Vrije Universiteit Brussel
Faculty of Science
Department of Computer Science
System and Software Engineering Lab

Dynamic Integration, Composition, Selection and Management of Web Services in Service-Oriented Applications

An Approach using Aspect-Oriented Programming

Bart Verheecke
September 2006

A dissertation submitted in partial fulfilment of the requirements of the degree of Doctor of Science

Advisor: Prof. Dr. Viviane Jonckers

Vrije Universiteit Brussel
Faculteit Wetenschappen
Vakgroep Informatica
Laboratorium voor Systeem en Software Engineering

Dynamische Integratie, Compositie, Selectie en Management van Web Services in Servicegeoriënteerde Applicaties

Een benadering gebruikmakend van Aspectgeoriënteerd Programmeren

---

Bart Verheecke
September 2006

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Promotor: Prof. Dr. Viviane Jonckers

Abstract

Web service technology is an open standards-based mechanism for communication over a network. Web services are simple, self-contained applications that perform functions, from simple requests to complex business processes. Client applications can integrate an existing Web service and communicate with it over a network, regardless of the hardware or platform used on either side of the wire. As Web services enable computer-to-computer communications in a heterogeneous environment, they are ideally suited for the Internet. We have observed that the implementation and deployment of a service-oriented application where several third-party Web services need to integrated, pose a variety of challenge.

The main concern of the research presented in this dissertation is the dynamic integration, composition, selection and management of Web services in client applications. The current approaches dealing with these processes fall short in providing the needed runtime flexibility to deal with various events in a dynamic services environment. Web services get hard-wired in the client, or only limited flexibility is provided to integrate functionally equivalent services. There is no explicit support for key requirements such as compositional mismatches, hot-swapping and service compositions. Furthermore, if client applications want to take into account Quality-of-Service properties in order to select the most optimal service for a given request, there is no way to enforce selection policies without explicitly providing code in the client in advance. Finally, we observe that invoking third-party services is far more complicated than making local method invocations. A variety of concerns needs to be dealt with, including exception handling, billing, logging, service monitoring, authentication, etc. All of these concerns need to be reflected in the code of the client. Current approaches require that code dealing with these concerns is provided explicitly and in advance, leading to tangled and scattered code while not being able to deal with unanticipated concerns.

In this dissertation we propose a mediation framework that takes care of all service related concerns for a client application. The framework, called Web Services Management Layer (WSML) introduces a flexible service redirection mechanism that allows for the runtime integration of services and compositions while taking into account selection policies and enforcing a range of client-side management concerns. As a lot of code dealing with the various identified service concerns results crosscutting in the client at those places where a service functionality is required, we opt to employ Aspect-Oriented Programming (AOP) to achieve a better separation of concerns. Service communication and composition details, selection policies and service management concerns are all ideal candidates to be modularized in aspects. As the WSML requires a lot of runtime flexibility, we opt to use a dynamic AOP approach, which allows for the hot deployment of aspects. The WSML offers support during the development, deployment and runtime cycle of service-based applications. To avoid that WSML administrators require aspect-oriented knowledge, the use of aspects is hidden by using aspect libraries and by doing automatic generation of aspect code. A prototype of the WSML has been developed in Java and JAsCo, a dynamic AOP language, and is deployed on a broadband provisioning platform of Alcatel.

Abstract

Web service technologie is een op open-standaarden gebaseerd mechanisme voor communicatie over netwerken. Web services zijn alleenstaande, zelfstandige applicaties die verscheidene functies uitvoeren, gaande van het afhandelen van simpele requests tot complexe bedrijfsprocessen. Klantapplicaties kunnen op eenvoudige wijze een Web service integreren en ermee communiceren, onafhankelijk van de technologieën die aan elke zijde van het netwerk gebruikt worden. Aangezien Web services de communicatie tussen computers in een heterogene omgeving mogelijk maken, zijn ze uitermate geschikt voor het Internet. We hebben vastgesteld dat het ontwikkelen en beheren van servicegeoriënteerde applicaties waar verscheidene Web services van derde partijen moeten geïntegreerd worden, heel wat uitdagingen stellen.

De hoofdinteresse van het onderzoek dat gepresenteerd wordt in deze thesis, is de dynamische integratie, compositie, selectie en management van Web services in klantapplicaties. De huidige benaderingen voor deze processen voorzien niet de nodige *runtime* flexibiliteit om te kunnen inspelen op de veranderingen die plaatsgrijpen in een dynamische serviceomgeving. Web services worden hardgecodeerd in de klantapplicatie, of enkel een beperkte flexibiliteit is aanwezig om functioneel equivalente services te integreren. Er is geen expliciete ondersteuning voor compositionele *mismatches*, dynamisch wisselen tussen services en servicecomposities. Verder kunnen klanten geen niet-functionele servicekwaliteit in acht nemen tijdens het service-selectieproces, tenzij hiervoor expliciet en op voorhand extra code in de klantapplicatie voorzien wordt. Tenslotte stellen we vast dat de invocatie van een service die onder het beheer van een derde partij valt, veel ingewikkelder is dan de invocatie van een lokaal object. Er dient rekening gehouden met verscheidene zaken zoals het afhandelen van uitzonderingen en fouten, betalingen, logging, monitoring, beveiliging, etc. Elk van deze concerns vereist extra code in de klant en de huidige benaderingen vereisen dat deze code op voorhand wordt voorzien op alle plaatsen waar service functionaliteit vereist is, wat leidt tot *crosscutting* code. Bovendien is het zeer moeilijk om concerns af te dwingen die niet op voorhand geanticipeerd zijn.

In deze thesis stellen we een mediatie framework voor dat zich tot doel stelt alle servicegerelateerde zaken voor klantapplicaties af te handelen. Het framework, Web Services Management Layer (WSML) genoemd, introduceert een flexibel serviceredirectiemechanisme dat runtime-integratie van services en composities realiseert. De WSML neemt de niet-functionele servicekwaliteit in acht tijdens het serviceselectieproces en biedt ondersteuning voor het afdwingen van verscheidene managementconcerns. Aangezien veel code die voorzien wordt in de klant voor deze concerns, *crosscutting* is, stellen we voor om aspectgeoriënteerd programmeren (AOP) aan te wenden om een betere modularisatie van de concerns te verkrijgen. Service-communicatiedetails, composities, selectieregels en managementconcerns zijn ideale kandidaten om in aspecten gemodulariseerd te worden. Aangezien de WSML in ruime mate flexibel moet zijn om veranderingen in de serviceomgeving op te vangen, maken we gebruik van een dynamische AOP-benadering, die toelaat om aspecten at runtime toe te voegen aan het systeem. De WSML geeft ondersteuning aan de applicatieontwikkelaar, zowel tijdens de ontwikkeling van de applicatie als *at runtime*.

Om te vermijden dat administrators aspectgeoriënteerde kennis nodig hebben, wordt het gebruik van aspecten verborgen door middel van automatische generatie van aspect code en het gebruik van aspectbiblioheken. Een prototype van de WSML is ontwikkeld in Java en JAsCo, een dynamische AOP-taal, en is ingezet op een breedband provisioning platform van Alcatel.

Acknowledgements

Writing this dissertation has been possible thanks to the help and support of many people. First and foremost, I would like to thank Prof. Dr. Viviane Jonckers for giving me the opportunity to obtain my Ph.D and for supporting me these past years. Next, I would like to express my appreciation to the persons I have worked with during the course of this dissertation: for the IWT Mosaic project I worked together with María Agustina Cibrán. A lot of the research described in this dissertation has been conducted in collaboration with her, for which I am very grateful. I want to thank Wim Vanderperren for his input on the aspect-oriented research and his invaluable technical support on JAsCo. I also want to express my gratitude to Ragnhild Van Der Straeten and Viviane Jonckers for proofreading this dissertation.

I would like to thank my Ph.D. committee members for taking the time to read this dissertation in detail and for providing me with valuable comments. Apart from my advisor Prof. Dr. Viviane Jonckers, the committee members are Prof. Dr. Bart Dhoedt, Prof. Dr. Theo D'Hondt, Prof. Dr. Geert-Jan Houben, Prof. Dr. Welf Löwe, Prof. Dr. Mario Südholt and Dr. Wim Vanderperren.

My other colleagues have always kept me motivated while helping in many ways. So thanks to Davy Suvee for implementing the incredible .NET QuoteWeb interface and for providing many apple diversions at the lab, and Bruno De Fraine for helping me to master Latex. Also thanks to Matthieu, Dennis, Niels and especially Maja and Ragnhild for putting up with the many lively research debates with Agus while writing their own theses in our open-plan office. In addition, I really appreciated the many medical chats with Dr. Ragnhild.

Next, I am extremely grateful for my family and friends who have supported me. My parents have always been there for me and helped in any way they can. Special thanks to my father for keeping me motivated when this was badly needed. He and my mother gave me the opportunity to study and to pursue any degree I wanted and they always supported any decision I made. I would like to thank Leen for being there for me these last years and for her moral support whenever I needed it. Also many thanks to Patje, Fredje and Dani-L. I realise how privileged I am to have so many friends who are part of my life and kept on motivating me during the last stressful months. A special thanks also to all my droomhuis roomies. Finally, thanks to Chantalle for the design of the cover page.

This research has been funded by the *Institure for the Promotion of Innovation through Science and Technology in Flanders (IWT)* and the *Research Foundation Flanders (FWO)*.

Contents

List of Figures

List of Tables

Code Fragments

Chapter 1

Introduction

1.1 Context

A Service-Oriented Architecture (SOA) is an architectural style for application integration. Its goal is to achieve loose coupling among interacting software agents, based on the notion of services [PG03]. A service is a unit of work done by a service provider to achieve a desired end result for its customers. One of the most suited technologies to realise a SOA is *Web Service Technology* because of its hardware and platform independent nature. Essentially, Web service technology provides a reasonably lightweight and open standards-based mechanism for computer-to-computer communications in heterogeneous environments. Web services are self-contained, self-describing modular components that communicate with clients using XML messages. Although a single Web service can implement and provide some useful functionality, the true value of Web services is realised when multiple Web services are composed together to offer some added value in a client application or another Web service. Possibly, a composed application or service needs to connect to a large collection of remotely hosted and managed Web services.

Web service technology has advantages in multiple domains. In the most straightforward scenario, a remote Web service is simply used to provide some fundamental piece of functionality to a client application. If the missing functionality is already available, it is more efficient to reuse it instead of rewriting it from scratch. Similar to the reuse of existing components in Component Based Software Development (CBSD), Web service technology can be adopted to reuse existing software components. The philosophy of CBSD is even taken to a next level, as Web services are already deployed components, realising resource sharing and component reuse in a hardware and platform independent manner.

Composing Web services together in a more complex fashion to realise a SOA also has potential in the Enterprise Application Integration (EAI) domain as Web services can enable interoperability between individual applications. The wide adoption of custom software throughout virtually every department of most companies has resulted in a vast array of useful but isolated islands of data and business logic [MSDN04]. Due to the varied circumstances under which each was developed, and the ever-evolving nature of technology, it is a complex task to create a functional assembly from these applications. By exposing the functionality and data of each existing application as a Web service it becomes possible to create a composite application that uses this collection of services to integrate an apparently disparate group of existing applications. As such, Web service technology is also a powerful approach to constitute end-to-end workflow solutions. These mechanisms are appropriate for long-running scenarios such as those found in business-to-business (B2B) scenarios. As different companies expose their business functionality to customers, a global market of services is realised. Service clients can then browse public market places to find services offering specific business functionality and integrate with them.

Basically, there are two ways to create a SOA: the first scenario involves the implementation of an inter- or intra-organisational process with a fixed number of partner roles. Based on a choreography or workflow specification, each partner implements a Web service to fulfil a specific role in the business process. Any modification to the process implementation in a later stage requires a new agreement between the partners before the modification can be deployed. The second scenario takes more advantage of the loosely coupled nature

of Web services: an application is built independently from any concrete services; partner roles are specified that need to be filled in by concrete services at runtime. In this approach, *just-in-time integration of services* becomes crucial: the entire process of service discovery, selection, composition and integration is deferred until runtime. This dissertation focuses on the second scenario: a client application is developed independently of the concrete Web services it requires; only at runtime, concrete third party Web services are integrated in the client.

1.2 Problem Statement

An impressive range of both commercial and open-source development tools enable the creation, deployment and management of Web services at the server-side [Chin03, KR03, Kirt00], but the dynamic integration of Web services at the client-side remains a hot research topic. After studying the current standards and practices in Web services, we observe a number of shortcomings that hamper the deployment of a SOA in a continuously evolving service environment. Although loose coupling is one of the key principles of a SOA and Web Service technology, we observe that service clients still need to co-evolve with the Web services they communicate with. We identified the following client-side issues:

1.2.1 Integration of Web services

Client applications that integrate Web services using current technologies are rather inflexible as they hard-wire references in the client application and offer only limited, anticipated support to redirect calls to other semantically equivalent services. This is however a serious limitation: because the context of Web services is an ever-changing business environment, the client application is not flexible enough to deal with changes in the functionality, displacement or usage of the integrated Web services. After all, services can become unavailable due to unpredictable network conditions or service failures, new services can become available and old ones can be taken out of business.

Furthermore, services can have incompatible interfaces requiring glue code in order to be integrated. It is also possible that services only offer partial compatibility, or that multiple services need to be combined together in a composition in order to be functionally compatible with the client. At the moment, a client application can only deal with these issues if code is explicitly provided in advance for these purposes. However, the code of the client application cannot deal with unanticipated events, code reusability is impossible as the code is context specific and last but not least, the code is hard to maintain as it may get spread around in the client at every place service functionality is needed. A more flexible service integration mechanism is needed that can adapt to these constant evolutions without having to stop, or worse, rewrite the client application code.

1.2.2 Selection of Web services

A second limitation encountered is that services can only be selected based on the functionality they offer. However, with a powerful and flexible service integration mechanism in place, the need arises to specify selection policies that guide the process of determining the most appropriate service for a given client request, based on non-functional properties. With the appearance of loosely coupled Web service technology, selection becomes more important as the whole integration and composition process becomes volatile. Today, selection might be based on the fact that all services must belong to a specific business partner, but tomorrow services need to offer a specific Service Level Agreement (SLA) in order to become eligible. Current service documentation does not support the explicit specification of non-functional requirements such as constraints based on Quality-of-Service, classes of service, access rights, pricing information, etc. The explicit specification of these non-functional properties at the service-side in a precise and unambiguous way would allow for a more intelligent and customised selection and integration of services at the client-side. This way, client applications are able to base their decision on business requirements in order to integrate the services that best fit the applications needs. Also Web service behavioural properties such as the actual response time or the uptime over the last month may be candidates to guide the selection process. In that case, more advanced monitoring of the services is required. Again, these requirements may evolve over time: today the fastest service might be preferred, but tomorrow the cheapest service might be more appropriate. At the moment, there is no uniform way to represent and specify these selection policies, independently from the client application, and to enforce them in the client at runtime. A more advanced mechanism dealing with service selection is needed.

1.2.3 Client-side Management of Web services

To aid in the construction of large-scale distributed systems, many software developers have adopted middleware approaches. Middleware facilitates the development of distributed software systems by accommodating heterogeneity, hiding distribution details, and providing a set of common and domain specific services. However, as pointed out in [CBR03], middleware itself is becoming increasingly complex; so complex in fact that it threatens to undermine one of its key aims: to simplify the construction of distributed systems. Additionally, [ZJ03] describes that the sheer volume of middleware standards and technologies as being a contribution to this complexity. Middleware particularly suffers from increased complexity when addressing concerns of a crosscutting nature.

In case of Web services, service invocation become increasingly complex as additional code is required to deal with various management concerns. Besides the fact that one is dealing with remote procedure calls, the issue arises that the client application is becoming dependent on services that are not under control of the owner of that application. This is referred to as *organisational fragmentation* of the application: the application becomes a multi-partner process, introducing a wide range of issues, including latencies, asynchronous Web service invocations, a myriad of separate failure nodes, security issues, etc. [Szyp01]. An important obstacle arises as the Web service most likely enforces a set of concerns on the client: e.g., the communication with the service needs to be encrypted, the client needs

to authenticate itself, payments are needed before the service can be used, etc. All of these concerns will need to be reflected in the code of the client. And as the services belong to different domain controllers, these requirements might change independently, on a frequent basis and even without notice.

Another issue involves the set of concerns the client wants to enforce to guide the service invocation process, for instance to deal with service failures. Other examples include the deployment of a caching mechanism to avoid expensive calls over the network, or the use of pre-fetching mechanism to avoid long waiting times. For low-level concerns such as providing a client ID in every message sent from a client to a service, *message handlers* provide a limited solution, as this approach provides a mechanism for adding, reading and manipulating the header blocks of the messages sent to and received from services. However, higher-level concerns whose deployment is not limited to the message handling level are not supported. To deal with issues such as multiple advanced billing scenarios, additional code has to be included in the client application, possibly resulting scattered and tangled with code addressing other issues. Even if this code is encapsulated in a separate reusable module, its execution has to be triggered repeatedly from the different points in the application where Web service functionality is required. As a consequence, management code results duplicated and scattered over the application, becoming an obstacle for future maintenance. Furthermore, these dependencies create a burden for the developer, as he has to reconfigure, stop and sometimes even rewrite client code whenever the service changes. A flexible client-side management mechanism is required to deal with these concerns in a transparent way for the client application.

1.3 Research Objectives and Approach

1.3.1 Objectives

The observation that service clients need to co-evolve with the Web services they integrate with is not surprising as it is practically impossible to eradicate incompatibilities between two independent parties through standardisation. Despite the wide variety of available standards and the continuous investment in standardisation initiatives, there will always arise problems due to political, technical, geographical or philosophical boundaries and due to required adaptations to the context in which the application is deployed. Instead of relying completely on standardisation it is a good design practice to assume incompatibilities will arise, and to put mechanisms into place to easily resolve them. However, the current Web service integration approaches fail largely in providing such mechanisms. Our goal is not to merely fix the flaws in the current Web services stack and technologies, but to provide a solution for dealing with dynamic and heterogeneous services environments without depending on the current status of the Web services standards, as they will evolve tremendously over the next years.

A global objective of this dissertation is to develop a mediation framework for the integration, composition, selection and client-side management of Web services targeted at a continuously evolving service environment. This framework must offer support during the development, deployment and runtime cycle of service-based applications while meeting

real-world requirements. The premise of this dissertation is that Web services are offered on a network, typically the Internet, by a series of third-party service providers, while imposing restrictions and requirements on their clients. It is up to the clients to comply with these requirements trough mediation rather than standardisation. Therefore, the proposed framework must allow for the complete decoupling of the Web services consumed in a client application and adapt to any requirements and evolution in these requirements, imposed by the service environment.

We observe that code dealing with service integration and composition result scattered and tangled in the client application. Furthermore, the enforcement of various selection policies and client-side service management concerns clutters the client code even more. As each of these concerns may evolve independently, we strive to a better *separation of concerns* [Parn72] in the implementation. To obtain our global objective, we propose to adopt *Aspect-Oriented Programming* (AOP) principles [KLM+97, EFB01, FICA04]. With AOP, crosscutting concerns are modularised in additional module constructs, named *aspects*. These aspects can be woven into the core application at well-defined points. We claim that service communication and composition details, selection policies and management concerns are all suitable candidates to be modularised in aspects. In addition, the runtime adaptations required in our service context can be realised by exploiting dynamic aspect-oriented technologies. With dynamic AOP, aspects can be introduced in the core application using runtime weaving techniques. By adopting dynamic AOP, the necessary flexibility is provided for successfully realising just-in-time service integration and to adapt to changes in the service environment. This leads us to the following thesis hypothesis:

> *"By modularising service related concerns into aspects and by exploiting dynamic aspect-oriented technologies, a reusable mediation framework for the just-in-time integration, composition, selection and client-side management of Web services in a continuously evolving service environment is realised."*

1.3.2 Our Approach

1.3.2.1 Web Services Management Layer (WSML)

To address the shortcomings of existing Web service integration approaches, which are analysed in detail in chapter 3, we propose an architectural framework for the mediation of Web services in client applications. An abstraction layer, called Web Services Management Layer (WSML), is placed in between the client and the world of Web services, as depicted in Figure 1.1. On the left-hand side, the client requests service functionality without referencing the concrete Web services. The WSML is responsible for intercepting the client requests and choosing the most appropriate service available on the right-hand side or combining a number of services together in a composition, invoke them in a manner that complies with any management requirements imposed by service providers and return the results to the client. Using the WMSL for service-oriented applications has the advantage that more robust and flexible applications can be developed without having to rewrite service-related code. Furthermore, replacing the Web service-specific invocations with generic service requests and extracting all extra web-service selection and management code from the client applications

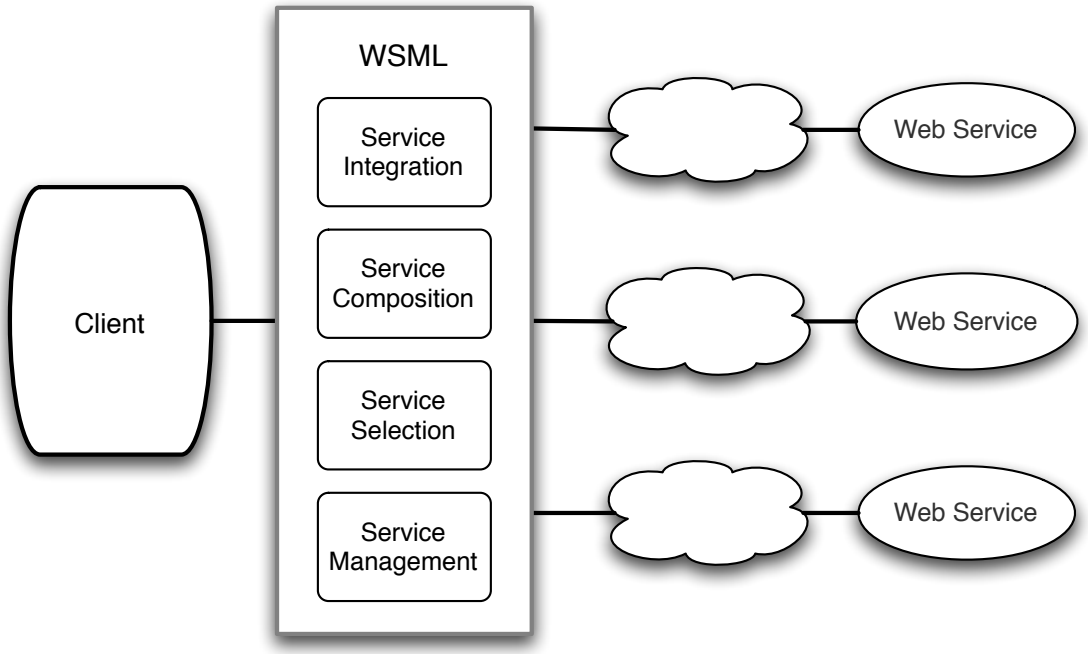

Figure 1.1: The Web Services Management Layer

facilitates future maintenance of the client application code. The main architecture of the WSML is presented in chapter 4.

In short, the WSML is responsible for the entire process of discovery, selection, composition, integration and client-side management of all Web services consumed in the client application. The WSML approach builds on top of existing Web service standards, and fits into the paradigm of the Enterprise Service Bus (ESB). An ESB is software infrastructure that simplifies the integration and flexible reuse of business components using a SOA. The WSML, with its capability to dynamically connect, mediate and control the communication between service requestors and service providers, fits perfectly in this vision. The WSML can run in the same environment as the client or it can be deployed on a different machine as a dedicated server. The latter scenario has advantages in a large-scale environment were multiple clients need to be managed.

To realise a flexible and dynamic service framework, we argue that Aspect-Oriented Software Design (AOSD) principles [KLM+97, EFB01] can be of great benefit. Using current approaches, code tackling different service related concerns, results tangled and scattered in the client. Using Aspect- Oriented Programming (AOP), these concerns can be nicely modularised and can evolve independently of the client. Furthermore, by using a dynamic AOP technology, these concerns can be plugged in and out at runtime, depending on the client and service environment. Chapter 4 gives an overview of the principles of AOSD and argues why AOP is a suited technology to implement the WSML. Also, the dynamic AOP language, JAsCo [SVJ03], is presented. With JAsCo, aspects can be specified in a reusable way, and deployed at runtime using dedicated connectors. The JAsCo technology

is based on a genuine runtime weaver [VS04, DVS05] that supports the physical weaving, unweaving and reweaving of aspects at runtime, even at previously unadvised joinpoints. The code examples, given in the remaining chapters are given in JAsCo, unless indicated otherwise, and the prototype of the WSML is implemented in this language. The prototype, and the tool support offered in our framework during the development, deployment and runtime cycle of a SOA are discussed in chapter 8.

1.3.2.2 Just-in-time Integration of Web Services with WSML

When the WSML intercepts a generic service request from a client, it must redirect this call to the most appropriate service. First, a pool of functionally compatible services must be available. Services are found on the network through a service discovery process. Next, functionally compatible Web services are integrated during a service integration process. After this, the WSML is able to redirect client calls to these integrated services. The WSML will decide which of the services in the pool is the most appropriate based on a set of, possibly cooperating, selection policies. This process results in the selection of the most optimal service, which will be addressed in the service invocation process. Alongside these processes, the WSML is also involved in the client-side management of all registered services.

Figure 1.2 depicts the development, deployment and runtime stages of a client application cooperating with the WSML. In the development phase, the client application is implemented and required service functionality is described by *Service Types*, i.e. abstract service interfaces that do not reference concrete services. At deployment time, the client application is deployed, together with the WSML hosting the service type(s). Next, concrete Web services can be integrated and addressed at runtime. This just-in-time integration process of Web services consists of four steps:

Service Discovery: Available Web services are discovered on the network using an automated look-up mechanism or a registration process. Typically, service providers register their services in a registry or repository where potential clients can find them.

Service Integration: A matchmaking process will determine the level of compatibility between the service types and the found Web services. A mapping process is needed to resolve incompatibilities between both parties. In a composition process multiple services are composed together to offer the required functionality.

Service Selection: Whenever multiple Web services or service composition are available to deliver the functionality described by a service type, a selection process is needed to determine which service(s) are most appropriate. This process is based on a set of selection policies, specified by the client application.

Client-Side Service Management: To cope with the more complex nature of service invocations, additional management concerns including security, logging and payments need to be enforced in the client.

Each of the processes will be discussed in further detail in the next four subsections.

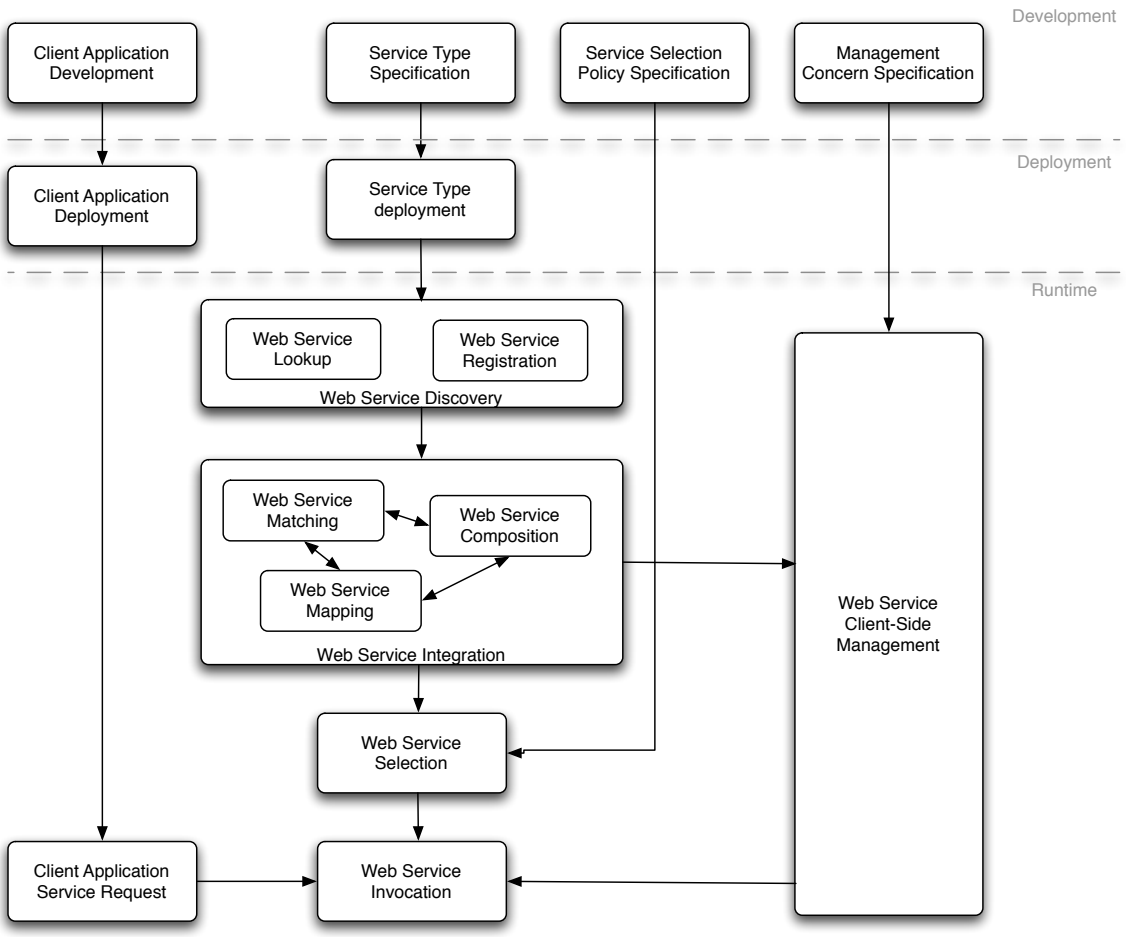

Figure 1.2: Just-in-time Integration of Web Services with WSML

1.3.2.3 Dynamic Service Discovery

Integrating a remote Web service in a client application starts with the process of service discovery¸ i.e., finding somewhere a service that actually delivers the functionality needed in the client. A Web service broker can be used for this purpose: the broker looks up a suitable Web service in a repository or registry and returns information to the client about the service so that the client can invoke it. This process depends heavily on the available service documentation: the more information is available in the repository, the easier it becomes to find the service that best fits the needs of the client. Adequate service documentation includes business information about the service provider, the service interface, the semantics of the offered functionality, technical binding details, etc. At the moment, Universal Description, Discovery, and Integration (UDDI) [BCC+04b] is the standard to discover Web services. The UDDI specification defines open, platform-independent standards that enable businesses to share information in a global business registry, discover services on the registry, and define how they interact over the Internet.

Currently, only private UDDI-based repositories are used, but it is expected that over time public repositories will become widely available. Typically, browsing for compatible Web service is done manually at development time of the client. Using taxonomies and classification schemas, compatible services can be found and integrated. Automating this process is a research domain on its own. Approaches can be found in the ontology domain: by semantically annotating all service capabilities [ABH+02, PV04] and by advanced querying of the repository, compatible services can be looked up. Researching more advanced service documentation, and automating service discovery is not part of this dissertation. For the main part of this work, we assume a mechanism is in place to create a service pool of functional compatible or partially compatible Web services. This mechanism can be a manual process where Web services are looked up in UDDI-like repository and added to the pool. Or it can be a more automated process, for instance based on ontologies. While it is not part of the core of this dissertation, we have done experiments [CVS+04b] with enriching Web service documentation and service requests by adding ontological data in order to assist in the process of service discovery and integration. These results are described in chapter 8.

1.3.2.4 Dynamic Service Integration

With a service pool of compatible services available to the client, a mechanism is needed to integrate and invoke these services at runtime. Current integration approaches are typically based on client-side proxies: a proxy is generated at client-side based on a functional service description in the Web Services Description Language (WSDL) [CCMW03]. WSDL provides an abstract way to describe the capabilities of Web services. It describes for service clients how to format service requests, and how service responses will look like. WSDL also defines a service's binding to a network transport protocol, usually HTTP. This information suffices to create a client proxy that acts as a local representative of the remote service. By invoking a method on this proxy, a Service-Oriented Access Protocol (SOAP) [GHM+03] message is created and sent to the service and the result is passed back to the client. However, this results in a tight coupling between the client and the service, as the service interface

is hard-coded in the client. Even more flexible solutions such as dynamic proxies and dynamic interfaces [Sun05] which allow for more decoupling still put many restrictions: only services that exactly implement the same interface can be integrated, or they leave the whole mapping between interface differences to the client. In our approach, the WSML takes care of the integration in a completely transparent way for the client. The client communicates with abstract *service types* that are mapped by the WSML to concrete services. Any kind of service that offers the required functionality can be addressed as our mechanism offers full support for compositional mismatches, i.e., allowing for the integration of services with different interfaces; and for service compositions, i.e. services that collaborate together to offer the required functionality to the client. The mechanism also supports conversational messaging, i.e., when both the client and the service need to keep track of the state of the conversation, for instance to improve performance, or to implement specific business logic.

With our approach, the application becomes more flexible as it can continuously adapt to the changing business environment and communicate with new services that were unknown or unavailable at deployment time. By realising full decoupling between the client and the services, *hot-swapping* can be realised when a service becomes unreachable due to network conditions or service-related problems. In the WSML, we modularise Web service communication details and composition patterns in *redirection aspects*, and we exploit advanced dynamic AOP techniques to realise this integration mechanism. The results of this research are the subject of chapter 5.

1.3.2.5 Dynamic Service Selection

With the powerful service integration mechanism of section 1.3.2.4 in place, the need arises to specify service selection criteria. As such, applications are able to base their decision on business requirements in order to integrate the services in a more intelligent and customised fashion. A selection policy specifies a constraint that should be enforced by the service integration mechanism. We have identified different kinds of selection policies, based on the kinds of events that may trigger a policy. Policies can be triggered due to client-side events, service-side events, or they can be based on non-functional properties advertised in service documentation or even on the runtime behaviour of the services. Depending on the type of selection policy, more advanced service documentation might be required. The documentation provided in WSDL format for example, does not support the explicit specification of non-functional requirements. Research efforts, e.g. the Web Services Offering Language (WSOL) [TPP03], are currently undertaken to provide a more expressive documentation language. In this dissertation we do not attempt to enhance the service documentation but expect that the current efforts will ultimately lead to a standardisation process. Our approach focuses on enforcing the selection policies at runtime and it is assumed this process can be made compatible with any kind of documentation format.

A selection policy can specify a hard constraint on an individual service or might specify a soft constraint on multiple services. Some constraints can be enforced at any given moment while other ones only over a period of time. In our approach, selection policies can be specified independently of the client, and enforced at runtime without changing any code in the client or service. At specification level, policies are specified in a dedicated XML configuration language, and at the implementation level, *selection aspects* represent policies.

Encapsulating selection policies in aspects has the advantage that the code representing the policy is modularised, and that they remain a first-class entity in the code. Using dynamic AOP, the policies can be enabled and disabled at runtime, reflecting at any time the business requirements of the client. To monitor the behavioural properties of services over time, we propose a mechanism based on *monitoring aspects* to introduce measuring points non-invasively. By providing a library of reusable monitoring and selection aspects in the WSML, service selection becomes possible through configuration and administration of the WSML, rather than through additional implementation efforts. This also eliminates the requirement for the administrator of the WSML to have aspect-oriented knowledge, which enhances usability. Chapter 6 discusses our research on service selection and monitoring.

1.3.2.6 Client-side Service Management

To cope with the more complex nature of service invocations, additional client-side code dealing with various management concerns is required. As these concerns may vary over time, a flexible mechanism is needed to enforce these concerns *non-invasively*. We propose a mechanism, complementary to SOAP message handlers, that deals with higher-level client-side management concerns. Again, we opt to cope with these concerns by modularising them in *management aspects* and deploy them for those services that require it at runtime by using dynamic AOP. While SOAP message handlers can only be triggered when messages come in or are sent out to a service, our approach is more flexible as it benefits from the richer expressiveness available in AOP languages. Using aspects to implement the management concerns has similar benefits as using selection policies: each concern is cleanly modularised in one aspect, non-anticipated concerns can be implemented in aspects and enforced in an oblivious manner in the client, and code reusability is achieved by generalising the concerns in patterns. An aspect library is available for a wide range of concerns, and configuration using an XML configuration language allows for the instantiation and deployment of the aspects. Again, this eliminates the need for the WSML administrator the have aspect-oriented knowledge.

An important issue that arises when various concerns are introduced in the system at runtime, possibly at the same points in the system, is *feature interaction* as a result of the fact that the concerns are not always orthogonal [Za03]. For example, if a billing mechanism is enforced to pay the service before it is invoked, and a caching mechanism retrieves cached results instead of actual invoking that service, then it is possible that both concerns interfere with each other. In AOP, several approaches have been proposed in order to make the composition of aspects more explicit, examples are Strategic Programming Combinators [LVV03] and treating aspect composition as function composition [WKL03]. In the WSML, a programmatic approach of JAsCo for explicitly representing aspect compositions, named combination strategies, is employed to avoid interaction issues.

Client-side management of Web services is the topic of chapter 7. Examples of supported management functionalities are:

- **Caching**: to avoid slow response times, lost packages, congested networks, and reduce the network traffic a caching mechanism can be considered. Instead of invoking the

services each time the application requests specific service functionality, the results are retrieved from a cache.

- **Billing**: services can have a variety of billing strategies. The client application might want to control how billing is applied, for instance for auditing purposes. Analogously, services might need to be billed as a requirement for their integration with a client application.

- **Logging**: many industries are required to provide tracing and logging capabilities for accounting as well as regulatory purposes.

- **Versioning:** new versions of a Web service, offering new capabilities, might be introduced alongside an older version, or replacing it.

- **Fallback:** a variety of client-side scenarios are possible to recover from a service failure, including compensation handling, hot-swapping to another service, sending of notification, etc.

- **Security**: Web services can use several security mechanisms (e.g., WS-Security) depending on the environment where they are deployed. Data sent over the network can be encrypted and user authentication and authorisation might be required.

1.4 Contributions

The following are the contributions of this dissertation:

- An analysis is made of the requirements needed for Web services middleware in dynamic service environments and it is shown how current approaches and state-of-the art tools fail in achieving these requirements

- Based on this requirements analysis, an architecture for a reusable service mediation framework, is designed. This framework is targeted at the the integration, composition, selection and management of Web services in service-oriented applications. Aspect oriented design principles are adopted to implement this framework and employ dynamic AOP technologies for flexible and dynamic configuration and to deal with evolutions in the service environment.

  - A dynamic service integration mechanism for Web services with support for compositional mismatches, service compositions, synchronous and asynchronous communication and conversational messaging is realised. This AOP-based mechanism works completely transparent for the client.
  - A service selection mechanism is proposed to guide the runtime selection of services based on the client or service context, and on non-functional and behavioural service properties. Using aspects, selection policies are enforced and monitoring points are installed at runtime in a non-invasive manner.

  - A client-side service management mechanism for the enforcement of various service and client-side driven concerns is proposed. Management aspects are employed to enforce these concerns obliviously in the client and explicit combination strategies help in resolving feature interaction problems.

- A prototype of the WSML written in Java and JAsCo and running on a production scale server has been developed. Tool support is available for the development of service-oriented applications in Eclipse, a state-of-the-art IDE. Runtime configuration of the WSML is possible through web-based administration interfaces and XML-based configuration languages.

- The WSML is successfully deployed on a Service Enabling Platform (SEP), an open telecom platform for broadband service delivery of Alcatel Bell. The WSML is integrated in a SEP prototype to facilitate the integration with different content and mobile phone providers by using Web service technologies and dynamic AOP.

These contributions are presented in various research conference and journal papers, both in the Web service and in the aspect-oriented community. First, the Web service related publications are listed: the main idea of introducing a mediation framework for Web services at the client-side and the adoption of aspect-oriented principles for its implementation were first introduced at, and published by [CV03] and in more detail at [VCJ03], focusing mainly on the dynamic integration mechanism. These results were updated and republished in [VCV+04]. The approach using aspects for client-side management concerns and solutions for feature interaction were presented at, and published by [CVJ03]. Adopting AOP for Web service selection and monitoring was the subject of a second publication at [VCJ04]. All of these topics were merged, updated and published in [CVV+06]. More advanced redirection using stateful aspects for conversational messaging was presented in a paper published at [VJ05]. The research results using ontologies for service documentation and integration appeared in the proceedings of [CVS+04b].

Publications in the aspect-oriented community include an analysis of the applicability of dynamic aspect technologies for Web services management at [VC04] and the explicit representation of business rules and selection policies by aspects for service composition at [CV05]. An experience report of the deployment of the WSML in a telecom environment was also presented at [VC05]. A complete high-level overview of the applicability of several aspect categories to unravel crosscutting concerns in Web services middleware is given in [VVJ06]. Validation of the research was possible through various formal research demonstrations at [VSV+04], [CVS+04a] and [VSC+04]. The WSML was also the subject of two demonstrators for the mid-term and end-term review of the industrial IWT-project MOSAIC[1], in collaboration with telecom partner Alcatel Bell.

[1]The research presented in this dissertation is partially conducted with the support of the Flemish Government Project MOSAIC, funded by the "Institute for the Innovation of Science and Technology in Flanders" (IWT) ("Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen")

1.5 Outline

The diagram on the next page outlines the remainder of this dissertation.

Chapter 2 – Web Services Technology
The key principles of Service-Oriented Architecture (SOA) are explained in this chapter, together with Web services as the middleware technology to achieve platform and hardware independent application integration. The current Web services stack is discussed, as well as the available tool support to implement and deploy Web services. The chapter concludes with a comparison of related middleware approaches.

Chapter 3 – Dynamic Web Services Environments
This chapter starts with the introduction of a running example of a travel agent application, deployed in a dynamic environment where changes occur continuously in the service environment, the network and the client. Therefore, the service integration process, the service selection and the client-side management of the Web services need a high level of flexibility. An evaluation of current practices is made with respect to the identified requirements and it is illustrated how these approaches fail at providing the needed flexibility to deal with any environmental changes.

Chapter 4 – Web Services Management Layer
To deal with dynamic service environments, we introduce a mediation framework for Web services, called Web Services Management Layer (WSML). This chapter presents several usage scenarios, the pursued development quality attributes and a high-level architecture for the framework. To avoid crosscutting code, we opt for a dynamic Aspect-Oriented Programming (AOP) approach. An introduction and motivation to AOP and an architecture of the WSML, based on aspects are presented. Also, the dynamic AOP language JAsCo is introduced.

Chapter 5 – Dynamic Service Integration

Chapter 6 – Web Service Selection

Chapter 7 – Client-Side Service Management

Continued on next page

Chapter 5 – Dynamic Service Integration
Just-in-time integration of Web services while avoiding hard-wiring service interfaces in clients, is the topic of this chapter. Redirection aspects modularising service communication and composition details are introduced and the usage of aspects in scenarios, such as asynchronous and stateful conversational messaging are discussed.

Chapter 6 – Web Service Selection
To realise more intelligent service selection, policies are enforced to guide the selection process. A categorisation of policies is made, and for each category it is shown how aspects are used to enforce them. Optionally, monitoring aspects are used to setup measurement points in the system to collect any required monitoring data.

Chapter 7 – Client-Side Service Management
Several service management concerns need to be enforced in the client when dealing with Web service invocations. In this chapter we discuss how modularising concerns in aspects helps in avoiding crosscutting code and how each concern can be enforced non-invasively at runtime while avoiding conflicts.

Chapter 8 Prototype
This chapter starts with an overview on how an implementation for the WSML aspects can be made. Several options including semantic matchmaking, high-level service composition specifications and aspect libraries are discussed. Next, a prototype of the WSML framework, implemented as a proof-of-concept in Java and JAsCo, is presented. Its architecture, the provided tool support and the realised development quality attributes are discussed. As a case study, the WSML prototype has been integrated with the Service Enabling Platform (SEP) of Alcatel Bell, a provisioning system for broadband Internet applications.

Chapter 9 Conclusions
This chapter concludes the dissertation. An overview of our ideas is given and future work is discussed.

Chapter 2

Web Services Technologies

Abstract The key principles of Service-Oriented Architecture (SOA) are explained in this chapter, together with Web services as the middleware technology to achieve platform and hardware independent application integration. The current Web services stack is discussed, as well as the available tool support to implement and deploy Web services. The chapter concludes with a comparison of related middleware approaches.

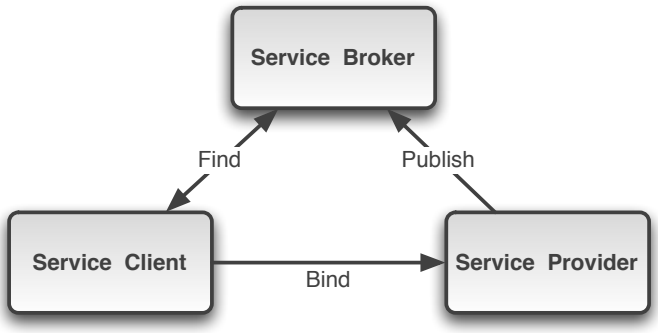

Figure 2.1: Service-Oriented Architecture

2.1 Introduction

A Service-Oriented Architecture (SOA) is essentially a collection of services and service clients that communicate with each other. The communication can involve either simple data passing or it can involve two or more services coordinating some activity. A SOA is called service-oriented because the central idea is that a service client needs a particular set of services in order to operate. Before the client can request the service, it needs to find a service provider. A service broker typically operates a repository to provide this location service. Upon request, the service broker returns a document that allows the client to first locate and then bind to the provider. Thus, the three key roles in an SOA are service client, service broker and service provider. Three fundamental operations exist: publish, find and bind. Service providers publish services to a service broker. Service clients find required services using the service broker and bind to them. This is depicted in Figure 2.1.

Service-oriented architectures are not a recent invention. In the past, clients accessed services using a tightly coupled, distributed computing protocol, such as Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), The Open Group Distributed Computing Environment (DCE) or Microsoft Distributed Component Object Model (DCOM). While these protocols are very effective for building a specific application, they limit the flexibility of the system. The tight coupling used in these approaches limits the reusability of individual services. Each of the protocols is constrained by dependencies on vendor implementations, platforms, languages, or data encoding schemes that severely limit interoperability. And none of these protocols operate effectively over the Web. As a result, these systems are expensive to implement and create proprietary one-to-one connections. The consequence was a multitude of Electronic Data Interchange (EDI) standards, with a distinct system for every industry.

One of the challenges of performing integration using these traditional middleware technologies is the lack of a universal protocol. With the advent of Web Services, the SOA concept has been more widely adopted because of the use of standards-based technologies. Web Services are based on the Extensible Markup Language (XML), a common language for describing data and the Service-Oriented Access Protocol (SOAP), which has become the primary de facto standard protocol for performing integration between multiple platforms and languages.

2.2 Web Services Definition

Web Services Technology allows applications to expose their business functionality as a service to be invoked by other applications over a network, in a *platform, language and vendor independent manner*. A Web service is a self-contained, modular application that can be described, published, located, and invoked over a network using industry standards. Web services do not provide a user interface, but instead, share business logic, data and processes through a programmatic interface across a network. Web services are characterised by a great interoperability and extensibility as they can be combined in a loosely coupled way in order to achieve complex operations. As such, organisations can communicate data without needing to know any details about each other's IT infrastructure behind the firewall.

It is a misconception that Web services are a reincarnation of distributed object technology [Vogels03]. Web services allow for the creation of distributed systems, as do CORBA, Java RIM and DCOM. The big difference is that Web services can be based on exchanging XML documents rather than doing invoking Remote Procedure Calls (RPCs) on distributed objects. Fundamental to Web Services is the notion that everything is a service, publishing an API for use by other services on the network and encapsulating implementation details. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services. Essentially, the term Web services describes a standardised way of integrating Web-based applications using a series of XML standards over a network. The following standards make up the most basic form of the Web services protocol stack (Each of these standards will be discussed in more detail in the next section):

- **Service Oriented Access Protocol (SOAP)** used to transfer the data between the entities.
- **Web Services Description Language (WSDL)** used for describing the services
- **Universal Description, Discovery and Integration (UDDI)** used for listing what services are available.

A simple definition for Web Services from Gartner reads: "*loosely coupled software components that interact with each other via standard Internet technologies*". In 2002 the Word Wide Web (W3C) consortium undertook an effort to standardise Web Services and their vocabulary. The Web Services Architecture Working group finished in January 2004 and published a Web Service Glossary [W3C04]. This document defines Web Services as follows:

> *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards."*

Because of their platform and vendor-neutral characteristics, Web Services have the advantage to succeed as middleware technology in the heterogeneous Internet environment. The following are the key properties of Web Service Technology:

- **Loosely Coupled**: Two systems are considered loosely coupled if the only mandate imposed on both systems is to understand the self-describing service interfaces and the messages sent between them. It also means that each system exists independently of the other systems it interacts with. This allows individual pieces of the application to be modified without impacting unrelated parts. Tightly coupled systems on the other hand, impose a significant amount of customised overhead to enable communication and require a greater understanding between the systems. Web Services are considered loosely coupled because of their use of open standards and how they interoperate.
- **Universal Data Format**: By adopting existing, open XML-based standards over proprietary, closed-loop communication methods, any system supporting the same open standards is capable of communicating and understanding autonomous and disparate systems that are exposed as Web Services. The broad industry adoption of these standards avoids that companies need proprietary technologies that may lock them in.
- **Ubiquitous Communication**: A wide variety of platforms and devices are being connected with each other over the Internet, therefore providing a ubiquitous communication channel. These platforms are becoming more diverse and employing standardised communication over existing ubiquitous transport protocols such as HTTP guarantees their interoperability. As will be discussed later, Web Services can run on a variety of communication protocols.

The Web services standards are maintained by independent, non-profit standards organisations composed of a diverse membership to drive re-use and interoperability. Members submit various requirements for the standards and agree to a specification after many review phases, resulting in free and open standards upon which any group can build Web service-compliant applications and tools. A few of the major Web services standards groups are the World Wide Web Consortium (W3C), OASIS and the Web Services Interoperability Organization (WS-I).

2.3 Web Services Interaction

A Web service typically exposes coarse-grained enterprise services that encapsulate some core business. A Web service client can communicate with a Web service in two patterns: RPC-based and document-based.

2.3.1 RPC-based interaction

With the RPC-based interaction, the operations of the publicly exposed interface of the Web service map directly to discrete operations exposed by the underlying application. The XML of the SOAP messages exchanges is formatted to map to the discrete operations exposed by that application. Typically, service invocations occur over a synchronous transport protocol such as HTTP, where the SOAP request and response uses the protocol-level request-and-response. However, asynchronous interaction patterns and other transport protocols are

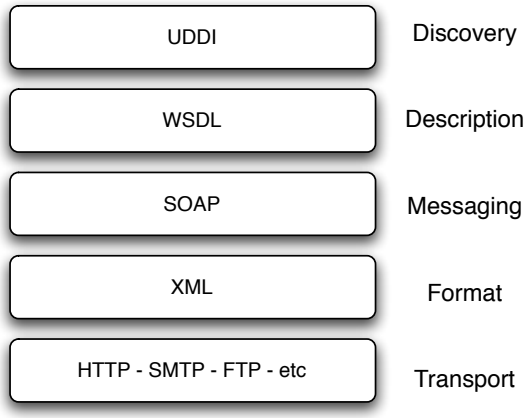

Figure 2.2: The Web Services Protocol Stack

also possible to form synchronous request-response interaction patterns [Tyagi04]. RPC is a traditional programming model and is at the moment the most popular interaction model for Web services.

2.3.2 Document-based interaction

Document-based interaction is a fairly new concept. In a document-based interaction, the service consumer interacts with the service using documents that are meant to be processed as complete entities. In other words, the document represents a complete unit of information and may be completely self-describing. These documents are typically formatted as XML but documents can also be sent in other formats. Such document-based interactions are typically long-lived in nature. Because the response cannot be returned immediately document-based message exchanges are typically asynchronous. The effort and complexity involved in building a document-oriented Web service is usually more than the effort involved in using an RPC-based architecture. This is because it involves extra steps, such as designing the schema for the documents that will be exchanged, negotiating and arriving at an agreement with business partners on that design, and validating the document against the schema [Tyagi04].

2.4 The Web Services Protocol Stack

In this section we take a closer look at the different layers upon which Web services technology is built. Figure 2.2 charts the used protocols in a Web Service Stack. Each of the following subsections deals with one layer of the stack. A reader who is already familiar with a concept can skip that particular section.

2.4.1 The Transport Layer

The Transport Layer defines how messages are sent from one service to another over the network. The upper layers of the Web Services stack do not pose any restrictions on which transport protocol is used, but typically HyperText Transfer Protocol (HTTP), is employed. HTTP is an application-level communication protocol for distributed, collaborative, hypermedia information systems and is being used since 1990 on the World Wide Web. It is a generic, stateless, request/response protocol. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred [FGM+99]. HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80 [RP94]. As a result, Web services running over HTTP can straightforwardly bypass the firewall of an enterprise, as port 80 is usually open to access the World Wide Web. Other possible transport protocols for Web Services include the Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), and Java Message Service (JMS). Most of the discussions and examples in this dissertation assume HTTP as transport protocol as it is the most commonly used and preferred choice of protocol with current SOAP-based systems.

2.4.2 The Format Layer

The information sent between two Web services is formatted using the eXtensible Markup Language (XML), a specification for defining and organising structured data in text format. XML is a markup-language that can be used to create self-describing, modular documents (data), programs, and even other languages, commonly referred to as XML grammars. The real value of XML is not in its innovativeness but rather its industry acceptance as a common way of describing and exchanging data between otherwise incompatible systems [Ngh02]. Every major vendor has announced support for XML in one form or another, and innovative uses for XML are emerging almost daily.

A *well-formed XML document* follows the syntax of XML and can be completely processed by an XML parser. If there are syntax errors in the document, then a parser typically rejects the entire document. A *valid XML document* is a well-formed document that can also be verified against a set of constraints, defined on the individual elements in the document through document type definitions (DTDs) and XML schemas. A DTD is an external document that acts as a template against which an XML document is compared. While DTDs are useful for data enforcement and validation of XML documents, they have a number of critical shortcomings. To address several issues and shortcoming of DTDs[1], the W3C produced the XML schema specifications [FW04]. XML schemas do provide support for namespaces, predefined and user-defined data types. With XML Schema, one can express syntactical, structural and value constraints applicable to the elements, e.g. to specify what elements are allowed in the documents, whether they are mandatory, and if there is an upper bound on the number of an element's occurrence.

Because XML is a text-based language, it is verbose and therefore human readable. As

[1] As described in [Ngh02], DTDs lack support for data types, data formats and namespaces (i.e. a mechanism to avoid name collisions, similar to the concept of packages in Java). Additionally, SOAP, one of the cornerstone technologies of Web Services, prohibits the use of DTDs in the document declarations

a downside, documents with complex data sets can quickly become very large. As it was never designed with conciseness of encoding or efficiency of parsing in mind, its usage for messaging purposes in Web services does have a performance impact. Web Services do suffer form poor performance compared to other distributed computing approaches such as RMI, CORBA, or DCOM. Data is represented inefficiently, and binding requires more computation. Research initiatives are undertaken to improve performance. For instance, the Fast Infoset standard draft specifies a binary format for XML infosets that is an efficient alternative to XML [STP04]. Fast infoset documents are faster to serialise and parse, and smaller in size, than the equivalent XML document. Note that the flexible and extensible nature of XML also is a weakness: without a standardisation process, different formats for the same data structures emerge, which results again in incompatibilities [Ngh02].

2.4.3 The Message Layer

The Message Layer of the Web Services stack consists of the Service-Oriented Access Protocol (SOAP) [GHM+03]. SOAP is a specification that defines the XML format for messages sent between Web services and clients. SOAP originally meant Simple Object Access Protocol, but the term has been unofficially redefined to mean Service-Oriented Access Protocol. It is a lightweight communication protocol, without any advanced features. SOAP is built on a messaging concept of passing XML documents from a sender to a receiver, also known as the *endpoint*. Any link in the processing chain that is not the endpoint is referred to as an *intermediary* [Ngh02]. The SOAP specifications allow an intermediary to process a SOAP message partially before passing it to the next link in the processing chain (which can be another intermediary or the endpoint).

A SOAP document is composed of three sections: the envelope, the header, and the body. The envelope is the container for the other elements in the SOAP message and provides a mechanism to identify the contents of a message and to explain how the message should be processed. The header element makes it possible to extend SOAP messages while still conforming to the SOAP specification, for instance to include authentication information or to specify an account ID for a pay-per-use Web service. The SOAP body contains the actual content that is being sent.

SOAP also specifies a transport binding framework with support for HTTP, SMTP, JMS and others. To encode data types SOAP provides a serialisation framework. Since SOAP had to support multiple languages and operating systems, it had to define a universally accepted representation for different data types such as float, integer, and arrays. Data can be passed using these SOAP encoding rules, but it can also be passed using a literal XML document that validates against some XML Schema. These two options are commonly referred to as *encoded* and *literal*[2].

Services may be designed to work on the raw XML payload, but it is more common for the payload to be mapped or bound directly to data types in the host language. This mapping is called *deserialisation* and is possible regardless of whether the payload is a

[2]The WS-I organisation (WS-I) has defined a number of profiles that specify how one should develop Web services to be interoperable. *Encoded* is not considered WS-I compliant. However, all current toolkits support it.

literal XML document or SOAP encoded as both support common features found in the type systems of most programming languages and databases. This includes simple types such as strings, integers, and floats, and complex types, such as structures and arrays. More complex data types (such as a Hotel class) require custom coding. The reverse process of mapping objects again to XML, for instance to send a response to the client, is called *serialisation.* Toolkits are available to automate this process, a well-known example in Java is the Axis SOAP toolkit [Apa02]. Typically, a similar serialisation/deserialisation process takes place at the client side. This is further discussed in chapter 3.

SOAP supports three categories of communication schemes:

- **RPC Style Messaging:** The SOAP RPC representation defines a programming convention that represents RPC requests and responses. Using SOAP RPC, the developer formulates the SOAP request as a method call with zero or more parameters. When the payload is constructed into a single structure the outermost element represents the name of the method or operation, and the innermost elements represent the parameters to the operation. The SOAP response is similar with an outermost element named in relation to the method name and the innermost parameters representing zero or more return parameters.

- **Document Style Messaging:** SOAP also supports more loosely coupled communications between two applications. The SOAP sender sends a message and the SOAP receiver determines what to do with it. It is entirely up to the SOAP receiver to determine, based on the contents of the message, what the sender is requesting and how to process it.

- **SOAP with Attachments:** By attaching additional files to a SOAP message, non-XML data can be transported between the client and a service. This specification uses MIME or DIME to encode messages.

Given the fact that there are two options to encode a SOAP message (literal and encoded), and that there are two major communication options (RPC and document) there are four combinations of binding style and data encoding. Many misconceptions exist on the impact of these different options on the interaction model (see section 2.3) of the Web service, mainly because of their unfortunate terminology. However, both concepts are orthogonal: RPC versus document does *not* imply that RPC-style should be used for RPC interaction models and that document- style should be used for document or messaging interaction models. It *only* indicates the formatting and the representation of the SOAP message on the wire. Which one is used is just a configuration option with no effect on the behavioural characteristics of the Web service[3]. For instance, in the .NET framework a variant of

[3]This does not imply that it is irrelevant for the Web services developer which option he/she chooses. Document-Encoded is never used in practice, but the other three combinations each have their own strengths and weaknesses with respect to overhead in the resulting SOAP message, affecting performance of the Web service; the XML validation process of the SOAP message; the human-readability of the SOAP message; the complexity of the WSDL-file that describes the Web service; platform support; etc. An analysis is made in [Butek05]

Document-Literal, called Wrapped Document-Literal[4] is used, but .NET clients and .NET Web services do communicate by default via an RPC programming model.

The implementation of SOAP toolkits by different vendors results in some interoperability issues, mainly caused by the infancy of the specification.

2.4.4 The Description Layer

The Web Services Description Language (WSDL) [CCMW03] was created in response to the need for unambiguously describing the various characteristics of a Web service. WSDL provides the grammar to describe the details of the Web service's publicly exposed interface, such as how it should be invoked, and supplies necessary information for the client to use in the service invocation. Minimally, the client will need to know the signature of the service, the wire protocol to be used to send the invocation message, and the location of the service [Tyagi04]. In essence, WSDL defines a contract that a provider is committed to support and describes how clients need to format their service requests over different protocols or encodings. WSDL is used to describe *what* a Web service can do, *where* it resides, and *how* to invoke it. In order to separate implementation from interface, WSDL does not specify how a Web service is implemented.

The interaction between the Web service and the service client can be characterised as a series of message exchanges: the Web service accepts the incoming message, may return an outgoing message or may throw an exception message. Each type of message can be described further by listing the data-types and order of its parameters. When the service has multiple methods, it is possible that some methods exchange the same messages and that some data types are common across messages. All of these elements are specified in the *abstract description* of a WSDL document. Additionally, a WSDL document contains binding and address information for the Web service, in a second logical part of the document: the *concrete description.* This part specifies the exact network address of the Web service, also known as the Web service endpoint, and the specific protocols the Web service understands. So, the abstract description specifies the *what* part, and the concrete description specifies the *where* and *how* parts. This is depicted in Figure 2.3.

Toolkits are available that support the automatic generation and parsing of WSDL-documents, although, due to the immaturity of the tools, the generated files might still require manual tweaking, not in the least to fix incompatibilities between different vendors [Ngh02]. Therefore, Web services developers should still understand the structure of a WSDL document. The **definitions** element is the root element containing the remaining five elements; it defines the name of the service and declares the namespaces used throughout the document. The following elements make up the rest of the document. First we list the elements belonging to the abstract description of a Web service (the *what* part):

- The **types** element contains the platform- and language-independent data type defi-

[4]To eliminate weaknesses of the other combinations, Microsoft has introduced a new variation, called Wrapped Document-Literal. There exists no specification of it, but it is used in the .Net framework. At the moment, Document-Literal and Wrapped Document-Literal are considered the best options in most circumstances.

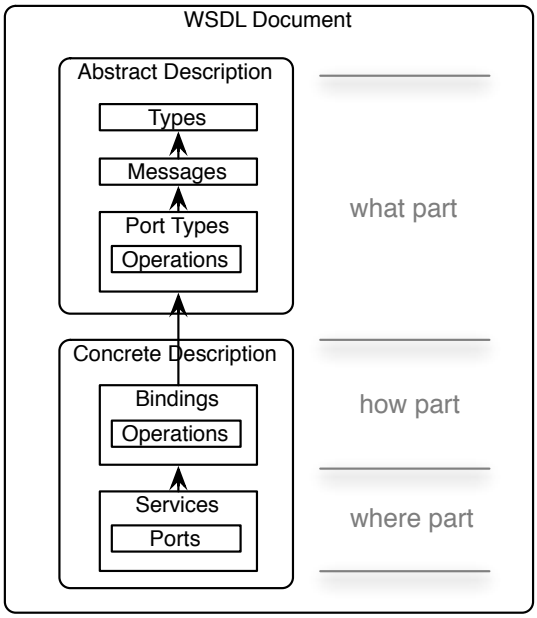

Figure 2.3: The different parts of WSDL document to describe a Web service

nitions used by the Web service

- The **message** elements contain input and output parameters for the service, and are used to describe different messages that the service exchanges. A message is sent between the requester and the provider (or vice versa). Each message element declares a name, value(s), and the type of each value; it does not specify whether the message is for input or output as this is the role of the next element.
- The **portType** element represents a collection of one or more operations, each of which has an **operation** element. Each operation element has a name value and specifies which message (from the message element) is the input and which is the output. An operation can be one-way, request-response, solicit-response, and notification. The latter two are simply the "inverse" of the first two, the only difference being whether the end point in question is on the receiving or sending end of the initial message.

The following elements make up the concrete description of a Web service:

- The **binding** element represents a particular portType implemented using a specific message and network protocol (the *how* part). In the WSDL-SOAP binding it is indicated whether the service uses RPC-style or document style messaging, and if they have a literal use or encoded use (see previous section 2.4.1.3). If a service supports more than one protocol the WSDL document includes a listing for each. In the remainder of this dissertation, we assume SOAP over HTTP as the binding of choice.

- The **service** element represents a collection of **port** elements, each of which represents the availability of a particular binding at a specified endpoint, usually specified as a URL where the service can be invoked. So, a port describes a network location for a binding (the *where* part).

2.4.5 The Publication Layer

This Publication Layer of the Web Services stack allows for the publication of Web services on a central market place where other parties can find and use them. Universal Description, Discovery, and Integration (UDDI) [BCC+04b] is a specification for repositories of Web services and their metadata. A UDDI repository contains entries about businesses, the services these businesses provide, and information on how those services can be accessed. Businesses use UDDI to register information about themselves and the services they offer in white, yellow, and green pages.

- **The white pages** contain general information about businesses, such as their names, text descriptions, contact information and other unique identifiers.
- **The yellow pages** classify businesses into taxonomies according to their industries, products/services, and locations. Examples include the Standard Industrial Code (SIC), the North American Industrial Classification System (NAICS) and the Universal Standards Products and Services Classifications (UNSPSC).
- **The green pages** specify the technical details of invoking specific Web services, including the address of the service, parameters, etc.

The entries in a UDDI directory are not limited to Web services; UDDI entries can be for services based on email, FTP, CORBA, RMI, or even the telephone. Both private and public UDDI repositories exist. Anyone can publish an entry in a public registry, which has no process to ensure the validity of its entries. Because an entry is not validated, there may be questions as to whether the business actually exists, whether the services are even provided, and whether the services are delivered at an acceptable level. Microsoft and IBM are UDDI operators of two public repositories, called business registries, which are the first public implementations of the UDDI specification[5]. A private UDDI on the other hand offers a company greater control over access to its system and its applications metadata, as it is easier to enforce certain criteria on an entry before it is published in the repository. Different variations exist [NGW04]:

- EAI registry. This is useful for large organisations that want to publish commonly used services by various departments or divisions.
- Portal UDDI. The registry is located behind a firewall. Therefore, the external users can search for entries, but only the operators of the portal can publish or update the entries in the portal.

[5]The Microsoft and IBM business registries are discontinued as of January 2006

- Marketplace UDDI. Only members of the marketplace (typically a closed environment) can publish and search for services. This type of registry is appropriate for vertical industries. The marketplace operator can establish qualifying criteria before an entry is added to the repository and can then provide additional fee-based services such as certification, billing, and non-repudiation.

The UDDI data model includes an XML schema that provides four major elements [NGW04]:

- The **businessEntity** element represents the owner of the services and includes the business name, description, address, contact information categories, and identifiers. Upon registration, each business receives a unique **businessKey** value that is used to correlate with the business's published service.
- The **businessService** element has information about a single Web service or a group of related ones, including the name, description, owner, and a list of optional **bindingTemplate** elements. Each service is uniquely identified by a **serviceKey** value.
- The **bindingTemplate** element represents a single service and contains all the required information about how and where to access the service. Each binding template is uniquely identified by a **bindingKey** value.
- The **tModel** element, short for technical model, is primarily used to point to the external specification of the service being provided. For a Web service, this element should ideally point to the WSDL document that provides all the information needed to unambiguously describe the service and how to invoke it. If two services have the same tModel key value, then the services can be considered semantically equivalent. This feature is discussed later on in Chapter 3, section 3.3.2.1.

Using a service repository such as a UDDI provides a level of indirection required for dynamic binding in clients. This is further discussed in the next chapter. Note that although WSDL and UDDI are part of the fundaments of Web Services, their use is not obligatory. A small Web service architecture consisting of a limited set of services, may not require a central repository to keep track of the available Web services. And a service that does not have a WSDL description can still be invoked using SOAP if its description is made available to the client by some other means. Therefore, Web Services are also referred to as XML Web Services, or SOAP Web Services.

2.5 The Web Services Protocol Stack Revisited

One of the advantages of Web services is that they rely on SOAP as a lightweight communication protocol. SOAP does not contain any advanced features. For this reason, SOAP is extensible through its header mechanism. Over the years, several new standards or proposals have been suggested in the Web services context. These standards are referred to as WS* standards, and providing an overview is almost an impossible task. Many of these standards

Business Domain Specific extensions	Various	Business Domain
Distributed Management	WSDM, WS-Manageability	Management
Provisioning	WS-Provisioning	
Security	WS-Security	Security
Security Policy	WS-SecurityPolicy	
Secure Conversation	WS-SecureConversation	
Trusted Message	WS-Trust	
Federated Identity	WS-Federation	
Portal and Presentation	WSRP	Portal and Presentation
Asynchronous Services	ASAP	Transactions and Business Process
Transaction	WS-Transactions, WS-Coordination, WS-CAF	
Orchestration	BPEL4WS, WS-CDL	
Events and Notification	WS-Eventing, WS-Notification	Messaging
Multiple message Sessions	WS-Enumeration, WS-Transfer	
Routing/Addressing	WS-Addressing, WS-MessageDelivery	
Reliable Messaging	WS-ReliableMessaging, WS-Reliability	
Message Packaging	SOAP, MTOM	
Publication and Discovery	UDDI, WSIL	Metadata
Policy	WS-Policy, WS-PolicyAssertions	
Base Service and Message Description	WSDL	
Metadata Retrieval	WS-MetadataExchange	

Figure 2.4: The Web Services Stack (revisited)

are still in their infancy compared to more mature distributed computing open standards such as CORBA. This wide range of standards is also called "acronym hell" because of the many new abbreviations they introduce. The following picture gives an overview of the current Web Services stack[6]. It is not intended as a complete categorisation, but only lists the most important standardisation efforts. Over the next chapters we will come back to some relevant standards when discussing service integration, selection and client-side management.

[6]**Data Source:** The Service-Oriented Architecture Practice Portal, February 2005

2.6 Web Services Development

2.6.1 Tool Support

Since the advent of Web Services, a lot of tools have been made available to develop and deploy Web services. Tool support is available in Integrated Development Environments (IDE) such as Eclipse and Visual Studio.NET to straightforwardly deploy applications as a Web service on an application server. Typically, this process is as easy as selecting the appropriate functionality that needs to be exposed by the Web service, for instance, in an object-oriented application, selecting all public methods of a class. The developer does not need to write any infrastructure code as the tool will automatically generate a WSDL-file and deploy the Web service on a server. These tools also work with existing applications: both applications written in popular languages such as Java and C/C++, and legacy systems can be easily Web service enabled.

During Web service development, a strong focus on the design phase of the Web service architecture is required. The design defines the services, data types, and message formats, and how they interact. The design can be created in modelling or other graphical tools, but Web services require that the design also be represented in WSDL documents, as these document serve as the interface towards other components and services in the architecture. This design practice is commonly referred to as *WSDL-first development* or *contract-first development*. A mayor advantage of this approach is that it should result in more stable WSDL documents.

Nowadays, Web services are primarily developed on two platforms: Java and .NET. We will briefly discuss the characteristics of both platforms.

2.6.2 Java Web Services

Web Service support is provided in the Java 2 Platform, Enterprise Edition platform (J2EE) through a set of APIs and tools that allow developers to design, develop, test and deploy both Web services and their clients. The three most important API's are:

- **Java API for XML-based RPC (JAX-RPC)** provides a platform for building Web services applications by hiding from the application developer the complexity of mapping between XML types and Java types and the lower-level details of handling XML and SOAP messages. JAX-RPC introduces a method call paradigm by providing two programming models: a server-side model for developing Web service endpoints using Java classes or stateless EJBs, and a client-side model for building Java clients that access Web services as local objects. In spite of its name, it supports both RPC-based and document-based interaction style Web services. It has become the most popular way to deal with Web services in Java [LS05]. The next release, version 2.0, will be renamed JAX-WS.

- **Java API for XML Processing (JAXP)** supports the processing of XML documents using DOM, SAP and XSLT. The JAXP API enables applications to parse and

transform XML documents independently of a particular XML processing implementation.

- **SOAP with Attachments API for Java (SAAJ)** enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments note.

Both commercial and open-source tools with Web services support are available. Web Services can be deployed on a wide range of J2EE compatible application servers.

2.6.3 Microsoft .NET

The .NET Framework is a common environment for building, deploying, and running Web services and web applications. It is developed and supported only by Microsoft but interoperability is guaranteed through the Web services standards. Two major tools exist:

- **Visual Studio.NET** is a multi-language development tool. The supported languages are: Visual Basic, Visual C++, and C#. Third parties support other languages.

- **The .NET Framework** is a standards-based, multi-language application execution environment that handles essential plumbing chores and eases deployment. The .NET Framework improves the reliability, scalability, and security of an application. It includes several parts: the Common Language Runtime (CLR), a rich set of class libraries for building XML Web Services, and ASP.NET for building dynamic web pages.

The initial advantage of .NET over Java was that Visual Studio.NET was the first to hide away all complexities of making Web service calls. By simply tagging any function call with a special attribute (called `WebService`), all infrastructure code to make the actual call is automatically generated. Nowadays, Java tools are also available that allow for easy deployment of Web services, so this advantage does not come into play anymore. Both platforms have their advantages and disadvantages, but it is outside the scope of this dissertation to go into detail on this. Important to remember is that there are different Web services implementations supporting the SOAP, WSDL, UDDI standards, and therefore both claim interoperability. In reality however, interoperability issues do arise, so developers need to take special precautions when working in mixed environments. In the remainder of this thesis, we will refer respectively to Java Web Services and .NET Web Services when the distinction needs to be made.

In the next chapter, we focus on the development of Web services clients, more specifically in the context of dynamic environments. We finish this chapter with an overview of some related middleware technologies.

2.7 Related Middleware Technologies

2.7.1 CORBA

Common Object Request Broker Architecture (CORBA) is an open standards-based approach for distributed computing. The Object Management Group (OMG) developed the specification for CORBA and specified the Internet InterORB (IIOP), the standard communication protocol between Object Request Brokers (ORB). With CORBA, client and servers can be written in any programming language. This is possible because objects are defined with a high level of abstraction provided by the Interface Definition Language (IDL). A compiler is used to do the mapping between an IDL file and a specific programming language. In order to enable IDL to be translated into various languages, it is limited to concepts found in all supported languages, thus representing a least common denominator. The communication between objects, clients and servers are handled to ORBs. In order to enable communication, compatible ORBs are needed on both sides of the connection. More complex features are offered through a variety of services. The latest CORBA specification can be found at [VIN97].

2.7.2 DCE

The Distributed Computing Environment (DCE) is a suite of technology services developed by The Open Group for creating distributed applications that run on different platforms. The framework includes a remote procedure call (RPC) mechanism known as DCE/RPC, and a variety of more advanced services dealing with security, authentication, naming directories, threading, distributed files, scalability and fault tolerance. The Open Group DCE RPC specification is available at [RKF92].

2.7.3 DCOM

The Microsoft Distributed Component Model (DCOM) allows calls to remote objects by using a layer that sits on top of the DCE RPC mechanism and which interacts with the COM runtime services of the Windows operating system. A DCOM server publishes its methods to the clients by supporting multiple interfaces. These are written in DCE IDL. A compiler, similar to the CORBA compiler, creates stubs and skeletons and registers them in a system registry. The protocol used is Object Remote Procedure Call (ORPC). Because of the binary level specification of DCOM, various languages can be used to code the server objects. DCOM supports a distributed garbage collection. Although DCOM is primarily associated with Windows, ports are available for other operating systems as well.

2.7.4 Java RMI

With Remote Method Invocation (RMI), distributed Java applications can be created, in which the methods of remote Java object can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialisation to marshal and unmarshal param-

eters and supports true object-oriented polymorphism and distributed garbage collection. The protocol used for the communication is the Java Remote Method Protocol (JRMP). No abstract description language is used to describe the remote objects. Clearly, Java is needed on both ends of the connection. Java RMI is part of the Java language specification and the latest 1.5 version can be found at [WAL98].

2.7.5 ebXML

Electronic business XML (ebXML) is a set of specifications that enables business to collaborate using XML-based technologies. With ebXML, businesses can find each other online and conduct business based on well-defined XML messages within the context of standard business processes, which are governed by standard or mutually negotiated partner agreement. ebXML can be considered a top-down approach for business integration, whereas Web services is a bottom-up approach [San03]. Though ebXML is an approved, robust standard, its applicability is far narrower than Web Services. As an evolution of EDI, it primarily addresses the B2B domain only. More details can be found in [HHK02].

2.7.6 Comparison

These middlewares can be considered similar as they follow a universal client/server architecture to make distributed systems. The differences are found in the different features that are supported as well as the level of complexity. They result in a tight coupling and the same middleware technology is needed on both sides of the wire. In addition, these middlewares are typically used for intranet applications, as it not straightforward to cross a firewall. More complex procedures like HTTP tunnelling could make this possible though. Note that the usage of one middleware technology in a system does not necessarily exclude another one. For instance, an existing CORBA-based application can be easily exposed as a Web service. Table 2.1 shows the parallels between the discussed middlewares and Web services. For an in-depth discussion, we refer the interested reader to [RAJ98].

2.8 Conclusions

Web services technology provides interoperability between software applications running on disparate platforms. Web services use open XML-based standards and protocols. By utilising HTTP, Web services can work through many common firewall security measures without requiring changes to the firewall filtering rules. The real value of Web services is not its innovative character but rather its industry acceptance as a common way to realise hardware and platform independent application integration. In recent years, many tools, both commercial and open-source have been made available to implement service-oriented applications. These tools hide away the complexity of writing and using Web services from application developers, which further drives adoption of the technology.

Web services standards for more advanced features such as transactions are currently non-existent or still in their infancy compared to more mature distributed computing open

Table 2.1: Comparison of Different Middleware Technologies

	CORBA	**RMI**	**DCE**	**DCOM**	**ebXML**	**Web Services**
Publication	COS Naming	Java Registry	CDS	System Registry	ebXML registry	UDDI
Description	CORBA IDL	Java Interface	DCE IDL	DCE IDL	CPP	WSDL
Messaging	GIOP	Stream	PDU	/	ebXML Messaging, SOAP	SOAP
Format	CDR	Serialized Java	NDR	/	XML	XML
Transport	IIOP	JRMP	RPC CO	ORPC	HTTP, SMTP, ...	HTTP, SMTP, ...

standards such as CORBA. It is expected, however, that over time more standards for these features will emerge. A disadvantage of Web services is the poor performance, compared to other distributed approaches such as RMI, CORBA and DCOM. But also here, research and new approaches are being proposed to improve the wire efficiency of XML messaging.

Chapter 3

Dynamic Web Service Environments

Abstract In this chapter a running example that will be used throughout the remainder dissertation is introduced. Next, requirements for the service integration process, the service selection process and the client-side management process of Web services are defined, with respect to changes and evolutions that may occur in the Web services environment, the individual web services, the network and the client environment. An evaluation is made of existing Web service technologies and approaches against the defined requirements. This evaluation targets tools and technologies available for the client-side development of a SOA using object-oriented programming languages and that are based on the Web services standards introduced in the previous chapter.

3.1 Running Example

Nowadays, if a person wants to book a holiday via the Internet, he or she does not have a single point of entry to start searching for information and to make reservations. A wide variety of web sites is available on the web, each providing travel information and allowing for online bookings. However, a person needs to browse all of these web sites, get acquainted with the different user interfaces and layouts, and repetitively enter search criteria such as the destiny, dates, number of persons, etc.) in order to compare different offerings. This reduces to a large extent the usability of the Internet for this task. Portal sites and web sites comparing pricing information of several shopping sites can reduce this negative impact somewhat. However, only sites included in these portal sites can be compared, and typically, these sites are only available in large countries with a high Internet penetration.

If one wants to create a system linking the different systems of multiple holiday operators in order to deliver a unified end-user experience, one has to integrate the different back-end systems, a process that will lead to a variety of integration and compatibility issues. Web service technology is an ideal middleware technology for this scenario. The rest of this dissertation will use the travel agent system as a running example. We assume holiday operators make three different kinds of Web services available:

- **Hotel Services:** a Web service that returns a list of hotels in a specific destination. A destination can be a country or a city. The service can also return a list of hotels with available rooms for a given period of time or a given number of nights. A hotel reservation can be made for a given hotel and a given period. Additional search and reservation constraints can be specified including the rating, price range, accessibility and extra services.

- **Flight Services:** a Web service that returns a list of possible flights from a given departure city to a given destination at a given date. The service can look for available seating, and make and cancel reservations. Additional search and reservation constraints can be specified including class, food type, seating preference, etc.

- **Car Services:** a Web service that returns a list of cars that can be rent at a given destination for a given period of time. The service can look for available cars and make and cancel reservations. Again, additional constraints can be specified including the type of vehicle, insurance options, etc.

The travel agent needs to integrate with multiple of these Web services in order to offer a wide variety of holidays via its web site to its customers. Depending on continuously evolving business requirements and changing network and service conditions, different services will be integrated at a given time. However, automating this process is far from straightforward: the Web services belong to different domain controllers and as a result might differ on several points including syntactical and semantical differences in the service interface, security measurements, Quality-of-Service, billing mechanisms, etc. All of these variations need to be reflected in the travel application, which clearly is a hindrance for a smooth integration process.

3.2 Introduction to Dynamic Service Environments

Web services do not have a user-interface but a programmatic interface. They are typically integrated in one or more client applications, such as the travel agent application described above. Based on the standards discussed in chapter 2, development tools have been made available to create clients that communicate with Web services. Typically, proxies are created at the client side to represent the remote Web service, and it is up to the client to manage these proxies. This is a practical and suitable solution for a SOA consisting of a small number of services, for example in intranet environments where the services remain under control of the same body that controls the client. If any changes need to be made to the system, they will need to be applied in a controlled manner to deal with changes in the service that might break the client. In large-scale systems, the number of needed proxies will grow, and if the Web services belong to different controllers, they will require more management.

A SOA consisting of these so-called *third party Web services* poses additional requirements on the underlying middleware technology, and this is the subject of this chapter. A suitable middleware technology requires a high level of flexibility to deal with a wide variety of runtime changes. These runtime changes can occur everywhere in the environment. In the following subsections we identify a set of requirements based on events that can take place in the environment. We distinguish the following four environments:

- **Web Services Environment:** the Web service environment is defined as all Web services available in the SOA, including both the services of the third-party service environment and the management services environment. Each Web service is possibly hosted by a different provider, and can therefore evolve independently. Web services can be added all the time to the environment, and existing ones may be removed.
- **Individual Web Service:** a Web service may be subjected to a number of changes, ranging from physical relocation of the service, to changes in the Web service interface or its behaviour. Furthermore, a service may be temporarily unavailable due to maintenance or failures. A Web service is documented by a functional and, optionally, a non-functional description.
- **Network Environment**: a remote Web service is reachable over a network. This can be the unpredictable Internet or a controlled, shielded intranet where certain assumptions regarding reliability or speed can be made. Network failures, congestion, loss of packages, routing issues, etc. can result in communication failures with a remote service.
- **Client Environment**: the environment in which the client is deployed may vary too. Clients can run on different platforms and devices, affecting the way communication with the service environment will take place. Also the state of the client, possibly depending on the end-user status, may affect service communication.

In this chapter, the applicability of the current Web services technologies and practices is analysed with respect to runtime changes and evolutions that may occur in any of the

four environments listed above. This evaluation targets tools and technologies available for the client-side development of a SOA using object-oriented programming languages such as Java and C#. The evaluation is not exhaustive, but focuses on approaches 1) made available by industry and/or standardisation bodies, 2) that are common practice today and 3) that are based on SOAP, WSDL and UDDI. In the following section, we define the requirements for the *service integration process* and discuss current approaches, based on proxies and dynamic interfaces. Requirements for the *service selection process* and the lack of current approaches is the subject of section 3.4. A requirements analysis for *service related management concerns* and a discussion of current approaches, including message handlers, is presented in section 3.5. Finally, we conclude in section 3.6.

3.3 Service Integration Process

3.3.1 Analysis of Requirements

We define the service integration process as the process of finding a Web service that is able to provide the functionality needed by the client and integrating that service in the client for the purpose of invoking it. The collection of all compatible services that are integrated in the client is called the *service pool.* We will now lists the client requirements needed to deal with specific events in the service environment, the individual services and the network that have an impact on this service discovery and integration process.

Category: Services Environment - **Event:** Multiple Services Available

Dynamic Binding: The client must be able to address all compatible services available in the service environment. Full support for dynamic integration of Web services requires dynamic binding of a concrete Web service implementation with the client. By fully decoupling the client and the services, client requests can be redirected to multiple Web services and unanticipated Web services can be integrated on the fly.

Hot-swapping: at any time it must be possible to switch between available services in the service pool. We make the distinction between **client-initiated hot-swapping** and **transparent hot-swapping**, where the client is unaware of the process of service switching.

Category: Services Environment - **Event:** Partial Service Matches

Reusable Service Composition: Composing Web services is the process of orchestrating multiple services in order to let them work together in some way to perform the needed functionality. This is required when no service can deal with a client request on its own. A composition can be the combination of one service doing some pre-processing or conversion of data before passing it on to the main service, but it can also be a complete workflow where multiple services representing different business identities collaborate in a business process. Composition should be adaptable to deal with short-term changes and should be able to evolve to embrace long-term changes.

Multiple Services Binding: binding multiple services at the same moment to a client is an alternative solution when the services present in the service environment only offer partial matches. In this case, the Web services are complementary to each other and any request coming from the client is redirected only to one of the services at a given moment. This contrasts to service composition, where the services have to collaborate to offer the needed functionality.

Conditional Service Binding: A conditional binding between a Web service and the client implies the binding only takes place if the Web service is able to handle the current client request. The characteristics of the request are tested against some condition evaluating whether the service can handle the request and thus allowing or disallowing a binding with the client.

Category: Services Environment - **Event:** Service added or removed

Service Discovery: A service look-up mechanism makes it possible to find new services available in the service environment and remove old ones that are no longer available. If no look-up system is needed or available, a registration mechanism can be used to set up a pool of services that are functionally compatible with the client.

Service Matching: a mechanism is needed to be able to categorise the available services based on their functional compatibility with the client requests.

Category: Individual Web service - **Event:** Functional Mismatches

Interface Mapping: If there are mismatches between the requested functionality by the client and the functionality offered by the service, a mapping is needed to mediate between the two parties and resolve any incompatibilities. Mismatches can occur both on the syntactic and semantic level. Other features dealing with functional mismatches are service composition and multiple service binding.

Category: Individual Web service - **Event:** Service Versioning

Notification or Detection: Whenever a service provider changes the interface, behaviour or non-functional documentation of its service, the client must be able to detect these changes and resolve any incompatibilities, for instance by redoing the mapping process. Support for notification and detection is further discussed together with service selection (chapter 6).

Category: Individual Web service - **Event:** Service de-localisation

Dynamic endpoint references: The most rudimentary form of runtime flexibility is offering support for delocalisation of Web services. If a remote Web service moves to another location, this change needs to be reflected in the client by updating the corresponding service endpoint reference.

Category: Individual Web service - **Event:** Latencies; Long running processes

Asynchronous Communication: As performance is a weak point in Web service communication, and as services may implement long-running business processes, it might be better to invoke services asynchronously, thus not blocking the client while waiting for a service response.

Category: Individual Web service - **Event:** Message Protocol

Conversational Messaging: In more complex service-client interactions, the service might keep state of the conversation with its clients. In that case, the client must follow the intended communication protocol as specified by the service. Otherwise the service invocation will result in runtime exceptions.

Category: Individual Web service - **Event:** Failures; Anomalies; Maintenance

Exception Handling: Because of the complex nature of distributed computing, a large variety of errors and unexpected situations can occur. For instance, there can be problems with the connection, serialisation, deserialisation, low-level socket problems, issues with the parsing of the WSDL file or the service can return declared exceptions or exceptions of unexpected situations. Also more advanced exceptions related to security issues such as expired sessions or content may occur and the client must deal with all of these exceptions.

Hot-swapping: In case of service failures or maintenance, another compatible service can be addressed.

Category: Network Environment - **Event:** Failures; Anomalies; Maintenance

Hot-swapping: In case of network failures another compatible service, preferably on another network, must be addressed. In case of general network failure, adequate exception handling is needed.

3.3.2 Evaluation of Current Practices

3.3.2.1 Proxies

Currently, WSDL (see section 2.4.1.4) is the standard description language for Web services. A typical usage of a WSDL description is to generate a client-side proxy out of it. A proxy class defines methods that represent the actual methods exposed by the remote Web service. This principle is based on the *Proxy design pattern* [GHJ95]. The proxy acts as a surrogate or placeholder for the remote Web service and the local components of the client can communicate with this local proxy. Although a proxy acts as an implementation of

the Web service class, it is not a pure class instance. When a proxy method is invoked, the proxy will transparently initiate the communication with the service by serialising this method invocation into a SOAP message and sending it to the actual Web service. Upon receiving response, it will deserialise the response back into the client application runtime environment and return it to the method invoker. As this process happens in a transparent manner for the component that invokes the proxy, it hides the complexity by treating the Web services as regular internal software components. Programmers do not need to take into account that they are actually dealing with remote procedure calls (RPCs). Typically, the generated proxy supports the ability to invoke the Web service both in a synchronous or an asynchronous manner. For instance, the proxy of a hotel Web service providing a method `bookRoom`, would contain three methods:

- `bookRoom`: starts the synchronous method invocation of the Web service `bookRoom` method.
- `beginBookroom`: starts the asynchronous method invocation of the Web service `bookRoom` method.
- `endBookRoom`: used to obtain the result of an asynchronous method invocation of the Web service `bookRoom` method.

Tool support[1] is available for developers to automate the process of creating proxies from an available WSDL document. Besides creating a proxy, the tool will also make a mapping for all elements in the WSDL document into the destination language. For instance, in Java a mapping is made between XML schema types and Java classes and a mapping is made between WSDL operations and Java methods. This mapping is used at runtime to determine which XML Schema type or Java class should be used when serialising between Java and XML. The mapping between Java methods and WSDL operations is used when there are overloaded methods.

This RPC programming model of invoking Web services, fits perfectly in the object-oriented programming paradigm and this immediately explains the popularity of the approach: languages such as Java and C# all provide mechanisms to serialise objects into SOAP messages and back. For instance, in Java, JAX-RPC (see section 2.6.2) creates a Java object for each XML element, building a directed, acyclic graph when serialising to RPC/encoded SOAP messages, or a tree when using document/literal SOAP messages. As denoted in [LS05], JAX-RPC essentially makes SOAP messaging look like Java RMI. Primitive types and collections are natively supported. For user-defined types (such as a Hotel class or a Flight class), typically only the member fields that are publicly accessible (e.g. using the JavaBean convention of getters and setters) are serialised and deserialised. An important side effect of user-defined types is that the client needs to understand the custom types to be able to make use of the Web service, which makes it less accessible in the first place. What's more, analysis of user-defined data types could reveal inaccuracies

[1] WSDL2Java is a command-based tool used to translate WSDL to Java. The counterpart in .NET is WSDL.exe that translates a WSDL file to any of the available .NET languages, including C#, Visual basic.NET and JScript.NET

and integration issues: one implementation's serialisation might not match another's deserilisation [GGT03]. And there are also performance implications: for every new data type the proxy class has more work to do when it serialises and deserialises the SOAP message and therefore could contribute to slower response times.

As described in section 2.3, Web service communication does not necessarily have to be RPC-based. It could also be document-based: instead of sending a SOAP message that represents a method invocation, a whole document is sent to the service for processing. The proxy approach is also suited for this kind of communication, although there are some tradeoffs when dealing with large and complex XML documents [Ana05]. A proxy for a document-based hotel Web service can have a method `process (ReservationDocument)` and a Java wrapper for the `ReservationDocument` can be generated at client-side. [TYAIG04] contains an overview of different strategies to realise document-style Web services.

The next two subsections introduce two proxy approaches: **static proxies** and **dynamic proxies**. A short description of their typical usage is given, together with an evaluation with regard to the requirements of dynamic web environments of Table 3.2.

A) Static Proxies

A *static proxy*, also known as a stub, is a proxy created at development time, representing a single Web service. In Visual Studio.NET 2003 [Ms03], the development suite of the .NET framework, the developer has to provide a web reference of the Web service to the tool. Then, the WSDL document of the Web service is automatically looked up, analysed and code for a proxy class is generated. Typically, various parameters can be set to indicate which parts of the WSDL file need to be generated into the destination language. The proxy class defines methods that represent the actual methods exposed by the remote Web service. The instantiation of the proxy is very straightforward as it boils down to a simple instantiation of a regular class.

Using static proxies does not offer any runtime flexibility, as there is no decoupling between the service interface and the service implementation: the proxy generated at client side reflects the interface specified at service side. If the developer needs to find a service, he can use a UDDI registry and do keyword-based searches. This approach is most appropriate for small applications integrated with one or a small number of fixed services running on the same machine or in a controlled environment such as an intranet. The main advantage of static proxies is that the client does not need to parse the WSDL file when the service functionality is requested at runtime, which boosts performance. On the downside, if the remote WSDL file changes, the proxy has to be re-generated and the type mapping has to be redone, requiring changes in the client code. Also, if the client uses n Web services, all the n proxy stubs need to be maintained at the client side. If one of the Web services changes its specification or behaviour, it will break the client application.

The only level of flexibility offered in this approach is the location of the Web service endpoint: by using a variable or a configuration file it is possible to defer the specification of the actual location of the service endpoint to deployment time. Obviously, the Web service at the specified endpoint must have exactly the same WSDL description that was used to generate the proxy. Syntactic or semantic differences between the Web service interface and the actual implementation will result in runtime exceptions or incorrect behaviour of the

client. Binding of unanticipated services at runtime and transparent hot-swapping is not possible. There is also no explicit support for conversational messaging: the client needs to make sure it follows a specified conversational protocol by invoking the proxy in the right order. Failure in doing so will result in runtime errors.

A primitive form of service composition is supported as WSDL allows for the definition of different port types, each with a different binding. Suppose *HotelService1* takes in a zip code to return a list of hotel descriptions and *HotelService2* takes in a hotel description to return an indication on room availability. If a programmer wants to combine the functionality of these two services in a simple composition, he has to integrate and manage two proxies in its client. Furthermore, a `HotelDescription` obtained from *HotelService1* cannot be passed on as a parameter to *HotelService2*, even if they are identical types. The reason is that both Web services are distinct identities described by their own WSDL files and therefore each proxy will have its own representation of the `HotelDescription` in a separate namespace, even if they are serialised into identical XML structures[2]. Solutions include copying values between instances of the different `HotelDescription` classes or manually changing the proxy code generated by the tool to make sure both services share the XML structure of the data type and the namespace.

A solution for .NET described in [Lind05] suggests defining a single interface WSDL file at client-side with a binding for each of the concrete Hotel Services and generating a proxy out of that WSDL file. However, this still does not achieve any encapsulation in the client application. The client is still aware of the two separate bindings, as there are two different proxy classes. The client is also well aware that the two services live in separate locations, since the client still needs to configure and maintain explicitly the URLs of the endpoints used to access the services. At the same time, there is nothing to indicate to the client application that the two services were implemented separately, at different times, by different teams. The only benefit is that types can be shared between the two Web services in the client code, which eliminates configuration efforts or data structure copying efforts. Note that this approach only works if both Web services use the same shared types, an assumption that does not hold in our heterogeneous service environment. Finally, this approach leaves the specification of the composition, i.e. how the services should work together, entirely to the client.

B) Dynamic Proxies

A more advanced approach to integrate Web services is *dynamic proxies* [Sun05]. Instead of creating a proxy class from a WSDL-document, only interfaces and mappings are generated. For example, the Wsdl2java tool in Java can translate a WSDL document automatically into Java Interfaces. The client is programmed against these Java interfaces. This is depicted in the upper part of Figure 3.1. Next, at runtime, a look-up mechanism will provide an actual instance of the Web service interface. A dynamic proxy class is used to create a type safe proxy object for an interface without requiring generation of a proxy at compile time. This principle is based on the *Factory Pattern* [GHJ95]. In the middle part

[2] A new version of the Microsoft® .NET Framework 2.0 will offer more support for sharing types between Web services. When generating proxies from multiple WSDL files of different Web services it will be possible to signal the tool that any common types should be shared between the generated proxies. Common types are elements with the same local name and namespace, defined in the same schema file.

of the figure, the role of factory is played by a registry that retrieves the WSDL file of the service, generates a proxy from this file and return it to the client. An implementation of such a registry, is for instance provided in Systinet Server for Java [Sys05], so programmers do not have to implement this mechanism from scratch. Once, the factory has returned a valid proxy, the client can invoke methods on it, which will result in the invocation of the remote Web service. This is shown in the lower part of Figure 3.1.

This approach, which requires more complex code than static proxies, offers better support to decouple the service interface from the service implementation. The service interface is wired in the client at compile time but the binding with the actual service is deferred to runtime by means of a service factory that looks up the WSDL file, parses it and generates a proxy. This approach is less performing than static proxies, but offers more flexibility. At runtime, a new Web service, matching the integrated service interface can be integrated by requesting a new proxy to the service factory. As such, the process realises dynamic bindings with support for dynamic endpoint references.

The approach does not offer support for glue code, hot-swapping, conversational messaging or multiple service bindings. Making a binding to a syntactically or semantically incompatible service will result in runtime exceptions or unexpected behaviour. It is assumed that a WSDL document is stable, as prescribed by the WSDL-first development approach of Web services. This concept is further strengthened by the concept of *tModels* in UDDI (see section 2.4.1.5). The idea is to fix industry specific service interfaces and post them in a UDDI registry, using tModels. Web services can refer in their WSDL-documentation to a tModel. As such it becomes possible to dynamically look-up Web services complying with a specific tModel and integrate them on the fly. This approach assumes some kind of standardisation process for service interfaces, something that has not happened yet, and very well may never happen.

Note that changes in a service interface (and thus in the WSDL file) will still break the client application. In this respect, it is important to point out that document-style interaction is preferred over RPC-style interaction, as document-style interaction offer better abstraction and better withstands changes to the underlying implementations [Burn03]. There are no methods, parameters and return types that can differ from what is expected in the client. However, the format of the exchanged XML-documents might still differ and evolve as the Web services change, and again, the client needs to adapt to these changes. It is a development choice of the Web service developer to adopt one or the other interaction model, and as indicated earlier, document-style services are harder to implement, as they require more design planning. In the end, the client might need to adapt to either of the two interaction models, and possibly, in a mixed environment there might be RPC-based and document-based Web services both offering the same functionality, a situation that cannot be handled by a single proxy.

As a matter a fact, it can be argued that using proxies for RPC-style Web services communication forces a too restricted view on Web services as being some kind of distributed objects. As discussed in [Vogels03], there are fundamental differences between them: Web services do not have the notion of objects, object references, factories, life cycles and they are typically stateless. The problem is that looking at Web services as plain XML document processors does not offer any help for developers in actually building Web services

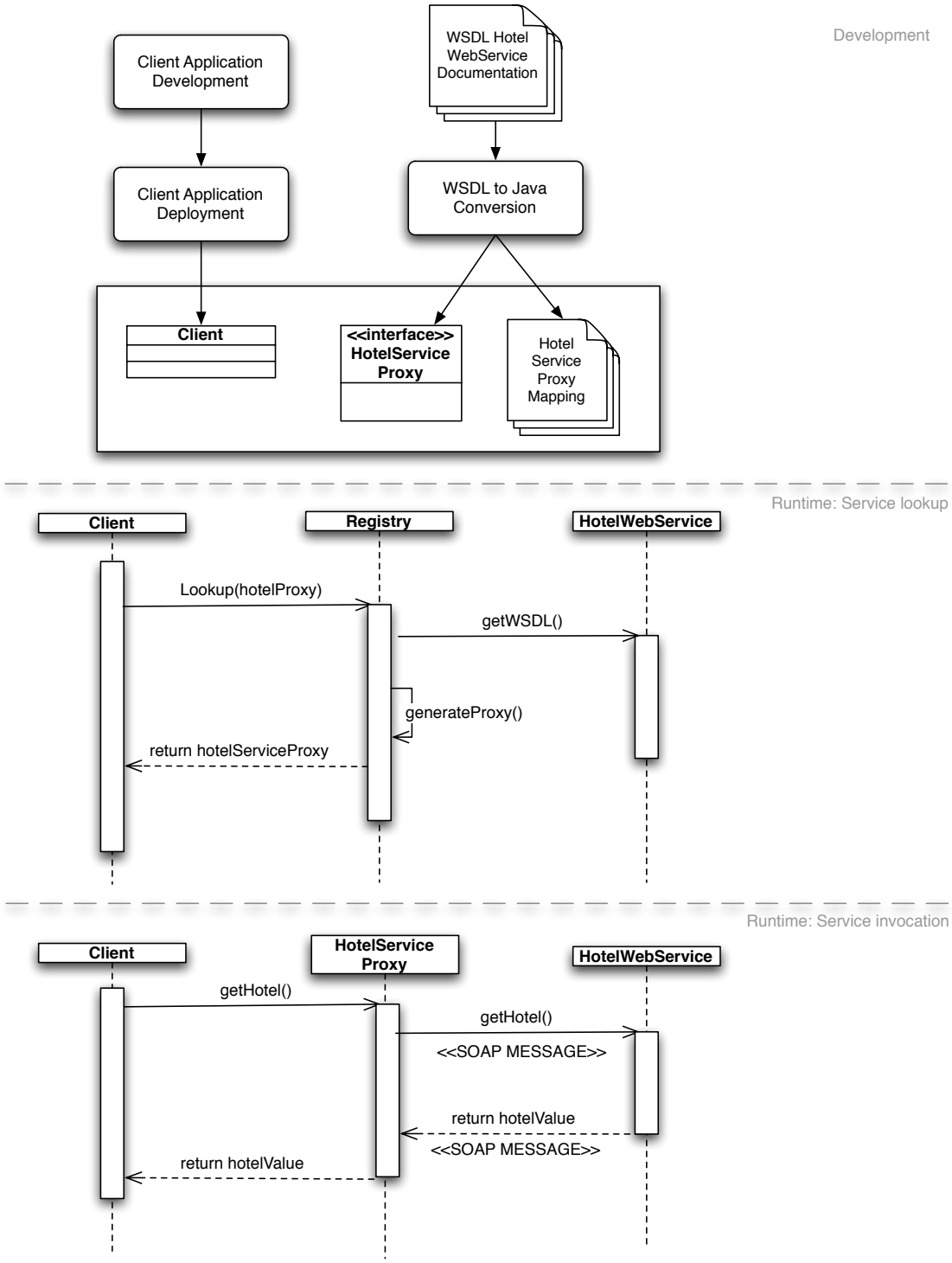

Figure 3.1: Invoking a Web service Using a Dynamic Proxy

and clients. By providing a proxy-based infrastructure, tool support can be provided and traditional procedure calls can be applied to Web services. After all, it is easier for a developer to work in his/her native object-oriented programming language and not having to deal with raw XML documents.

An example of how far one can go in forcing the distributed objects view on Web services is presented in [RS05]. The approach discussed in this paper makes the public methods of user-defined types returned by Web services available in the client. For example, if a client retrieves an instance of the class Hotel from a Web service, then the client could directly call the `bookRoom` method on this object. While this approach builds further the object-oriented paradigm, it is clear that it completely breaks the concept of loose-coupling and language independence of the Web service paradigm.

In [LS05] the opposite argumentation is made: the authors make the case that making a mapping between XML Schema and an OO language such as Java in the proxy approach is inherently flawed because it is too complex and brittle. This is caused by the fundamental differences between the type system of XML Schema and Java. Problems include the fact that some XML elements cannot be straightforwardly mapped to a Java class and vice-versa; XML names cannot always be mapped to Java Identifiers; serialisation of enumerations, graphs of objects and Java exceptions can be an issue; etc. To overcome these problems, the authors propose a lightweight SOAP stack for Java, called Alpine, that takes an XML centric approach. No mapping is provided but instead access to the SOAP messages is provided using XML support libraries.

3.3.2.2 Dynamic Invocation Interface (DII)

An alternative to proxies for service invocations *is Dynamic Invocation Interfaces (DII)* [Sun05]. With DII, a client application can call a remote procedure even if the signature of the remote procedure or the name of the service is unknown until runtime. In contrast to a static or dynamic proxy client, a DII client does not require runtime classes generated by a tool. However, the source code for a DII client is more complicated than the code for the other two types of clients. During runtime, the client needs to supply the name of the operation or the names and values of arguments. Next, the WSDL document from a specified URL is downloaded, the correct operation is selected, the arguments are mapped and the target Web service is called. With DII clients, no runtime classes generated by mapping tools are required like it is the case with proxies. In JAX-RPC, this is done in three steps (see also Figure 3.2:

1. **Creating a Call Object:** The Call object represents all information needed to invoke the operation on the Web service.
2. **Filling the Call with actual values:** all information about parameters and values of the service operation in addition to operation name need to be provided, including the operation to invoke, the port type for the service, the address of the endpoint, the name, types, and modes (in, out, in-out) of the arguments and the return type.
3. **Invoke operation:** after all this information is provided, the Call object can now invoke the target operation.

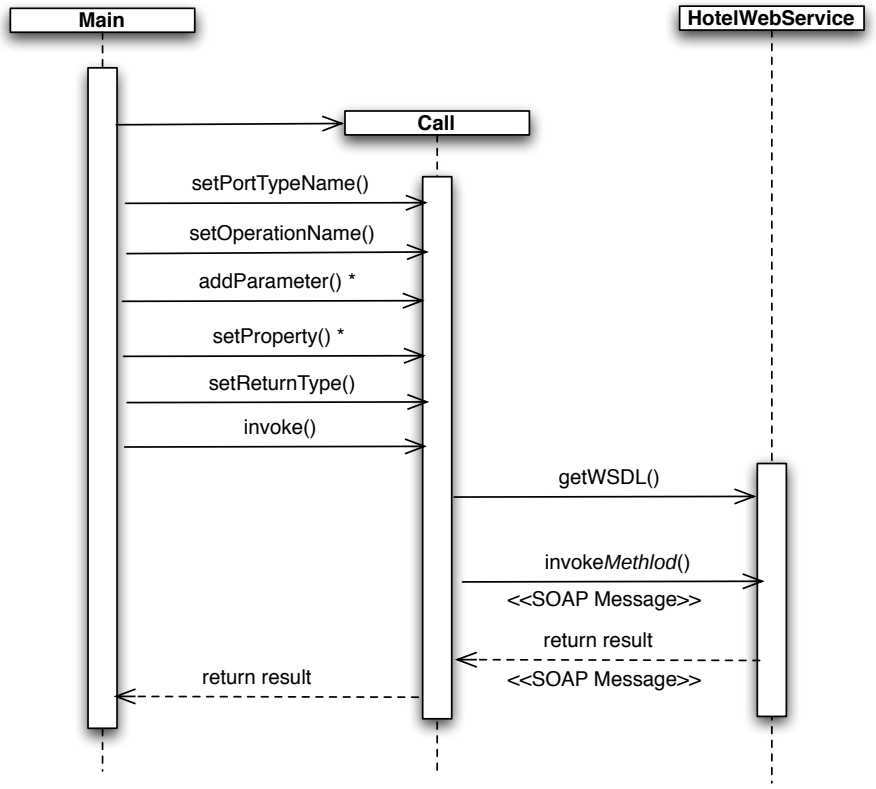

Figure 3.2: Invoking a Web service using DII

Using DII realises a very flexible form of dynamic binding. The client can specify at runtime service related items including the endpoint reference and method signature. The advantage of using a DII Call interface is that a client can call a remote procedure without development-time knowledge of the WSDL URL or the Web service operations' signatures. This makes the code easy to adapt if the Web service details change [Port03]. Using a service factory, a service instance and call instance are created. It is entirely up to the client to determine which method to invoke with the appropriate parameters, and there are no guidelines on how to resolve this: the client should retrieve this information by other (unspecified) means. Clearly, this high flexibility results in more complex code. Also, this approach does not offer explicit support for hot-swapping, service composition, conversational messaging or multiple service bindings. Code needs to be explicitly provided in the client implementation to anticipate these concerns. Only limited support for syntactical mismatches is available, as the method signature does not need to be fixed. Making a binding to an incompatible service will result in runtime exceptions or unexpected behaviour. All exception handling must be provided manually.

3.3.3 Table of Comparison

Table 3.1 compares the different Web service integration approaches. A (+) indicates full support for the requirements identified in Section 3.3.1, a (+/-) indicates partial support, possibly through additional mechanisms, and a (-) indicates there is no support at all as the developer needs to deal with the requirement by explicitly providing code.

Table 3.1: Comparison of Different Service Integration Approaches

	Static Proxies	**Dynamic Proxies**	**DII**
Discovery	-	+/-	+/-
Matching	-	-	-
Mapping	-	-	+/-
Changeable endpoint ref.	+	+	+
Dynamic binding	-	+	+
Multiple Services Binding	-	-	-
Conditional Service Binding	-	-	-
Asynchronous communication	+	+	+
Reusable Service Composition	-	-	-
Notification / Detection	-	-	-
Transparent Hot-swapping	-	-	-
Conversational Messaging	-	-	-
Exception Handling	-/+	-/+	-/+

3.4 Service Selection Process

3.4.1 Analysis of requirements

We define the Service Selection Process as the process of selecting the most appropriate Web service amongst all integrated Web services in order to deal with a specific client request. We now list the requirements for an advanced service selection process flexible enough to deal with evolutions in the service, network and client environment.

Category: Services Environment - **Event:** Multiple Services Available

Service Selection Enforcement: Based on a set of selection policies as specified by the client, the most appropriate service must be selected from the set of compatible services to deal with the client requests for some service functionality.

Category: Individual Web Service - **Event:** Service properties or behaviour changes

Notification or Detection: Changes in the service properties or behaviour must be noticed by the client in order to trigger the selection process as these changes may alter which of the available service is the most appropriate. Possibly, some changes such as the average monthly response time are only noticed after a period of time.

Category: Network environment - **Event:** Network influences service reachability

Notification or Detection: Changes in the network must be noticed by the client in order to trigger the selection process as the network influences greatly the availability and reachability of the services.

Category: Client environment - **Event:** Client specifies a new selection policy

Selection Policy Specification: The client specifies policies that drive the runtime selection of the services. These policies may be based on the context of the client, the context of the service, or based on non-functional or behavioural properties of the service. The specification of the most appropriate service may change over time depending on changes in the client and/or service environment.

Advanced Deployment: as policies can be based on service documentation, behaviour, service-context or client-context, an advanced deployment mechanism is needed to enforce the policy at runtime.

Reusable implementation: Code enforcing a policy must be reusable in different contexts.

Service Monitoring: if the policy involves service behavioural properties, a service monitoring mechanism is required to measure the behaviour of the Web service(s).

Category: Client environment - **Event:** Client specifies multiple selection policies

Feature interaction control: In case of multiple selection policies, feature interaction control is necessary in case of interference between the policies. Multiple policies must cooperate to select the most appropriate service to deal with a client request.

Category: Client environment - **Event:** Client changes one ore more selection policies

Hot Deployment: New policies must be deployed and enforced at runtime to accommodate to changes in the specification of the selection policies of the client.

3.4.2 Evaluating current practices

Evaluating the support for service selection, is quite straightforward as there are to our knowledge no standards or common approaches for this purpose. Doing service selection with proxies or DII is possible, but has to be implemented manually. For instance, if a client primarily communicates with *HotelServiceA*, but resorts to *HotelServiceB* for backup reasons, then the client will have to maintain two proxies and the logic to switch between the two proxies has to be provided manually. If more backup services are added later on, the code has to be changed. Furthermore, more advanced selection policies based on non-functional service properties or service behaviour may be applicable to select the most appropriate proxy. These policies, driven by constantly evolving business requirements, might need data from various sources including the Web service documentation, the Web service behaviour or the client state. All these points need to be intercepted to gather the required data, which becomes an impossible task if the system needs to deal with unanticipated selection policies.

We assume there are multiple reasons for the lack of more sophisticated selection mechanisms. First of all, doing dynamic discovery and integration is quite complex and cumbersome as illustrated in the previous section; and second, current service documentation in WSDL format does not support non-functional properties. But besides these technical and practical obstructions there is also the fact that up until now there has not been any *need* for dynamic selection, as today SOAs are realised in a hard-wired fashion. First, business agreements are made, followed by an implementation process involving all partners. Selection will become more important as Web services are combined in a temporal fashion.

3.5 Client-Side Service Management Process

3.5.1 Analysis of Requirements

Web services technology is simple: it only provides the necessary protocols to exchange XML documents, unlike other distributed object technologies (see section 2.7) that offer support for advanced features such as reliability and transactions. This simplicity means that many of the more complex distributed applications cannot be easily built without adding other technologies to the basic Web services [Vogels03]. We assume that over time, these features will become available in an interoperable manner, just as it is the case with WS-Security [ADH+02], a recent standard dealing with security extensions for Web services. It is not our objective to investigate or analyse any standards or specifications for a specific concern, nor do we attempt to propose our own. In this dissertation we focus on mechanisms to enforce these concerns dynamically in the client. The concerns may be imposed by the client, or by the service. We now list the requirements for this management process.

Category: Individual Web Service - **Event:** Service enforces Management Concern

Management Concern Enforcement: Invoking a service may impose a number of requirements and restrictions which the client must comply with before being able to

invoke the service. Examples include authentication, encryption and payment procedures. These criteria, referred to as *service-imposed management concerns*, may change over time, even without notice.

Exception Handling: In case of faults during service invocation, management related concerns need to be adequately dealt with.

Reusable implementation: Code enforcing a management concern must be reusable in different contexts.

Category: Individual Web Service - **Event:** Service enforces multiple concerns

Feature interaction control: In case of multiple management concerns, feature interaction control is necessary to deal with possible interference.

Category: Individual Web Service - **Event:** Service changes concern(s)

Hot Deployment: New concerns must be deployed at runtime to avoid communication errors with the services and to avoid a negative impact on the client application.

Notification or Detection: Changes in the client-side management requirements must be noticed by the client.

Category: Client environment - **Event:** Client enforces one or more concern(s)

Management Concern Enforcement: A client can specify additional management concerns, such as pre-fetching or caching. These criteria, referred to a *client-imposed management concerns*, can evolve too when the client or business environment changes.

System-wide triggering: Possibly, specific elements of the client-context need to be passed along in order to be able to enforce the concern.

3.5.2 Evaluation of Current Practices

3.5.2.1 Manual Configuration

Some toolkits allow creating proxies or call objects that can be further configured. Code fragment 3.1 shows a typical piece of Java code required to invoke a remote Web service. The code deals with various concerns including redirection, user authentication, the actual invocation, logging and exception handling. Clearly, the code for each of these various concerns is tangled. Moreover, in other places in the core application where a service invocation is required, similar or even identical code can be found: the code is also scattered.

In the code fragment, a simple form of authentication is used, but possibly the communication with the service needs to be encrypted, or the service needs to be paid in advance.

```
1  Import javax.xml.rpc.Stub;
2
3  Public class HotelClient {
4
5      private String endpoint, username, password;
6
7      public static void main {String[] args} {
8             try {
9
10                endpoint = args[0];
11                Stub proxy = createProxy ();
12                proxy.setProperty (Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
13                HotelInterface hotelService = (HelloInterface) proxy;
14
15                hotelService.setProperty (Stub.USERNAME_PROPERTY, username);
16                hotelService.setProperty (Stub.PASSWORD_PROPERTY, password);
17
18                Hotel hotel = hotelService.getHotel Park Hotel, Bruges);
19
20                log (HotelService invoked);
21         }
22
23         catch (Exception ex) {
24                ex.printStackTrace();
25         }
26     }
27
28     private static Stub createProxy () {
29         return (Stub) {new MyHotelService_Impl().getHotelInterfacePort();
30     }
31 }
```

Code fragment 3.1: Invoking a Web Service in Java

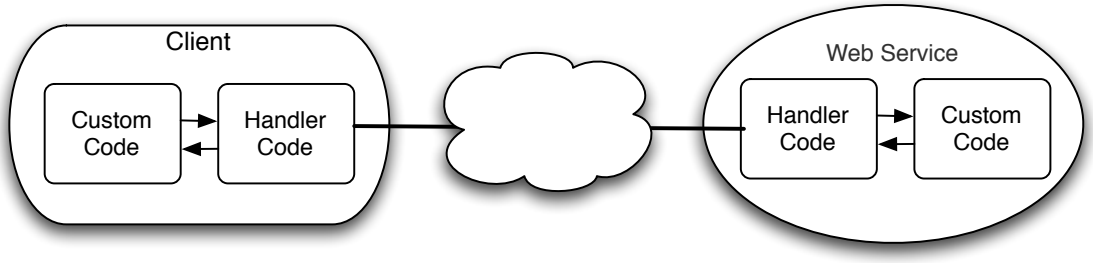

Figure 3.3: SOAP Message Handlers at Client and Service Side

All these concerns, enforced by the Web service, will be reflected in the code of the client. Therefore, the client is obliged to co-evolve with the service, even while loose-coupling is one of the key features of Web service technology.

3.5.2.2 SOAP Message Handlers

Another way to do client-side management of Web services is through *Message Handlers*, a mechanism to customise the SOAP messages sent to and from a Web service. Remember from section 2.4.1.3 that SOAP messages contain a body and a number of (optional) headers. The latter contain application-specific meta-data, such as security-related information. Handlers provide a mechanism to access, insert and remove these headers from a SOAP message. Handlers also have access to the rest of the SOAP message, including the body, so handlers can be used for example to encrypt and decrypt the data in the body of the message. Handlers are organised into chains and can be used both at the client-side and service-side as depicted in Figure 3.3. Chained handlers are invoked in the order in which they are configured. When a handler completes its processing, it passes the result to the next handler in the chain.

In JAX-RPC, a handler must implement a specific interface with specific methods to handle requests, responses and faults. Next, the handler must be registered at a specific position in the chain of a client proxy or service endpoint. This can be done programmatically or though deployment descriptors. It is possible to pass objects between handlers of the same chain to share state information specific to one request, but a handler always remains stateless [MTSM03]. Useful scenarios for message handlers include security (e.g. encryption and decryption of messages), processing of metadata (e.g. including headers with context information), data validation (e.g. validating a message against an XML schema before it is being processed), handling data content (e.g. handling attachments) and optimising performance (e.g. caching frequently accessed results in the server).

Developers write handlers as individual units that do not need to be aware of other handlers and are thus reusable. Also, because handlers are combined in chains, the order has to be explicitly configured, which can be crucial in some scenario's. For example, if a client sends an encrypted request in a compressed format, the handlers on the service side must first decompress and then decrypt the input [MTSM03]. Message handlers are very suited for

specific service management concerns dealing with reading or manipulating SOAP headers, however, higher-level concerns whose deployment is not limited to the message handling level are not supported. To deal with these issues, additional code has to be included in the client application, possibly resulting scattered and tangled with code addressing other issues. Even if this code is encapsulated in a separate reusable module, its execution has to be triggered repeatedly from the different points in the application where Web service functionality is required. As a consequence, management code results duplicated and scattered over the application, becoming an obstacle for future maintenance.

3.5.3 Table of Comparison

Table 3.2 compares the different client-side Web service management approaches. A (+) indicates full support for the requirements identified in Section 3.5.1, a (+/-) indicates partial support, possibly through additional mechanisms, and a (-) indicates there is no support at all as the developer needs to deal with the requirement by explicitly providing code.

Table 3.2: Comparison of different Service Management Approaches

	Manual Configuration	**SOAP Handlers**	**SOAP Handlers with deployment descriptors**
Exception Handling	-	+/-	+/-
Client-imposed Management	+/-	+/-	+/-
Service-imposed Management	+/-	+/-	+/-
Reusable implementation	-	+	+
Hot deployment	-	-	+
Feature Interaction Control	-	+/-	+/-
System-wide triggering	-	-	-

3.6 Conclusion

Each of the existing service integration approaches has its own advantages and disadvantages, and they have proven to be worthwhile in existing Web service applications, where Web services and their clients are developed at the same time, possibly by the same team, or where there are pre-existing agreements between all parties. Various tools, made available by both vendors and the open-source community help in creating object-oriented clients via proxies or DII. These tools provide the infrastructure that allows traditional procedure calls to be applied to Web services. It remains open for debate if these tools do not limit the potential of the Web services technology as Web services are all about interoperability

and heterogeneity, and viewing Web services as distributed objects with a set of remote methods does not favour these two elements.

Either way, the existing approaches do not offer the required flexibility to integrate a client with a wide variety of *existing third party Web services* in a dynamic Web service environment. Services do get hard-coded in the client, or only limited support is given to integrate services with deviant interfaces or to compose services together in a transparent way for the client. For service selection, no support is given to specify policies and to do any runtime redirection. Finally, support for client-side management concerns is provided through message handlers. This is only a useful approach for concerns that only need to be applied locally in one part in the client and where access to the message fields is sufficient. To address these shortcomings, we introduce our approach in the next chapter.

Chapter 4

Web Services Management Layer

Abstract To deal with dynamic service environments, we introduce a mediation framework for Web services, called Web Services Management Layer (WSML). This chapter presents several usage scenarios, the pursued development quality attributes and a high-level architecture for the framework. To avoid crosscutting code, we opt for a dynamic Aspect-Oriented Programming (AOP) approach. An introduction and motivation to AOP and an architecture of the WSML, based on aspects are presented. Also, the dynamic AOP language JAsCo is introduced.

4.1 Introduction

To address the shortcomings of the existing Web service integration approaches, we propose an architectural framework for the mediation of Web services in client applications. More concretely, all service related code is removed from the client application and placed in a mediation layer, called **Web Services Management Layer (WSML).** The WSML is deployed in between the client application and the service environment and its overall goal is to mediate between the (possibly static) client application and this constantly evolving service environment. This approach offers the following advantages for the client application:

- The client application becomes more flexible as it can transparently adapt to the changing business and service environment and communicate with new services and service compositions that were unknown or unavailable at deployment time.
- By weakening the link between the application and the service, hot-swapping functionality becomes possible. When a service becomes unreachable due to network conditions or service-related problems, this mechanism enables switching to other services or service compositions, based on a set of selection policies.
- Replacing the Web service-specific invocations with the generic request of service functionality and extracting all extra web-service selection and management code from the client applications facilitate future maintenance of the application code.

When deployed in a SOA, the WSML will reside in the centre of the SOA with interfaces on its four boundaries, as depicted in Figure 4.1. Each quadrant will involve different stake-holders fulfilling another role. On the west border, communication with the client environment takes place. The client environment consists of one or more clients requesting service functionality. The WSML redirects these client requests to the third-party service environment at the east border . The third-party service environment consists of a number of third-party service providers, each providing a number of services. These services remain under control of the service providers. Additionally, a number of management services reside in the management services environment on the south border. Management services offer generic functionality including AAA, accounting, logging, billing, etc. These services may be hosted by third-party service providers, or can be deployed in the client domain. On the north border, the administrative environment resides, responsible for configuration, signalling and event handling coming from both the client and third party service environment. Note that although communication with the four borders may involve Web services technology, this is not necessarily the case. In the remainder of this dissertation, the convention is followed that all illustrations including the WSML, will depict the elements of the environment in their corresponding quadrant.

The philosophy of our approach is that the client application is developed while leaving out all service related concerns. The service concerns will be entirely dealt with by the WSML. For this purpose, the WSML must be instantiated and configured to the needs of the client. Ideally, this can happen entirely at the administrative level, meaning no coding efforts or expert skills of the WSML are required. As will be explained in Chapter 8, automatic code generation and reusable templates are used for this purpose. Note that each

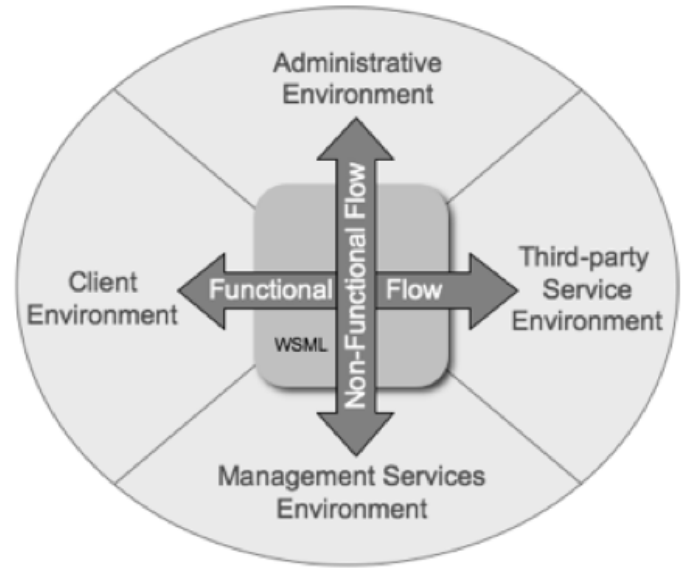

Figure 4.1: The WSML Interfaces

of the four quadrants may be managed and controlled by a different organisation. Possibly, the quadrants of the north, south and west border may be under the same organisational control implying that the entities responsible of developing, deploying and maintaining the client application, are also in control of administering the WSML and/or maintaining the (local) management Web services. However, it should be noted that the philosophy of our approach is that, if the entities are not the same, they are not required to be sharing the same technical knowledge of the other systems. This implies that the developer of the client does not require to have in-depth technical knowledge about the implementation of the WSML and vice versa. The WSML can be configured and deployed on an administrative level, requiring only business knowledge from the client (e.g. the knowledge of which kinds of non-functional service properties are to be preferred when selecting a service and which management concerns need to be enforced), while the WSML can operate to a large extent without knowledge about the client implementation. Some exceptions exist, for instance if services are to be selected based on client context, information of this context has to be made available in the WSML.

Before discussing the architecture of the WSML, we first list a number of envisioned usage scenarios in the following section. Next, the development quality attributes of the WSML are listed in section 4.3 and its main architecture is presented in section 4.4. As already hinted before, our approach relies heavily on aspect-oriented principles: in section 4.5, we give a general introduction to AOP and a motivation why AOP is an ideally suited paradigm to realise the WSML. The section ends with a detailed overview of the architecture of the WSML, based on aspects. We conclude this chapter with an introduction to JAsCo, the dynamic AOP language used to implement a prototype of the WSML, and present conclusions in section 4.6.

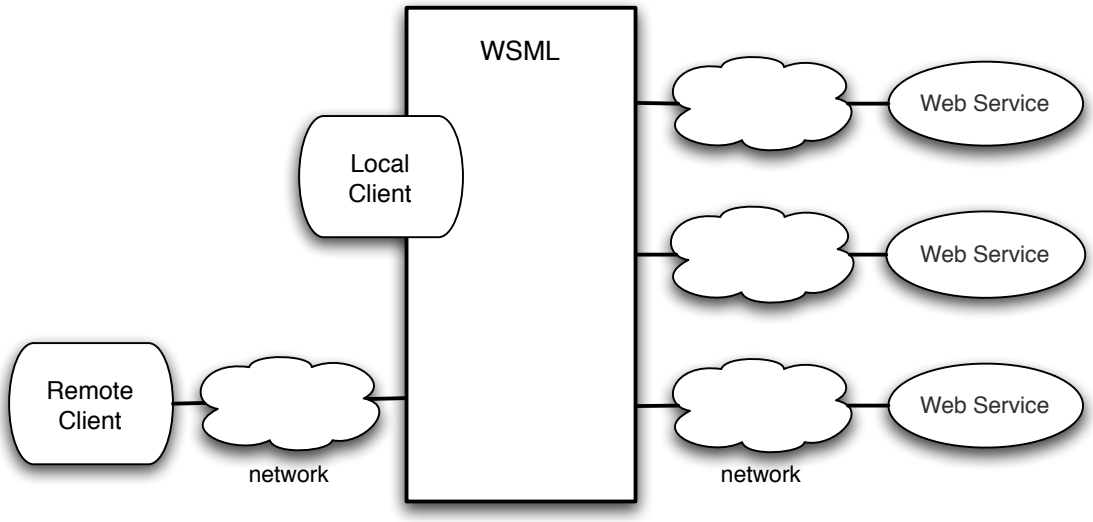

Figure 4.2: The WSML as Mediation Layer and/or Service Broker

4.2 Usage Scenarios

4.2.1 Web Services Mediator

As indicated before, the primary usage of the WSML is to act as a mediation layer between the client and the services and to manage and to configure the client-side usage of those services. Changes in the service or network environment only have an impact on the WSML while the client application remains unaware of any changes. This scenario is suited to deal with services in a fragmented environment with multiple independent domain controllers and unreliable network conditions. Two setup variations are possible: the WSML serves a single application and runs with that application in a single environment or, in a large-scale setup the WSML is deployed as a server, with the possibility to serve multiple clients. Both setups are depicted in Figure 4.2.

Example: a company relies on external shipping handlers to deliver its products to the customers. Each handler is specialised in dealing with specific product deliveries in specific areas, so depending on the product and its destination, a specific shipping handler is addressed. The company can integrate its software with the shipping Web services to get pricing information, place orders and follow-up deliveries. The WSML is used in this scenario to mediate between the different shipping service interfaces and deal with additional non-functional requirements. For instance, when a shipping service changes its WSDL-file, or installs a different login mechanism or requires a stronger encryption protocol to encrypt its messages.

4.2.2 Web Services Broker

The WSML acts as an application-tailored broker for Web services. Client requests are redirected to the most appropriate services in the resource pool, based on the required service levels, cost structures, policies, and priorities specified by the client. This scenario has the advantage that the client can reduce the negative impact of "bad services", i.e. services that are often unreachable or provide unreliable results. Reconfiguring the WSML can easily change the runtime characteristics of the client.

Example: a software company delivers stock quotes to its customers and relies on a wide range of different Web services to provide the stock information. These services are in high competition with each other and offer various delivery schemes, ranging from delayed stock quotes provided at specific intervals to real-time up-to-date quotes. Different pricing schemes are also possible, including micro payments per requested stock to monthly or annual subscriptions. In this scenario, the WSML is used to address the appropriate stock service(s) taking into account the client subscription, the payment schemes and the reliability of the stock information amongst others.

4.2.3 Web Services Grid

By presenting the available services as a pool of heterogeneous system resources, a grid can be created that handles all client requests. Grid computing allows you to unite pools of servers, storage systems, and networks into a single large system This resource-level virtualisation is ideally suited when multiple services are available and it is of no importance which concrete services handle the client requests.

Example: An application involved in life sciences, such as genome research and pharmaceutical development, can use parallel and grid computing to process, cleanse, cross-tabulate, and compare massive amounts of data. The WSML can be used to distribute requests evenly over all available services to optimise processing of the data or to optimise requests for information spread over many data repositories by broadcasting the requests and merging the results.

4.2.4 Web Services Intermediary Stub

Web service communication between two (legacy) systems can be intercepted by the WSML for a wide variety of reasons including re-routing, monitoring and other management concerns as depicted in Figure 4.3.

Example: Consider an existing proprietary client application that communicates with a server using Web service technology to request video downloads. By deploying the WSML in between the client and the server it can become possible to for instance count the number of downloads, store this information in a database, and update a billing server. Neither the client, nor the server needs to be changed for this. Therefore, this scenario is suited to adapt or extend existing legacy software.

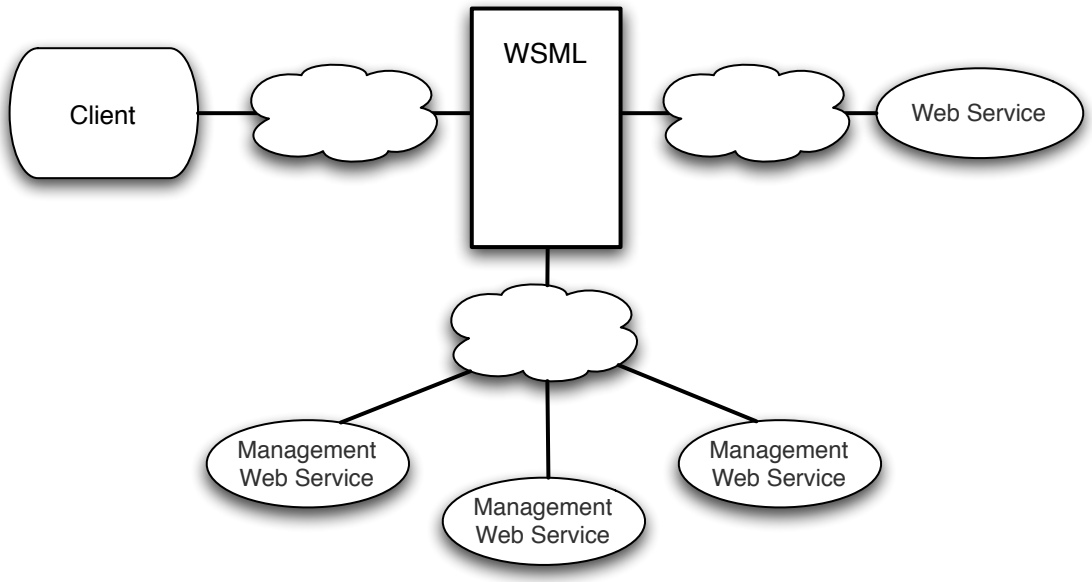

Figure 4.3: The WSML as Intermediary Component in a Client/Server Model

4.2.5 Web Services Ubiquitous Environments

The philosophy of writing an application once and deploying it many times, in different environments is a noble one, but difficult to achieve. The WSML offers support for ubiquitous computing as the mediation capacities of the WSML can make a client application adapt to different environments, as depicted in Figure 4.4. By taking into account the different characteristics of the client environment, more efficient service selection and communication can be done.

Example: Consider a video player application, showing movies originated from external Web services. If the video player runs on a digital media centre connected to a broadband internet connection, it is able to show any kind of movie, regardless the quality and size of the movie file. It might therefore prefer to communicate with Web services offering movies in the best resolution at an affordable price. On the other hand, if the same application is run on a mobile device with a wireless connection, more restrictions apply. Only movies with a low resolution can be played and only small files can be transferred over the low bandwidth connection. Furthermore, the device has limited processing and memory resources and a limited power source. Now, the application will have to choose a movie Web service that offers the movie in a compatible format, in a low resolution and in a small-sized file. The WSML can be used to adapt to the different devices and environments the client application is deployed in, and based on that, select the most appropriate Web services. In this setup, the WSML also optimises communication patterns over the expensive wireless network, for instance by pro-actively removing unreachable Web services or caching service results.

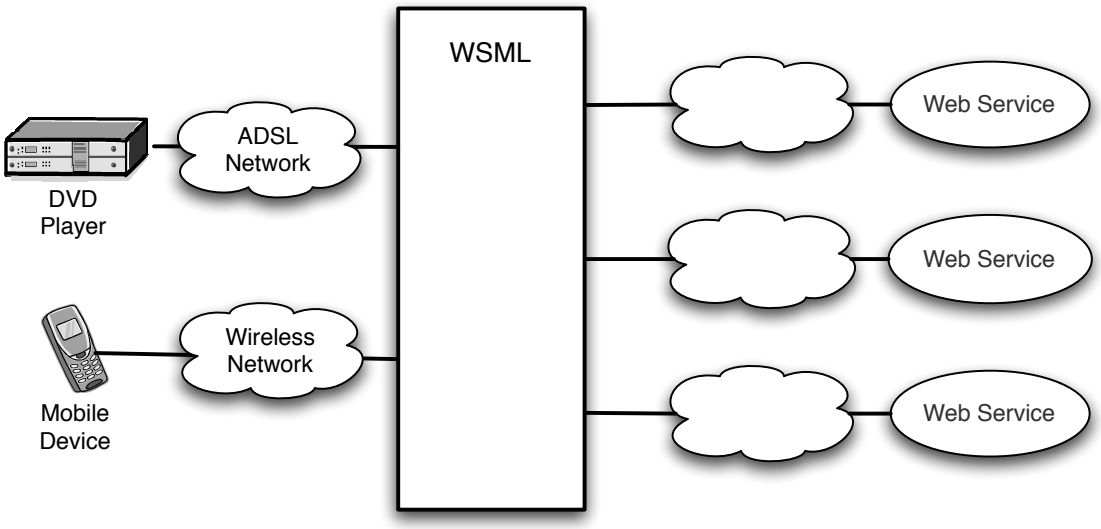

Figure 4.4: The WSML in a Ubiquitous Environment

4.3 Development quality attributes

The requirements we pursue for the WSML follow automatically from the analyses made in Chapter 3 in the context of dynamic Web services environments. However, before drawing up an architecture for the WSML and finding a suitable implementation technology, we will discuss overall development quality attributes, we pursue too. A first category includes quality attributes dictated by the Web services context:

- **Web Services standards compliance:** as the philosophy of the WSML is to realise integration of Web services "as-is", the existing standards are to be re-used. It is outside the scope of this dissertation to propose new standards or adaptations of existing standards to realise the WSML. Instead, we opt to benefit from the work already realised by current Web services standardisation efforts and to comply with the existing Web services stack and to allow for extensions by future standards.

- **Performance and scalability:** making a mediation framework that is configurable at runtime requires inevitably a very flexible runtime adaptation mechanism, which can affect performance and scalability of the framework. As it was mentioned in Chapter 2, performance is one of the weaknesses of Web services technology, partially because of the verbose nature of XML. Most performance gains can be realised by optimising the (de)serialisation and the compactness of the SOAP messages. Therefore, the performance goal of our framework is to achieve results comparable to existing integration approaches. Scalability will be dealt with by the underlying server platform, and the scalability of our overall approach should be limited by that platform, and not by our framework on top.

Next, we also pursue a set of development qualities, intrinsic to the development of a framework:

- Reusability: designing and implementing a middleware framework is a complex task, therefore the WSML must be a domain independent mediation platform for Web services reusable in multiple scenarios. The WSML must be a generic framework that can be deployed and configured for a single concrete client application or groups of applications. As such, context-dependent code must be limited as much as possible.

- **Runtime Configurability:** the WSML must be completely configurable at runtime for a specific context. Optionally, this requires specifying unanticipated configuration settings that need additional code to be inserted at runtime. By introducing additional abstraction levels, this configurability must become preferably an administrative effort rather than an implementation effort. We envision different kinds of configuration approaches, including an administration interface directed to humans, an administration service and a dedicated configuration language.

- **Runtime Extensibility:** As it is impossible to develop a framework able to deal with all current and future service management concerns, extensibility of the framework is of vital importance. Also, as the WSML is central hub in SOA, possibly executing long-running business processes or critical applications, it is no option to stop and rewrite the code. The framework should therefore be extensible at runtime to incorporate and enforce new service concerns.

In order to achieve a framework that is reusable and flexible enough to deal with runtime changes, clean *modularisation* of the code is of vital importance. The next section introduces the architecture of the WSML, where all service related concerns are modularised in separate modules.

4.4 WSML Architecture

The WSML is a mediation framework for Web services for service integration, selection and client-side management that takes care of service integration issues, enforces a set of service selection policies and client-side service management concerns and must function transparently for the client. The service concerns must be enforced at runtime and possibly they were unanticipated at development time or deployment time. Therefore, we need to modularise every concern in separate modules and have a flexible mechanism to dynamically deploy these modules. Figure 4.5 illustrates the architecture of the WSML. The left-hand side of the figure illustrates an application requesting Web service functionality. In order to make it possible for the application to make requests without referencing concrete services the *Service Type* concept is introduced. A service type represents some abstract service functionality without referencing concrete services. The right-hand side shows semantically equivalent services that are available to answer the request. Semantically equivalent services are services that offer the same functionality but can differ in the way they provide it, such as method names, number of parameters, etc. Note that Web services B1 and B2 need to cooperate together in a composition. Inside the WSML, separate modules take care of the service integration, service selection and service management process:

- **Service Integration Module:** The integration module realises the redirection of client invocations on the service type towards a functionally compatible Web service.

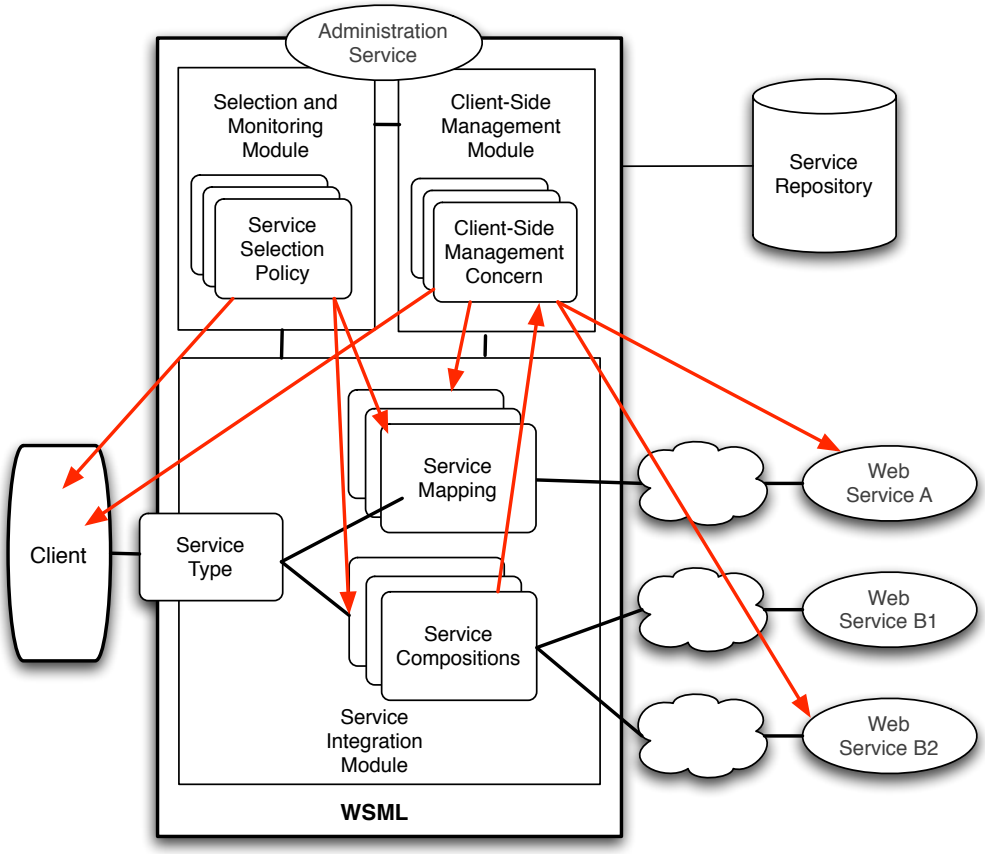

Figure 4.5: Modularised Architecture of the WSML

If needed, a mapping is made between the service type interface and the Web service interface. Optionally, multiple services are combined in a service composition to offer the functionality needed by the client. New services can be integrated at runtime.

- **Selection and Monitoring Module:** This module enforces a set of selection policies that are used by the integration module to redirect client request to the most appropriate service. These policies can be determined and change at runtime. Optionally, service monitoring is needed to collect the necessary data to enforce the selection policies.
- **Client-Side Service Management Module:** Various client-side service concerns are dealt with in the management module. Examples include caching, logging, billing, pre-fetching and exception handling. Which concerns are to be enforced can vary at runtime. Furthermore, anticipated concerns may be added and feature interaction issues must be taken into account.

A big issue in this framework are the interaction between the modules. The red lines in the Figure indicate some of the possible interactions. Each module needs to communicate with the other modules, or even with the client or the remote Web services, possibly in an

unanticipated fashion. Using traditional software engineering approaches, code implementing these concerns will result tangled and scattered with other concerns. Our approach is to use Aspect-Oriented Programming (AOP), a software engineering paradigm that aims at achieving a better separation of concerns. An introduction to AOP and a detailed motivation to use AOP in the WSML are the subject of the next section.

4.5 Aspect-Oriented Programming in the WSML

4.5.1 Introduction to AOP

Aspect Oriented Programming (AOP) [KLM+97, EFB01, FICA04] is a software engineering paradigm that aims at achieving a better *separation of concerns* [Parn72], a crucial property for realising comprehensible and maintainable software. Software development addresses multiple concerns, both at the user/requirements level and the design/implementation level. Often, the implementation of one concern will result scattered throughout the rest of the implementation. These concerns are called *crosscutting* because the concern virtually crosscuts the decomposition of the system. What is crosscutting is a function of the particular decomposition of the system and the underlying support environment. A particular concern can be crosscutting in one view of an architecture while being localised in another [FICA04].

Using traditional software engineering methodologies, the implementation of a crosscutting concern will result tangled and scattered, meaning the code for different concerns becomes intermixed, possibly at multiple places in the system. Because the crosscutting concerns are spread and repeated over several modules in the system, it becomes very hard to add, edit, verify, test or remove such concerns individually. Moreover, the scattered and tangled code seriously hampers the evolution of (1) the concerns and (2) the base application. Typical examples of crosscutting concerns are debugging concerns such as logging [KLM+97] and contract verification [VSJ03], security concerns [DJ04] such as confidentiality and access control, and business rules [CDJ03, DJ04, OT01] that describe business-specific logic.

The goal of AOP is to allow developers to cleanly modularise crosscutting concerns. Therefore, AOP introduces an additional module construct, named an *aspect*. Traditional aspects consist of two main parts: a pointcut definition and an advice. Points in the program's execution where an aspect can be applied are called *joinpoints*. The *pointcut* language allows for a quantified description of a set of joinpoints where the aspect should be applied. The *advice* is the concrete behaviour that is to be executed at a certain pointcut, typically *before*, *after* or *around* the original behaviour identified by the joinpoints. The additional logic defined in a before or after advice has to be executed before or after the original behaviour respectively. An around advice replaces the original behaviour, but is still able to invoke it if necessary. The advice language typically consists of the host language augmented with a limited number of special keywords that offer aspectual reflection and control over the execution of the original joinpoint. In order to apply the advices at the joinpoints specified in the aspect's declared pointcut(s), the aspect needs to be *weaved* with the base application. Traditionally, weaving takes place at compile-time, which means

that the advices are inserted into the target application at the source or byte-code level. Currently, more advanced approaches allowing to weave in aspects at runtime at previously unadvised joinpoints are pursued too [VS04]. Other techniques are also possible, for instance in *Steamloom*, weaving of advices is postponed until runtime by introducing AOP concepts into the execution model underlying the virtual machine [BHMO04].

Although AOP is a rather new paradigm, numerous aspect-oriented approaches have been proposed and are currently reaching maturity. These include *AspectJ* [KHH+01], *HyperJ* [TOH+99], *Adaptive Programming* [LOO01], *Composition Filters* [BA01], *AspectWerkz* [Boner04], *JBoss/AOP* [FR03], *Spring/AOP* [JHA+05] and *JAsCo* [SVJ03]. Most approaches focus on object-oriented software development (OOSD), while some, including the latter three, also offer support for component-based software development (CBSD). Similar to OOSD, several concerns are encountered in CBSD that crosscut multiple components in the system. In [SVJ03] it is argued that combining the AOSD and CBSD principles is a valuable contribution to both paradigms. Firstly, integrating the principles of AOSD in CBSD helps in achieving better separation of concerns over multiple components. And secondly, integrating CBSD principles in AOSD is valuable as well as CBSD puts a lot of stress on the plug-and-play characteristic of components; for example, it should be possible to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, makes aspects reusable and their deployment easy and flexible. AOP approaches supporting CBSD typically have some kind of *deployment constructor* that is used to deploy a reusable aspect in a specific context. As services are the logical extension of components in a distributed fashion, AOP can also help in achieving better separation of concerns in this context.

4.5.2 Motivation for AOP in the WSML

As indicated before in Figure 4.5, using traditional software approaches, the client code dealing with the various service integration, composition, selection and management concerns, as discussed in detail in chapter 3, results scattered and tangled in the client code.

Service Integration: When using current approaches based on traditional software engineering approaches, the service integration process takes either place at development time (e.g. using static proxies), or it takes place at runtime (e.g. using dynamic proxies or DII) but leaving all coding efforts to the programmer. The code, necessary to setup a proxy or dynamic interface, and to invoke a Web service, can become quite complex, especially if a robust implementation is necessary. This service integration code will appear at any place in the client where service functionality is required. As a result, changes and evolutions in the service interface will need to be reflected at all these places in the client code. Glue code, dealing with mismatches and typically written on a per service-basis, will result scattered in the client and there is no or very limited possibility for code reuse as code is implemented for a specific context. Furthermore, similar to the plug-and-play characteristics of components, services should be interchangeable at runtime. But replacing one service with another functionally equivalent service may again require changes at multiple locations in order to deal with variations in the interfaces. In case service compositions are needed to fulfil the functionality needed in the client, the situation is aggravated, as at those places in the client, multiple proxies will need to be maintained. As each proxy represents a service

that possibly belongs to a different provider, each proxy may impose further requirements or restrictions. For instance, as discussed in Chapter 3, section 3.3.2, data retrieved from one service cannot be straightforwardly passed along to a next service, but may require additional configuration efforts or data structure copying efforts.

Service Selection: Taking into account not only the functional compatibility of a service, but also expressing requirements on the non-functional level, will inevitably result in more complex client code. Selection policies expressing Quality-of-Service criteria the services needs to uphold, or taking into account the client requests or the client state to select the most optimal service will require data from a variety of places, resulting in scattered and tangled code. For instance, monitoring logic, required to observe the service behaviour in order to determine fair and neutral QoS descriptions requires multiple measurement points in the system. Furthermore, these data must be fed into a selection mechanism that enforces a set of selection policies determining the most optimal service at any given time. As selection policies are driven by businesses requirements, they can be regarded as business rules [Bus00] that tent to evolve more frequently than the core application functionality, as discussed in [CDJ03]. It is crucial to separate them from the core application in order to trace them to business policies and decisions, externalise them for a business audience and change them whenever deemed necessary. Selection policies thus need to be kept separated from the services and the applications that integrate them in order to enhance maintainability, reusability and adaptability. Using traditional software approaches, monitoring and selection logic will result tangled and scattered in the client at those places where service functionality is needed. In this context it is important to note that it is required to not only encapsulate the code implementing the policy, but also the code that links the policy to the client code, as this linking code may result crosscutting as well. In addition, it is impossible to anticipate each and every kind of policy and which kind of monitoring it will require. A system that is able to access all available data and runtime events in an oblivious manner is required. This system must be able to introduce all required monitoring points and enforce selection policies at runtime, while encapsulating code in first-order entities.

Service Management: the wide range of management concerns a service might impose on its clients, including authentication, encryption, payments will be reflected in the code of the client. Therefore, the client is obliged to co-evolve with the service, even while loose coupling is one of the key features of Web service technology. These concerns, together with the management concerns imposed by the client, including logging, caching and pre-fetching will result in tangled and scattered client code. This was also illustrated in Code fragment 3.1. Note that similar to selection policies, the management concerns that need to be enforced in the client will vary over time. For instance, enforcing some kind of caching makes sense in case of network bottlenecks while monitoring is only necessary when the service behaviour needs to be analysed. Finally, some complex kinds of service management need to be implemented in a distributed manner. Examples include distributed monitoring where monitoring points are set up in the client, the network layer and the service environment, or advanced transactional management. Implementing these concerns in an encapsulated manner is impossible with traditional software practices.

An important criterion that was mentioned in each of the three identified categories is the possibility to enforce concerns dynamically. Due to the evolving nature of the Web services

environment, it is desirable to plug in and out the integration, selection and management code at runtime. Either to swap services, reflect business changes for selection policies or to enforce client-side management concerns, there is a need to dynamically alter the service-related code of the client. It is impossible to anticipate all kinds of possible alterations needed in the client in the future, so hard-coding each concern at all possible places where the concern might seem applicable and making sure that the concern captures all data that is possibly relevant (e.g., name, target, arguments and possible exceptions), is impossible.

By opting for an AOP approach, each of the aforementioned concerns can be cleanly modularised in separate modules, i.e. the aspects, and enforced in the code in an oblivious_manner enhancing the evolution and maintenance of the code. Early AOP approaches weave aspects with the core application at compile time, which requires recompiling the application every time a set of aspects have to be integrated or removed. This approach is not suitable in a service-oriented environment where applications need to minimise as much as possible their downtime. Therefore, an AOP technology that provides support for runtime addition and removal of aspects is more appropriate. With a *dynamic AOP-approach* it becomes possible to anticipate changes in the client, network and service environment without having to stop and alter the code of the client application. With dynamic AOP, the aspects can be plugged in and out at runtime, and as such enforce various selection policies and management concerns in the client. This is particularly important in critical applications dealing with long-running intra- or inter-organisational processes that cannot be stopped easily. This is the main focus of the remainder of this thesis.

While some requirements such as the dynamic binding of Web services could also be realised using other approaches (for example a regular Object-Oriented framework that exploits late binding), we observe that for many others, including the enforcement of unanticipated selection and management concerns, the use of AOP has many clear advantages that no other approach offers. The fact that we want to enforce these crosscutting concerns in an oblivious manner by inversion-of-control indicates AOP is an ideal approach. While we could emulate this also with OO by extensively adopting the Observer Pattern [GHJ95] (i.e. by making everything observable and having observers that register themselves where needed), it is clear that we are in this case in fact making or own pseudo AOP-approach, which will be less powerful and expressive than a dedicated AOP technology. As we wanted to use a unifying technology to realize all concerns, we have adopted AOP, not only for the selection and management concerns, but also for the dynamic integration of Web services. As we will discuss in the next chapters ,the flexible runtime capabilities of dynamic AOP allowed us to realize all identified requirements in a natural fashion, omitting the need to resort to another approach.

4.5.3 WSML Architecture Based on Dynamic Aspects

By modularising every service related concern in the WSML in separate reusable aspect, the advantages AOP offers in the OOSD and CBSD context are also applied in the Web services context: each concern is modularised in a first-class entity in a reusable manner, and can be applied at runtime at every possible joinpoint that can be described by the pointcut language of the used AOP language. Figure 4.6 illustrates the architecture of the WSML. Aspects are used for implementing the generic functionality of the management

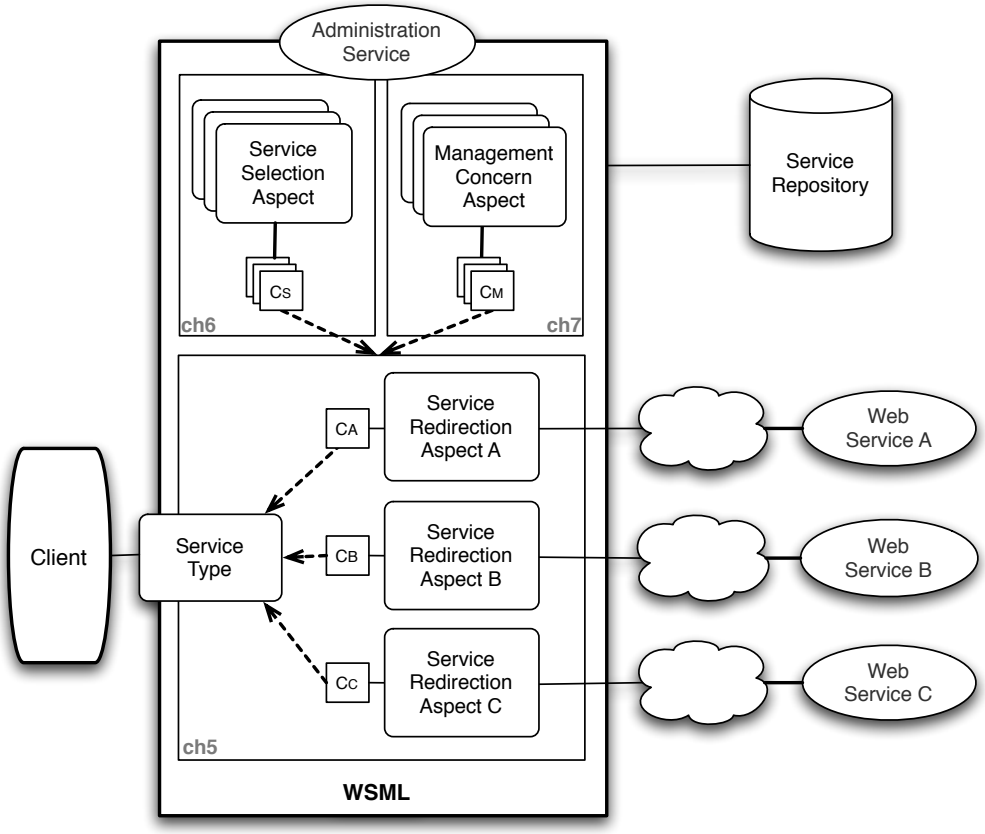

Figure 4.6: Detailed Architecture of the WSML with Aspects

layer while deployment descriptors (indicated with C) specify when these aspects need to be deployed. This is depicted in Figure 4.6. In the WSML, we distinguish three categories of aspects:

- **Service Redirection Aspects:** Web service communication details and service composition details are encapsulated in redirection aspects. By employing the dynamic capabilities of an AOP technology, a flexible integration and redirection mechanism for Web services and service compositions is realised. This is the topic of the next chapter.

- **Selection and Monitoring Aspects:** Service selection policies, encapsulated in selection aspects implement a generic selection mechanism that can be easily enforced and configured at runtime. Additional monitoring aspects are used to gather service behaviour data to drive the selection process. This is further discussed in chapter 6.

- **Client-Side Service Management Aspects:** management aspects deal with a wide variety of client-side service management concerns such as caching, logging, billing and exception handling. These concerns can be enforced at runtime, while taking into account possible feature interaction issues. Management is the subject of

chapter 7.

There is lively ongoing debate in the AOP-community on what is exactly crosscutting. In [FECA04], the two main properties required for AOP are identified as being *quantification* and *obliviousness*. The quantification property implies that aspects are added to an application via quantified program statements, i.e. statements that have effect on many places in the underlying code. Obliviousness states that one cannot tell that the aspect code will execute by examining the body of the base application. This is exactly what is necessary to achieve better separation of concerns. In case of the service redirection aspects in the WSML, we find that the obliviousness property is only partially fulfilled. After all, requesting some service functionality is part of the client logic, unlike for instance adding an optional caching or selection concern. However, the added complexity of setting up proxies or dynamic interfaces, doing mappings through context specific glue code, composing services together, etc. is not part of the client logic. While we do not claim that AOP is the only way to achieve better separation of concerns in this particular case, we do have chosen for a symmetrical approach where the three identified service concern categories are implemented in aspects, particularly because all concerns require dynamic plug-and-play characteristics. As we will illustrate in Chapter 5, a lot of features required for a dynamic service integration mechanism are readily available in a dynamic AOP technology such as JAsCo. For the selection and management aspects both the quantification and obliviousness properties hold as these concerns need to be applied in multiple places in the client, network or service environment, requiring static or even dynamic quantification, while applying them obliviously avoids tangled and scattered code.

Before continuing with the technical details of chapters 5 to 7, we will first conclude this chapter with a relative brief introduction to JAsCo, the dynamic AOP-language chosen to implement the prototype of the WSML. All coding examples in the next chapters are given in JAsCo, therefore a basic introduction to the language and its runtime characteristics can be useful for readers unfamiliar with the JAsCo technology. JAsCo was chosen as an implementation technology because of its dynamic capabilities combined with the possibility to write reusable aspects independently of their deployment context. Also, JAsCo is being developed in the same lab as the WSML, which resulted in a setup where new features of JAsCo could immediately be deployed and tested in an extensive AOP-centered framework and vice versa, lacking features that were desired in the WSML context and beneficial to the AOP capabilities of JAsCo, could be implemented more easily and rapidly in JAsCo.

4.5.4 JAsCo

4.5.4.1 The JAsCo Language

JAsCo [SVJ03, Van04, VVSV03] is a dynamic AOP language, originally targeted at component-based software engineering. JAsCo enables weaving advices to sets of joinpoints in a base application so that the advices are executed before, around or after the execution of the joinpoints. The main contributions of the JAsCo language with respect to other AOP approaches are its highly reusable aspect modules and its strong aspectual composition mechanism for managing combinations of aspects. The JAsCo technology sup-

ports dynamic integration and removal of aspects with minimal performance overhead. The JAsCo language is an aspect-oriented extension for Java that stays as close as possible to the original Java syntax and concepts. It introduces two important additional entities: *aspect beans* and *connectors*:

- **Aspect Beans**: an aspect bean is an extended version of the standard Java bean component that is able to contain several *hooks* that capture the crosscutting behaviour. Hooks define when the normal execution of the system needs to be intercepted in an abstract way. In addition, hooks specify the extra behaviour that needs to be executed at that moment in time.
- **Connectors**: a connector is responsible for connecting an aspect bean to a concrete context. Furthermore, expressive combination strategies and precedence strategies can be defined in order to manage the collaboration of several aspect beans and components. As such, a partial solution for the notorious *feature interaction problem* [PSC+01] is provided.

An aspect bean is an extended version of a regular Java bean that is specified independently of concrete component types and APIs, which makes it highly reusable. An aspect bean contains one or more logically related hooks that describe the crosscutting behaviour itself. The advices of a hook are used to specify the various actions that a hook needs to execute when the hook is triggered. Hooks are able to define three main types of advice (i.e. before, around and after), and four more advanced advices:

- **before():** the advice is executed before the joinpoint is executed.
- **around()**: the advice is executed around the joinpoint.
- **around returning(TypeX):** the advice is executed around the joinpoint when an object of TypeX is returned. As such, the return value of the joinpoint can be altered.
- **around throwing(ExceptionX):** the advice is executed around the joinpoint when an exception of type ExceptionX is thrown. The thrown exception can be altered re-thrown, or a normal result can be returned instead.
- **after():** the advice is executed after the joinpoint regardless of how the joinpoint executes (i.e. whether a normal result or an exception is executed).
- **after returning(TypeX):** the advice is executed after the joinpoint when an object of TypeX is returned. The type Object can be used to capture all returns.
- **after throwing(ExceptionX):** the advice is executed after the joinpoint when an exception of type ExceptionX is thrown. The type Exception can be used to capture all exceptions.

Code fragment 4.1 illustrates an aspect bean that captures a basic access control security concern. The crosscutting behaviour, in this case intercepting the method invocation and

```
1  class AccessManager {
2      PermissionDb p_db = new PermissionDb();
3      User currentuser = null;
4      Vector listeners = new Vector();
5
6      public boolean login(User user, String pass) {}
7      public void logout() {}
8
9      hook AccessControl {
10         AccessControl(method(..args)) {
11             execution(method);
12         }
13
14         isApplicable() {
15             return !p_db.isRoot(currentUser);
16         }
17
18         around() {
19             if(p_db.check(thisJoinPointObject,currentuser))
20                 return proceed();
21             else throw new AccessException();
22         }
23     }
24 }
```

Code fragment 4.1: JAsCo Aspect implementing an AccessManager

throwing an exception when the current user does not have the required credentials, is captured in the `AccessControl` hook (lines 9 to 23). A hook includes a special kind of constructor that defines in an abstract way when the hook has to be triggered (lines 10-12). A constructor receives several abstract method parameters that are bound to one or more concrete methods at the time the aspect is deployed. The constructor body specifies when the hook needs to be triggered. In the case presented in Code fragment 4.1, the `AccessControl` hook is triggered whenever one of the methods bound to the abstract method parameter method is executed.

JAsCo supports more involved triggering conditions in a constructor body, for example control flow conditions that can be combined using logical operators. Often, the execution of a hook does not simply depend on conditions about the static and dynamic program information that are defined in a constructor. The `isApplicable` method is able to specify an additional triggering condition using the full expressiveness of Java (lines 14-16). In many scenarios the conditions that determine whether a hook should be triggered cannot be specified in a hook constructor. For example, a hook implementing an anniversary discount business rule is only triggered when the current date is the client's birthday. The `isApplicable` method allows specifying these additional conditions. In our example, the `AccessControl` hook is triggered whenever the current user is not the root user. Checking access for the root user is useless because by definition this user has access to the complete system.

The `AccessControl` hook defines an around advice that contains the main access control behaviour (lines 18-22). The `thisJoinPointObject` keyword makes it possible to fetch the

```
1 static connector PrintAccessControl {
2     AccessManager.AccessControl control =
3         new AccessManager.AccessControl(Printer.doJob(PrintJob));
4         control.around(); //optional
5 }
```

Code fragment 4.2: Connector for the AccessManager Aspect

target object of the currently encountered joinpoint. As such, the advice makes sure that whenever the user does not have the correct permissions to access the target object of the current method invocation, an exception is thrown. Otherwise, the original execution continues by employing the proceed keyword. In case other aspects are also applicable to the same joinpoint, the proceed method proceeds to the next aspect's around advice, realising a chain of around advices that ends at the method originally replaced.

This aspect is reusable because the application context is not hard coded. Furthermore, by defining the abstract context in a constructor, JAsCo is able to guarantee type safety. For instance, a hook can specify that it expects a method with an object of type integer as first argument. When the hook is deployed, this abstract context can be statically checked. This is in contrast to typical framework-based approaches such as JBoss/AOP [FR03], Spring/AOP [JHA05] and AspectWerkz [Boner04]. In these approaches, the aspect has to include possible type-unsafe casts. In order to further increase the aspect reusability, an aspect bean is also able to defer the implementation of context-specific behaviour to the definition of the connector using refinable methods. As such, JAsCo aspect beans are able to support aspectual polymorphism [MO03].

Abstract aspects are deployed onto a concrete context using a connector. A connector can explicitly instantiate and initialise one or more hooks. Code fragment 4.2 illustrates a connector that instantiates the `AccessControl` hook onto the `doJob` method of the *Printer* component. As a result, the method abstract method parameter of the `AccessControl` hook is bound to the `doJob` method. The hook is thus triggered whenever that `doJob` method is executed and the current user does not have root permissions. Note that in JAsCo it is also possible to bind abstract method parameters to several concrete methods. Moreover, JAsCo connectors also support wildcards to easily specify a range of methods for aspect application.

By default, all defined advices of a hook are executed when the hook is triggered. Optionally, a connector can selectively specify which advices have to be executed (line 5). When several hooks are instantiated in a connector, this mechanism enables specifying the precedence of the hooks on a per-advice type basis. In addition, connectors are also able to specify more expressive combination strategies in order to manage the cooperation among several aspects that are applicable at the same joinpoint. A combination strategy acts like a filter on the list of applicable hooks and is implemented using plain Java. An example of a combination strategy is a mutual exclusion strategy that specifies that whenever a certain hook is triggered, other hooks cannot be triggered. These precedence and combination strategies offered by JAsCo are an important step towards solving the well-known aspect-oriented feature interaction problem [PSC+01].

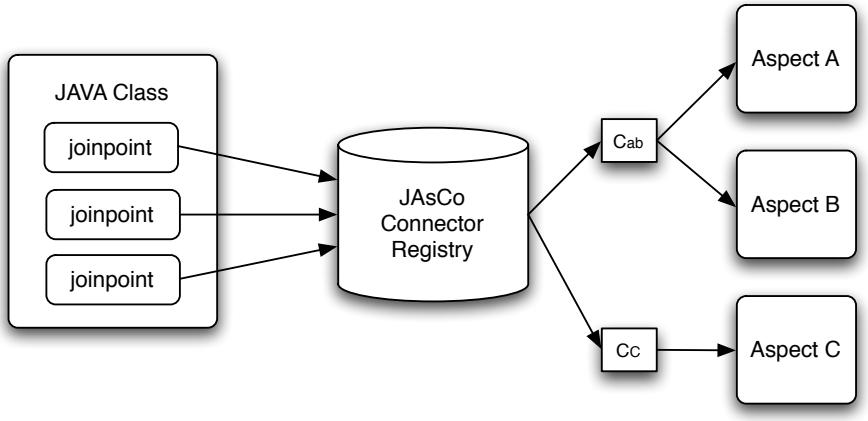

Figure 4.7: JAsCo Runtime Architecture

4.5.4.2 JAsCo Runtime Infrastructure

The JAsCo technology is originally based on a new, backward compatible component model where traps that enable aspect interaction are already built-in. Ideally, new software is shipped employing this new component model. This way, attaching and removing aspects to components does not require any adaptation whatsoever to the target classes. Of course, expecting all components to be developed using this new component model is rather utopian. Therefore, it is also possible to automatically transform a regular Java class into a JAsCo class by employing a pre-processor that inserts the traps using byte-code adaptations.

The JAsCo technology is also very flexible to support *unanticipated runtime changes*. Connectors and their corresponding aspect instances can easily be loaded and unloaded at runtime by using the JAsCo runtime infrastructure API. In addition, JAsCo includes a very flexible system for remotely (from outside the application) adding and removing connectors. The main advantage of this trapped component model consists of the portability of the approach. JAsCo does not depend on a specialised virtual machine nor on some custom interfaces only available at certain systems. For example, a runtime environment optimised for embedded systems (JAsCoME) and an implementation of JAsCo for the .NET platform have been realised [VVSV03]. The drawback of a trapped approach is of course that a performance overhead is experienced for all these traps, even if no aspects are applied.

JAsCo employs a central *connector registry* that manages registered connectors and aspects at runtime. This connector registry serves as the main addressing point for all JAsCo entities and contains a database of connectors and instantiated aspects. Whenever a connector is loaded into or removed from the system at run-time, the connector registry is notified and its database of registered connectors and aspects is automatically updated. The left-hand side of Figure 4.7 illustrates a JAsCo-enabled class from which the joinpoint shadows are equipped with traps. As a result, whenever a joinpoint is triggered, its execution is deferred to the connector registry, which looks up all connectors that are registered for that particular joinpoint. The connector triggers the advices of the applicable aspects.

To improve the runtime performance of JAsCo, the *Jutta* and *HotSwap* systems were introduced in [VS04]. Jutta is an aspect-oriented just-in-time compiler that is able to generate a highly optimal code fragment containing the combined aspectual behaviour for a certain joinpoint. HotSwap is a runtime instrumentation framework that is able to insert traps at only those joinpoints where aspects are applied. With HotSwap, physical weaving, unweaving and reweaving of aspects at runtime at previously unadvised joinpoints becomes possible. When aspects are added or removed, the corresponding traps are added or removed. By employing Jutta and HotSwap, JAsCo is able to significantly improve on current state-of-the-art dynamic AOP approaches, like AspectWerkz and JBoss/AOP, performance-wise. JAsCo is even able to improve on the statically weaved language AspectJ in some cases [VS04]. All code examples in this dissertation are based on JAsCo version 0.8.6.

Two advanced extensions of JAsCo are also applied in this thesis: *stateful aspects* and *distributed aspects*. To avoid that this first introduction to JAsCo gets overloaded, we will introduce the technical details of these two extensions at the moment we first need them. Stateful aspects are therefore introduced in Chapter 5 in the context of stateful service conversations. Distributed aspects are discussed in Chapter 7.

4.6 Conclusions

In this chapter we have introduced the outlines of our approach: we propose to remove all service related code from a client application and put it in a separate module, called Web Services Management Layer (WSML). The WSML is responsible for taking care of all service concerns, including finding and integrating functionally compatible services. Additionally, the WSML takes care of selection, monitoring and other client-side management concerns for the client. As a result, the client code is not cluttered anymore with service specific code. The WSML can be deployed in various scenarios. The most obvious one is where the WSML acts as a service mediator or service broker, with an option of running the WSML as part of the client, or in a remote setup, where the WSML acts as a server for possibly multiple clients.

As the code, dealing with various service related code, ends up scattered and tangled with code dealing with other concerns, we suggest to use Aspect Oriented Programming (AOP) to cleanly modularise each concern in a separate entity, called *aspect*. AOP identifies scattered and tangled code as crosscutting concerns. By moving these concerns into aspects, the base code is freed from this additional code, and the aspects can be deployed obliviously at the places where needed. A pointcut language allows specifying the joinpoints, i.e. those places in the base code where an aspect should be triggered. To ensure that we can introduce and remove at runtime the aspects, which encapsulate the service related concerns, the need for a dynamic AOP approach was identified. The concrete dynamic AOP approach chosen in the context of this thesis is JAsCo. We have discussed the JAsCo language, which mainly introduces two concepts on top of Java: aspects and connectors. The aspects contain the crosscutting concern in a reusable manner, while connectors are used to deploy one or more aspects in a specific context. Finally, a brief discussion on the runtime infrastructure of JAsCo was made.

Now that we have introduced our main approach (WSML), the paradigm for its implementation (AOP) and a concrete technology (JAsCo), we can discuss each of the identified aspect categories in more detail. Aspects for service integration is the subject of the next chapter, aspects for service selection and monitoring are presented in chapter 6 and aspects for management concerns are considered in chapter 7.

Chapter 5

Dynamic Integration of Web Services

Abstract Just-in-time integration of Web services while avoiding hard-wiring service interfaces in clients, is the topic of this chapter. We discuss in detail how dynamic integration of Web services is realised by employing AOP-techniques in the WSML. Service types are introduced as a generic description of the service functionality required by the client application but without any reference to concrete services. Client requests on service types are redirected to the appropriate Web services or service compositions through service redirection aspects. These aspects modularise all service communication or composition details. We will illustrate how dynamic binding, asynchronous communication, stateful conversational messaging, and reactive compositions are realised using AOP.

5.1 Service Types

A primary goal of a dynamic redirection mechanism for Web services is avoiding that concrete service interfaces get hard-wired into the client. Therefore, service requests have to be formulated in the client application in an abstract way and it is up to the WSML to *redirect* these requests to a concrete service. To this end, we introduce *Service Types*. A service type is a generic description of the service functionality required by the client application but without any reference to concrete services. A service type can be considered a contract specified by the application towards the services and allows hiding the syntactical differences between semantically equivalent services. While still complying with the same service type, Web services based on RPC-based interaction might differ on the following levels:

- Web method names
- Synchronous / Asynchronous methods
- Parameter types & return types
- Semantics of parameters & return values
- Method call sequencing
- Etc.

By introducing Service Types, the heterogeneity of concrete services can be hidden. This approach differs from the concept of *tModels* (see section 3.3.2.1) where the services must have identical interfaces. Suppose the *HotelServiceType* for the case study introduced in section 3.1 specifies the following methods:

- `HotelList getHotels (BeginDate, EndDate, CityCode)`
- `Reservation bookHotel (BeginDate, EndDate, HotelCode)`

Client applications can invoke the methods of the *HotelServiceType* whenever they want to retrieve a list of hotels in a city, check for room availability and make a reservation. How invoking the service type methods will result in the invocation of one (or possibly multiple) Web services is of no interest to the client. In a black-box fashion, the service type encapsulates all service related behaviour and exposes a functional interface to the client. At the implementation level, this interface can be made available in two ways:

- **Local Object:** the WSML is part of the runtime environment of the client and any defined service type is provided as a class that can be instantiated. Requesting service functionality is possible by simply instantiating the service type class and invoking a method on it. One could look at the service type as an advanced service proxy, a local representative of all remote Web services offering the service type functionality.

- **Remote Web Service:** a service type is deployed as a Web service on the WSML, running as a server, and possibly hosting multiple clients. Requesting service functionality involves invoking the Web service representing the service type. This approach fits in the vision that everything realised by using or composing one or more Web services is exposed again as a value-added Web service.

These two approaches map to the distinction made in Figure 4.2 of chapter 4, where the WSML runs together with a local client, or as a dedicated server. Note that if the WSML is running in server setup, some of the issues the WSML is intended to address, may reoccur between the client and the WSML. For instance, an unreliable network connection between the client and the WSML server will result in issues that clearly cannot be addressed by the WSML. Therefore, the WSML server setup should preferably be used in a controlled environment such as an intranet, where the WSML can be used as the communication gateway between the local clients and the the Web services on the internet.

Now, as a service type can be instantiated, it should exhibit some behaviour. This is the *default behaviour* of the service type, and will be executed whenever no Web service or service composition associated with the service type is available or able to return a valid result. Possible default behaviours for a service type include:

- Throwing an exception
- Returning a default value
- Invoking a default Web service
- Etc.

Specifying a new service type boils down to specifying a set of methods with their appropriate parameters and return types as they would fit into the client, together with the default behaviour of each method. Clearly, the client must be adapted to this default behaviour: if for instance the service type returns a `NoServicesAvailableException` by default, then the client should know how to deal with this exception. Note that while a service type is intended to hide away all service related concerns, it might not accomplish this completely: if for instance no Web service is available to deal with a request, this will still have an impact on the client, i.e. by throwing the aforementioned exception.

Based on the requirements for a flexible service integration mechanism targeted at dynamic service environments as discussed in section 3.3, we will now continue with a discussion on how we realise dynamic binding of Web services using aspect-oriented techniques. First, we deal with basic Web services engaging in simple request-response communication in the next section. More advanced Web service communication patterns are discussed in subsequent sections: asynchronous communication in section 5.3, conversational messaging in section 5.4 and finally, service compositions in section 5.5. Related work and conclusions are presented respectively in sections 5.6 and 5.7.

5.2 RPC-based Web Services

5.2.1 Mappings

Suppose two semantically equivalent services *HotelServiceA* and *HotelServiceB* offer the functionality described in the *HotelServicetype*.

HotelServiceA provides the methods:

- `HotelList listHotels (CityCode, BeginDate, endDate)`
- `Reservation bookHotel (Date, Nights, HotelCode)`

HotelServiceB provides the methods:

- `HotelList getHotels (BeginDate, EndDate, CityName)`
- `Reservation bookHotel (BeginDate, EndDate, HotelCode)`

Another service *CityCodeLookupService* offers the functionality for converting a given city name into a city code and vice versa.

- `CityCode getCityCode (CityName)`
- `CityName getCityName (CityCode)`

As one can see, the operations do differ as they have different names or vary in the parameters they accept. Figure 5.1 shows how the operation `getHotels` of the *HotelServiceType* can be mapped onto two concrete Web service interactions. In the first example, both the name and the order of the arguments of the service type differ from the ones specified in *HotelServiceA*. More complex mappings may require additional pre-processing of the parameters, e.g. to convert a begin date and an end date provided by the `bookHotel` service type method into a begin date and a number of nights as required by *HotelServiceA*. Additionally, also post-processing of the returned result may be required, i.e. to convert the result provided by the Web service into a result expected by the service type.

In the second mapping a simple service composition is needed, as *HotelServiceB* requires the city code to be converted into a city name first. This additional method is provided by the third service *CityCodeLookupService.* We leave the discussion of this composition to section 5.5.

5.2.2 Dynamic Binding

To enable the redirection of requests on the *HotelServiceType* to concrete hotel Web services, *Service Redirection Aspects* are introduced. They specify *what* behaviour to execute, in this case which concrete methods to invoke on a specific Web service, *when* some service functionality is requested in the application.This behaviour is specified in aspect hooks;

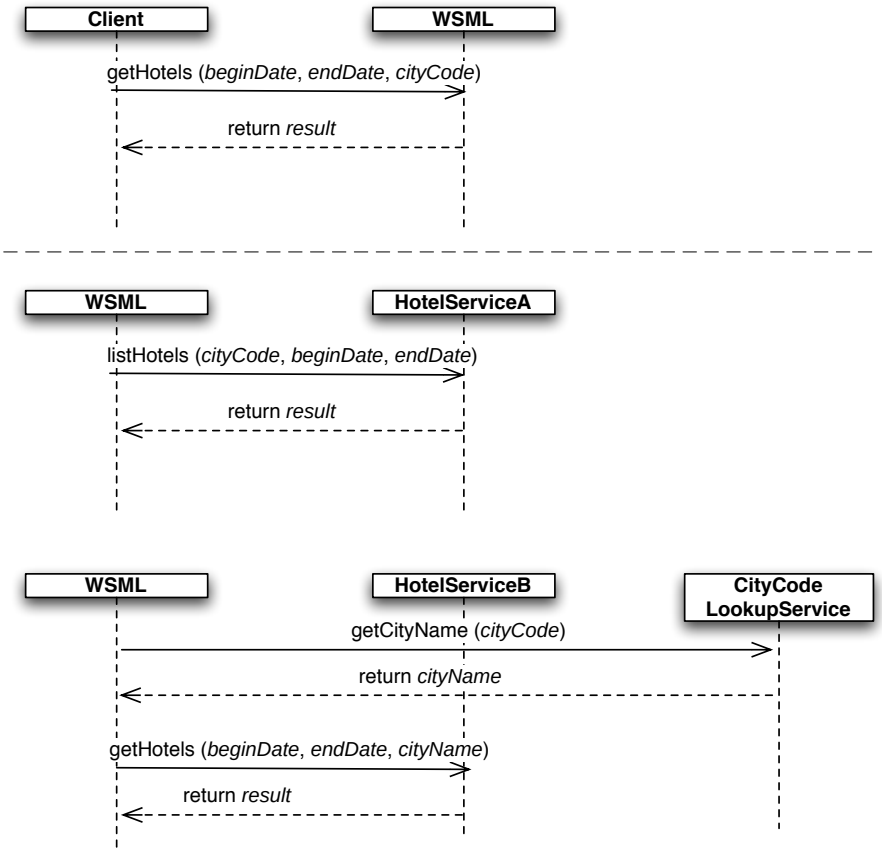

Figure 5.1: Two Possible Mappings for the Hotel Service Type

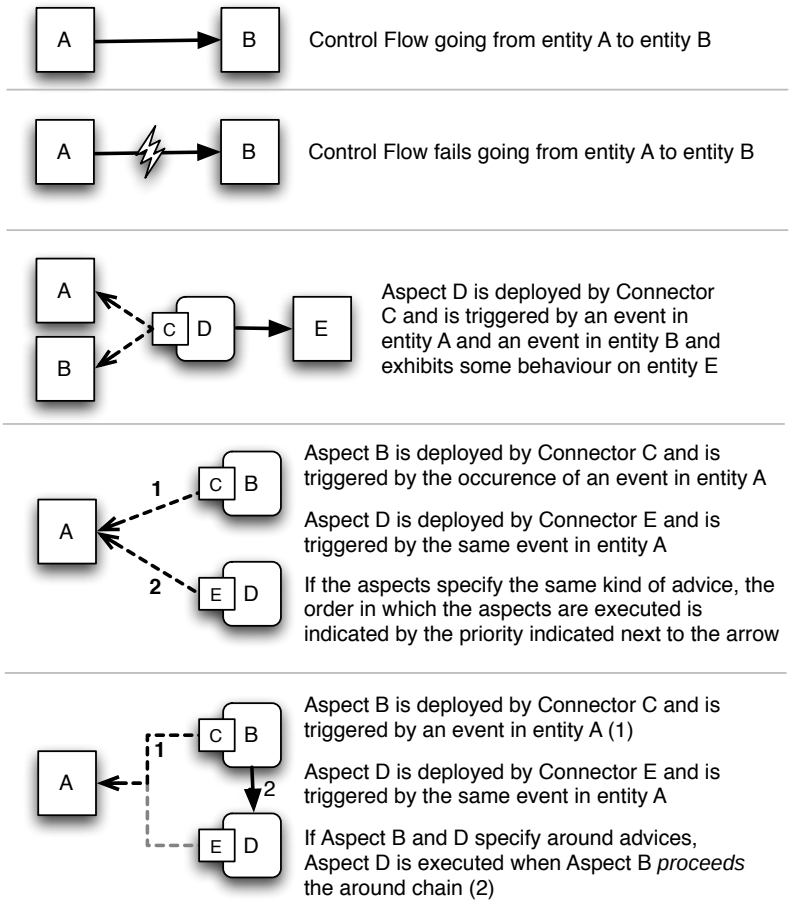

Figure 5.2: Listings for the Figures

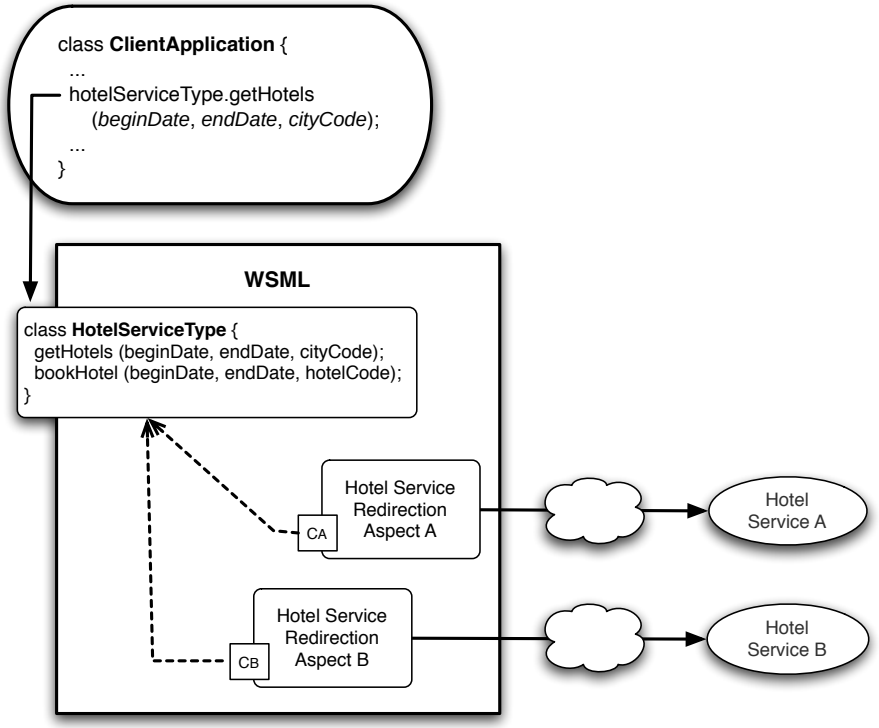

Figure 5.3: Dynamic Service Binding using Redirection Aspects

there is a hook for each request the Web service can fulfil. As such, a service redirection aspect encapsulates all *communication details* for a specific Web service. In case of our example, there will be one redirection aspect for *HotelServiceA* and one for *HotelServiceB*. JAsCo aspects are specified without referencing concrete context information: a connector is used to provide this context information and to specify exactly *where* advice behaviour of the aspect hooks has to be executed. In our case, a service redirection aspect has to be triggered each time the methods `getHotels` and `bookHotel` are called in the application. This is depicted in Figure 5.3. The full lines indicate the control flow and the dashed lines indicate the hooking of an aspect on one or more joinpoints (in this case on all methods of the `HotelServiceType` class). The complete list of notations is included in Figure 5.2.

Code fragment 5.1 shows an example implementation of the *HotelServiceA redirection aspect* in JAsCo. The aspect contains two hooks: one for the `getHotels` request (lines 4 to 17) and one for the `bookHotel` request (lines 19 to 33). Each hook defines, besides a constructor, an around advice which is going to invoke the actual web method of the corresponding Web service using a static proxy (line 11 and lines 27). If the invocation of the Web service fails for whatever reason, exception handling is specified in the catch blocks. We come back to this in the next section. Note that the advice can contain additional pre-processing code (e.g. in line 26 a parameter required by the Web service is calculated) or post-processing code (e.g. to convert the result returned by a Web service to a format expected by the client). In case multiple advices require the same *glue code*, it is possible to specify a specific method in the aspect for this purpose to avoid code duplication.

```
1  Class HotelServiceARedirection {
2      Proxy proxy = new Proxy (http://www.hotel.com/service.wsdl);
3      ...
4      hook GetHotelsHook {
5          GetHotelsHook(method (Date beginDate, Date endDate, String cityCode)) {
6              execution(method);
7          }
8
9          around() {
10             try {
11                 return proxy.getHotels(cityCode, beginDate, endDate);
12             }
13             catch (Exception e) {
14                 return proceed();
15             }
16         }
17     }
18
19     hook BookHotelHook {
20         BookHotelHook(method (Date beginDate, Date endDate, String hotelCode)) {
21             execution(method);
22         }
23
24         around() {
25             try {
26                 int nights = endDate - beginDate;
27                 return proxy.bookHotel(cityCode, beginDate, nights);
28             }
29             catch (Exception e) {
30                 return proceed();
31             }
32         }
33     }
34 }
```

Code fragment 5.1: Service Redirection Aspect for Hotel Service A

In our example, a static proxy is used for the service invocations for three reasons: 1) Performance: static proxies perform better than dynamic proxies or DII as it is not necessary to analyse the WSDL-file or generate client-side code at runtime. 2) Simplicity: using static proxies results in clean and less complex code, enhancing readability and maintainability. 3) Static typing: as the type of the proxy is static, it is possible to do service method look-up at compile-time, unlike with DII. This facilitates implementing the aspect and reduces runtime errors. Also, whenever the WSDL-file of a service changes, the static proxy can be regenerated, and any mismatches that occur due to changes in the interfaces can be more easily detected and resolved at compile time, before redeploying the aspect.

Now we need a connector to deploy the service redirection aspect for the corresponding service type(s). This way, JAsCo connectors encapsulate all *deployment details*. Code fragment 5.2 illustrates the *HotelServiceAconnector*, which specifies the mapping between the requests of *HotelServiceType* and the aspect of Code fragment 5.1. This connector instantiates the hooks defined in the aspect, which results in the execution of the around

```
1  static connector HotelServiceAconnector {
2      HotelServiceARedirection.GetHotelsHook getHotelsHook =
3          new HotelServiceARedirection.GetHotelsHook
4              (HotelList HotelServiceType.getHotels (Date, Date, String));
5      HotelServiceARedirection.BookHotelHook bookHotelHook=
6          new HotelServiceARedirection.BookHotelHook
7              (Reservation HotelServiceType.bookHotel (Date, Date, String));
8      getHotelsHook.around(); //optional
9      bookHotelHook.around(); //optional
10 }
```

Code fragment 5.2: Connector for Hotel Service A

behaviour specified in those hooks, as soon as a service type request is performed.

As one service redirection aspect encapsulates all communication details for one Web service, it is easy to accommodate to Web service versioning. If an integrated service changes for instance its interface or the message format it expects, then these changes only need to be reflected in the corresponding aspect. Of course, the service can only change within the boundaries specified by the service type. The service can no longer be integrated in the client if it does not deliver the required functionality anymore. Determining this "compatibility" is a complex process, as it requires analysing the semantics of the service type and the Web service. This is discussed in chapter 8. The other way around is also possible: a Web service may provide the functionality needed by the service type, but might need additional information to do so. For instance, to book a hotel room, a hotel service might require the birth date of the customer. It is very well possible that this information is somewhere available in the client system, but if it is not passed along as a parameter of a service type request, it cannot be made available to the Web service. Using traditional OO, this can only be fixed by storing the data explicitly somewhere to make it available at the moment of the service invocation, or by passing it along as an extra parameter of the service type, which will undoubtedly involve changes in multiple places in the client. Using AOP, the service redirection aspect can be used to capture the unavailable data in the client in an oblivious way without having to adapt the client code, and pass this context along to where it is needed. Capturing unavailable data that otherwise requires crosscutting changes in an application is an approach also suggested in [CDS03] for the purpose of gathering business rules related data.

5.2.3 Hot-Swapping

Hot-swapping means the replacement of a software program or a part of a program while the whole software system remains in operation [FGWP01]. In the Web service context, this means that one service is replaced by another without having an impact on the client. Problems inherent to the process of hot-swapping include the *Referential Transparency Problem*, indicating that all objects receiving a handle to an object that is about to be hot-swapped, need to be updated with a reference to the new object; the *State Transfer Problem* stating that the state of an module that is replaced, needs to be transferred to the new module for consistency reasons; and the *Mutual Referential Problem*, indicating that if

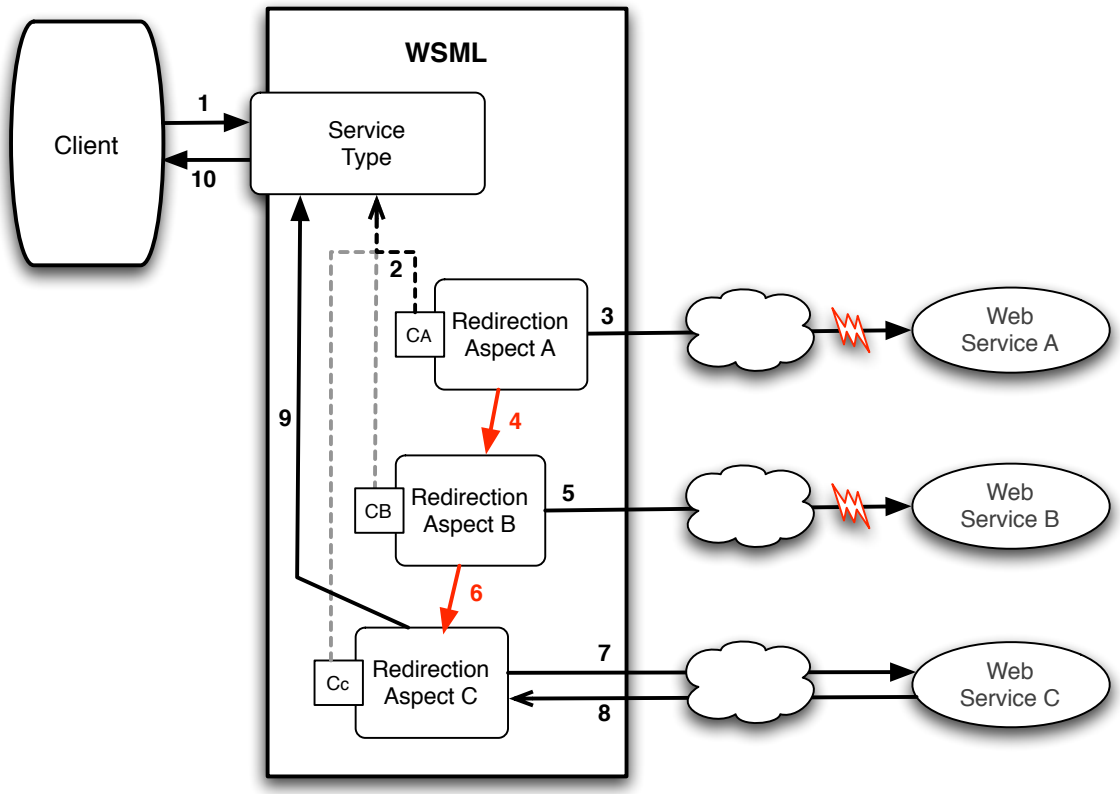

Figure 5.4: Hot-swapping Services through Around Advice Chaining

a module depends on other modules, multiple modules must be swapped in one transaction and the order of swapping may be important.

In our running example, whenever the client application invokes the *HotelServiceType,* either *HotelServiceA* or *HotelServiceB* should be addressed. By default, the "first" service redirection aspect is triggered, and its corresponding Web service is invoked. If the invocation is successful, the result is returned to the client method that invoked the *HotelServiceType*. If the invocation fails (e.g. because the network or the Web service is down), the second redirection aspect is triggered and as a result, the next Web service is invoked. In case of consecutive failures, this process is repeated until all Web services are tried. If the last one is tried and fails too, the default behaviour of the service type, as described in section 5.1, is executed. This whole process, as depicted in Figure 5.4, happens in a transparent way for the client application and is straightforwardly realised by an AOP technique called *Around Advice Chaining.* Because there are multiple service redirection aspects present, there are multiple around behaviour methods employed on the same joinpoint. Around advice chaining means that the firstly specified around method is executed first. If this around method invokes the original behaviour, the second around behaviour is executed instead of the original behaviour and so on. In Code fragment 5.1 this is realised in lines 14 and 30 by using the proceed keyword, which indicates to proceed with the around advice chain and thus, to continue with the next service redirection aspect.

The JAsCo connectors realise a dynamic binding between the client application and the Web services. If a Web service is temporarily unavailable, a service redirection aspect can be taken out of the chain by disabling the corresponding connector. As such, it is very easy to avoid unnecessary calls to unreachable Web services. This means that the proposed hot-swapping mechanism can pro-actively select the most optimal services, and therefore minimise the impact of a switch on the application performance. Furthermore, as JAsCo allows for the runtime deployment of aspect beans and connectors, it becomes possible to dynamically alter which services are integrated. By creating a new redirection aspect and corresponding connector at runtime, a new Web service can be included in the around chain. As such, our approach can easily cope with the volatility of a Web service environment while leaving the client application untouched. Additionally, it is also possible to realise client initiated hot-swapping with this approach. By providing a mechanism that can be used by the client to notify that a hot-swap is desired, the appropriate connectors of the redirection aspects can be enabled or disabled. It is also possible to dynamically alter the order of the around chain in order to put more appropriate services at the beginning of the chain. This is the subject of chapter 6.

In our solution, the referential transparency problem is circumvented through the use of service types, as no object in the client refers directly to a Web service. The mutual referential problem is not applicable as Web services are independent autonomous modules, and we assume no existing dependencies. The state transfer problem can however be an issue. Up until now we assumed simple request-response communications between the WSML and the Web services: a hot-swap can take place at any time. If more advanced business processes need to be executed between services that keep state, switching to another service becomes more difficult. It is clear that in this case, a hot-swap cannot take place at any moment and that compensation actions might be required to rollback the state of the service and maybe even the state of the application. This is discussed in section 5.4.

5.2.4 Changeable endpoint references

You may already have noted that the redirection aspect shown in Code fragment 5.1 contains a static proxy in line 2 to invoke the Web service. Static proxies already support changeable endpoint references (see section 3.3.2.1) by using a variable or a configuration file. If for instance in line 2 a variable was used instead of a hard-coded reference, then it is possible to assign a new value for the endpoint reference whenever the service is relocated. At deployment time, the endpoint reference can be passed from the connector to the aspect through an explicit constructor. The advantage is that the aspect does not need to be updated and recompiled. To detect service re-locations at runtime, a dedicated advice can be triggered whenever the service is about to be relocated, and update the reference in the aspect.

Code fragment 5.3 contains an updated version of the Service Redirection Aspect. A dedicated `setProxy` method (lines 4 to 6) updates the endpoint address of the proxy. Additionally, a new `endpointChangedHook` (lines 15 to 19) is added. This hook is triggered whenever a new endpoint needs to be set. The following Code fragment shows the connector for this aspect.

```
1  Class HotelServiceARedirection {
2      Proxy proxy;
3
4      void setProxy (String newEndpoint) {
5          proxy = new Proxy (endpoint);
6      }
7
8      public HotelServiceA (String newEndpoint) {
9          setProxy (newEndpoint);
10     }
11
12     hook getHotelsHook {...}
13
14     hook bookHotelHook {...}
15
16     hook endpointChangedHook {
17
18         after () {
19             setProxy (newEndpoint);
20         }
21     }
22 }
```

Code fragment 5.3: Service Redirection Aspect with Changeable Endpoint References

```
1  static connector HotelServiceAconnector {
2      HotelServiceARedirection serviceA =
3          new HotelServiceARedirection (http://www.hotel.com/Service.wsdl);
4      HotelServiceARedirection.getHotelsHook getHotelsHook =
5          new HotelServiceARedirection.getHotelsHook
6              (HotelList HotelServiceType.getHotels (Date, Date, String));
7      HotelServiceARedirection.bookHotelHook bookHotelHook =
8          new HotelServiceARedirection.bookHotelHook
9              (Reservation HotelServiceType.bookHotel (Date, Date, String));
10     HotelServiceARedirection.endpointChangedHook endpointChangedHook =
11         new HotelServiceARedirection.endpointChangedHook
12             (wsml.WebService.setProxy (String));
13 }
```

Code fragment 5.4: Connector for Hotel Service A with Endpoint Reference Setting

In lines 2 and 3 of Code fragment 5.4 the explicit constructor of the *HotelServiceA* aspect is called with the URI of the proxy as a parameter. The two first hooks are instantiated and specify the respective *HotelServiceType* methods as joinpoints, similar to what we did in Code fragment 5.2. The third hook (lines 10 to 12), which contains the advice to set the endpoint reference, defines as a joinpoint the moment the endpoint reference changes. The code example assumes the WSML is notified of this change and hooks on this moment. This could be realised through the WS-Eventing standard [BCC+04c] that makes it possible for clients to subscribe to event messages coming from a service. If the Web service does not offer such functionality, periodical polling can be done to detect changes (e.g. if the URI contains a forward reference to the new location of the service). Another approach is employing a distributed joinpoint model, where the aspect directly hooks on joinpoints specified remotely on the Web service. This advanced AOP approach is further discussed in chapter 7.

5.2.5 Exception Handling

The proposed service integration mechanism is very straightforward, easy to understand and realise. However, the service redirection aspects as proposed in section 5.2.2 actually encapsulate two separate concerns: firstly, they encapsulate all communication details for a Web service, but secondly, they also contain the implementation dealing with service invocation failures. In the Code fragment 5.1, the catch blocks (lines 13 to 15 and lines 28 to 30) specify that whenever the service invocations generates an exception, the next service is invoked using the proceed() method. While this may be valid behaviour in many cases, it might be too restrictive in others. A more flexible approach is to split up the code in service redirection aspects, which only contain code to invoke a service, and in *fallback aspects,* which specify what to do in case of an invocation failure. This is achieved by changing Code fragment 5.1 so that an exception is thrown whenever the service invocation fails. Without any fallback aspect into place, this exception will be returned to the client. Any next call from the client application on the service type will continue to be redirected to the same service, even if it fails.

Deploying a fallback aspect to intercept the exception can change this behaviour. Figure 5.5 displays the case where a fallback aspect catches any service invocation exception and then continues the redirection chain, thus realising hot-swapping functionality. In the picture, *WebServiceA* fails to return a valid result and the fallback aspect continues the chain by proceeding to *WebServiceB* that does return a result. If one wants to capture possible exceptions that are thrown in `RedirectionAspectB`, the fallback aspect should be applied there too. An alternative is to deploy the fallback aspect on the service type itself. As the service redirection aspects specify advices *around* this service type, any exception occurring in the redirection aspects is captured. As a concrete example, consider the next Code fragment showing a fallback aspect for hot-swapping that limits the redirection chain to a fixed length, and thus limiting the number of service invocation retries. Lines 11 to 15 show an *around throwing advice* that is triggered as soon as a `ServiceInvocationException` occurs in the joinpoint. If the number of retries is lower than the `maxInvocations` the chain is continued (line 13), otherwise the exception is simply re-thrown (line 14).

When a Web service invocation fails, the hot-swapping aspect continues the chain until

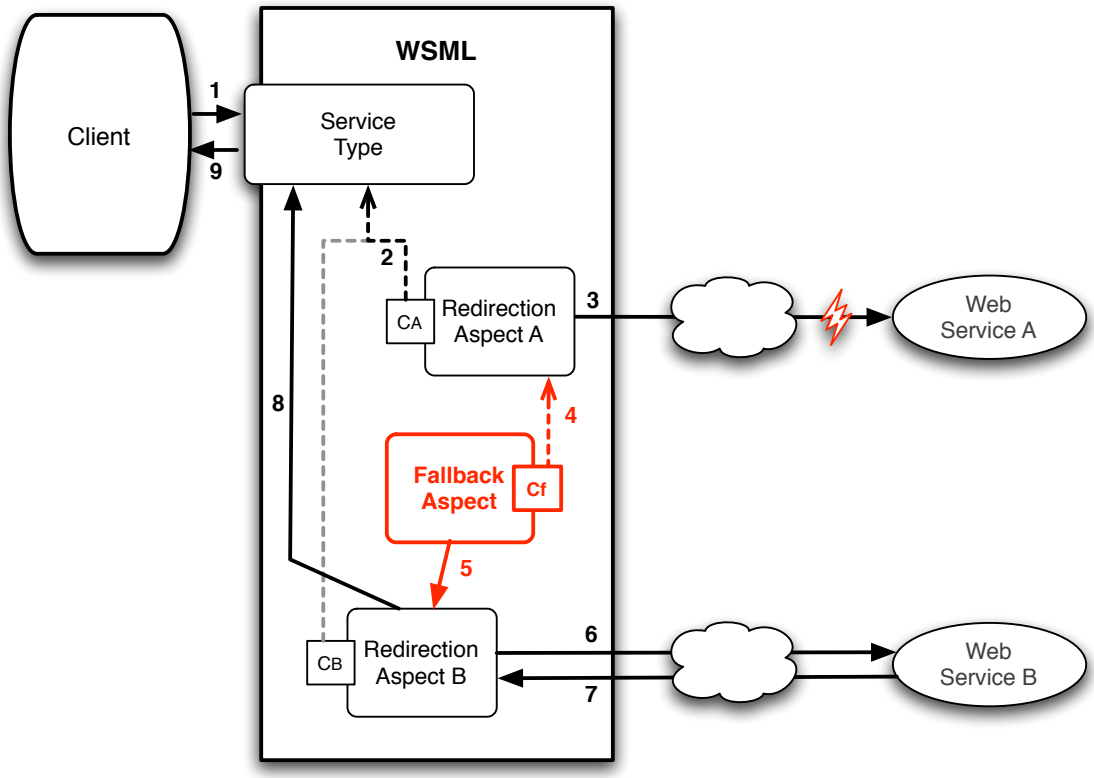

Figure 5.5: Exception Handling of Services using a Fallback Aspect

```
1  class HotSwappingAspect {
2      private static int maxInvocations = 3;
3
4      hook InvocationHook {
5          int invocation = 0;
6
7          InvocationHook(method(..args)){
8              execution(method);
9          }
10
11         around throwing(ServiceInvocationException e) {
12             if (invocations++ < maxInvocations)
13                 return proceed();
14             else throw e;
15         }
16     }
17 }
```

Code fragment 5.5: Fallback Aspect for Hot-swapping

a working Web service is found. The next time the service type is invoked, the chain will be triggered again in the same fashion. This means that all the services that resulted in an exception will be retried first. This behaviour may be undesired, especially as invocations on unavailable Web services result in an exception after a, possibly lengthy, time-out period has gone by. A straightforward solution is to remove the corresponding service redirection aspects temporarily from the chain or to move them to the back of the chain. With JAsCo, this is achieved by simply disabling or reordering the connectors. The implementation of the fallback aspect can be adjusted to realise this through a dedicated connector API.

Examples of other possible fallback behaviour that can be nicely modularised in aspects include:

- **Invocation Retrying:** instead of continuing with the next available Web service, a fallback aspect could be deployed to retry the same Web service a number of times. This aspect would look identical to the hot-swapping aspect of Code fragment 5.5, except in line 12, it should state to invoke the joinpoint again. In JAsCo this is possible as follows:
  textttthisjoinpoint.invokeAgain().
- **Service Monitoring:** as a service invocation failure might imply something is wrong with the Web service, a monitoring aspect could temporarily remove that service from the chain and start to periodically check the service for availability. Monitoring is further discussed in Chapter 6.
- **Administrator Alert:** an aspect can be deployed to raise an alert as soon something goes wrong. For instance, an email message could be sent to an administrator to indicate the problem.

Fallback aspects are a first example of management concerns that can be straightforwardly encapsulated in aspects. Chapter 7 will illustrate more cases, including client-side management issues such as service billing, broadcasting and caching.

5.2.6 Conditional Service Binding

In specific scenarios not all available Web services can deal with all kinds of requests: each service may be specialised in a subset of the possible client requests. For instance, *HotelServiceA* deals only with hotels in Europe while *HotelServiceB* handles hotels in Africa. In this case, the parameter values of the client requests (e.g. the `CityCode` or `CityName` parameters in the methods of the *HotelServiceType*) must be taken into account by the service binding mechanism. This means that we need a more fine-grained control over when a hook in our service redirection aspect should be triggered. Otherwise said: if multiple aspects are present on a joinpoint, a condition evaluation is needed to determine which aspect advice(s) should be triggered.

In AspectJ, a condition can be added to the pointcut specification using an *if* pointcut designator. In JAsCo an condition check can be added to a hook. Code fragment 5.6 shows an extended version of the `getHotelsHook` of Code fragment 5.1. In lines 8 to 10 an

```
1  ...
2  hook getHotelsHook {
3      getHotelsHook(method (Date beginDate, Date endDate, String cityCode)) {
4          execution(method);
5      }
6
7      isApplicable() {
8          return isValidDestination(cityCode);
9      }
10
11     around() {...}
12 }
```

Code fragment 5.6: Conditional Binding with a Service Redirection Aspect

`isApplicable` method is added containing a condition check on the hotel location. This condition is checked before the advice in the corresponding hook is triggered. In the method body it is calculated whether the current parameter(s) are valid input for the Web service. A method provided in the aspect can do this evaluation, or it may be left to a Web service method. At first it might seem strange to invoke a Web service to check whether that service can deal with the provided input. However, taking into account that a Web service can trigger a long-running business process, or can require a payment, it is worth considering to do pre-validation of the input.

Using the `isApplicable` method filters the Web services at two levels: first, services are pro-actively filtered out by enabling or disabling connectors as explained before. Second, the conditions of the remaining services are checked against the current request. Only the services that are left over are candidates to be invoked, as only their corresponding aspects are considered. Otherwise said: only service redirection aspects with enabled connectors and conditions that evaluate to true remain in the redirection chain.

5.2.7 Multiple Services Binding

It is also possible to realise multiple services binding with service redirection aspects. Remember from chapter 3 that this involves the binding of multiple complementary Web services to the client at once. In our example *HotelServiceType* method `getHotels` can be redirected to *HotelServiceA*, a Web service providing fast results and method `bookHotel` can be redirected to *HotelServiceB*, being the cheaper Web service. Obviously, this kind of binding is achieved at the connector level of our approach, as the connectors bind the client with the redirection aspects that invoke the Web services. Up until now we used one connector to link a whole redirection aspect to a service type. However, it is also possible to use a different connector for each service type request. This way it becomes possible to use several parts of multiple services to provide the functionality of one Service Type at a given time, thus enabling multiple services binding. The *HotelServiceType* example is depicted in Figure 5.6: two connectors per Service Redirection Aspect are used to bind the two hotel Web services to the client. Which service is invoked for either request depends on which connectors are enabled or disabled and the order of the connectors. Note that with multiple services binding there are x sets of connectors to be maintained and ordered: Ca1...Cy1 to

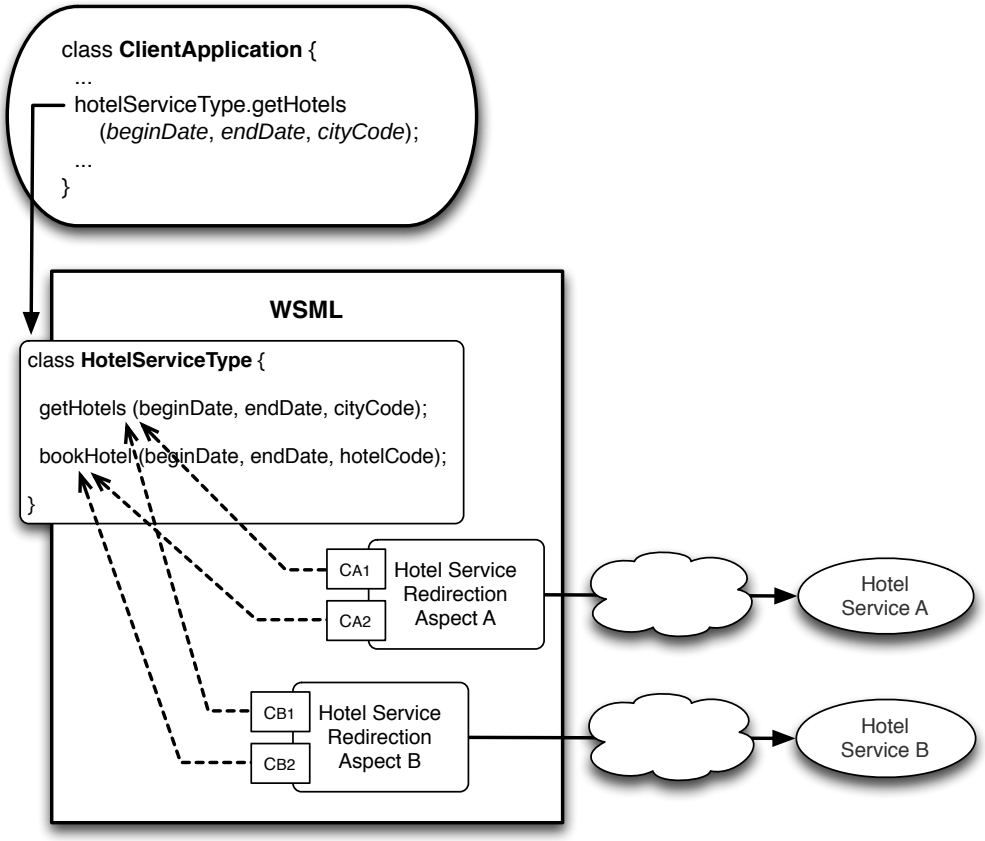

Figure 5.6: Multiple Services Binding in the WMSL

Cax. . . Cyx with x equal to the number of service type methods and y equal to the number of Web services linked to the service type (in our example: x=2 and y=2). [1]

5.2.8 Summary

Before moving on to the more advanced kinds of Web service interactions, we briefly summarise the mechanism as described up until now. Figure 5.7 illustrates how a service redirection aspect looks like. There is a Hook for each service type method. This hook consists of a constructor, an optional applicability method and an around advice. As soon as a service type is invoked by the client, the connector triggers the applicability condition in case of conditional binding to check if the Web service is suitable for the request at hand. If so, the around advice is executed. An optional pre-processing step consists of glue code for the parameters. It maps the parameters provided by the client to the parameters expected by the Web service. Next, the Web service is invoked, and optionally the result is mapped back in a post-processing step. Finally, the result is passed back to the client. In case of an error in this process, an exception is thrown. If this exception is not caught in a fallback aspect (for example to retry the process or proceed to the next service), the exception is passed back to the client. If no service is available or the last service is tried and has failed, the default behaviour as specified in the service type is executed.

Remember from section 2.3 that more advanced Web services might prefer to communicate by exchanging XML-documents with their clients. For instance, booking a hotel would not involve invoking a `bookHotel` method but rather sending a `HotelReservation` document. As stated in Chapter 3, while reviewing proxies, Web services using document-style interaction abstract away the underlying service implementation and better withstand changes to that implementation [Burn03]. The proxy approach is also suited for this kind of communication. For example, a proxy for a document-based hotel Web service can have a method
textttprocess (ReservationDocument) and a Java wrapper for the `ReservationDocument` can be generated at client-side. So, in the redirection aspect, the document can be prepared as required and then sent to the Web service. Any result can be translated back into an element of the client language.

5.3 Asynchronous Web Services

5.3.1 Introduction

Up until know only synchronous Web service invocations are discussed. In synchronous invocation, the client sends a request to the service and then halts operation while waiting for a service response. However, synchronous invocation is not suitable when the service must perform a very complex computation or when the service will respond after an undetermined

[1] The current implementation of the WSML, version 0.5.1, supports both regular dynamic binding and multiple services binding. This means that for a service type with x methods, x+1 connectors maintain the binding with one Web service: 1 connector for a regular dynamic binding and x for the multiple services binding. In case of y Web services there are y(x+1) connectors in x+1 sets.

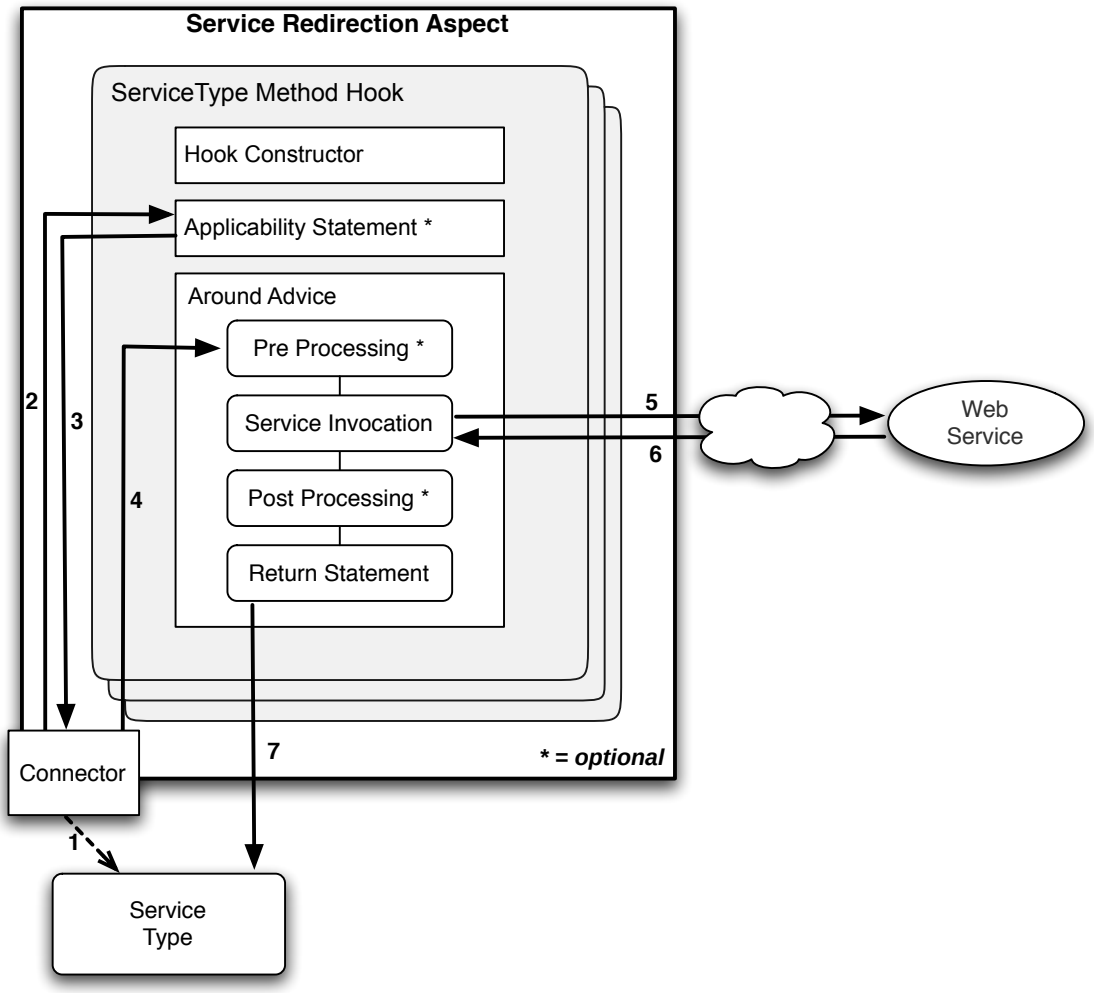

Figure 5.7: The Different Parts of a Service Redirection Aspect

delay, for instance because it started a long-running business process, possibly involving human action. The client would wait until it finally received the response, blocking other processes, wasting system resources and keeping an unused transport channel open. In these cases, asynchronous invocation is more appropriate. In asynchronous invocation, the client does not wait for the service response. It can continue independent computation until the response is received. This way, the client becomes event driven and uses its resources more efficiently by processing responses when they are ready.

Typically, the sender will send its service request to a message handler who will put it in an outgoing message queue and return control back to the client so it can continue further processing. Each message will get an ID to map it to the result returned by the service. A pool of threads manages this message queue. There are two possible mechanisms for getting the results of asynchronous invocation: polling and call-backs. The *call-back mechanism* fits better the object orientation paradigm, using client listeners, referred to as call-backs. These call-backs are notified by the service side when the results to an asynchronous invocation are ready. With the *polling mechanism* the client periodically requests the service for a response.

A common way to do asynchronous invocations based on call-backs is based on begin and end methods (see also section 3.3.2.1). The begin method starts the asynchronous communication with the Web service method while the end method is used to get the result of the asynchronous call. The begin method will return an `AsynchronousConversation` object, which acts as a handle for an asynchronous invocation once it has started. Using this interface, the client can obtain information about the invocation state or register the callback that will get called once the response comes asynchronously from the service. The class implementing the callback needs to provide some methods that will be called in specific events (i.e. when the result is received, when the operation times out, etc.). Code fragment 5.7 shows the simplified code of a client application invoking the `bookHotel` method of *HotelWebServiceA* asynchronously, using Systinet Server for Java [Sys05]. The asynchronous invocation is started (line 6) and a callback is set (line 7). In this case, the same class is registered to be called back when the invocation returns (lines 11 to 15) or when the invocation times out (lines 18 to 21).

5.3.2 Asynchronous Service Redirection Aspects

Our dynamic service binding mechanism can be adjusted to cope with asynchronous Web service invocations. Firstly, the communication between the client and the service type needs to become asynchronous to allow the client to continue processing while waiting for a service response. And secondly, the client needs to know how and when it should collect the service result. For this purpose, the service type must provide a begin and end method for each request. Code fragment 5.8 shows an *Asynchronous Service Redirection Aspect* with a hooks for both the begin and end methods. The around advice for the begin method (lines 10 to 20) invokes the Web service (line 12) and additionally a client object is registered for the callback (line 13). Whenever the result becomes available, the callback object is notified. The callback object can retrieve the result by invoking the end method on the service type. This causes the around behaviour of the end hook to be executed (lines 32 to 35) where the result is fetched from the Web service.

```
1  public class Client extends GenericAsyncCallback {
2
3      // asynchronously call service
4      private void doAsyncInvocation() throws Exception {
5          HotelServiceA proxy = (HotelServiceA) Registry.lookup(WSDL_URL, HotelService.class);
6          AsynchronousConversation async = proxy.beginBookHotel(args);
7          async.setCallback(this);
8      }
9
10     // implementation of the callback
11     public void onResponse(AsyncConversation async) {
12         Reservation response = proxy.endBookHotel(async);
13         async.finish();
14         System.out.println("Response received: " + response.toString());
15     }
16
17     // implementation of the time out
18     public void onTimeout (AsyncConversation async) {
19         System.out.println(method invocation timed out);
20         async.finish();
21     }
22 }
```

Code fragment 5.7: Asynchronous Invocation of a Web Service

When the end method is invoked, a mechanism is required to keep track of which service redirection aspect was triggered for the service invocation. Assume a hot-swap takes place between the invocation of the begin method and the end method of the service type. This would lead to an erroneous redirection of the end method at the moment the result becomes available. This can be easily avoided by using the `isApplicable()` construct of JAsCo. It can be used to test if the `AsynchronousConversation` object belongs to the corresponding redirection aspect (lines 28-30).

Note that our approach can also be deployed to use asynchronous services for synchronous service types and vice versa: an synchronous Web service could provide the functionality for an asynchronous service type (the result would be available immediately) or an asynchronous Web service could be used with an synchronous service type (the service type would halt the client until it receives the asynchronously fetched result).

5.4 Conversational Web Services

5.4.1 Introduction

Web services are generally considered to be stateless: there is no shared context between the client and a Web service, and a Web service can deal with each individual request of a client without needing any state information. However, in practice Web services can often maintain state or resources on behalf of an individual service client or within a particular business context across multiple interactions. For example, the *FlightService* in our case study (see section 3.1) implements an airline reservation system and needs to be able to

```
1  Class AsyncHotelServiceARedirection {
2      Proxy proxy; //client proxy of HotelServiceA
3      Set conversations;
4
5      hook BeginBookHotelHook {
6          BeginBookHotelHook (method(args, Callback subj)) {
7              execution(method);
8      }
9
10         around() {
11             try {
12                 Async async = return proxy.beginBookHotel(args);
13                 async.setCallback (subj);
14                 conversations.add (async);
15                 return async;
16             }
17             catch (Exception e) {
18                 throw new (ServiceFailureException (HotelServiceA, e);
19             }
20         }
21     }
22
23     hook EndBookHotelHook {
24         EndBookHotelHook(method(Async async)) {
25             execution(method);
26         }
27
28         isApplicable() {
29             return conversations.contains (async);
30         }
31
32         around() {
33             conversations.remove (async);
34             return proxy.endBookHotel(async);
35         }
36     }
37 }
```

Code fragment 5.8: Asynchronous Web Service Redirection Aspect

maintain the status of flight reservations across multiple independent interactions of each customer and therefore needs to identify which state (i.e. which reservation) needs to be queried or updated. There is a discussion in the Web services community whether Web services should be stateful or stateless. In [Vogels03] the view that Web services have no notion of state is given and that interaction with Web services is stateless. In [PWWR03], contextualisation is proposed as a way of modelling stateful interactions with stateful services. A consensus is that Web services themselves should remain stateless while possibly acting upon stateful resources and manipulating them based on the messages they send and receive. The responsibility for the management of the state is delegated to another component such as a database or a file system. As pointed out in [FFG04], statelessness in the implementation of the service itself tends to enhance reliability and scalability: a stateless Web service can be restarted following failure without concern for its history of prior interactions, and new copies of a stateless Web service can be created and destroyed in response to changing load. Thus, statelessness is generally viewed as good engineering practice for Web service implementations.

Web services that interact with stateful resources will need to provide explicit state information in the messages exchanged with their clients. Typically, correlation information is added explicitly in the messages for this purpose. For instance, a Reservation ID could be passed along explicitly in the parameters between the client and the *FlightService*. Some consider this correlation data at the application level burdensome in the client application and therefore, techniques for implicit correlation and session creation have been proposed. These approaches include using features of the underlying communication protocol (e.g. cookies with HTTP). All these approaches have not led to an agreed upon convention or standard that promotes interoperability between clients, stateful Web services and other stateful resources.

One suggested approach is the WS-Resource Framework (WSRF) [WSRF05] to model stateful resources in Web services. The state is kept in a separate entity, called a *resource*, which stores all state information. Each resource has a unique identifier, so clients can engage in stateful interaction with a Web service by instructing it to use a particular resource. A pairing of a Web service with a resource is called a WS-Resource. The WS-Resource is described by a WSDL-file so that clients can query and manipulate the resource through message exchanges. WSRF uses the *Implied Resource Pattern* [FFG04], which relies on WS-Addressing to reduce the client burden while not limiting to a specific transport protocol or requiring unique support in the Web services client runtime. WS-addressing [BCC+04a] is a more powerful mechanism for addressing entities managed by a service than what is possible using URIs. A WS-Addressing endpoint reference can contain additional reference properties, in this case a stateful resource identifier that allows for the unambiguous identification of a stateful resource.

WSRF-based Web services follow the Factory Pattern: clients must invoke a dedicated method to receive a WS-Addressing endpoint reference. This method is referred to as the WS-Resource factory. The returned endpoint reference points to a new WS-Resource, which has been composed from a newly created stateful resource and its associated Web service. The client must recognise that the endpoint reference is a WS-Resource-qualified endpoint reference. This reference contains a stateful resource identifier in its reference properties component, and the client must redirect all communication with the service using that

endpoint reference and include the stateful resource identifier in the SOAP header of all messages it exchanges with the service.

In this context, the importance of the reliability of the network should be underlined. One of the fallacies of [Deu91] states the network is assumed to be reliable, something that does not hold as Web services are most often accessed over unreliable, stateless protocols such as HTTP. When a client invocation that changes the state of a Web service resource fails, the client cannot make assumptions anymore on the state of the Web service resource, as the service may have processed the request before the failure, or the request could have been lost before reaching the service. It is important to employ a reliability protocol such as WS-Reliable Messaging (WSRM) [BBC+05] or WS-Atomic Transactions (WSAT) [CCF+05] to ensure that non-idempotent operations of stateful Web service are executed reliably.

5.4.2 Conversational Web Services

From the introduction above it is clear that the client needs to perform certain actions in a specified order. First the client must invoke the *WS-Resource factory* and then redirect all messages to the received endpoint reference. Next, a conversational protocol imposed by the service and its underlying business process, may further restrict the communication. For example, the *FlightService* can specify that clients must either create a new Reservation account or login using their existing account. Next, clients can browse for available flights and add or remove flight segments to their reservation. Once a client has finished detailing his flight itinerary, he/she must checkout, do a payment, and logout.

- `Login (userName, password)`
- `newReservation (Name, Address, Phone)`
- `openReservation (ReservationNumber)`
- `browseFlights (Source, Destination)`
- `addFlightSegment (FlightNumber, Date)`
- `removeFlightSegment (FlightNumber, Date)`
- `checkout (ReservationNumber)`
- `payment (ReservationNumber, BankCardDetails)`
- `logout (userName)`

Services that engage in such a conversation with their clients are referred to as *conversational services.* In [FFG04] a conversational service is defined as a service that implements a series of operations such that the result of one operation depends on a prior operation and/or prepares for a subsequent operation. The service uses each message in a logical stream of messages to determine the processing behaviour of the service. The behaviour of a given operation is based on processing preceding messages in the logical sequence.

This communication protocol complicates the scenario to achieve dynamic service binding and hot-swapping. After all: switching from one service to another in the middle of a conversation is not straightforwardly possible. In our example, it is required that the client logs in on the service, starts a new reservation, browses for flight segments, add one or more to the itinerary, checks out and finally logs out. The client needs to execute these methods in the specified order on the same Web service, except in the case where the Web service resource can migrate to another service, a scenario that is not further discussed here.

5.4.3 Conversational Service Types

The dynamic service integration mechanism described in section 5.2 does not guarantee that each request will be redirected to the same service, as hot-swapping takes place in a transparent way. We will need to adapt our approach to ensure that when the client starts to communicate with one particular service, all subsequent messages will be redirected to (the same instance of) the same service. First, we will make our service types *conversational-aware.* An initial service type request will start a multi-step conversation between the service and the client. As shown above, the service will maintain a conversational state with the client (e.g. through a WS-Resource). This initial request typically returns some form of conversation ID (e.g. a stateful resource qualifier, or a session ID) and any further communication message must somehow carry this ID (e.g. explicitly in the parameters or implicitly in the SOAP header). A service type request can have any of the following three roles in a conversation:

- **Start:** this request starts a new conversation. Each call on this request creates a new conversation context and an accompanying unique conversation ID. In our example, this is the login method.

- **Continue:** this request is part of a conversation in progress. The conversation ID is used to correlate each call to an existing conversation. This includes methods that are intended to be called subsequent to the conversation's start and before its finish as well as requests for responses with additional information, requests for progress or status, and so on.

- **Finish:** this request finishes an existing conversation. A call to a finish request marks the conversation for destruction. At some point after a finish method returns successfully, all data associated with the conversation's context is removed. In our example, this is the logout method.

Next, we have to make sure that the binding of the Web services through service redirection aspects respects the communication protocol. The aspects will need to maintain state about the conversation it is having with the Web services. This can be realised by making the redirection aspects stateful.

5.4.4 Stateful Aspects

Most of the current joinpoint approaches feature a dynamic joinpoint model, i.e. a model where the joinpoints are runtime events of the program execution. As such, it becomes possible to invoke aspect behaviour based on runtime types, call-stack context (e.g. AspectJ's `cflow()` pointcut [KHH+01]), dynamically evaluated expressions, etc. Describing the applicability of aspects in terms of a sequence or protocol of run-time events however, is generally not supported. With the exception of the cflow() pointcut, the pointcuts of current mainstream AOP languages cannot refer to the history of previously matched pointcuts in their specification. In order to trigger an aspect on a protocol sequence of joinpoints, one is obliged to program code for maintaining a state regarding the occurrence of relevant joinpoints, as such implementing the protocol by hand. Explicit protocols are nevertheless frequently encountered in a wide range of fields such as Component-Based Software Development [FS02], data communications [Tane88] and business processes [ACD+03] such as Web services conversations.

In [VSCD05] it is argued that protocols are valid targets for aspect application, and that it is desirable to support them in the pointcut language itself while delegating the actual control-mechanism implementation to the weaver. An extension of the JAsCo programming language for stateful aspects was proposed. An implementation based on a deterministic finite automaton, has been made and added to the current version. With *JAsCo stateful aspects* it is possible to describe a protocol-based pointcut expression. Every line in the expression defines a new transition within the protocol. Each transition specifies one or more destination transitions that are matched after the current transition is fired. On every transition, advices can be attached which are executed when the transition fires. This principle is illustrated by an example in the next section.

5.4.5 Conversational Service Redirection Aspects

Code fragment 5.9 shows a *conversational service redirection aspect* for the *FlightService*. This aspect keeps track of the state of the conversation, and only allows to progress in the conversation based on the specified protocol in lines 5 to 12.

Each of the advices (lines 19 to 26) contains code responsible for invoking the *FlightService*. This is analogue to the code of the service redirection aspects as presented in section 5.2. However, only if a transition with the corresponding name is fired, the advice is triggered. This way, the conversation protocol is enforced in the client. Only if the client first invokes the login on the service type, the corresponding advice in line 19 will be triggered and the service will be invoked. This is specified in the first line of the protocol in line 7. Next, the user can start a new reservation (line 8), or open an existing one (line 9), add flight segments to his itinerary (line 11), or remove them again (line 13). Next, the user can checkout (line 14), do a payment (line 15), and finally, log out (line 16). If the client invokes a method not conforming this conversation, the service will not be invoked, as no advice will be triggered. Using a stateful aspect ensures that the service will only be invoked if the client has the correct conversation history with that service. This mechanism also avoids unnecessary calls to the service: adding a flight segment without a valid login would result in an exception anyway: this mechanism avoids the call altogether. It is also possible to

```
1  Class FlightServiceARedirection {
2      FlightService proxy;
3
4      Hook BookFlightHook {
5          BookFlightHook (methodA(args)... methodH(args)) {
6
7          login:exection(methodA) > newReservation || openReservation;
8          newReservation:execution(methodB) > addFlightSegment;
9          openReservation:execution(methodC) >
10                             addFlightSegment || removeFlightSegment || checkout;
11         addFlightSegment:execution(methodD) >
12                             addFlightSegment || removeFlightSegment || checkout;
13         removeFlightSegment:execution (methodE) > addFlightSegment || checkout;
14         checkout:execution (methodF) > payment || logout;
15         payment:exection (methodG) > logout;
16         logout:execution (methodH) > login;
17     }
18
19     around login () {... }
20     around newReservation () {... }
21     around openReservation {} {...}
22     around addFlightSegment () {... }
23     around removeFlightSegment () {... }
24     around checkout() {... }
25     around payment () {... }
26     around logout (){... }
27
28     around complement() {
29         throw new CommunicationException (the conversation protocol was not followed);
30     }
31 }
```

Code fragment 5.9: Conversational Service Redirection Aspect

specify an advice on the complement of the protocol as shown in lines 28 to 30. In this example, an exception is thrown if the conversation protocol is not respected.

Clearly, a new instance of the stateful redirection aspect is needed for each new conversation. A typical aspect language implementation, such as JAsCo, allows advanced control over the instantiation of the aspects. In this case, we want to instantiate a redirection aspect whenever the start request occurs and an ID is assigned. Any further communication with this ID needs to be redirected to the same redirection aspect instance. JAsCo allows specifying custom aspect factories for this purpose. Our custom factory creates an aspect instance for the start request, and couples this instance to the conversation ID calculated by the Web service. Any next request on the service type belonging to the same conversation will be redirected to the appropriate instance.

As mentioned before, many implementations of conversational Web services apply the factory pattern. A special service instance factory provides a method to create instances of the service for a specific conversation. When the client invokes this method, a specific endpoint reference is sent back. All future communications need to be sent to this reference. This approach, as for instance applied in BEA WebLogic Server, is completely compatible with our proposed client-side redirection mechanism, as it can be specified in the appropriate around advices to invoke the service via a particular endpoint reference.

5.4.6 Dealing with Multiple Conversational Web Services

In case multiple flight Web services are available, multiple stateful redirection aspects as shown in Code fragment 5.9 will be present. When the client starts to communicate with one particular service, all subsequent messages need to be redirected to (the same instance of) the same service. Otherwise said, a running conversation is redirected to one service, but a new conversation, marked by the start request, may be redirected to another one. As the hot-swap does not happen immediately, we refer to this process as *delayed hot-swapping* for Web services. For instance, if the service used for current conversations becomes too expensive, future conversations can be redirected to a cheaper service, while the current conversations are still settled with the more expensive service. Again, this can be realised by simply reordering the JAsCo connectors.

A problem with delayed hot-swapping is that it does not offer a solution for imminent issues such as service failures. Only start-requests can be hot-swapped, the continuing requests and final requests need to be redirected to the same service. If that service fails no other services can be addressed to deal with the request. The reason is because no other redirection aspect will have made the transition to the required state in the conversation protocol. *Immediate hot-swapping* for these requests can still be made possible with two approaches: Pro-active conversational synchronisation and reactive conversational replay. With *pro-active conversational synchronisation* the conversational state with all available Web services is synchronised. In case of our example, the start request (i.e. the login) is broadcasted to all available services. This can be simply realised with an around advice in a special dedicated broadcast aspect. If Code fragment 5.10 is applied on the login request, then the proceed() in line 3 will trigger the around chain and the login advice of each redirection aspect will be triggered. The disadvantage is that the client is automatically

```
1 around () {
2     for (int i=0; i<nbrOfAvailableServices; i++)
3         proceed();
4 }
```

Code fragment 5.10: Pro-actively Login in all Web Services Through Broadcasting

logged in on all services, while not all of them might be used. Furthermore, this approach is only valid up to a specific point in the conversation. For instance, broadcasting the checkout request to all available services is not favourable, as the user would end up with multiple flight reservations. Finally, this approach generates a lot of network traffic overhead.

A better alternative is *reactive conversational replay*. The conversation being executed with one service is saved, and in case a hot-swap is required, replayed on another service. Again, aspects are ideally suited for this purpose: a dedicated aspect can monitor all points where data is passed on to the service and store it for possible future usage. If a service invocation fails, and a hot-swap is needed, the aspect is triggered to replay the conversation. Code fragment 5.11 shows a naïve implementation of a generic advice that records the conversation between the client and the service type using Java reflection and context information of the joinpoint. The `RecordHook` (lines 10 to 19) is triggered before each method invocation on the service type and stores both the request name and the arguments of the request. In case of a failure, the `PlayHook` (lines 21 to 40) is triggered, disables the service that caused the exception (i.e. disable the corresponding connector) and replays the conversation on the service type (lines). Another service redirection aspect will be triggered and invoke another Web service. The whole conversation will be replayed and the stored conversation will be flushed. A `FlushHook` (lines 42 to 50) can also flush the conversation in specific points in the conversation, i.e. to avoid that a payment is done multiple times.

With this replay aspect it becomes possible again to do hot-swapping in a transparent way for the client. Note that this mechanism assumes all services behave exactly in the same way. Suppose another flight service is addressed which does not offer a particular flight segment as specified earlier by the client. In that case replaying the conversation will fail and the client will need to restart the conversation. To avoid these problems the number of hot-swaps should be kept to a minimum. The distinction must be made between necessary hot-swaps (e.g. initiated because of unavailability of a service) and optional hot-swaps (e.g. initiated because another service becomes cheaper). In the latter case, it might be better to delay the hot-swap to avoid problems impacting the client.

Also note that any uncompleted conversation on a failing service might need to be rolled back. If a Web service offers a compensation method, it can be invoked to roll back previous actions as soon as the service becomes available again. Note that a service might define a penalty rate, i.e. a price the client needs to pay the service provider when he/she cancels the service request after the time-out period to rollback has expired. A fallback aspect could for instance be defined to trigger whenever a service invocation fails, poll it at regular intervals to check its availability, and invoke a compensation method as soon as the service becomes available again.

```
1  Class ConversationalReplayAspect {
2      List methods;
3      List parameters;
4
5      public flushConversation() {
6          methods = new List();
7          parameters = new List();
8      }
9
10     hook RecordHook {
11         RecordHook (method (args)) {
12             execution(method);
13         }
14
15         before() {
16             methods.add (thisJoinPoint.getName());
17             parameters.add (thisJoinPoint.getArgumentsArray());
18         }
19     }
20
21     hook PlayHook {
22         PlayHook (method (args)) {
23             execution(method);
24         }
25
26         around throwing (ServiceInvocationException e) {
27             try {
28                 wsml.WebService.disableService (e.getServiceName);
29                 Class serviceType = Class.forName (name);
30                 for (int i = 0; i < methods.size(); i++) {
31                     Method method = serviceType.getMethod (methods[i]);
32                     method.invoke (parameters[i]);
33                 }
34                 flushConversation();
35             }
36             catch (Exception e) {
37                 throw new Exception (The conversation could not be replayed);
38             }
39         }
40     }
41
42     hook FlushHook {
43         FlushHook (method (args)) {
44             execution(method);
45         }
46
47         before () {
48             flushConversation();
49         }
50     }
51 }
```

Code fragment 5.11: Conversational Replay Aspect

5.5 Service Compositions

5.5.1 Introduction

Web service composition involves composing applications and processes using Web services technologies without regard to the details and differences of those environments. Otherwise said, service composition is the *orchestration* of a number of existing services to provide a richer composite service assembled to meet the requirements of the client. The composition can be the combination of one service doing some pre-processing or conversion of data before passing it on to the main service, but it can also be a complete workflow where multiple services representing different business identities collaborate in a business process. As explained in [CJ01], a differentiation can be made between pro-active compositions and reactive compositions. *Pro-active compositions* are composed off-line for deployment in a stable, always-up, resource rich platform. Typically, the composition is specified with a fixed set of partners in mind. Possibly, the Web services are even implemented after the specification of the composition in some orchestration or workflow language, to make sure that the Web services will exactly fulfil their role in the composition. Any modification to the process implementation in a later stage typically requires a new agreement between the partners before the modification can be deployed. *Reactive compositions* on the other hand are compound services that are created on the fly based on more volatile partner agreements, often optimising for real-time parameters such as available network width. Reactive compositions reference partner roles that are filled in at runtime. Our dynamic service binding mechanism supports both kinds of compositions, making it possible for compositions to be adaptable, as for instance, non-responding services in the composition can be replaced by a semantically equivalent one. The compositions can also deal better with long-term changes and evolution, as the composition is modularised as a first-class entity: a composition aspect can be easily changed and recompiled without affecting the rest of the system.

5.5.2 Service Composition Redirection Aspects

To illustrate support for compositions, consider our running holiday planning example. Remember from the mapping in section 5.2.1, that invoking *HotelServiceB* requires the conversion between a city code, as provided by the client, and a city name, as expected by the service. A second service, called *CityCodeLookupService*, offers this conversion functionality. The two Web services can be easily combined as shown in Code fragment 5.12. An around advice (lines 10 to 13) can contain multiple service invocations, and as such, the composition is modularised as a series of Java statements. By implementing and deploying a service composition redirection aspect, one is specifying a *pro-active* composition: by compiling the aspect and instantiating it by means of a connector, a fixed composition is deployed for a service type. If any of the Web service in the composition change, the composition needs to be adapted. Note that, as the binding mechanism as explained in section 5.2 is reused, hot-swapping still remains possible if multiple service compositions and/or Web services are present for the service type.

To create *reactive* compositions that not reference concrete services, the aspect could

```
1  Class HotelServiceBRedirection {
2      HotelProxy hotelProxy;
3      ConversionProxy conversionProxy;
4
5      hook getHotelsHook {
6          getHotelsHook(method (Date beginDate, Date endDate, String cityCode)) {
7              execution(method);
8          }
9
10         around() {
11             String cityName = conversionProxy.getCityName (cityCode);
12             return proxy.getHotels(beginDate, endDate, cityName);
13         }
14     }
15
16     hook bookHotelHook {...}
17 }
```

Code fragment 5.12: A Service Composition Redirection Aspect for Hotel Service B

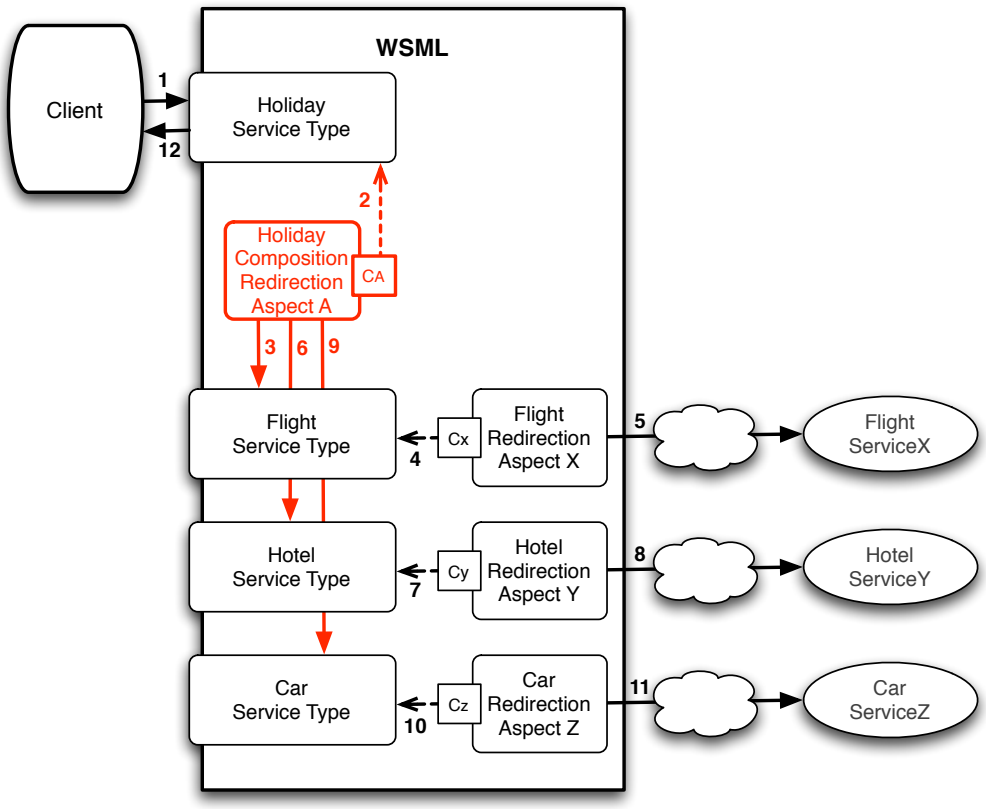

Figure 5.8: Holiday Service Composition

```
1  Class HolidayComposition {
2      HotelServiceType hotelST;
3      FlightServiceType flightST;
4      CarServiceType carST;
5
6      hook BookHolidayHook {
7          BookHolidayHook (method (String homeCode, String destCode,
8          String hotelCode, Date beginDate, Date endDate)) {
9              execution(method);
10     }
11
12     around() {
13         FlightReservation flightReservation;
14         HotelReservation hotelReservation;
15         CarReservation carReservation;
16
17     flightReservation=flightST.bookFlight (homeCode, destCode);
18         if (flightReservation!=null)
19             hotelReservation = hotelST.bookHotel (beginDate, endDate, hotelCode);
20         if (hotelReservation!=null)
21             carReservation = carST.bookCar (BeginDate, endDate);
22         return new Object[] {flightReservation, hotelReservation, carReservation);
23     }
24 }
```

Code fragment 5.13: A Service Composition Redirection Aspect for Holidays Bookings

be adjusted to reference service types again. For instance, assume the client prefers to deal directly with a *HolidayServiceType* combining the hotels, flights and cars services. Figure 5.8 shows how invoking the *HolidayServiceType*, indirectly triggers the invocation of *FlightServiceX*, *HotelServiceY* and *CarServiceZ* trough the respective service types. In the Figure, the lines indicating returned values are omitted for clarity reasons.

Code fragment 5.13 shows a naive around advice to book a complete holiday. After booking the flight (line 17), the hotel (line 19) and the car are booked (line 21). A more advanced composition could allow for more customization from the client perspective.

Invoking service types from within a service composition allows for the specification of a composition in a generic way without hardwiring concrete service interfaces. It also avoids the explosion of the number of service compositions that need to be specified in case multiple partners are available to play a specific role in a composition. By specifying for each individual service type which service(s) and composition(s) can be used a request, a temporal composition is created that best fits the criteria of the client. The complete redirection mechanism, supporting transparent mappings between service types, Web services and service compositions is illustrated in Figure 5.8. A service type is either fulfilled by a single concrete Web service, or by a service composition. A composition is composed out of concrete Web services and/or service types.

An important implication of reactive compositions is that a temporal composition is "created" at the moment a client request comes in. Each service type the composition refers to is a partner role which is being filled in be a concrete service. As service types

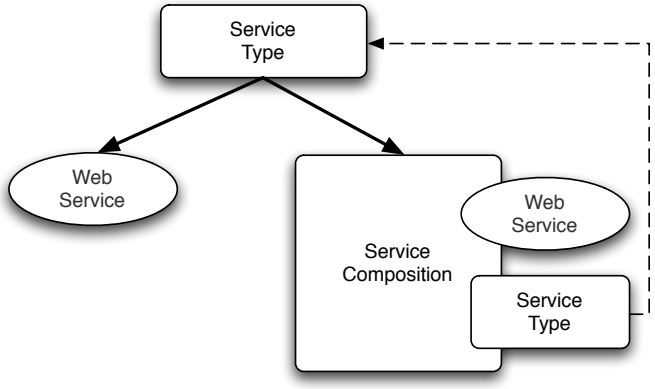

Figure 5.9: A Service Type is mapped to a Web Service or a Service Composition

support transparent hot-swapping, introducing the concept of *dynamic partner roles:* each partner in a reactive composition can be replaced by another one. In case of stateless communication, this hot-swap does not have an impact on the composition. However, as explained in section 5.4, in case of stateful communication it can become a problem. If a hot-swap occurs it might be needed to replay the conversation on the new service and specify additional compensating actions.

5.5.3 Relation with Web Services Composition Languages

A lot of research is going on in the field of service composition. High-level composition languages have been proposed: among them are the Web Services Conversation Language (WSCL) [BBB+02], Yet Another Workflow Language (YAWL) [AH05] and the Web Services Business Execution Language (WS-BPEL) [ACD+03]. These high-level composition languages have as a goal to specify, and possibly execute, business processes by combining several Web services together. These languages can be divided into *orchestration languages* and *choreography languages*. Web service orchestration describes executable business processes that interact with Web services in a certain prescribed order. As such, the advices of our service composition redirection aspects specify an orchestration. Choreographed processes on the other hand only describe the publicly visible behaviour of the messages exchanged between Web services. They do not fully specify the internal business logic of the processes, and are therefore not executable. Choreography's usefulness lies in the ability to validate the business protocol of a Web service. The specification of the stateful *FlightService* in section 5.4 requiring to log in, browse for flight segments, etc. is a good example of a choreography protocol.

Most service composition approaches realise centralised composition by specifying a centralised workflow with a number of service participants. This composition is typically deployed on a central engine. WS-BPEL [ACD+03] is becoming the de-facto standard for specifying centralised business processes. It supports the definition of both executable business process and abstract business process by providing mechanisms to specify common core concepts of both types of processes with essential extensions for each process. *Executable*

business process is defined as the model of the internal, actual behaviour of a participant in a business interaction (i.e. orchestration), while *abstract business processes* define mutually visible message behaviour of each party involved (i.e. choreography). Besides offering some way to model the business logic, composition languages may also offer support for advanced constructs such as transactions, exception handling, state, etc.

These languages are complementary to our approach where aspects are used to enable flexible mediation between Web services and clients. In our approach, the actual business process is part of the client, although it is possible to express more complicated business process constructs in the aspects, including more advanced control flow such as conditional executions, loops and parallel branching. Other features such as exception handling, compensations, stateful context, etc. are supported through additional aspects, as explained before. The result will be a collection of aspects working together to realise a composition. This more decentralised approach differs from dedicated composition languages such as WS-BPEL where all concerns are tangled and scattered in one big monolithic specification.

The main purpose of the WSML approach is to avoid the tangling and scattering of service-related code in an application integrating with multiple services. As a BPEL-like process communicates also with a set of Web services, the same issues arise: code dealing with various concerns such as exception handling, access control, authentication, business rules, etc. often does not fit into the process-oriented modular structure of a web service composition and as a result, this code is scattered around the processes and tangled with the specification of other concerns within a single process. Furthermore, composition languages such as WS-BPEL only support predefined and static composition logic: the composition is fixed and cannot be altered, again limiting its usability in a dynamic Web services environment. As discussed in the next section, some AOP extensions for BPEL have been proposed to tackle these issues.

As WSML and BPEL are complementary, they can be easily integrated with each other. Service types as offered by the WSML can be integrated in a BPEL process and fulfil the specified partner roles. As a service type can be exposed as a Web service, this does not pose any technical issues. A result of this integration is that all service-related concerns, which are tangled and scattered in the composition can be removed from the composition and modularised into aspects. The BPEL specification will only contain the core composition, which can than be easily changed at runtime by deploying WSML aspects. Redirection aspects can be used to integrate semantically equivalent services in BPEL, something that is not possible by default because BPEL offers no support for glue code to deal with compositional mismatches. The WSML selection and management aspects can be used to enforce selection policies and other client-side management concerns in a non-invasive manner without polluting the core composition.

Another approach is to integrate a service composition specified in BPEL in the WSML and fulfil the functionality required by a service type. A WS-BPEL engine is able to interpret a BPEL specification, and expose that composition itself as a Web service. Therefore, redirecting a client request to a BPEL composition is not more difficult than redirecting to a regular Web service. In the prototype, discussed in section 8.4, it is shown how a business process engine is integrated in the WSML. The advantage is that more complicated compositions can be specified in a dedicated language such as BPEL and that this composition

can be integrated in a client through the WSML.

5.6 Related Work

5.6.1 Adaptive Integration Approaches

There are many composition approaches and dedicated languages to specify service compositions, workflows and business processes using Web services. We limit the discussion to approaches focussing on adaptations and runtime flexibility.

The *Web Services Invocation Framework (WSIF)* [APA03] supports a Java API for invoking services irrespectively of how and where they are provided. WSIF mostly focuses on making the client unaware of service location migrations and changes in protocols. This is done by using WSDL as a normalised description for disparate software modules. Different middleware technologies including SOAP, Enterprise Java Beans and Java Message Service can be used amongst others. Our approach is complementary to the work realised by WSIF, as the WSML focuses on hiding away both syntactic and semantic differences of Web services. Services using the WSIF and described using WSDL can be integrated in the WSML too.

In traditional workflow systems, the need for adaptability and flexibility is provided using a variety of implementation approaches, [CIJ+00][EH99][HHJ+99]. Their main contributions are formally founded methods to make the workflow process capable of efficiently adapting the tasks to be performed and their execution order (e.g. adding or removing tasks, changing control flow paths, etc.) in reaction to changes in the environment conditions like changes in the type of the participants, their role in an organisation and the infrastructure reconfiguration. The process adaptations can be classified according to whether they are performed at design-time or runtime and either at the process schema or process instance level. For instance, eFlow [CIJ+00] is an approach that supports the specification, enactment, and management of composite e-services, modelled as processes that are enacted by a service process engine. It uses several constructs to achieve adaptability, including dynamic service selection and binding, parallel execution of multiple equivalent services, etc. A migration manager assists users to modify running process instances without violating a predefined set of behavioural consistency rules. But adaptability remains insufficient and vendor specific [EMP05, KB04]. Moreover, many adaptation triggers, like infrastructure changes, considered by workflow adaptation are not relevant for Web services as services hide all implementation details.

In [SGHS05] a software architecture, called *Aspect-Oriented Web Services (AOWS),* is presented. It is targeted at describing crosscutting concerns between web services to give more complete description of Web services, supporting richer dynamic discovery and seamless integration. An implementation is made on the .NET platform and all AOWS subsystems and their relationships have been formally modelled. While aiming to achieve similar goals as the WSML, AOWS does not support third-party independent services as services need to be modelled in an AOWSDL language, and registered in a dedicated AOUDDI registry. Clients communicate with AOconnectors, which address the Web services through

adaptors; if necessary multiple services are bundled in an AOComposite. The aspectual features of the AOWS framework are used to provide more efficient and effective dynamic description, discovery and integration. Similar as in our approach, service related code is extracted from the client, and the client only needs to communicate with the AOconnectors.

5.6.2 Aspect-Oriented Composition Approaches

The most well-known approach for Web services composition is the *Web Services Business Process Execution Language (WS-BPEL)* [ACD+03], presented before as an XML-based specification language for business processes. Because of the limitations discussed in section 5.5.3 (namely the lack of support for crosscutting concerns and the impossibility to dynamically change a composition), some aspect-oriented extensions for BPEL were recently proposed. With *Ao4BPEL* [CM04], aspects can be (un)plugged into the composition process at runtime. Since BPEL processes consist of a set of activities, joinpoints in AO4BPEL are well-defined points in the execution of the processes: each BPEL activity is a possible joinpoint. The attributes of a business process or certain activity can be used as predicates to choose relevant joinpoints. Since BPEL processes are XML documents, XPath [23], a query language for XML documents, has been chosen as the pointcut language. Like AspectJ and JAsCo, AO4BPEL supports before, after and around advices. An advice in AO4BPEL is an activity specified in BPEL that must be executed before, after or instead of another activity. When the advice logic cannot be expressed in BPEL syntax, it is possible to use code segments in Java by using JBPEL, although this breaks the portability of the BPEL process. As a proof-of-concept, a BPEL engine is made aspect-aware by adding an aspect manager to it and by extending the process interpreter function to check before and after each activity whether an aspect is present. The main advantage of the AO4BPEL approach is that both the composition and the aspect advices are written in the same language, requiring the programmer to only learn one technology. In the WSML, aspects are written in JAsCo, while the client can be implemented in any other kind of language on any kind of platform. On a downside, the pointcut language of AO4BPEL is less expressive than the JAsCo pointcut language used in the WSML. Another recent WS-BPEL extension is *AdaptiveBPEL* [EMP05]. It is closely related to AO4BPEL but adds policy-driven adaptations and selection of aspects to enable client-specific customisation of Web services.

In [CE05] an aspect-oriented approach to compose dynamically Web Services in a decentralised manner is proposed, in contrast to the WSML, which realises centralised composition. *Aspect-Sensitive Services (CASS)* is a distributed aspect platform that targets the encapsulation of coordination, activity life-cycle and context propagation concerns in service-oriented environments. CASS also has the notion of redirection aspects. All web services need to be CASS-enabled, and aspects are deployed on different domain controllers for each composition context. This approach is therefore not suited in an environment with independent service providers.

In [SFCV+05] we have made an evaluation of FuseJ [SDV06], as another approach for decentralized composition. FuseJ is an architectural description language aiming at the unification of aspects and components. The motivation behind FuseJ is that all new approaches introducing new programming languages or frameworks for modularising crosscutting concerns will treat aspects as a different kind of entity. However, the behaviour implemented

Table 5.1: Comparison of Evaluated Web Service Composition Approaches

	FuseJ	WS-BPEL	AO4BPEL	CASS	WSML
Seamless AOP	+	-	+-	+-	+-
Explicit process description support	-	+	+	+	-
Reusable composition	+-	+-	+-	n/a	+-
Selective composition	+	-	+-	+	+
Dynamic composition	+	-	+-	+	+
Automatic discovery	+	-	-	-	+
Compatibility requirements	+-	+	+	+-	+-

by aspects is not that different from ordinary component behaviour. Both implement some functionality required within the application and only the way in which they interact with other software entities differs. FuseJ proposes to apply aspect-oriented composition mechanisms upon existing module constructs. As such, independently specified components can be deployed in both a regular and an aspect-oriented fashion, achieving a seamless integration between aspects and components. While not targeted at service composition, we have evaluated FuseJ as a composition approach and compared it with WS-BPEL, WSML, Ao4BPEL and CASS. We briefly discuss the evaluation criteria as listed in Table 5.1, borrowed from [SFCV+05]. *Seamless AOP* indicates whether explicit aspect constructs are used and exposed to the user of the approach. The (+/-) for the WSML results from the fact that the use of AOP hidden for the administrator of the WSML but there are still explicit aspect constructs in the implementation. *Explicit process description support* denotes whether explicit constructs are present to specify processes, something the WSML does not support. *Reusable compositions* (whether services are hard-wired in the composition or not), *selective composition* (indicating if partner services can be selected based on predefined conditions), *dynamic composition* (specify whether services can be hot-swapped) and *automatic discovery* (indicating if services can be looked up and integrated at runtime) are all supported. For the compatibility requirements (indicating whether the approach imposes specific requirements on the Web services), the WSML received a (+/-) as automatic discovery and matchmaking is only possible if services are semantically annotated (as discussed in section 8.3.3).

5.6.3 Semantic Approaches

Ontologies are a key enabling technology for a Semantic Web [BFD99, MPM+05] of services whose properties, capabilities, interfaces, and effects are encoded in an unambiguous, machine-understandable form. The realisation of the Semantic Web is underway with the development of markup languages such as *DAML-S* [ABH+02]. DAML-S is an ontology-based approach to the description of Web services developed as part of the DARPA agent markup language program [HM00]. DAML-S aims at providing a common ontology of services and its ultimate goal is to provide an ontology that allows agents and users to discover, invoke and compose Web services. Currently the structure of the DAML-S ontology is threefold and consists of a service profile for advertising and discovering services, a process model which gives a detailed description of a service's operation and a service grounding

which provides details on how to interoperate with a service via message exchange.

The *Web Service Modelling Framework (WSMF)* [FB02] provides an approach based on Semantic Web technology for developing and describing Web services and their composition. The aim of WSMF is to enable fully flexible and scalable e-commerce based on web services. Goal repositories are used to define the problems that should be solved by web services; they are specified in terms of pre-conditions and post-conditions of services together with a service model. WSMF aims at strongly de-coupling the various components that implement a Web service application while at the same time providing a maximal degree of mediation between the different components. WSFM builds on comprehensive ontologies such as DAML-S and provides the concepts of goal repositories and mediators to solve complex service requests.

While not being the core focus of this dissertation, we have extended our service integration mechanism with support for the Semantic Web by documenting Web services and service types with the *Web Ontology Language (OWL)* [BSP+01], a successor of DAML-S. A matchmaking algorithm to determine compatibility between them has been developed and is discussed in section 8.3.3.

5.7 Conclusions

In this chapter we have presented our approach for dynamic integration of Web services and compositions. Client applications communicate with a generic service type and client requests are redirected at runtime to one of the available semantically equivalent Web services. At implementation level, service communication details are modularised in service redirection aspects. This AOP mechanism realises a mapping between the service types and the concrete Web services, offers support for dynamic binding and hot-swapping through around advice chaining. Additional exception handling is dealt with in separate dedicated fallback aspects. Conditional bindings and multiple service bindings are possible trough the expressivity of an AOP pointcut language and the fine-grained deployment mechanism as provided by connectors. Support for conversational messaging is provided through stateful aspects. With stateful aspects, the conversation protocol between the client and a Web service is mapped to a protocol-based pointcut expression. As such, the conversation protocol is enforced in the client. In this case, hot-swapping is supported by synchronising the conversation between multiple services, or by replaying the conversation on another equivalent service.

Finally, service compositions are supported by modularising composition details in the redirection aspects. Concrete Web service interfaces are not hard-wired in the composition by referring back to service types. As such, reactive compositions, where each partner in the composition is determined at runtime, are made. As Web services are used in an unpredictable network environment such as the internet, the service composition can adapt itself to temporarily unreachable services that belong to different domain controllers. A complex long running composition needs to dynamically adapt itself to short-term changes, for instance by replacing a non-responding service by a semantically equivalent one. Selecting the most appropriate service for a given role based on a set of service selection policies is

the subject of chapter 6. Long-term evolvability is supported by only specifying the core service composition in a redirection aspect. If the core composition itself changes, the corresponding aspect can be easily rewritten and recompiled on the fly. This as a result of the composition being a first-class identity: one composition is modularised logically and physically in one reusable aspect. Any other management concern such as logging, monitoring, billing, security, caching, etc. can be easily added through additional aspects whenever the business environment requires doing so. This is the subject of Chapter 7.

Chapter 6

Web Services Selection

Abstract In this chapter, the service integration process is made more intelligent by incorporating a more advanced service selection mechanism. The client application can specify a set of selection policies that are enforced are enforced to select the most appropriate service for a given client request. A categorisation of selection policies is made based on the location of the required triggering points, and for each category it is shown how aspect-oriented programming is used to implement them in a modular fashion. Optionally, monitoring aspects are used to setup measurement points in the system to collect any required monitoring data.

6.1 Introduction

In the previous chapter we demonstrated how a flexible dynamic binding mechanism for Web services is realised using aspect-oriented programming techniques. The next step is to make this mechanism more intelligent and customisable by incorporating non-functional requirements as part of the client requests. This is important in order to cope with the volatile nature of the business environment of a SOA. Non-functional requirements can guide the process of selecting the most appropriate Web service for a given service request: today, the client might prefer a Web service because it belongs to a specific business partner, but tomorrow the service is required to offer a specific service-level agreement to become eligible. In order to cope with the evolving business environment, it must be possible for client applications to specify their non-functional requirements as *service selection policies* when requesting service functionality. These selection policies must be decoupled from the Web services and the client as they are driven by business requirements and need to dynamically reflect changes in the environment. Keeping selection policies separately enhances maintainability, reusability and adaptability.

One limitation of the current service documentation in WSDL (see section 2.4.1.4) is that it only allows for the description of service functionality while not offering support for the specification of non-functional properties. Many WSDL extensions such as the Web Service Offering Language (WSOL) [TPP03] are however being proposed that enable the specification of non-functional properties on Web services. Using a dedicated language to specify these properties enables service providers to describe the *Quality-of-Service (QoS)* of their services. Clients can base their selection process on this description: for example, selection policies could be specified to take into account the cost or physical distance of a Web service. Some other non-functional properties of a service can however not be described but need to be determined at runtime by doing service monitoring. These properties are based on the runtime characteristics of the services and we refer to them as *service behavioural properties*. Examples of such properties are average response time, number of successful invocations, network bandwidth and service reliability. Other selection policies can be based on the state of the client, or even the Web service itself. As argued in Chapter 3, current Web service integration approaches provide little or no support for the runtime enforcement of selection policies during the service binding process. The programmer needs to provide code for this purpose manually and switch between the service proxies present in the client.

In this chapter we will illustrate how aspects are ideally suited to modularise service selection policies and service monitoring concerns. These selection aspects will guide the dynamic service binding and as a result, whenever a client invokes a service type, the most appropriate service will be invoked. We identify several categories of Web services in section 6.2. Next, section 6.3 discusses the applicability and advantages of AOP for service monitoring and selection. A first category of service selection, based on QoS, is presented in section 6.4. The subsequent section deals with selection based on client requests and service responses. Selection based on client or service context are discussed in section 6.6 and related work and conclusions are presented respectively in section 6.7 and section 6.8.

6.2 Service Selection Policies Classification

Service selection policies can be classified as imperatives and guidelines. An *imperative* is a constraint on a service that has to be satisfied at all time, i.e. it is an invariant. Imperatives can describe absolute conditions (e.g. the cost of the service can not exceed a fixed amount, the response time of a service can not drop under some threshold, etc.), or can involve interrelationships with other services or the system (e.g. a service needs to be cheaper than the average of the cost of all registered services). In order to get approved for selection and integration, a Web service has to comply with all specified imperative selection policies. At the moment a service does not satisfy at least one imperative selection policy, it is *disapproved* or *disqualified* and therefore not considered to be invoked any longer.

To determine which service to address if two or more services are approved, *guidelines* are employed. A guideline specifies that, if multiple approved services are available for the required functionality, one is preferred over the other (e.g. the cheapest service or the service with the highest encryption level). This implies that services are compared with each other and a ranking is made. This is an application of the well-known *multi-criteria problem* [BV85]. Note however that, if an approved service does not satisfy a specific guideline, e.g., it is not the cheapest at a given moment in time, it might still be considered for selection later on, if it becomes the cheapest service available or if cheaper services fail. We can deduce two elements from a selection policy specification:

- **Triggers:** The policy is triggered whenever an environmental change occurs that affects the constraint's enforcement. One can locate these triggering points in various places ranging from the client, throughout the network, and to the actual Web services.
- **Action:** A selection policy's action typically includes qualifying, disqualifying, and prioritising services for the current or future client requests.

We have identified several categories of selection policies as depicted in Table 6.1 and Table 6.2. The first table lists the most common selection policies, namely those based on the QoS. Several kinds of imperatives and guidelines can be specified on the non-functional service properties. We make the distinction between two kinds of properties: firstly, there are documented properties advertised in the service documentation (for example, constraining the cost to use a Web service). The triggering points' location of policies specified on these properties will depend on the kind of property, the documentation, and the notification mechanism used by the service. Secondly, there are service behaviour-based properties indicating properties that need to be monitored at runtime over a period of time (for example, the downtime in the past month). Typically a set of measurement points to collect the required QoS data is required to enforce service behaviour-based properties.

A second main category are selection policies that are based on context. In case of policies based on the client-context, the constraint applies to explicit or implicit client-side business logic, for example: "If the client application's user has a gold subscription, use the fastest service." The triggering points reside in the client. On the other hand, the policy could be based on explicit or implicit service-side business logic, for example: "During peak hours, the service's capacity is limited to a certain number of requests for each client." The triggering point depends on the documentation and notification mechanism the service uses.

Table 6.1: QoS Service Selection

	Non-Functional QoS Properties	
	Documented	Service Behavioural
Static QoS Imperatives	A static invariant applied on a documented service property. Service approval is determined by evaluating one service at a time.	A static invariant applied on a monitored service property. Service approval is determined by evaluating one service at a time.
Dynamic QoS Imperatives	A dynamic invariant applied on documented service property. Service approval is determined by evaluating all services at a time.	A dynamic invariant applied on monitored service property. Service approval is determined by evaluating all services at a time.
QoS Guidelines	Service ranking is made based on a documented service property.	Service ranking is made based on a monitored service property.
Qos Normalized Guidelines	Service ranking is made based on a set of normalized documented and/or monitored service properties.	

A third category are policies that apply on the client requests and/or service responses. The type of client request may guide the selection process. For instance, hotel descriptions are retrieved from the fastest service, while hotel bookings are made on the cheapest one. Policies can also be applied on the service responses, for instance retrieving hotel descriptions from the service that returns the longest hotel list. In the latter case it is possible that multiple services are invoked, but only filtered results are returned to the client. Table 6.2 lists selection policies based on state and requests/responses.

Table 6.2: Service Selection Based on the Client or the Service

	Client	Web Service
Context	The client context determines which services or set of services to select.	The service context determines whether that services is selected.
Request-Response	The client request determines which services or set of services to select.	The response returned by a service is evaluated before being returned to the client.

6.3 Towards a flexible implementation of selection policies

After having identified what selection policies are, the challenge is how to implement them flexibly while avoiding changes in the client application every time the business requirements change. Previous work by Cibrán et al. [CDJ03, CDS03, CSH+04] focuses on decoupling the business logic rules in the context of software applications developed using object-oriented or component-based software development techniques. These applications have substantial core application functionality and are normally driven by business policies. Thus, they need to constantly cope with changes in the business requirements embodied in their policies. In this context it is increasingly important to consider business rules as a means to capture some business policies explicitly. A business rule is defined by the Business Rules Group as a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behaviour of the business [Bus00]. As business rules tend to evolve more frequently than the core application functionality [Hall01, KRRS96], it is crucial to separate them from the core application, in order to trace them to business policies and decisions, externalise them for a business audience, and change them. Contrary to typical approaches in the field that only focus on decoupling the business rules themselves [RDR+00, GKPG+05], this work identifies the need to decouple the code that links the business rules to the core application functionality. This linking code crosscuts the core application and thus need to be separated in order to achieve highly reusable and configurable business rules. That research shows how AOP (and in particular JAsCo) is ideal to decouple the crosscutting business rules links.

Successful experiments using AspectJ [KHH+01] and JAsCo (see section 4.5.4) were performed achieving the identified requirements. In the Web services context, selection policies are driven by business requirements and need to dynamically reflect the changes in the environment, analogously to business rules. The same way AOP resulted ideal to separate business rules links and connect them with the core application, we claim AOP can be successfully used to plug-in and out selection policies that govern the selection and monitoring of services.

We propose *service selection aspects* that cleanly modularise selection logic that is able to approve or select the most appropriate Web service for a given business requirement. Each selection policy is represented by one aspect; *it maps the triggers to joinpoints and the actions to advices*. For example, a selection aspect can implement a simple policy stating a service's maximum allowed price. A change in the pricing of the Web service will trigger this selection aspect, resulting in the qualification or disqualification of the Web service as specified in the advice of the aspect. More complex scenarios involve service behaviour-based policies, based on data monitored over a period of time. By using *service monitoring aspects*, the monitoring logic that would otherwise remain scattered over the client application, is modularised in one place, and monitoring points can be easily set up in a non-invasive way. As such, the monitoring aspects are employed for observing system, environmental and service changes and are also able to monitor the execution of the services themselves.

The use of selection and monitoring aspects in the context of the WSML is depicted in Figure 6.1. Whenever a change in the QoS of a Web service occurs (step 1), a monitoring

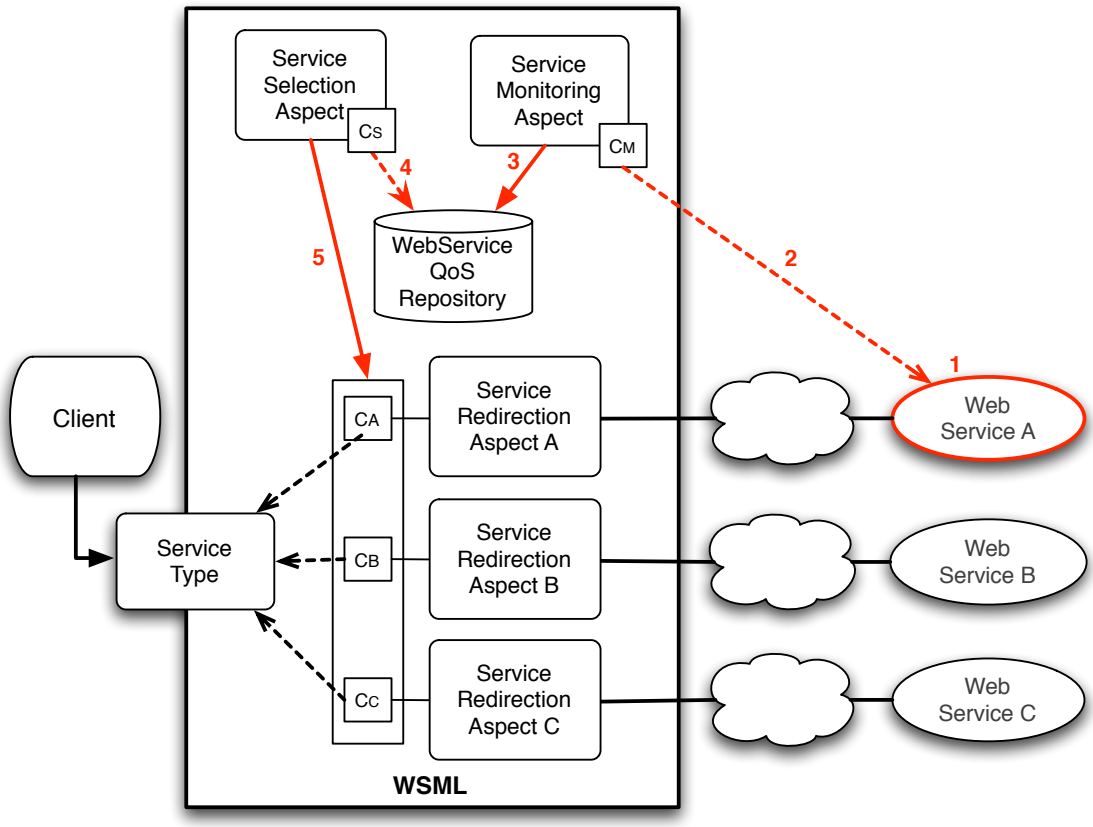

Figure 6.1: Service Monitoring and Service Selection Aspects in the WSML

aspect will detect this change, e.g. by monitoring the runtime behaviour of the Web service or by polling specific service properties (step 2). The monitoring aspect will compute and store a new property value in the QoS repository (step 3) where it is available for selection aspects. One or more selection aspects are triggered by these property changes (step 4) and re-evaluate if the service (and possibly other services) needs to be qualified, disqualified or re-prioritised (step 5). Alternatively, changes in the client or service context or specific client requests or service results may also trigger the selection. These scenarios are not depicted. In the following subsections we will discuss the various selection policies categories in depth.

6.4 Selection Based on Quality of Service

6.4.1 Non-Functional Properties

Functional requirements define what a software application is expected to do. Non-functional requirements define how the software operates or how the functionality is exhibited [Chung91]. Non-functional requirements, such as accuracy, security, cost and performance, state global constraints on how an application should exhibit its functionality.

Using traditional software engineering approaches, functional requirements are implemented gradually into the software application during the software development phases. At the end of the development, all functional requirements are implemented in such a way that the software satisfies the requirements as designed during the first development phases. Non-functional requirements, however, do not follow the same implementation manner as the functional requirements. The term non-functional implies fairly complex functions that need to be implemented in order to provide and enforce a certain property [LSS05]. So they are usually satisfied to a certain degree as a consequence of the design decisions that were taken for implementing the software functionality. The problem of a non-functional property is not its possibly complex functional implementation, but the explicit or implicit reference to this implementation spread across the several functional modules of an application. In order to avoid the mixture of both functional and non-functional properties tangled in the same code fragments we need to have a good separation of these concerns, as offered for example by AOP.

Systems that use a number of Web services can specify certain requirements on the non-functional properties of these services as selection policies. As applications are subject to changing business requirements and changing third-party Web service providers, the incorporated policies must be able to adapt to these changes. Existing state-of-the-art technologies for publishing and finding services, such as WSDL and UDDI use static descriptions for service interfaces. WSDL documentation is insufficient for this purpose because WSDL cannot express non-functional properties and doing key-based searches in UDDI registries does not take into account non-functional criteria. As a result, the approaches that use these technologies to integrate services do not take into consideration dynamic service selection based on the assessment of non-functional properties. At the moment, a lot of research is being done to describe Quality of Services (QoS), which allows to define the concept of service quality. Several languages are being proposed to build a QoS Ontology, which defines the semantics of QoS parameters and their relationships. A non-exhaustive list of QoS related languages currently under development, includes The Hierarchical QoS Markup Language (HQML) [GNYW01] , the Web Ontology Language (OWL-S) [BSP+01], The Web Service Level Agreement (WSLA) [KL03] The Web Services Offering Language (WSOL) [TPP03] and the Web Service Modeling Framework (WSMF) [FB02] .

Examples of QoS properties for Web services include, but are not limited to:

- **Scalability:** Scalability q_{sc}(ws) of a Web service is the ability of providers to consistently serve the requests despite variations in the volume of requests in a given period. Scalability is defined in a range [0,1] and is related to the service performance (see next section) [Ran03].
- **Maximum Response Time:** Maximum Response time q_{maxRes}(ws) is the guaranteed maximum time required to complete a client request. It is measured from the moment the request arrives at the service and the moment the result leaves the service.
- **Maximum Throughput:** The maximum throughput q_{maxThr}(ws) is the maximum number of simultaneous requests of Web services served in a given period with the guaranteed response time.

- **Availability:** The availability q_{av}(ws) is the probability that the Web service will be available at some period of time. Larger values represent that the service is always ready to use while smaller values indicate unpredictability service availability [MN02].
- **Time-to-Repair:** Related to the service availability is the Time-to-Repair (TTR) value. *TTR* represents the time it takes to repair a service that has failed. Ideally smaller values of TTR are desirable [MN02].
- **Accessibility:** The accessibility of a Web service q_{acc}(ws) represents whether the Web service is capable of serving the client's requests [SA03]. It may be expressed as a probability measure denoting the success rate or chance of a successful service instantiation at a given time [MN02]. There could be situations when a Web service is available but not accessible because of high volume of requests. This property is therefore related to the scalability.
- **Transactions:** Web services providing an undo procedure to rollback the service execution in a certain period without any charges can be an important selection criterion [LNZ04]. This transactional property can be evaluated in two dimensions: whether an undo procedure is supported and what the time-out of the undo procedure is. The first is denoted by q_{tx}(ws)=0/1 where 1 indicates support and 0 lack of support and the latter is denoted by q_{cons}(ws), indicating the duration where an undo is allowed.
- **Pricing:** the cost, q_{pr}(ws), of the service execution of a Web Service as specified by the provider. Web services either directly advertise the pricing information of their services, or they provide means for potential clients to inquire about it. Related to pricing are the **Compensation Rate** q_{comp}(ws), the percentage of the original price that will be refunded when the service provider cannot honour the committed service, and the **Penalty Rate** q_{pen}(ws), the percentage of the original price the client needs to pay the service provider when he/she cancels the service request after the time-out period for transaction to rollback has expired [LNZ04].

By using a Web service that has the QoS described by some documentation language, we can use these properties to enforce selection policies that guide the service integration process. It is important that these properties are described in terms of a QoS ontological model. After all, conflicts in the semantics between the client and the different service providers may occur. An ontology defining a basic set of QoS parameters and their relationships, together with the possibility to extend an ontology with domain-specific parameters is required. Already, multiple ontologies for Web services have emerged, each introducing their own semantics, relationships and categorisations [BBD+05], [OWL-S05], [LNZ04], [PSL03], [TBE05]. The QoS properties of the Web services can be stored in a central *QoS registry*, similar to a UDDI registry. We do not propose our own documentation language to describe non-functional properties but assume that over time a standard will emerge for this purpose, similarly to what happened for Web services security with the WS-Security standard [ADH+02].

Finally, note that the documentation as obtained by the service provider may not be neutral. One is relying on the documentation as provided, which may be undesired depending on the nature of the provided information: e.g. billing information versus guaranteed

uptime per month. In the first case, the information will most likely be accurate, but in the second case, the provider might provide too optimistic numbers. In the latter case, one could turn to specially dedicated third-party QoS monitoring services that endorse or rate a particular service independently, or one could gather monitoring data oneself, as described in the next section.

6.4.2 QoS Service Monitoring

6.4.2.1 Introduction

Service monitoring is a term typically used in the larger context of the server-side process of testing, debugging, deploying and observing the functionality and performance of running Web services in order to provide the optimal quality-of-service to its customers. The importance of this process is underlined by the fact that several companies have made it their core competence. While not totally unrelated, we use this term in a slightly different context: service monitoring as the client-side process of monitoring the behaviour and/or performance of the available Web services in the service environment *from the client point of view* in order to select the most optimal one. Suppose the client application of our holiday operator (see section 3.1) only wants to communicate with hotel Web services that return a result in 5000 milliseconds, in order to offer an optimal end-user experience. As mentioned in the previous section, the client could rely on the non-functional service documentation describing the QoS, or it could monitor the available Web service itself and use the monitored data for the selection process. Doing active monitoring in the client has the advantage that the data is collected from the actual consumption of the service and therefore will always be up-to-date and objective. Examples of monitored properties for Web services include, but are not limited to:

- **Actual throughput:** The actual throughput of a Web service is the actual number of consumers accessing a particular Web service in a given period of time. In a multi-organisational setup, this number cannot be measured from the client point-of-view but needs to be provided by the service provider. A single instance of the WSML is only capable of monitoring the throughput of requests passing through it.
- **Performance:** The performance of a Web service represents how fast a service request can be completed. It can be measured in terms of throughput, response time, latency, execution time, transaction time, etc. [LNZ04, MN02, SA03] In general, faster Web services should provide higher throughput, faster response time, lower latency, lower execution time, and faster transaction time. Network factors will have a direct affect on this property when measuring performance from the client point-of-view.
- **Execution Time:** The execution duration $q_{ex}(ws)$ measures the expected delay in seconds between the moment when a request is sent and the moment the result is retrieved. The execution duration is computed using the expression $q_{ex}(ws) = q_{process}(ws) + q_{trans}(ws)$, with $q_{process}(ws)$ being the actual processing time and $q_{trans}(ws)$ being the actual transmission time. From the client point-of-view only the $q_{ex}(ws)$ can be measured without knowing the individual values of $q_{process}(ws)$ and $q_{trans}(ws)$.

- **Processing Time:** The time $q_{process}$(ws) the service takes to process the result. It is calculated as $q_{process}$(ws) = t_i(ws) - t_d(ws) with t_d(ws) the timestamp when the request entered the Web service and t_i(ws) the timestamp when the result was sent out. The processing time can only be measured by the service provider or through active monitoring by the underlying middleware.

- **Regulatory:** Regulatory of a Web service is a quality aspect dealing with issues of conformance of services with the rules, the law, compliance with standard, and the established Service Level Agreement. The regulatory can be determined by ratings the WSML keeps over time for each provider, based on its own experiences. Additionaly, these ratings can be combined with additional ratings from other clients or based on heuristics such as continuous usage or based on third-party monitoring services [TBE05].

- **Reputation:** The reputation q_{rep}(ws) of a service is a measure of its trustworthiness. It mainly depends on the clients' experiences of using the service. The value of the reputation is defined as the average ranking given to the service by clients.

- **Consumption Rate:** The consumption rate is obtained by the ratio of the actual throughput to the number of hits of a Web service. The number of hits can for instance be determined by the number of times the WSDL file of the service has been retrieved. Alternatively, the consumption rate can be obtained by the ratio of the actual throughput to the maximum throughput [TBE05].

- **Reliability:** Reliability is the quality aspect of a Web service that represents the degree of being capable of maintaining the service and service quality. The number of failures per month or year represents a measure of reliability of a Web service. In another sense, reliability refers to the assured and ordered delivery for messages being sent and received by service clients and service providers [MN02].

- **Stability:** The stability of a Web service is the change cycle of a Web service [Ran03], it is the measure of the frequency of change related to the service in terms of its interface, QoS information and/or implementation. The value of the stability is defined as the number of times a Web service changed by the provider in specific time interval.

Leaving the monitoring to the client requires a complex framework is set up for this purpose. If large number of service providers need to be polled constantly this monitoring framework might become quite complex. Furthermore, this framework should be able to deal with a large level of flexibility. The client might be interested in fast services one day, but prefer more reliable services tomorrow. Using traditional software engineering approaches, it would not be straightforward to introduce the, possibly unanticipated, monitoring logic at run-time in a non-invasive way. Encapsulating the monitoring concern in reusable aspects has the advantage that monitoring logic can be dynamically deployed in order to intercept Web service communication and start generating monitoring data.

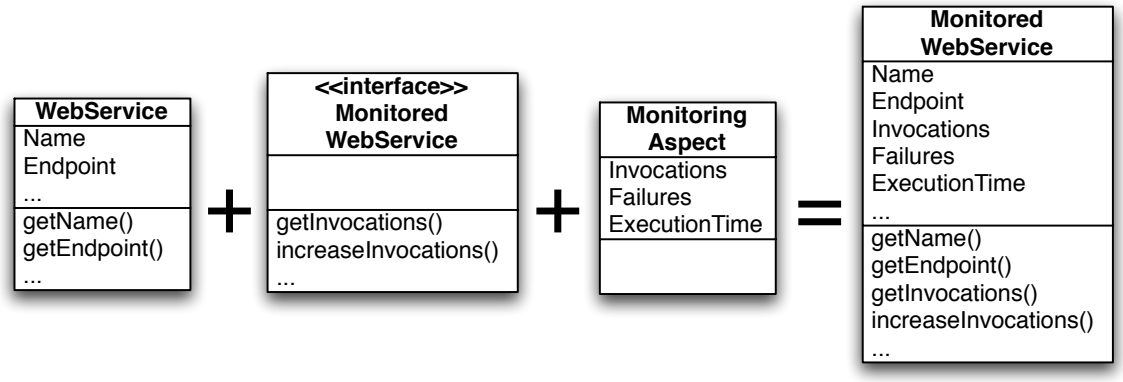

Figure 6.2: Monitoring a Web Service with Virtual Mixins

6.4.2.2 Service Monitoring Aspects

A monitoring aspect can be used to encapsulate all monitoring logic. The aspect can be deployed at a wide variety of measurement points by specifying appropriate joinpoints in the environment. When a monitoring aspect is deployed to collect data, it should make that data available for the selection aspects in order for them to reason over. This might be achievable by hooking the selection aspects on the monitoring aspects, but conceptually this concept is difficult to understand. The concept of aspects-on-aspects, defined in [DFS02] as aspects being "visible" for other aspects, is supported in some AOP languages such as JAsCo, but may result in technical difficulties, e.g. to do debugging. A second approach is to store the collected data in a collection somewhere, so that other aspects can access it. In this case, the `wsml.WebService` class, representing a Web service in the WSML, is a good candidate to host the data collection. Through dedicated getter and setter methods such as `addProperty (name, value)` and `getProperty (value)` the properties can be added (by monitoring aspects) and queried (by selection aspects). Although following object-oriented principles, this still leaves the conceptual problem that the values of the properties (which are part of the crosscutting monitoring concern), are not stored in the aspect itself, but rather outside it. Note also that the monitoring aspect frequently needs access to the values (e.g. to calculate an average result), so one might decide to keep also a copy of the values in the aspect itself, leading to data redundancy. A third approach is called *Mixins*, an AOP approach that allows inserting an implementation of an interface in target classes so that other aspects can depend on that. In AspectJ [KHH+01], this feature is known as *Intertype Declarations*. This is ideal for communicating information, such as monitoring data, between aspects.

Figure 6.2 shows a `WebService` class that is "extended" with a `MonitoredWebService` mixin interface through the `MonitoringAspect`. This aspect contains the implementation of the methods specified in the interface. Code fragment 6.1 shows an example of a `MonitoredWebService` interface providing methods to get and set the execution time of the service invocations, and methods to get and increment the number of invocations and invocation failures on that Web service. This interface can be inserted into a target class by simply declaring both a hook that implements this interface and a corresponding connector.

```
1 interface MonitoredWebService extends jasco.runtime.mixin.IMixin {
2     public int getExecutionTime();
3     public int getInvocations();
4     public int getFailures();
5     public void setExecutionTime (int speed);
6     public void incrementInvocations();
7     public void incrementFailures();
8 }
```

Code fragment 6.1: Monitored Web Service Virtual Mixins Interface

In our example, the wsml.`WebService` class is the target class.

The implementation of this interface is provided in an aspect, as shown in the next Code fragment. The first hook (lines 3 to 23) shows an `IntroduceHook` with a straightforward implementation. There are three attributes for the execution time, the number of total invocations and the number of failed invocations and corresponding getter and setter methods. A second `InvocationHook` (lines 25 to 51) contains the actual monitoring logic. The three advices in this hook will be triggered whenever a Web service invocation occurs. First, a before advice will increase the invocations counter and start a timer (lines 33 to 38). Second, an after returning advice will stop the timer and update the execution time (lines 40 to 44). In case of an invocation failure (which will result in an exception as shown in section 5.2.5 of the previous chapter), an after throwing advice will be triggered to update the failures counter (lines 46 to 50).

The connector deploying the Monitoring Aspect is shown next in Code fragment 6.3. First, the `IntroduceHook` is deployed, and introduces the `MonitoredWebService` interface in the `WebService` class. The `perobject` keyword indicates a unique hook instance will be created for every target object instance. This avoids that the attributes in the hook are shared amongst multiple `WebService` objects. Note that in JAsCo this introduction happens through lazy evaluation for performance reasons: the hook instance is only created the first time one of the methods of the introduced interface is called. In lines 4 and 5 the `InvocationHook` is deployed on all methods of the `HotelService` proxy (see section 3.3.2.1), causing the monitoring to be performed for this particular Web service. Note that through the use of wild cards, a more fine-grained control over the deployment of the monitoring concern is possible. The `perall` keyword ensures a new hook instance is created for each encountered joinpoint.

Actually, the `MonitoredWebService` interface should only contain the getter methods, and expose those to the outside world. The setter methods should be either private, or non-existing as the `InvocationHook` should be able to directly manipulate the appropriate variables. However, in JAsCo this is impossible as variables that are global to an aspect are shared amongst all aspect instances. As a result, in the `InvocationHook` (lines 33 and 34 of Code fragment 6.2) aspect reflection is needed to determine on which Web service the invocation took place. This makes the aspect code less comprehensible and reusable.

In this example the measured execution time is the sum of the time to serialise the request into a SOAP message, the time to sent the message over the network, the time it takes by the service to process the request, the time to send the result back over the network

```
1  class MonitoringAspect {
2
3      hook IntroduceHook implements MonitoredWebService {
4
5          private int executionTime= 0;
6          private int invocations = 0;
7          private int failures = 0;
8
9          IntroduceHook(void method(..args)) {
10             introduce(method(args));
11         }
12
13         public int getExecutionTime() {return executionTime };
14
15         public void setExecutionTime (int lastExecution) {
16             if (executionTime==0)
17                 executionTime= lastExecution;
18             else executionTime=
19                 ((executionTime * (invocations-1)) + lastExecution) / invocations;
20         }
21         public int getInvocations () {return invocations};
22         public void incrementInvocations() {invocations++};
23         ...
24     }
25
26     hook InvocationHook {
27
28         private long startTimer = 0;
29
30         InvocationHook(method(..args)){
31             execution(method);
32         }
33
34         before () {
35             MonitoredWebService ws = (MonitoredWebService)
36             WSML.getWebService (thisJoinPoint.getClassName());
37             ws.incrementInvocations();
38             startTimer = System.currentTimeMillis();
39         }
40
41         after returning (Object result) {
42             ws = ...
43             stopTimer = System.currentTimeMillis();
44             ws.setExecutionTime (stopTimer  startTimer);
45         }
46
47         after throwing(ServiceInvocationException e) {
48             ws = ...
49             ws.incrementFailures();
50             throw e;
51         }
52     }
53 }
```

Code fragment 6.2: Service Monitoring Aspect with Virtual Mixins

```
1 static connector IntroduceMixin {
2     perobject MonitoringAspect.IntroduceHook introduce =
3         new MonitoringAspect.IntroduceHook(* WebService.*(*));
4     perall MonitoringAspect.InvocationHook invocation =
5         new MonitoringAspect.InvocationHook(* HotelService.* (*));
6 }
```

Code fragment 6.3: Connector for Monitoring Aspect

to the client plus the time to deserialise the SOAP message back into object(s). A more advanced aspect could make the distinction between them and measure more detailed and accurately. The added complexity in this case is that more measurement points are needed, in a distributed fashion: for instance, to measure the service response time, measurement points in the service are needed. This type of *distributed monitoring* is further discussed in Chapter 7.

To ensure that the data describing the QoS of the available Web services is fair and accurate, the best option is to have information from multiple places. Besides the information provided in the service documentation, and monitored data, one could also consult feedback from other clients. For example, pricing information can be provided by the service provider, performance can be computed by monitoring aspects, while service reputation can be based on clients' feedback. Frameworks, as described in [LNZ04] and [TBE05], can be installed that allow clients to enter their feedback. Possibly, additional mechanisms are needed to prevent manipulation of the data.

6.4.3 QoS Service Selection

Given the detailed service information made available either through service documentation, feedback or through advanced monitoring, and given the fact that this information is obtained in a fair and open manner, a more intelligent and client-tailored service selection procedure based on QoS, can be realised. In the next subsections, we will discuss the various QoS policies from Table 6.1 the client can specify to guide the selection process and show how aspect-oriented programming is ideally suited to implement the policies.

6.4.3.1 Static Imperatives

A first category of QoS policies includes static imperatives. A policy can constrain the value of a non-functional property of the service by specifying fixed minimum and/or maximum boundaries. This policy states that a functionally compatible Web service, will only be addressed if a specific property value of the Web service falls in specific range. For instance, one could state that the price of using the Web service should have an upper limit of 0,10 euro per invocation. If the price is higher, the service is disqualified until the price changes. In that case, the qualification status will be re-evaluated. A selection aspect can be implemented to enforce this policy. From the policy specification we can deduce the joinpoints and advices of the aspect:

Identification of joinpoints: Remember from chapter 5 that clients communicate

with service types (see section 5.1) and that, using a dynamic binding mechanism one of the available Web services is addressed to deal with a client request. The policy stated above is clearly specified on the level of the service type: all Web services registered for the service type must have the service property in between specific boundaries. Therefore, a first joinpoint will be the moment a new Web service is registered for the service type. A second joinpoint is the moment one of the registered Web services changes its QoS, either explicitly by changing its documentation, or implicitly, by changing its (non-functional) behaviour. In that case, the service property constrained by the policy may change, requiring a re-evaluation of the service's approval or disapproval status. To detect changes in the service documentation, there are basically two practical solutions: notification or polling. In case of notification, the Web service notifies the client (in our case the WSML) about a change in its QoS (e.g. using WS-eventing [BCC+04c]). In case of polling, the documentation of the Web service could be polled with regular intervals. In either case, the moment when the change is detected, will be a joinpoint for the selection aspect. When the property is monitored dynamically, the joinpoint will be the moment the monitoring aspect calculates and saves a new value of the property.

Identification of advices: The action is to disapprove or approve the Web service based on the condition evaluation. As explained in Chapter 5, the dynamic binding mechanism to integrate a Web service into the client relies on service redirection aspects and connectors. The connectors hook the redirection aspects on one or more service types. Whenever a client invokes the service type, a redirection aspect is triggered and a Web service is invoked. By default, the connectors are ordered in the way they were added to the system. As result the same service will always be addressed by default. Changing the binding mechanism to incorporate selection policies involves changing the connectors, for instance if a service is too expensive, disabling its connector and thus preventing the corresponding Web service from being invoked.

A selection aspect can be written that hooks on the identified joinpoints, while implementing an after advice that enables or disables a service redirection connector depending on the evaluation of the service property. An aspect implementing a pricing imperative and an aspect realising a response time imperative are in essence the same. A generic selection aspect for both purposes can be implemented, and next, configured through a connector.

6.4.3.2 Collaboration Between Multiple Imperatives

The selection aspect described above will filter out services and up until now it was assumed that only one policy, and thus one aspect was deployed at a given moment. In a real world situation however, it is obvious that multiple policies will be specified and that these policies may interfere with each other. If we assume multiple imperatives are defined, then a Web service needs to be approved by all of them. If at least one imperative disapproves the service, it will be disqualified. To realise this, the selection aspects will need to cooperate in order to come up with a single list of approved services (which may be reordered by an additional guideline policy as we show later on). The most elegant way is that each aspect stores its own list of approved and disapproved services, and remains unaware of the existence of other selection aspects, hence promoting modularity of the code. The following code example shows a simplified version of a basic selection aspect that allows approving

and disapproving the services registered for a service type according to the values of a particular property.

In the `ServicePropertyImperative` aspect, the hooks `WebServiceAddedHook` (lines 13 to 21) and `PropertyChangedHook` (lines 23 to 32) approve or disapprove a Web service depending on the value of the property the policy is initialised with. Their corresponding after advices are executed when a new Web service is registered for the first time or when the value of the observed property changes. The evaluation of the service property with respect to the minimum and maximum boundaries is done through a dedicated compare method. This method is not included in the Code fragment. Property values can be compared with each other when the classes that represent them implement the java.lang.Comparable interface. This interface imposes a total ordering on objects of a class that implements it. This ordering is referred to as the class's *natural ordering*, and the `compareTo` method is referred to as the *natural comparison method*.

The `GetApprovedServicesHook` (lines 34 to 45) realises cooperation of multiple deployed selection aspects. The hook is able to retrieve only those Web services that satisfy all selection policies among the ones registered for a particular service type. First, in line 40, the proceed method continues with the enforcement of other selection policies. Next, in lines 41 and 42, all services disapproved by the current policy are removed from the list. As a result, only services approved by all policies is returned. This hook will be triggered at the moment a list of all approved services is needed (e.g. to enable/disable connectors, to order them according to a guideline, to show them in a GUI, etc.). The attributes in lines 3 to 5 can be specified in a connector. For instance, the selection policy to approve hotel services based on price can be realised by a connector deployed on the `HotelServiceType`, with a minimum of 0 euro, a maximum of 1 euro for property price. Additionally it should be specified what to do with a service if the constrained property is not defined (i.e. approve or disapprove such a service).

Note that if a new Web service is registered, service behavioural properties such as the *average speed* property will not be set until the service is invoked at least once. As a result, these services have no reputation yet, which prevents them from being selected. To circumvent this problem, their monitoring could be based on dummy invocations and/or their selection could be based on the recommendations or endorsement by trusted third parties using more advanced selection policies. We would also like to point out that selection policies can be specified individually at the level of each service type request to realise an even more fine-grained selection. For instance, for browsing hotels the fastest service could be preferred, while for making reservations, the cheapest service could be favoured.

As multiple selection policies are specified on the same service type, this could imply a certain performance overhead at run-time. It is important to consider that searching for the "best" solution can be computationally expensive. Moreover, because the "best" might change over time, unwanted oscillations can seriously affect the overall performance of the application. Consider a dynamic (monitored) property of a service that changes frequently (e.g. the response time of a ping[1] signal of the Web service server). In that case, it might

[1]**Pinging** is used to check for basic 2-way connectivity between two computers. It is the process of sending signals (packets) to another computer, typically a server, on a network to see if it sends a return or an 'echo.' If the signals 'time-out' the remote computer may down, it might be disconnected from the

```
1  class ServicePropertyImperative {
2
3      private List approved, disapproved;
4      private Object minimum, maximum;
5      private String property;
6
7      public void checkWebService (WebService ws) {
8          if (compare (minimum, ws.getProperty (property), maximum))
9              approved.add (ws);
10         else disapproved.add (ws);
11     }
12
13     hook WebServiceAddedHook {
14         WebServiceAddedHook (method(Webservice ws)){
15             execution(method);
16         }
17
18         after () {
19             checkWebService (ws);
20         }
21     }
22
23     hook PropertyChangedHook {
24         PropertyChangedHook (method(String mproperty, ..args)){
25             execution(method);
26         }
27
28         after() {
29             WebService ws = (WebService) thisJoinpointObject;
30             checkWebService (ws);
31         }
32     }
33
34     hook GetApprovedServicesHook {
35         GetApprovedServicesHook (method()){
36             execution(method);
37         }
38
39         around() {
40             List listPreFiltered = (List)proceed();
41             List myApprovedServices = (List)approved.clone();
42             myApprovedServices.retainAll(listPreFiltered);
43             return myApprovedServices;
44         }
45     }
46 }
```

Code fragment 6.4: Service Selection Aspect for an Imperative Policy

be necessary to limit the number of evaluations. Preventing the monitoring aspect from updating the property too often is one way; only evaluating and updating the preference ranking in the selection aspect after a number of property updates is another. A dedicated aspect could be used to categorise all services according to their properties (e.g. fast and cheap, slow and expensive, etc.) and specify the policies based on these categories.

6.4.3.3 Dynamic Imperatives

The imperatives discussed earlier represent selection policies that express a constraint on a service property whose value needs to be situated in between statically specified boundaries. These boundaries remain fixed. A problem that may occur is that the Web services evolve and over time all get disqualified. Consider the constraint specifying that the response time of a service must be lower than 100 ms. This policy can work well for some time, but it is possible that over time all services become disqualified due to increased service loads and network congestions. In order to avoid this situation, a dynamic policy can be specified stating that only Web services with a response time at most 50 ms slower than the highest response time measured in the last 24h are qualified. This policy ensures that the slowest Web services are disqualified without the risk of having no service at our disposal at all. We can adjust our selection aspect of Code fragment 6.4 to deal with dynamic boundaries.

Identification of joinpoints: The joinpoints are the moment when a new Web service is registered, when the property of a registered Web service changes and when a registered Web service is removed.

Identification of advices: Whenever the aspect is triggered through one of the three identified joinpoints, it will have to calculate new minimum and maximum *derived* values for the boundaries. After all, each of the three identified events has a possible impact on the boundaries. In the advice, an iteration over all registered Web services is necessary to obtain the property and to calculate the boundaries. If one of the boundaries has changed, a second iteration is needed to re-evaluate all Web services and to qualify or disqualify them. The issue that too many evaluations may occur and that they might be computationally expensive is even worse in this case, so it might be necessary to take into account a buffer on the boundaries. In between this buffer the boundary limits may vary without requiring re-evaluation of all Web services.

Up until now, the order in which the imperative policies are employed does not matter. Each imperative policy filters out Web services it considers "bad", and the order in which this occurs is of no importance. This commutative behaviour of the selection policies is an important property of our solution, as an AOP language does not always support aspect precedence and combination strategies to deal with feature interactions. However, in the case of dynamic boundaries, the order does matter. Consider four Web services with a price of respectively 0.5, 1.0, 1.5, and 4.0 euro. A static imperative states that services cannot cost more than 2 euros. A dynamic imperative states that the price of any service cannot be more of the average of all services. If the dynamic imperative is employed first, the average

network, or there is a problem with the network itself. Massive variations in the ping value may for instance occur in case of heavy server loads or fluctuations in the network connectivity.

is 3,5 euro, filtering out only service 4. If the static imperative is employed first, the average is 1 euro, filtering out services 2, 3 and 4. It will depend on the context which situation is preferred. The advantage of the second scenario is that the peak of the Web service 4 is not taken into account and therefore the calculated value better represents the average price of the remaining services. In case more dynamic imperative policies are specified, the complexity increases more, as the ordering of execution of each policy will result in a different outcome. A simple solution could be to detect and reject multiple of these conflicting aspects as they are deployed. Another possibility is to indicate at configuration or at deployment time a conflict is detected and make the ordering of the enforcement of the selection aspects explicit. As mentioned earlier, we can describe aspect ordering in the JAsCo language trough connector priorities or connector combination strategies and therefore implementation of these solutions is possible.

6.4.3.4 Guidelines

Specifying one or more imperatives will reduce the number of Web services that are candidates to be addressed. However, the chance is small that the size of the subset of Web services that survives all imperative filters is exactly one. Therefore, a guideline can be used to determine which service in the subset is preferred over another. Otherwise said, the non-functional properties of the service can be used to make a preference ranking. A straightforward example of a guideline is to prefer services with a high average response time in the last month. A monitoring aspect can deliver this data, and a selection aspect can order the services.

Identification of joinpoints: The joinpoints identified for an imperative policy are also applicable here: whenever a new service is registered and whenever its property changes, the policy needs to be triggered. Additionally, when a service is removed, it will have to be removed from the preference ranking.

Identification of advices: When the aspect is triggered, it should execute some evaluation and make a ranking of the available Web services. Using the dynamic service binding mechanism of Chapter 5, this ranking can be made by simply reordering the connectors of the service redirection aspects. If multiple joinpoints are applicable on the same joinpoints (as is the case with service redirection aspects), then the order of the connectors will determine which aspect will be triggered first. Therefore, a service redirection aspect with a corresponding connector with a higher priority will be executed before one with a lower priority. Another approach besides connector priorities is *connector combination strategies*, which are used to control the execution sequence of connectors. This is further discussed in Chapter 7 for management purposes. How the evaluation should be done depends on the kind of guideline, the natural comparison method of the java.lang.Comparable interface can be reused for this purpose.

Again it is possible to implement a generic aspect that does not depend on a specific property, ranking services according to price or according to speed is a similar process. A generic selection aspect can be configured by specifying which property should be used for the ranking, how the ranking should be done (e.g. ascending or descending) and what to do in case the property is not specified for a service (e.g. give it the highest or lowest

priority). Additionally it should be taken into account that properties of multiple services are not always comparable: they might have different units or they may be specified against a different reference schema or even use completely different QoS ontologies. Conversions might be required in that case.

6.4.3.5 Collaboration between Multiple Guidelines

Unlike imperative selection aspects, only one guideline can be applied on a set of Web services. If a ranking is made on one property, adding a second guideline ordering on another property will undo the ordering done by the first guideline. If a preference ranking is to be made while considering multiple QoS properties, a solution is to compute an open and fair QoS value for each Web service, based on all QoS properties. Then, a preference ranking can be made based on those values. Calculating a QoS value can be done by constructing the normalisation of a matrix out of all the properties of the services as described in [LNZ04] and [TBE05]. A QoS matrix is constructed with x rows and y columns where x equals the number of Web services and y the number of considered properties. On this matrix, a series of normalisations are applied. The purposes of normalisation are to allow for a uniform measurement of service qualities independent of units, to provide a uniform index to represent service qualities for each provider and to provide a threshold regarding the qualities. In [LNZ04], the client expresses his preferences by specifying an array where each element represents a weight assigned to a property. By applying this array on the matrix, individual QoS values can be calculated for each Web service. Alternatively, in [TBE05], a matchmaking algorithm is presented based on *Euclidean distance measuring* [Dun02]. Similarity distance measure is used to solve the problem of finding the relative difference between two values. In this case, the client preferences with respect to the ideal properties of a Web service are expressed in a vector. The Web service with a smaller Euclidean distance is preferred over a service with a larger distance.

Figure 6.3 shows a scenario where three selection policies work together. Six functionally equivalent Web services (C_a ... C_f) are present in the system. Their QoS is defined by five properties (q_1 ... q_5). First, a static imperative analyses property q_2 of each Web service with respect to statically defined boundaries through function f_1, resulting in the disqualification of Web services C_b and C_d. Next, a dynamic imperative analyses property q_5 of each Web service to define dynamic boundaries and evaluates each Web services with respect to these boundaries. This results in the disqualification of Web service C_c. Finally, a guideline will prioritise the remaining three Web service by calculating a normalised QoS value of the remaining properties q_1 , q_3 and q_4 for each Web service. This results in a ranking where C_e is assigned priority 1, C_a priority 2 and C_f priority 3.

6.4.4 Selection for Service Compositions

As described in section 5.5, it is also possible to dynamically bind service compositions to a service type. In that case, the redirection mechanism will have to choose between all functionally equivalent service compositions and/or Web services. Additionally, in case of reactive compositions, each role in the composition may be fulfilled by a different service at a given moment. The selection mechanism introduced in this chapter can be reused for service

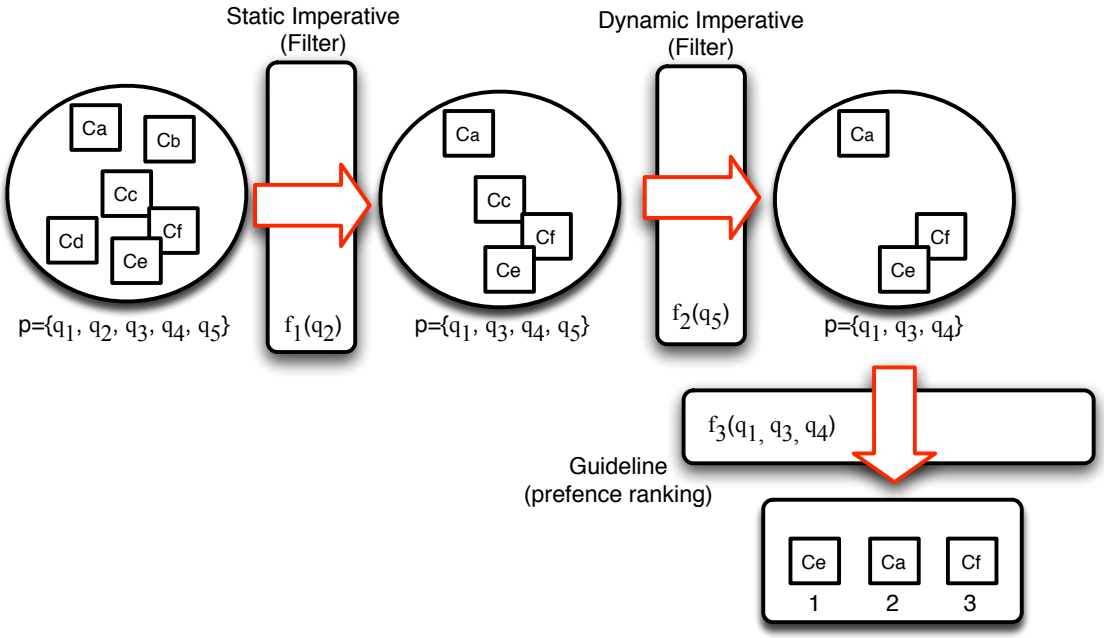

Figure 6.3: Cooperation of multiple Selection Policies

compositions. The only difficulty is defining an accurate QoS description of the composition. This is needed to verify whether a set of services selected for composition satisfies the QoS requirements for the whole composition. If such a description is available, the selection can be done as before: for instance, if the cheapest service functionality is preferred, then a service composition, which costs amount to 1 euro is cheaper than a single service doing the same thing for 2 euro. Describing the QoS of a service composition is done by aggregating the QoS dimensions of the individual services. For some properties such as service cost this is straightforward, but for other properties, especially the behavioural ones, the situation is more complex. For instance, calculating the execution time will require taking the control flow of the composition into account. It is outside the scope of this dissertation to discuss the functions required to calculate the QoS of a service composition. The interested reader is referred to [JRM04]. This paper focuses on QoS aggregation based on abstract composition patterns like sequence, loop and parallel executions.

We have done experiments in [VSCV+05] with *Adaptive Programming* as a technique to calculate service composition properties. Adaptive Programming (AP) [LOO01] aims at providing support for a very different kind of crosscutting concerns than the ones tackled by typical aspect-oriented approaches. When an operation involves a set of cooperating classes, one can either localise this operation in one class or split the operation over the set of associated classes. Localising the operation in one class causes hard-coded information about the structural relationships between these classes and is as such a violation of the well-known *Law of Demeter* [LH89]. The other alternative, namely distributing the operation over the set of involved classes, conforms to the Law of Demeter, but causes the logic of the desired behaviour to be spread over different classes making evolution very difficult.

To capture an operation that involves several cooperating classes, AP introduces adaptive visitors, which allow visiting the objects contained within an application without explicitly describing the structural relationships among these objects. Traversal strategies are responsible for specifying the abstract visiting process for an adaptive visitor.

In the case of service composition, the logic to calculate a service composition property depends on the property values of the services it is composed of, as well as the behaviour of the composition itself, possibly in an unanticipated fashion. While it was possible to calculate simple service composition properties by using the visitor pattern [GHJ95], this implementation led to a lot of visitor classes, containing in many cases duplicated processing logic. In addition, the implementation of these visitors had to be manually adapted whenever the property data structure changed as unanticipated properties could not be calculated. In order to keep the implementation of the visiting process reusable and structure-shy, adaptive visitors, implemented as JAsCo aspects are employed. With a JAsCo Adaptive Programming extension [VSCV+05] it is possible to implement an adaptive visitor as a regular JAsCo aspect. These visitors will *visit* each service of the composition in order to calculate a value for the service composition. While we have implemented a set of reusable adaptive visitors which allow calculating the sum, average, minimum, maximum, etc. of compositions, we envision more complicated properties can be calculated in a similar fashion. The visitors are resistant to structural changes in the property data structure of the WSML, reusable for multiple types of properties and combination strategies can be employed to guide how the visiting process must be executed.

6.5 Request/Response Initiated Service Selection

By employing the selection policies presented in the previous section, a particular service is chosen regardless of the client. Without taking into account the current client state, or the particular request that is being handled, a most optimal service is selected *pro-actively*. In this section, we discuss the case where the client requests and their arguments are taken into account (in subsection 6.5.1) and where service responses influence the selection process (in subsection 6.5.2).

6.5.1 Service Selection Based on Client Requests

6.5.1.1 Example

A more advanced scenario is a selection policy that chooses a service *depending on the information provided in the request* from the client application. Suppose the hotel service providers from our case study each represent different hotel companies, and therefore have different price offerings for the same information. For example, *HotelServiceA* offers discounts when booking hotels in Europe, but it is very expensive for hotels outside Europe. *HotelServiceB* on the other hand is specialised in U.S. based hotels, but is more costly in booking European hotels as it is required to charge additional booking fees. In this case, the parameter values of the client requests (e.g. the `CityCode` or `CityName` parameters in the methods of the *HotelServiceType*) must be taken into account when choosing the

cheapest service for the request. This scenario differs from the selection policies discussed in the previous section, as these policies can be enforced pro-actively: filtering out services by imperatives or prioritising services can take place at any time, and as a result identical client requests may be redirected to different services at different times. In the current scenario, the client request needs to be analysed first, in order to select the optimal service.

6.5.1.2 Conditional Binding

A first attempt to realise this kind of selection in a modularised fashion with aspects is re-using the conditional binding mechanism as presented in Chapter 5. Remember from section 5.2.6 that this mechanism enforces a conditional check to evaluate client requests before the service is being addressed, to avoid calls to services that are unable to deal with a specific request. Whenever a client issues a request, each available Web service is analysed by means of a conditional check in the service redirection aspects, and those services that cannot deal with the request are taken out of the redirection chain. The same mechanism could be used to check for additional constraints, enforced by selection policies. A drawback of this solution is that two concerns are encapsulated inside one aspect. Namely, both the redirection mechanism for a Web service and the selection logic for the service are encapsulated inside a single aspect. Furthermore, the selection logic will need to be present in all service redirection aspects, leading to code redundancy.

A logical next step is decoupling the selection logic from the redirection aspect. This promotes the reusability of the same selection aspect for multiple services. Encapsulating this selection policy in a separate aspect requires having access to the context of the joinpoint, i.e. the parameters of the client request. We identify two possible scenarios as depicted in Figure 6.4. Scenario A shows a Service Selection Aspect hooking on the Service Redirection Aspects to control the further execution of these aspects. In Scenario B a Selection Strategy is deployed to control which Service Redirection Aspect is preferred for a particular client request. We discuss both scenarios in the next sections.

6.5.1.3 Scenario A: Aspects on Aspects

Identification of joinpoints: A set of possible joinpoints where the arguments can be intercepted is the service redirection aspects themselves (scenario A of Figure 6.4). As such, the selection aspect will interfere with the execution of a redirection aspect as it might prohibit the execution of a redirection advice (e.g. because the service is too expensive) or prefer one redirection advice over another advice (e.g. because one service is faster than another). Aspects that interfere with other aspects are better known as aspects on aspects, defined in [DFS02] as being "visible" for other aspects. The *ServiceSelectionAspect* of Figure 6.4 encapsulates an imperative constraint on the request arguments. The selection aspect triggers whenever a service redirection aspect is executed. This scenario can be implemented in an AOP language supporting aspects-on-aspects such as JAsCo. In our case, the selection aspect will *hook on the advice* of the service redirection aspect.

Identification of Advices: If the advice of a service redirection aspect is executed, the selection aspect will replace this behaviour by means of an *around* advice. In this

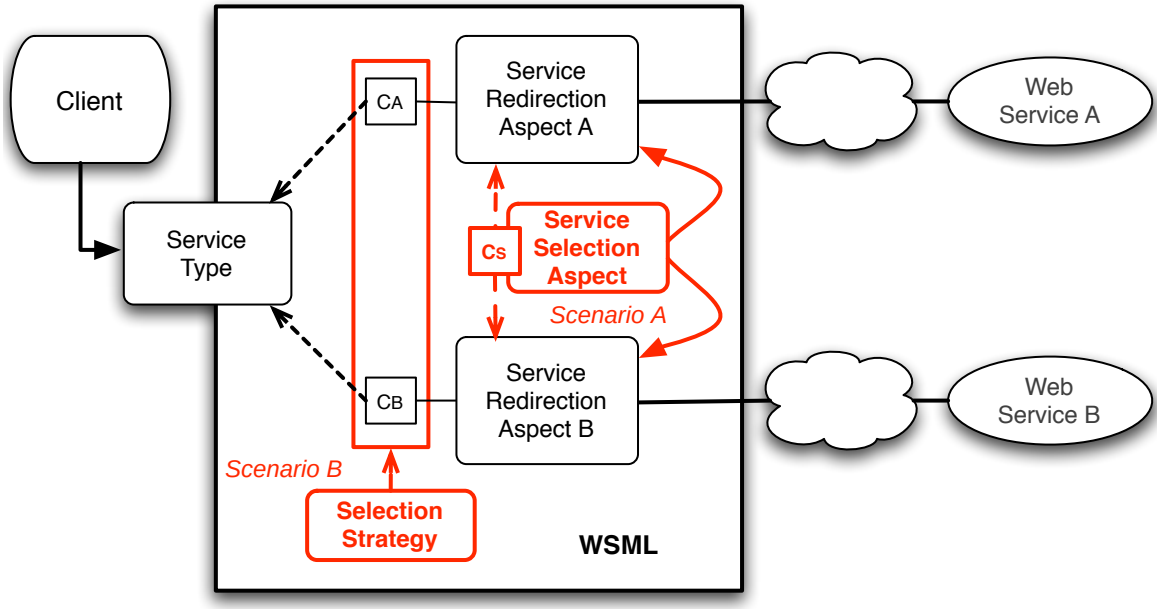

Figure 6.4: Two Scenarios for Service Selection Based on Client Requests

around advice the context of the original joinpoint of the redirection aspect, namely the provided arguments of the client request, are checked against the imperative constraint. If the arguments are valid the selection policy should not interfere and let the original redirection aspect *proceed* with its redirection logic and continue with the invocation of the Web service. Or, if the selection policy disapproves the service for the current context, it will block further invocation of the service by not proceeding the redirection but rather throwing an exception. As explained in Chapter 5, section 5.2.5 a fallback aspect can be used to capture any exceptions generated during the redirection process. The fact that a selection aspect prohibits the invocation of a Web service can be treated as just-another kind of exception. The fallback aspect can capture this exception, and continue with another service redirection aspect. In that case, the selection aspect can again interfere and analyse the applicability of the service for the request.

Note that the example given above includes an imperative constraint. A guideline constraint complicates the situation because in that case, the redirection aspects need to be reordered according to some preference ranking, based on an evaluation of the arguments' requests. This is not possible with this solution, as the selection aspect is triggered after a redirection aspect is already triggered. In that case, it can simply block the further execution; re-ordering the aspects is not possible anymore. Therefore, we turn to scenario B.

6.5.1.4 Scenario B: Aspect Combination Strategies

Another solution is applying the selection logic at the level of the service type and treating it as a problem of feature interaction (scenario B of Figure 6.4). Indeed, all available service redirection aspects hook on the service type, and we want to solve the problem of

deciding which aspect should actually be triggered. Otherwise said, which of the aspects should be triggered at a certain joinpoint, and in which order. This is referred to in the AOP community as aspect interaction and aspect composition problems [PSC+01], describing the issue on how aspects, which are all implemented in isolation, should be deployed together without interfering, but rather cooperate together. One simple AOP technique to achieve this is already explained in Chapter 5: with *around advice chaining*, all available service redirection aspects are placed in a logical chain: the chain is continued when a redirection aspect fails to invoke a service. A second technique we already employed is *connector priorities* in JAsCo. By assigning a (global) priority index to a connector, its corresponding aspect always has precedence over one with a lower priority index. This technique was used to implement the guidelines in previous section.

However, in the case of selection policies that need to work on a per-joinpoint basis, more advanced strategies are needed. To resolve this, a more expressive way of declaring how the aspects should cooperate is required. For example, specifying that when aspect A is triggered, aspect B cannot be triggered for this joinpoint, or that one aspect has priority over another. Making this possible in an AOP implementation can be done by introducing new keywords in the language: for instance, adding a new connector keyword *exclude* which specifies that aspect A excludes aspect B. However, as discussed in [Van04] other aspect combinations require additional keywords and it seems impossible to be able to define all possible combinations in advance. A more flexible and extensible system is proposed in JAsCo that allows defining a combination strategy using regular Java. A `CombinationStrategy` interface is introduced that needs to be implemented by each concrete combination strategy. A JAsCo combination strategy works like a filter on the list of hooks that are applicable at a certain point in the execution.

The combination strategies as available in version 0.8.6 of JAsCo, however, only have access to the hooks belonging to the connector it is specified upon. What is needed here is combination strategy that works on all hooks of a particular joinpoint and not just the ones that were instantiated together with the combination strategy as it is working now. This would imply that hooks can be added to the combination strategy in more than one connector. At the moment, the hooks are instantiated in the connector and added directly to one combination strategy by passing the hooks in the constructor of the combination strategy. Thus, the combination strategy is only accessible from one connector that instantiates the hooks. Therefore we propose a combination strategy that is accessible from multiple connectors by offering an extended interface that allows to add hooks.

In languages lacking the possibility to describe combination strategies, one would have to resort to using the reflection capabilities of the AOP language, as a workaround to reason about the aspects and their applicability and to modify their interactions. For example, in JAsCo the connector registry can be consulted and modified, in AspectWerkz [Boner04] a manager class for aspects can be consulted and most other AOP languages offer similar infrastructures.

6.5.2 Response-Based Selection

As described earlier, the QoS is an important driver to prefer one service over another. If the QoS is described in the service documentation, the client can make certain assumptions on the service and its delivered functionality. The QoS description may also include assertions over the response the service returns, for instance: the *FlightService* provider may state that the seat availability for its flights is not older than 15 minutes. This assertion is important to avoid that end-users attempt to book flights that are no longer available. An additional policy could be implemented to do quality tests on the results and verify the QoS. In case of approval, the result is passed back to the client, in case of disapproval, the result is discarded and another service is invoked. Again, aspects are ideal to encapsulate this constraint checking after service invocations.

Identification of joinpoints: an aspect that will check the results of a service invocation will be enforced for a given service type. As soon as a service redirection aspect has invoked a service, and is about to return this result to the client, this result-checking aspect will need to interfere. Therefore, the methods of the service type on which the result constraints apply, will be joinpoints for the new aspect.

Identification of advices: as the result of a service invocation is to be intercepted, an around returning advice is appropriate to check the result and continue the returning of the result in case of an approval, or to do something else (e.g. pass the result to a next service by continuing the service redirection chain) in case of a disapproval. Note that this aspect will not interfere with a fallback aspect (Chapter 5, section 5.2.5) as a fallback aspect contains around throwing advices, i.e. advices that are only triggered when an exception is thrown in the joinpoint. So, if a service invocation succeeds, the result-checking aspect is executed to check any constraints on the service. If a service invocation fails, the fallback aspect is executed to deal with the exception: both types of aspects are complementary.

An interesting application of result-based service selection is result averaging, i.e. invoking a set of services and taking the average of the results. Suppose the travel agent of the running example wants to include weather information to the pages with holiday destinations. To include accurate temperate information, it could invoke three weather services and take the average of the three returned results.

6.6 Context-Based Service Selection

6.6.1 Example

The selection policies identified up until now are based either on QoS (section 6.4) or by analysis of client requests or service responses (section 6.5). A category that has not been discussed yet is service selection based on the client context or end-user preferences. The client context can also influence the Web service selection process: for instance, in a ubiquitous environment, the holiday booking application could run on either a mobile device with limited processing power, a narrow-band wireless network connection and a small display, or it could run on a standard computer with more resources and a large display. Depending on this setup, the application should address Web services that return limited results (e.g.

short hotel descriptions with low resolution pictures) in the first case versus Web services that provide extensive results (e.g. detailed descriptions with multiple high resolution pictures). Other examples of client contexts that can influence the selection process include end-user properties (e.g. a user with a gold subscription may access Web services that allow the booking of VIP arrangements), and specific other related non-functional requirements (e.g. certain communication patterns such as payments that require stronger security measurements).

6.6.2 Client-Context Monitoring Aspect

Again, a solution based on AOP has the advantage that selection policies can be encapsulated in separate aspects in a flexible manner independently of that client, but more importantly, the client application can be obliviously *observed* by an aspect. A change in the client context can be detected immediately by a monitoring aspect, as depicted in Figure 6.5, and as such, the service selection process can be triggered. This illustrates the usability of AOP for *context passing*.

Identification of joinpoints: Evidently, the monitor aspect has joinpoints in the client application at those places where context changes can be detected. These joinpoints will have to be identified depending on the intended scenario: in the example where the holiday application runs on different systems this will be the moment the system starts up, in the example with different end-user subscriptions it will be the moment a new user logs on, or when the user changes its subscription status. Clearly, our AOP-approach is only capable of capturing context information that is somewhere available in the environment. Note that this is the first time, the link between the client and the WSML (i.e. the service types), is not sufficient to communicate all service-related issues, i.e. the current client-context is not passed along to the WSML. Without AOP, one would need to resort to re-factoring of the object-oriented client code as at those places where a client context transition occurs, additional information should be passed along explicitly to the WSML, for example as an additional parameter in a service type request. However, this is highly unwanted: besides the fact that the client code needs to be changed, it will also result in tangled service selection code in the client, at all places where context-changes may occur. And if later on, the selection policies change, this would result in redundant information being passed along to the WSML. An alternative, such as the Observer Pattern [GHJ95] is no solution for this problems, the code still needs to be re-factored as classes where context changes occur need to become Subjects where Observers can subscribe to.

Identification of advices: As shown in Figure 6.5, the solution presented here maps each client context to a different set of applicable selection policies. If a context transition happens in the client (step 1), the monitoring aspect unloads the corresponding selection aspects of the last context and loads the corresponding aspects of the new context (step 2). For instance, if the client runs on a mobile device, a guideline policy ordering services on size of the results is employed together with an imperative disregarding all services that return results in rich text formatting as the mobile device cannot show these results. When the client runs on a normal computer, price might be the main concern, and therefore an imperative restricting the costs is enforced. This modular approach has as advantage that the selection aspects remain totally independent of the client application and its contexts.

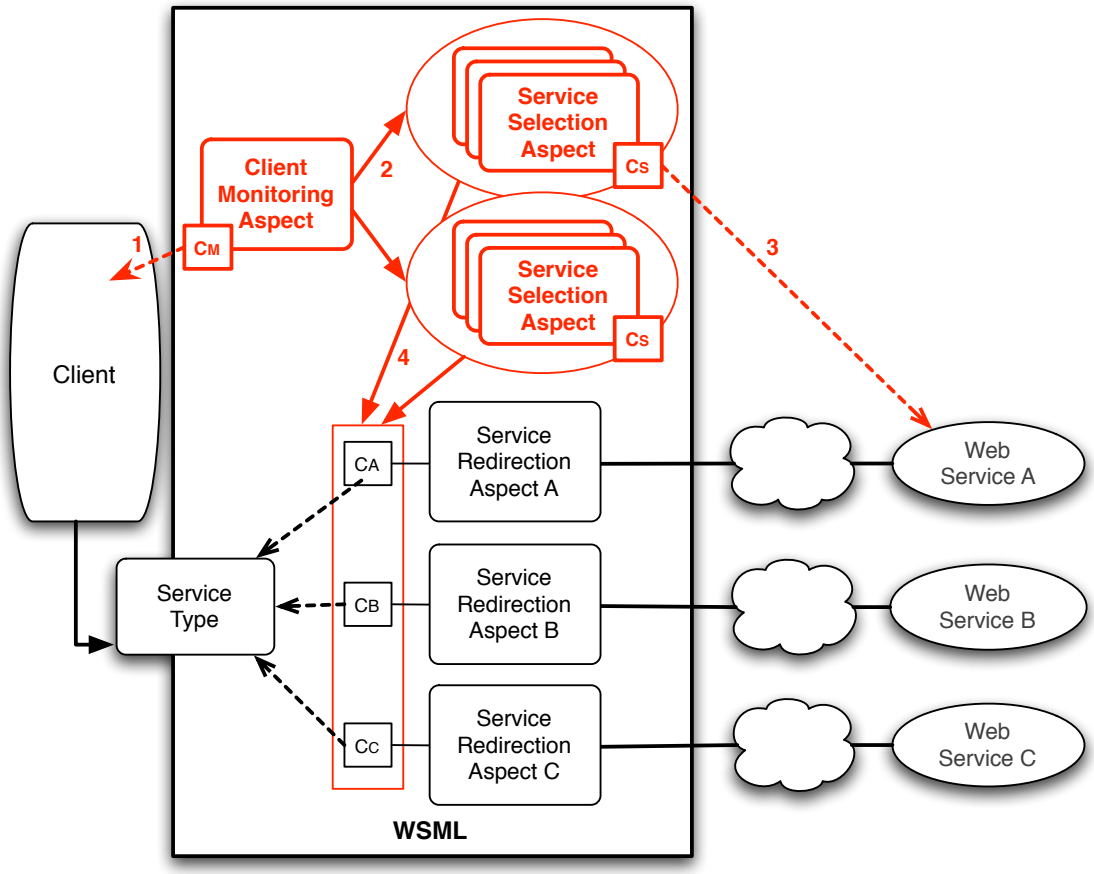

Figure 6.5: Client-Context Monitoring Aspect

The logic to monitor the different client contexts is encapsulated in a separate client context monitoring aspect. As a result, the service selection aspects for imperatives and guidelines as introduced in the QoS context in section 6.4.3, can be reused. In the picture, the service selection aspects are triggered by a change in the Web service environment (step 3), after which the appropriate service is selected (step 4). Examples of other monitoring aspects include aspects that observe the properties of the end-user logged on into the system, and aspects that monitors security-related contexts. If different context observing aspects are deployed at the same time, they impose different selection policies, which can collaborate if they impose constraints on similar services.

The presented solution does not incorporate a strict mapping between the particular Web service and the context of the program. Having an explicit mapping between the context and the concrete Web services would require a revision of those mappings each time a new service is introduced. One way of keeping a more direct mapping is introducing an index reflecting how well a Web service is suited for each client context. One or more monitoring aspects could maintain these indexes.

An important consideration is how the client and the WSML are set up. If we assume the

client and the WSML run on the same host, i.e. on the same virtual machine, the monitoring aspect can run locally. However, if the client and the WSML run in a distributed setup (as depicted in Chapter 4, Figure 4.2), then the joinpoints reside in the client, while the advices are executed in the WSML: in that case, we need explicit support for distribution in our AOP-approach. Few AOP-approaches offer this support, namely D [Vid97], JAC [PSDF01], DJ-cuter [NST04] and JAsCo. With a distributed AOP-approach it is possible that aspects intercept joinpoints executing on other hosts (i.e. distributed joinpoints) and to execute advices on other hosts (i.e. distributed advices). The Distributed JAsCo implementation (DJAsCo), which we employ here, and its corresponding AWED language are described in detail in [BSV06].

In DJAsCo, it is possible to specify remote joinpoints by simply adding `joinpointhost(HostAddress)` in the hook constructor of an aspect. The `HostAddress` is the IP address and an optional port of the remote host. So in this case, the monitoring aspect would run in the WSML, and the client is a joinpoint host. Specifying multiple clients to be monitored is possible using logical operators. In order to execute advice that trigger on remote joinpoints, the joinpoint information is distributed to all interested hosts (in our case from the clients to the WSML). To this end, the JAsCo connector registry (see Chapter 4, section 4.5.4.2) has been adapted to intercept all joinpoints, prepare them for transmission[2] and send them to the remote hosts. Note that in a distributed setup, each host maintains its own connector registry, as a central approach would not scale and become a performance bottleneck. More advanced features of distributed AOP, including aspect distribution and aspect context sharing, are discussed in Chapter 7 in the context of Web services management.

Similar to client-based service selection, it is also possible that the service(s) themselves select the most optimal service. This "service-based" service selection seems to be more appropriate in a more controlled environment than the original setup of this dissertation of having independent third-party services. Nevertheless, a policy could state that a service will distribute its resources evenly over a set of (possibly fixed) clients. In order for the client to make sure this policy is enforced, it requires runtime service information. Analogue to what we discussed above for client-based selection, a dedicated monitoring aspect with remote joinpoints inside the service could be used for this purpose. Alternatives include relying on notifications from the service or on the documentation.

6.7 Related Work

Most related approaches on Web services selection for SOAs is QoS-based, thus covering the same grounds as the selection mechanism described in section 6.4. Most of these approaches suggest to advertise the QoS of the services together with the QoS preferences of the client

[2]The current DJAsCo implementation version 0.8.6 uses Java Serialization to transmit objects from one host to another. To locate and send joinpoint information to other interested remote hosts, the JGroups framework is employed [Ban02]. JGroups is a well-known toolkit for reliable multicast communication. In addition, JGroups supports a wide range of network protocols, which makes the system independent of specific network technologies.

in a common ontology, and to use centralised matchmaking mechanisms to find the most optimal service. Our approach does not focus particularly on proposing new ontology models or matchmaking algorithms, but rather on enforcing selection policies non-invasively in the client for third-party Web services and compositions. Therefore, we will discuss the related work more in terms of how each approach affects the Web services and the client implementation.

6.7.1 QoS-enabled Service Repositories

A first category includes approaches that offer a QoS-enabled service registry or repository. Instead of looking up a service in a UDDI using only a functional description, these registries or repositories also take into account QoS preferences of the client. The *QoS-IC Framework* has been recently proposed in [TBE05] as a centralised repository to store all QoS specifications of services. It tracks changes in QoS properties and notifies the clients in case of changes. The QoS properties are grouped in two groups: obtained QoS properties and computed QoS properties, which map to the documented properties (section 6.4.1) and service behavioural properties (section 6.4.2) of our approach. When service providers publish their services, or when clients look up services in this repository, they must extend their SOAP messages with QoS specifications. The repository models all QoS data in an ontology, manages updates of the QoS properties and notifies clients. A validation manager validates business information and verifies provider's claims of QoS properties. The matchmaker algorithm is based on the Distance Measure Function [Dun02]. QoS-IC relies on UDDI for functional service matching; meaning only services with the same interface can be integrated. Glue code and service compositions are not supported. These limitations do not apply for the WSML. Furthermore, when the QoS-IC framework returns the most optimal service, it still needs to be integrated in the client (e.g. using proxies). Another disadvantage is that clients need to do a service look-up with both a functional and a non-functional part in its body. However, as we argue that the non-functional part tends to evolve faster than the functional part, we have separated the two in our solution: the client specifies a functional request and passes it along to the WSML. Next, the WSML will, while considering non-functional selection policies, address the appropriate service. If necessary, the WSML could still be made compatible with the QoS-IC approach: clients can send their functional request to the WSML, which will add a non-functional part to the message (based on the specified policies) and pass the request along to an intelligent third-party broker to obtain the most appropriate service.

6.7.2 QoS-based Service Selection Frameworks

A second category includes approaches where clients communicate *through* a framework that takes care of the selection process, similar to what we propose with the WSML. The advantage of these approaches is that the service integration is done by the framework, and that this framework *acts on behalf of the client*, but independently from it. In [MS04] an agent-based solution for QoS-based service selection is presented. Agents represent autonomous service clients and providers and collaborate to dynamically configure and reconfigure services-based software applications. Agencies gather QoS data from agents,

and store, aggregate, and present it back to the agents. This approach is made in the *Web Services Agent Framework (WSAF)*. WSAF incorporates service selection agents that use a QoS ontology and an XML policy language that allows service clients and providers to expose their quality preferences and advertisements. An agent exposes the services' interface, while augmenting it with agent-specific methods. These methods are used by the client to specify its QoS preferences prior to using the service's methods. This is similar to the service types in our approach, which also mediate autonomously between the client and the available services. In the WSML, clients can also express their selection policies in a dedicated XML configuration language which will result in the instantiation of one or more selection and monitoring aspects, as shown in Chapter 8. The behaviour of the agents can be changed through scripting, but the paper is not clear on how much flexibility this offers in terms of changing the quality-degree matching process. WSAF relies on UDDI for service matching, resulting again in the limitation that the framework can only address services with the same interface. It uses dynamic proxies for this purpose.

GlueQoS [WTM+04] provides an mediation mechanism to support the dynamic management of QoS features between consumers and providers. The approach provides a declarative language for specifying the QoS feature preferences and conflicts, and a middleware-based resolution mechanism, GlueQoS Policy Mediator (GPM), that reasons using these specifications. GlueQoS is an extension of the WS-Policy [BCH+03] language. They assume fixed ontologies of features and their interactions explicitly and a-priori defined. Both on the client and server side a GPM is responsible for negotiating QoS settings at service integration time, using a policy mediation meta-protocol. The GPM determines a compatible QoS feature composition, when possible, and otherwise declares that the partner's policies are incompatible. This is done at runtime in an open dynamic environment, thus liberating the deployment expert from considering all possible client-server QoS pairings, and certainly also liberating the application developer from tangling application logic with QoS considerations. This promising approach does however not fit the premises of our dissertation as a GPM must be deployed in all Web services in order to work with this approach.

6.7.3 Request/Response Initiated Selection Approaches

Another category of related work encompasses approaches that strive at providing personalised services for a particular context. This relates to our selection mechanism based on client requests and service responses (section 6.6). In [BW03] an approach is discussed where client requests are annotated explicitly with user-specific preferences. The requests are expanded with information found in user profiles and domain knowledge and are matched with Web services that are semantically described. Their approach is mainly targeted at allowing human interaction to inject personalisation in the service selection process and does not deal with ontologies or interoperability issues. In that paper, the distinction is made between hard constraints and soft constraints. Hard constraints cover the required service functionality for a specific client request (e.g. select a flight service that can book a flight to New York next Monday) while soft constraints cover additional requirements that can be relaxed (e.g. prefer a non-stop flight). An algorithm is presented where semantically described services are looked-up through keyword-based searches. Next, the services are evaluated by matching their method signatures with the specified hard constraints, and

afterwards a ranking is made based on the soft constraints. No implementation of the approach is available, so it is unclear to which extend such a context-tailored selection process can be automated without end-user intervention.

Another approach to get personalised behaviour is to make the Web services context-aware by submitting the context inside the request. In [KK04], a context framework is proposed to facilitate the development and deployment of context-aware adaptable Web services. Web services are provided with context information about clients that may be utilised to provide a personalised behaviour. This setup does not match the premise of this dissertation, as it requires adaptations to the third-party Web services in function of the client.

6.8 Conclusions

This chapter discusses our approach for customised service selection in the WSML. Selection policies are decoupled from the client application and enforced through selection aspects. One aspect encapsulates one policy: the triggering points in the environment where changes happen that influence the enforcement of a policy map to joinpoints of the aspects, and the advices contain code to approve, disapprove and/or re-prioritise Web services. A variety of selection policies are discussed, including selection based on Quality of Service, service behavioural properties, service and client context, client requests and service results.

To monitor service behavioural properties and to detect service and client context changes, we suggest employing monitoring aspects. As such, monitoring points can be introduced in the environment non-invasively. In the case of remote triggering points (e.g. in a third party web service or in a remote client), one can apply a distributed joinpoint model. Note that approach is only realistic when both hosts belong to the same organisation. Another solution is to use a notification mechanism such as Web Services (WS)-Eventing or a polling mechanism to detect remote changes and trigger advices accordingly.

Using aspects to implement the selection policies has the advantage that policies are treated as first-class identities. One aspect represents one policy as a logical unit. Even though a policy might need data from various places to execute, it isn't scattered among multiple points in the code. This modularisation makes it easier to implement and maintain policies. Furthermore, aspects can enforce a wide range of unanticipated policies in a unified manner without having to stop the client or rewrite any code. Aspects can generalise many policies in a reusable manner. For instance, consider a policy that states, "Whenever a property changes above a specific threshold, the policy should decide on disqualifying the service." Implementing the policy in an aspect creates a library of reusable aspects that can be easily instantiated in a specific context.

Chapter 7

Client-Side Web Services Management

Abstract Several service management concerns need to be enforced in the client when dealing with Web service invocations. In this chapter we discuss how modularising each concern in aspects helps in avoiding crosscutting code and how each concern can be enforced non-invasively at runtime. Also conditional, meta-level and distributed service management concerns are considered.

7.1 Introduction

To aid in the construction of large-scale distributed systems, many software developers have adopted middleware approaches. Middleware facilitates the development of distributed software systems by accommodating heterogeneity, hiding distribution details, and providing a set of common and domain specific services. However, as pointed out in [CBR03], middleware itself is becoming increasingly complex; so complex in fact that it threatens to undermine one of its key aims: to simplify the construction of distributed systems. Additionally, [ZJ03] describes that the sheer volume of middleware standards and technologies as being a contribution to this complexity. Middleware particularly suffers from increased complexity when addressing concerns of a crosscutting nature.

In case of Web services, service invocations become increasingly complex as additional code is required to deal with various management concerns. For example, the client is required to make a payment before the service can be invoked, or the client needs to authenticate itself, or needs to encrypt all SOAP messages. These concerns are imposed by the service provider and additionally, the client might also deploy some concerns, for instance doing pre-fetching or caching to optimise performance. Many of these concerns cannot be easily modularised and therefore become entangled in the system, thus decreasing understandability and potential for reuse.

Furthermore, it will vary over time which concerns are required to be enforced. Therefore, a flexible mechanism is needed to enforce these concerns non-invasively. In this chapter, we propose a mechanism based on AOP that deals with a variety of service management concerns. We opt to cope with these concerns by modularising them in *management aspects*. Using dynamic AOP, the concerns are only deployed at runtime for those services that require them. While SOAP message handlers (see section 3.5.2.2) can only be triggered when messages come in or are sent out to a service, our approach is more flexible as it benefits from the richer expressiveness available in an AOP pointcut language. Using aspects to implement the management concerns has similar benefits as for selection policies: each concern is cleanly modularised in one aspect, non-anticipated concerns can be implemented in aspects and enforced in an oblivious manner in the client, and code reusability is achieved by generalising the concerns in patterns.

In Chapter 5, we already discussed some concerns, including exception handling, conversational messaging, etc. In Chapter 6 we discussed service monitoring for the purpose of realising a more intelligent selection mechanism. In this chapter, we discuss some other concerns and implement them with a mechanism that builds on top of the service redirection mechanism introduced in Chapter 5. The next section presents some management examples that are modularised in aspects: billing concerns are discussed in section 7.2.1 and global and local caching solutions are presented in section 7.2.2. The subsequent sections discuss more advanced management topics. In section 7.3, feature interaction between multiple concerns is discussed, section 7.4 covers the conditions that may trigger the enforcement of a concern, meta-level management concerns are discussed in section 7.5 and approaches to realise distributed management are the topic of section 7.6. Related work is presented in section 7.7 and we conclude in section 7.8.

7.2 Examples of Management Concerns

7.2.1 Billing

7.2.1.1 Introduction

To get revenues, service providers may charge clients for the use of their Web services. Typical examples of payment schemes are pay-per-use, where the client pays for the actual usage of the service, and subscription models, where the client buys access to the service for a period of time. Several variants including pre- and post billing, micro-payments, charge-per-items, etc. are possible.

Typically, it is up to the service provider to keep track of the payments and charge the client for the correct amount (although dedicated brokerages that take care of billing can be used too). At the client side, a billing mechanism is also required. First of all, it is highly likely that the client must include additional information (such as a client ID) in each of the messages it sends to the service. Next, if an automated payment mechanism is put into place, the client needs to make the actual payment, for instance using a dedicated bank Web service. Which bank service is to be used may depend on the service provider. And finally, in case of post-paid payment models, the client might want to keep track of its usage of the service for auditing purposes to check whether the service provider takes on correct billing procedures.

Billing can become quite complex for the client, as each service provider might enforce different procedures, using different protocols and different third-party bank services. Furthermore, it is possible that the client application will charge its respective customers for the usage of the Web services. The Travel Application of section 3.1 uses a third party hotel service for hotel reservations. If that service charges a fixed fee for each booking, the travel application may charge this cost to its own customers. Depending on the kind of customers this amount may vary: e.g. VIP customers may get a discount. This simple example illustrates that billing can become complex, and may require interaction with several parts of the client, besides the obvious communication with the Web services. As billing is a service related concern, the WSML is ideally suited to deal with it independently from the client application and the concrete Web services used.

Depending on the applicable business model, payments may be *reversed*. Instead of getting paid, the Web services may need to pay the client application to be integrated. The travel agent lists all available hotels it retrieves from the hotel services. Service providers that prefer a higher position in the list submitted to the customers may need to pay for this as it increases the possibility that the customer will use their service. In this setup, the WSML will need to keep track of this payment information, and bill each Web service. It is also possible that the WSML acts as a wholesale broker (see section 4.2.2). In that case, it might offer a specific functionality to its customers for a fixed price, and buy this functionality from third-party services that are in competition with each other.

Finally, we would like to point out that billing becomes even more complex when taking into account Service Level Agreements (SLA). If a service provider describes its Quality of Service (QoS) and an SLA between the client and the service exists, it is not uncommon

that the service provider needs to pay a *compensation rate* if it does not meet its agreed upon QoS. As discussed in section 6.4.2.1, constant monitoring can be used to determine this, for instance by monitoring aspects. Alternatively a third party Web Service Auditor could be used to make sure response rates are in line with what is defined in the SLA.

7.2.1.2 Billing Aspects

A simple billing mechanism, where the client is charged for each invocation of a single service can be straightforwardly implemented using message handlers. A billing message handler is triggered each time a message passes the handler chain. However, in more complex cases with multiple services as described above, this basic triggering mechanism is not sufficient and the expressiveness of an AOP pointcut language is needed. Suppose the Hotel, Flight and Car Services start charging additional booking fees to their clients to compensate for administrative costs. These variable fees are to be paid after a booking is made and a corresponding confirmation is sent to the client. The travel application does not support this natively and only charges its customers for the costs of the purchased items (the hotel booking, flight reservation and car renting). The travel application decides to charge these additional booking fees to its customers, unless they make three bookings in a row. In that case, regular customers get 50% discount if they make three bookings while for VIP customers the fees are completely annulled. The VIP status of customers is internal information for the travel agent and is never communicated to the Web services.

Figure 7.1 shows our solution, using a separate *PayPerUseBillingAspect* to pay the Web services and a *TravelFeeBillingAspect* to charge the customers of the travel agent for the booking fees. The *PayPerUseBillingAspect* is triggered every time a booking is done on a particular Web service, and a corresponding bank service is invoked to make the payment. The *TravelFeeBillingAspect* is triggered every time a customer logs in and makes a booking (no matter what kind of booking) and calculates the applicable booking fee total.

7.2.1.3 Pay-Per-Use Billing Mechanism

First, we discuss the actual payments done from the client to the Web services by the *PayPerUseBillingAspect*. This aspect is shown in Code fragment 7.1: a `billHook` (lines 5 to 24) is triggered whenever a booking is made on a hotel, flight or car service and an after advice makes the payment. The actual bank service to be used may vary for each service; therefore a dedicated method (lines 10 to 12) can be used to set the appropriate bank service. If necessary a *BankServiceType* could be used to avoid hardwiring of the bank service interface in the billing aspect. The amount the aspect needs to pay depends on the billing mechanism of the service:

- **Fixed cost:** The cost is fixed and therefore determined at the moment the Payment mechanism is enforced. For instance, a method `setCost()` can be provided in the aspect; it can be called from the connector at deployment time of a `PaymentPerUseAspect` instance.
- **Fixed cost varying over time:** the cost is looked up each time a payment is to be

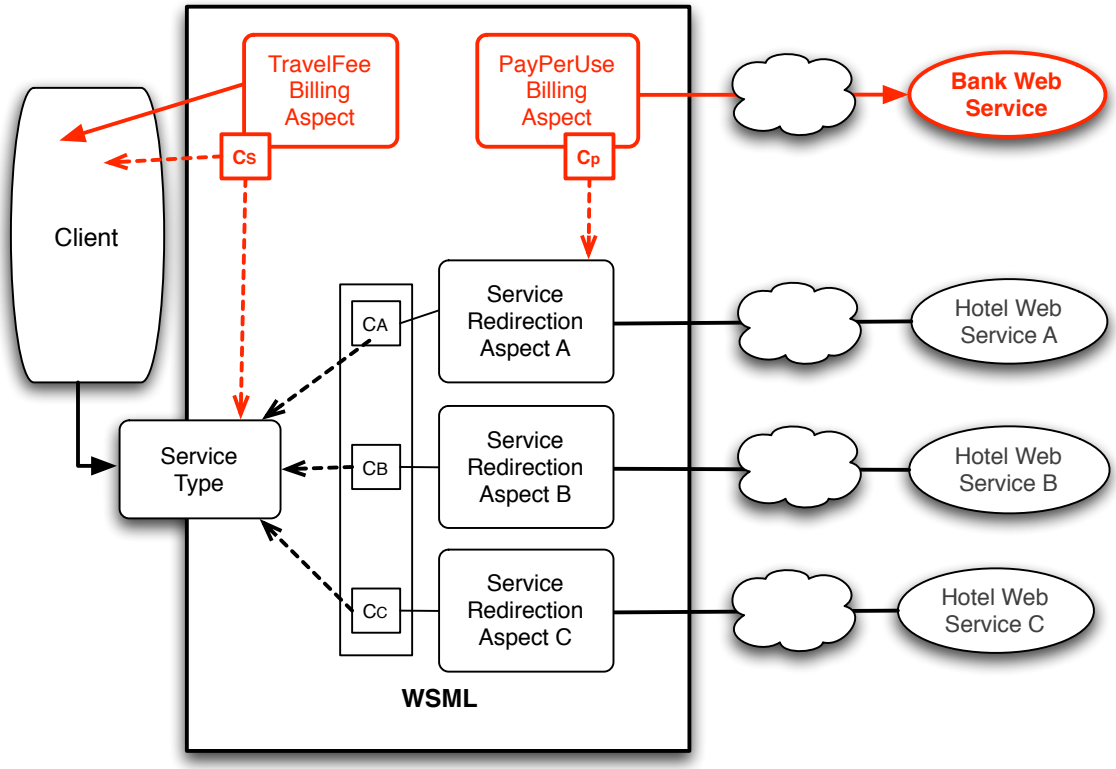

Figure 7.1: Enforcing Service Payment Procedures through Billing Aspects

```
1  class PayPerUseBillingAspect {
2      BankService bankservice;
3      int cost;
4
5      hook BillHook {
6          BillHook (method(..args)){
7              execution(method);
8          }
9
10         public void setBankService (String URL)
11             bankService = new BankService (URL);
12         }
13
14         public void setCost (int newCost)
15             cost = newCost;
16         }
17
18         after() {
19             ((WebService) thisJoinPointObject).getCost(thisJoinpoint.getName());
20             String from = TravelAgent;
21             String to = serviceName;
22             bankService.makePayment (from, to, cost);
23         }
24     }
25 }
```

Code fragment 7.1: Service Payment Aspect for Pay-per-use

made. Changes in the cost can be detected by polling or notification mechanisms, for instance enforced by a monitoring aspect as described in Chapter 6, section 6.4.2.2.

- **Variable cost:** the cost depends on which functionality of the service is invoked and/or how it is invoked. Additional code is required to determine the actual cost. In the Code fragment, line 16 looks up the cost using a dedicated operation provided by the Web service (the argument is the name of the method originally invoked on the service). Alternatively, the costs can be advertised in the documentation, or it can be included in the message containing the service results.

The generic billing aspect, as shown in Code fragment 7.1, can be deployed and customised for all services that adopt the same type of billing. In case the invocation of the web method fails (e.g. due to a service of network failure), an exception is thrown and the after advice of the `BillHook` is not executed. As a result, no billing will take place.

The connector in Code fragment 7.2 deploys the Service Payment Aspect for each booking performed on *HotelServiceA* using the * quantifier. The methods `setBankService` and `setCost` are invoked once at the moment the connector is initialised.

7.2.1.4 Discount Booking Fee Mechanism

Next, we deal with the charging of the fee from the travel application to its customers by credit card. Code fragment 7.3 shows a stateful aspect counting the number of bookings,

```
1 static connector BookPaymentconnector {
2     PayPerUseBillingAspect.BillHook billHook =
3         new PayPerUseBillingAspect.BillHook (* HotelServiceA.book* (*));
4             billHook.setBankService (http://www.NationalBank.be/webService);
5     billHook.setCost(5);
6 }
```

Code fragment 7.2: Billing Connector for Service Payment Aspect

and applying the additional fees the moment the customer of the travel application checks out. A first hook (lines 5 to 11) registers the individual booking fees of each Web service. The second hook (lines 13 to 26) is stateful and registers a logged in customer and if that customer executes three bookings, a discount is given. The third hook (lines 28 to 38) is triggered whenever the customer checks out and an amount is charged on his or her credit card. An around returning advice will update the amount charged to the customer by adding the correct booking fees.

Deploying this aspect enforces payments of booking fees enforced by the Web services in the travel application, while it was not originally designed for it. In our example, the `bookMethod` in the `BookHook` and `DiscountFeeHook` will map to the `bookMethods` provided in the *Hotel-, Flight-* and *CarServiceType*, the login method of the `DiscountFeeHook` will hook on the `travelLogin` method residing in the Travel Agent Application to allow customers to login the system, and the `travelCheckoutMethod` will map to the `travelCheckout` method in the client that is used by customers to checkout and pay for their booked holiday.

7.2.2 Caching

7.2.2.1 Introduction

In environments where resources such as the network bandwidth are limited (e.g. wireless connections) or where communication with Web services is expensive, it can be a good option to cache service results. Instead of actually invoking a Web service when the client requests this, a result stored in a cache is returned instead. Another important criterion to opt for caching is the fact that one of the biggest drawbacks of using Web services is performance (see Chapter 2). Two main types of caching are possible:

- **Output caching:** the results of service requests are stored in a server's cache. Subsequent client requests for the same service functionality with the same parameters will get the values cached on the server without actually invoking the service methods. For instance, Web services deployed in .NET have native support to be equipped with an output cache.
- **Client caching:** the results of service requests are stored in a client's cache. Subsequent client requests for the same service functionality will not be sent to the Web service, but rather, the values cached in the client will be reused. Data that is used often but does not change frequently is a good candidate to be stored in client cache.

```
1  class TravelFeeBillingAspect {
2      private int feeTotal;
3      private Customer customer;
4
5      hook BookHook (bookMethod(args)) {
6
7          after() {
8              WebService ws = WSML.getWebService (thisJoinPoint.getClassName());
9              feeTotal += ws.getProperty(BookingFee);
10         }
11     }
12
13     hook DiscountFeeHook (bookMethod(args), travelLogin (Customer c) {
14         login:execution (loginMethod) > booking;
15         booking:execution(bookMethod)[3] > login;
16
17         after login() {
18             customer=c;
19         }
20
21         after booking() {
22             if (customer.isVIP())
23                 feeTotal = 0;
24             else feeTotal = feeTotal/2;
25         }
26     }
27
28     hook CheckoutHook (travelCheckoutMethod(args)) {
29         CheckoutHook (method(..args)){
30             execution(method);
31         }
32
33         around returning (int result) {
34             total = result + feeTotal;
35             feeTotal = 0;
36             return total;
37         }
38     }
39 }
```

Code fragment 7.3: Billing Aspect for Booking Fees

In the context of the WSML, we are interested in enforcing *client caching*. Enforcing a cache is a service-related concern that should have no impact on the client application. As an illustration, remember our Travel Agent Application wants to collect a description and room availability for a given hotel. The hotel description is rather static and will not change very often while the room availability changes constantly. The client application wants to retrieve both pieces of data, and inside the WSML, a cache can be employed for hotel descriptions, while up-to-date room availability information can be retrieved from the appropriate Web service. In the next two subsections we show two possible approaches relying on *caching aspects*.

7.2.2.2 Global Caching Mechanism

A *caching aspect* works in a similar way as the regular service redirection aspects from Chapter 5 (section 5.2.2), with the difference that it fetches its results from a cache database instead of invoking the Web service itself. When using global caching, one cache database is employed for *all services* that realise the same service type. Figure 7.2 shows a *CachingAspect* deployed in the WSML. When a client request comes in (step 1), the *CachingAspect* is triggered (step 2) and looks up if it has valid results in its cache (step 3). A cached result is valid if it was obtained for an identical client request and if the result has not expired. If a valid result is available, it is returned to the client. Otherwise, the chain of available service redirection aspects is started (step 4). In this particular case, Web service B answers the request and returns a result (step 5 to 10).

Code fragment 7.4 shows a possible implementation of a caching aspect. The after returning advice of the `CacheUpdateHook` (lines 4 to 20) is triggered after a service has been invoked and a result is returned to the client. The advice stores the intercepted result in a cache and returns it back to the client. The `isApplicable` statement (lines 13 to 15) ensures that this is only done when the cache does not already contain a valid result. The `RetrieveCachedResultHook` (lines 22 to 34) defines an around advice that is triggered whenever a cached result is to be returned to the client instead of actually invoking a Web service.

By deploying the aspect above on one or more methods of a service type, one is enforcing a global cache: first the cache is checked whether a valid result is available. If that is the case, the result is returned, otherwise the default service redirection mechanism proceeds, and the obtained result is stored in the cache. This is for example done in Code fragment 7.5 for the `getHotels` method of the *HotelServiceType*. Note that the Payment-per-use mechanism using the *PaymentPerUseAspect* of the previous section continues to work correctly in the presence of the caching aspect: only when no cached value is available the actual service is invoked and the billing is done. This is because both the caching and the billing aspect are deployed on separate joinpoints.

7.2.2.3 Local Caching Mechanism

A *local* caching mechanism is a caching mechanism deployed for each Web service *individually*. The result of a concrete invocation on a particular service is cached so that it can

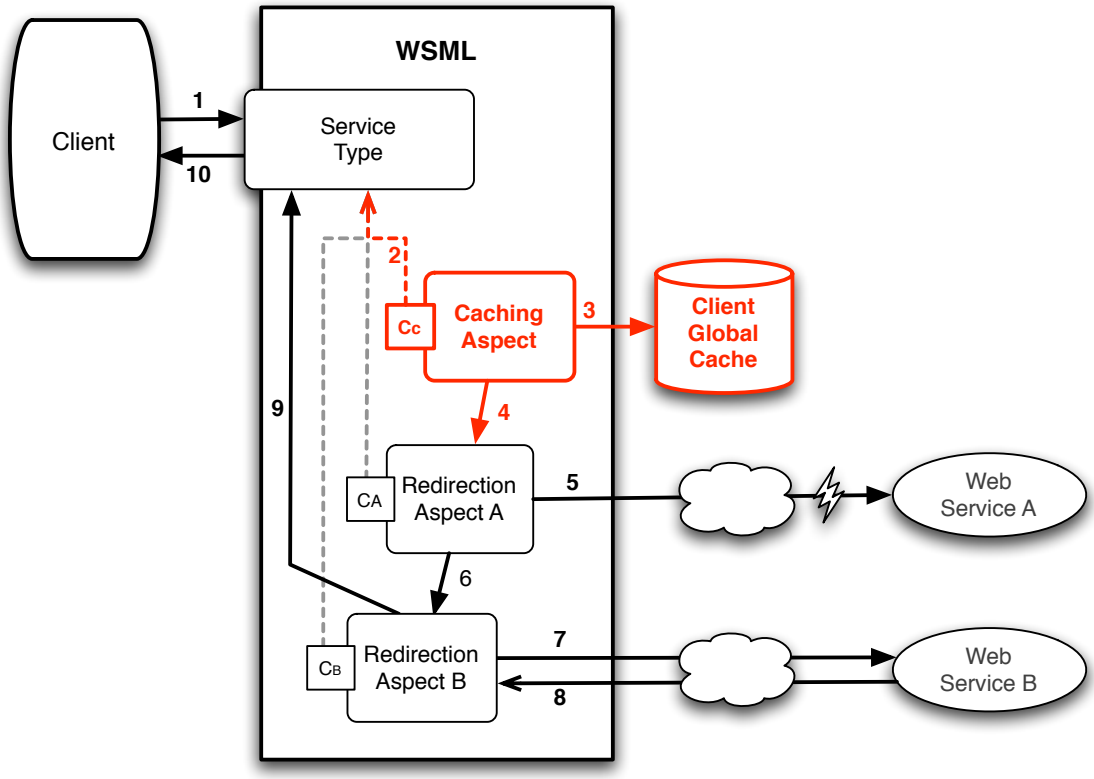

Figure 7.2: Global Caching of Service Results through a Caching Aspect

```
class CachingAspect {
    CacheTable cache;

    hook CacheUpdateHook {
        CacheUpdateHook (method(..args)){
            execution(method);
        }

        public void setExpirationTime(float expTime) {
            cache.setExpirationTime(expTime);
        }

        isApplicable() {
            return !(cache.hasValidResult(method, args));
        }

        after returning(Object result) {
            cache.storeResult(result, args);
        }
    }

    hook RetrieveCachedResultHook {
        RetrieveCachedResultHook (method(..args)){
            execution(method);
        }

        isApplicable() {
            return cache.hasValidResult(method, args);
        }

        around() {
            return global.cache.getResult(args);
        }
    }
}
```

Code fragment 7.4: Service Caching Aspect

```
static connector GlobalCachingConnector {
    CachingAspect.CacheUpdateHook cachingHook =
        new CachingAspect.CacheUpdateHook
            (String HotelServiceType.getHotels(Date, Date, String));
    CachingAspect.RetrieveCachedResultHook cacheRetrieveHook =
        new CachingRedirectionAspect.RetrieveCachedResultHook
            (String HotelServiceType.getHotels(Date, Date, String));
    cachingHook.setExpirationTime(15);
}
```

Code fragment 7.5: Global Caching Connector for the Service Caching Aspect

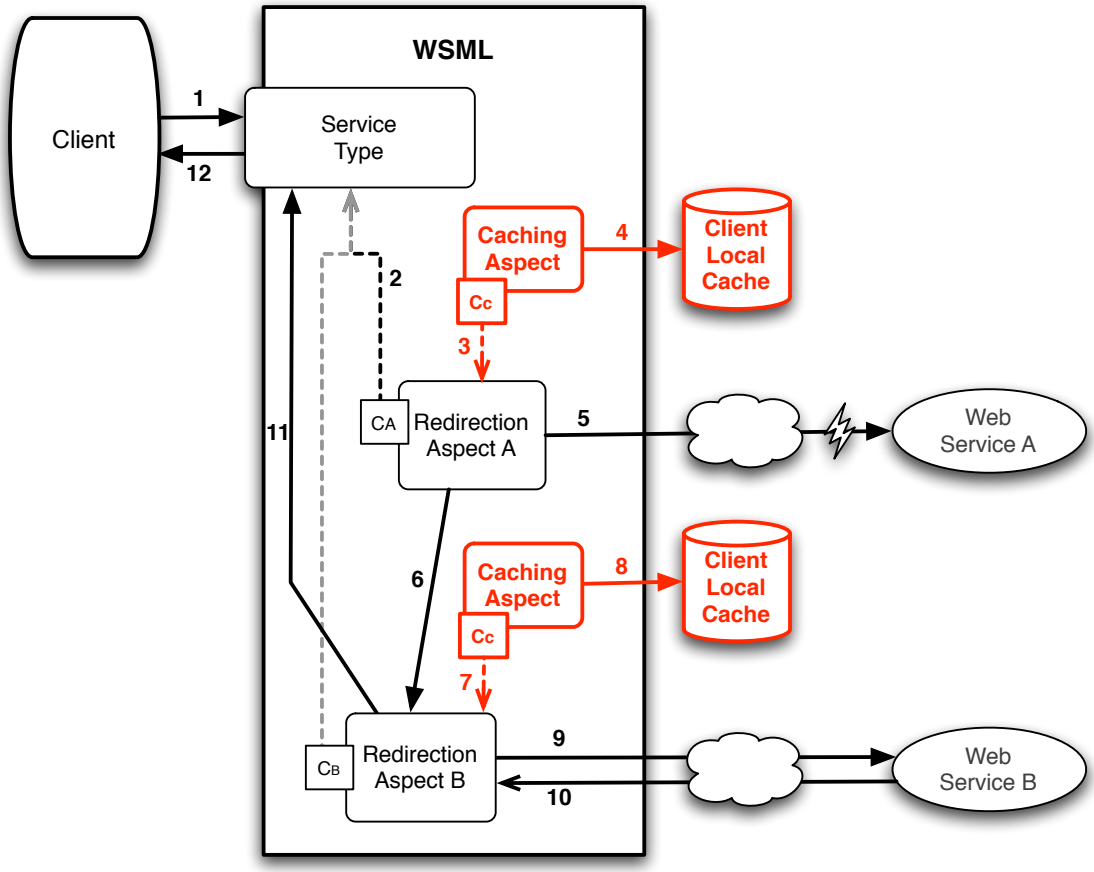

Figure 7.3: Local Caching of Service Results through a Caching Aspect

be reused for future requests. Figure 7.3 illustrates how the local caching works: whenever a service redirection aspect invokes a Web service, the caching aspect checks for a cached result first. If a cached result is available that satisfies the requirements, this value is returned and the service is not invoked. If no valid cached value is available for the request, the web method is invoked and the local cache is updated with the new result. In case the web method invocation fails, the next redirection aspect is triggered. This mechanism can easily be deployed by tweaking the global caching solution: the same caching aspect is used but deployed using a different kind of connector. Instead of hooking on the service type, the aspect must now hook on the method(s) that invoke the Web service.

A problem that arises in this scenario is feature interaction: both the *PaymentPerUseAspect* and the *CachingAspect* hook on the same joinpoint, but a natural conflict exists between the two concerns. This is further discussed in the next section. Another example of a concern that can be used to optimise performance is *pre-fetching* where a service invocation that will possibly take place in the future is already executed to enhance performance from the client-point of view. This is also a crosscutting concern, as it is triggered based on specific client or service behaviours. For instance, if the customer requests the hotel availability for a given period, the system can already pre-fetch available flight seats too.

The presented mechanism also functions for management concerns enforced by the service provider. Assume the *HotelService* provider adopts the WS-Security standard to provide quality of protection through message integrity, message confidentiality, and single message authentication [ADH+02]. The Travel Agent Application may prefer to enforce higher security measurements for its VIP customers and decide that if the logged in customer has indeed the VIP status, a binary security token needs to be encoded and attached to all SOAP messages sent to the *HotelService*. An aspect is ideally suited to implement this behaviour non-invasively, something that is not possible with message handlers, as the customer's VIP status is unavailable context information. As a Web service will employ some advertisement mechanism (e.g. through its documentation) which additional WS* standards are enforced, the WSML can operate a polling or notification mechanism to detect which management concerns need to be deployed. For instance, the mechanism can be used to detect that a Web service changes its encryption level and automatically deploy a dedicated encryption aspect.

In the next sections we discuss feature interaction between multiple management concerns; in section 7.4 we elaborate on conditional management concerns; meta-level concerns are the topic of section 7.5 and approaches for distributed management are discussed in section 7.6.

7.3 Feature Interaction

Multiple management concerns can interfere with each other, especially the ones enforced by the service upon the client and the ones enforced by the client itself. For example, a privacy concern that makes sure no private client information is sent to the services, may conflict with the payment requirements of the Web service. The term *feature interaction* [Za03] is used to reflect how feature combinations affect each feature's ability to function as it would separately. Feature interactions can be complex, subtle, and very difficult to identify. The effect of a feature is positive ($\rangle$ 0) when it acts to satisfy some non-functional requirement, negative ($\langle$ 0) when it prevents other components from satisfying functional or non-functional requirements and zero (0) when there is no observable change. The following are possible interactions that may occur between two features [WTM+04].

- **Orthogonal:** Two features A, B are orthogonal if their combined contribution to requirements fulfilment is exactly equal to the sum of their individual contributions. These features could reasonably be combined together or individually.
- **Complements**: Two features are complementary if their combined contribution is greater than the sum of their individual contributions. It is advantageous to combine these features together but they may be deployed individually.
- **Dependent**: A feature A is dependent on feature B if their combined effect is positive but the individual effect of A is non-positive. Feature A should only be deployed with feature B.
- **Conflicts**: Two features conflict if their combination has a negative effect on the behaviour of the application. The deployment of one feature should exclude (XOR)

the deployment of the other. The decision that an effect is negative is arbitrary but may include effects such as introducing deadlock or putting data in inconsistent states.

- **Prevents**: A feature A prevents feature B if their combined effect is equal to the individual effect of A. The deployment of A excludes B from effecting the system regardless of policy. This is different from conflicting because the effect is confined to the features themselves.
- **Equivalent**: Two features are equivalent if their individual effects are qualitatively the same. There is no need to deploy these features together but remote partners might only support one of them.

The problem of feature interaction also occurs in AOP, as multiple non-orthogonal aspects can be deployed at the same time in the same application. Several approaches have been proposed in order to make the composition of aspects more explicit, examples are Strategic Programming Combinators [LVV03] and treating aspect composition as function composition [WKL03]. In our framework, a programmatic approach, called *aspect combination strategies,* is used. As an example, assume the *PaymentPerUseAspect* and the *CachingAspect* are deployed locally on *HotelServiceA*. Both aspects are specified on the same joinpoint and will conflict with each other. To ensure that billing is not performed when the result is retrieved from the cache, several approaches can be employed. As each aspect implements the crosscutting concern *independently* we do not want to explicitly tangle code dealing with conflicts in one or more aspect implementations. Therefore, we prefer dealing with the feature interaction on the deployment level. As most AOP approaches have an explicit or implicit deployment descriptor, it is a good idea to deal with possible conflicts there. For instance, in JAsCo connectors one can specify precedence for multiple aspect advices, and as such specify explicit ordering of advice execution in a single connector. Additionally, more advanced control is introduced by aspect combination strategies. As we already briefly discussed in section 6.5.1.4, aspect combination strategies can be used to specify how interfering aspects should cooperate. Making this possible in an AOP implementation can be done by introducing new keywords in the language: for instance, adding a new connector keyword *exclude* which specifies that aspect A excludes aspect B. However, as discussed in [Van04] other aspect combinations require additional keywords and it seems impossible to be able to define all possible combinations in advance. A more flexible and extensible system is proposed in JAsCo that allows defining a combination strategy using regular Java. A `CombinationStrategy` interface is introduced that needs to be implemented by each concrete combination strategy. A JAsCo combination strategy works like a filter on the list of hooks that are applicable at a certain point in the execution. A predefined strategy is the exclusion strategy, which we need in the case of the *PaymentPerUseAspect* and the *CachingAspect*. This combination strategy, as deployed in Code fragment 7.6 ensures that the advices of the billing hook are excluded when the advices of the caching hook are applicable. As a result, the billing advices are only executed when the concrete Web service is invoked and not when the result is returned from the cache.

Both the *conflicts*, *prevents* and *equivalent* interactions can be solved in a similar way. Other combination strategies include the `TwinCombinationStrategy` [Van04] making sure that whenever one aspect is deployed, the other one is deployed too. This can enforce the *depends* feature interaction.

```
1  static connector LocalCachingAndBillingConnector {
2
3      CachingAspect.CacheUpdateHook cachingHook =
4          new CachingRedirectionAspect.CacheUpdateHook
5              (String HotelServiceA.getHotels(cityCode, beginDate, endDate));
6
7      CachingAspect.RetrieveCachedResultHook cacheRetrieveHook =
8          new CachingAspect.RetrieveCachedResultHook
9              (String HotelServiceA.getHotels(cityCode, beginDate, endDate));
10
11     PaymentPerUseAspect.BillHook billHook =
12         new PaymentPerUseAspect.BillHook
13             (String HotelServiceA. .getHotels(cityCode, beginDate, endDate));
14
15     ExcludeCombinationStrategy excludeBilling =
16         new ExcludeCombinationStrategy(cacheRetrieveHook, billHook);
17     addCombinationStrategy(excludeBilling);
18 }
```

Code fragment 7.6: Local Caching and Billing Connector for Service Caching Aspect

While advice precedence and aspect combination strategies prove to be sufficient in most cases, they have the drawback that removing or adding an aspect involves rewriting and recompiling the connector and re-instantiating the aspects. In some situations, this expressive way of specifying aspect execution order is not required: a simple ordering of the connectors is enough. Furthermore, in some cases it is more important to instantiate aspects in separate connectors in order to allow for a fast addition, removal, enabling and disabling of individual aspect instances, as it is the case in the WSML. To facilitate the management of these connectors, *connector priorities* and *connector combination strategies* were added to JAsCo. Connector priorities control the execution order of advices that are instantiated in separate connectors and defined on the same joinpoint. This feature was already used to implement service selection guidelines (see section 6.4.3.4) where they were used to make a preference ranking of service redirection aspects. Connector combination strategies on the other hand, make it possible to filter the list of all connectors at each encountered joinpoint. They allow for the addition and removal of aspects independently of each other in a much easier way while it is still possible to express aspect interactions in a modular fashion. While all the approaches for feature interaction can be applied at once, experience has proven that this complicates matters, as it becomes very difficult to reason about how multiple aspects relate with each other and how they will be executed.

7.4 Conditional Management Concerns

Modularising crosscutting service management concerns in aspects has the advantage that an expressive pointcut language can be used to deploy the concern where needed. Otherwise said, the developer has large control over how, where and when the concerns are enforced. Examples of enforcing the caching at the level of a service type and at the level of an individual Web service were given. It can also be possible to enforce the concern at the composition level, i.e. only when that composition is executed, the concern is triggered. In

general, management concerns can be deployed on any of these three levels:

- **Service type level:** the concern is triggered before, around and/or after one or more service types are invoked.
- **Service composition level:** the concern is triggered before, around and/or after one or more service compositions are invoked.
- **Web service level:** the concern is triggered before, around and/or after one or more Web services are invoked.

While this already offers a reasonable level of fine-grained control over the deployment of the concerns, it is possible that additional constraints are to be enforced. For instance, it could be desired to deploy a caching mechanism on a specific service type, only when the execution time of the available Web services becomes too long. For this, we need even more fine-grained control over when the caching aspect will be triggered. In AspectJ [KHH+01], an *if pointcut designator* can be used to test some condition. In JAsCo, the `isApplicable` condition in the aspect hooks is the most obvious location to add additional constraints. In this case however, the condition resides in the aspect code together with the crosscutting concern. To promote code reusability, aspect inheritance can be used. The main logic can reside in the super aspect, while a variety of conditional solutions can be implemented in sub aspects. Another way to implement a more generic conditional caching aspect is using an abstract method in the `isApplicable` and refining it in the connector or using a façade interface for implementing the conditional check. Still, the AspectJ approach where the condition is part of the pointcut specification is more favourable.

A less implementation-oriented approach is treating the conditions as business rules [Buss00]. As discussed earlier when introducing selection policies (see section 6.3), business rules are intended to assert business structure or to control the behaviour of the business, so the statement to only enabling a cache when services become too expensive or too slow can be treated as a business rule. As argued before, it is crucial to separate them from the core application and externalise them. Cibrán et al. [CDJ06] introduce a higher-level business rule language to specify business rules, after which they are translated into, possibly aspect-oriented, code.

7.5 Meta-level Management Concerns

The management concern examples discussed up until now somehow all interfere with the control flow from the client to the Web service(s). Either some billing functionality is added when a service is invoked or, a service invocation is replaced by a look-up in a cache. Other management concerns however, work on the meta-level, or on a mixture of both levels. These three categories are depicted in red in Figure 7.4. The top half of the picture shows the functional level of the WSML with a service type, redirection aspect with corresponding connector, and a proxy to invoke a Web service (i.e. the entities we have been discussing up until now). In the WSML however, each of these entities is represented by an instance of a corresponding meta-level class. These classes are responsible for generating and deploying

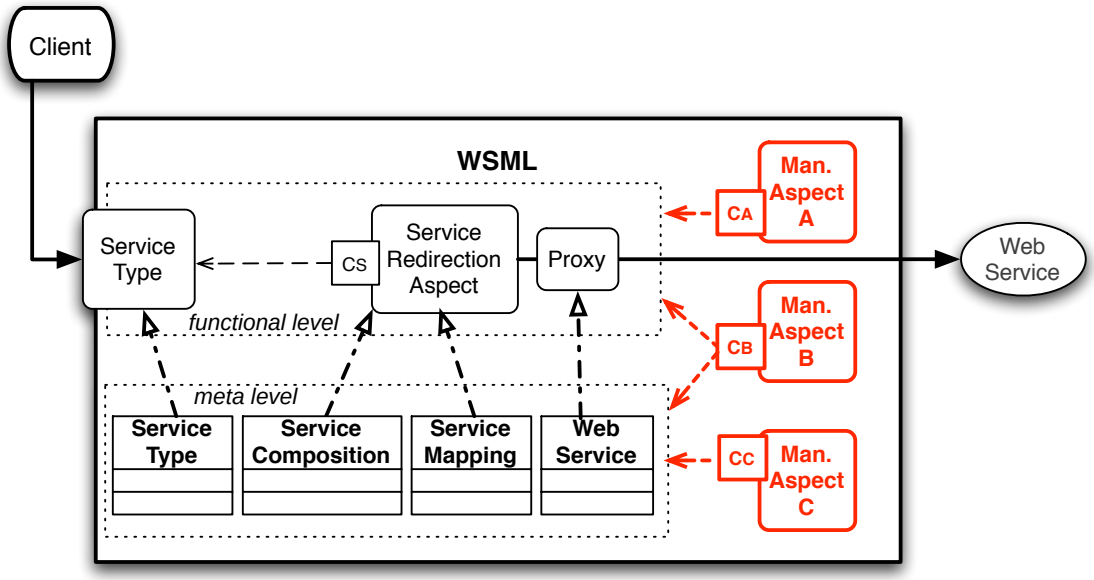

Figure 7.4: Functional and Meta-level for Joinpoints

the functional entities (something we will discuss in more detail in Chapter 8, section 8.4.2) and to store meta-data for those entities (e.g. the properties of a Web service are stored in a `WebService` instance).

We distinguish three kinds of management concerns, depending on where their joinpoints reside and where the advices execute. First, we have the functional aspects (category A) that are triggered by events at the functional level and have advices that are also specified on the functional level. Examples include all management concerns discussed earlier in this chapter. A second group of aspects include the hybrid aspects that work on both levels (category B). For example, the service monitoring aspects of Chapter 6 that measure the execution time of a service are triggered at the functional level, but they store their results on the meta-level (inside an instance of a `WebService` class using *mixins*). Service selection aspects also belong typically to category B. For instance, a selection aspect is triggered by an event on the meta-level (e.g. a change of a service property) causing a rearrangement of the service redirection aspects on the functional level. Finally, aspects in category C include meta-level aspects, whose joinpoints and advices both reside at the meta-level. An example includes a price composition aspect, which calculates the price for a service composition based on the price of the individual services it is composed of. This aspect will be triggered whenever the price of one of the services changes and set the corresponding meta-data of the service composition.

An important consideration is that in some cases there is no symmetrical quantification possible on both levels. For instance, hooking on the Web service invocation of *HotelServiceA* can be done by making the methods of the corresponding proxy as joinpoints. But hooking on the moment when the price of *HotelServiceA* changes (which is on the meta-level) means only one particular instance of the `WebService` class will need to act as a

joinpoint. An example of this quantification issue occurs in Chapter 6, Code fragment 6.3[1] for a monitoring aspect as one hook hooks on the functional level and the other one on the meta-level. Not all AOP languages support hooking on instances. In that case, the aspect will be triggered for any instance of the class and the aspect should check if it is the right one, causing a performance overhead.

7.6 Distributed Management Concerns

Depending on both the setup scenario of the WSML and the service environment, it can be useful to deploy management concerns in a distributed manner, meaning they are not merely enforced for and by the client application. For instance, in Chapter 6, section 6.4.2, service monitoring was defined as the client-side process of monitoring the behaviour and/or performance of the available Web services in the service environment from the client point of view in order to select the most optimal one. The monitoring concerns described in that section were enforced locally in the WSML by aspects. In this section, we discuss two solutions. Firstly, by introducing distributed joinpoints it becomes possible to setup measuring points in the services themselves. This is for instance useful to realise a more fine-grained measurement of the service execution time. By installing joinpoints on the moment a message arrives at the service, and the moment when a message leaves the service, it becomes possible to make the distinction between the service processing time and the network transmission time. Distributed joinpoints can also be useful for logging, support for business activities, load balancing, etc. Note that this requires code to run on the service provider platform. In the WSML prototype it is required that JAsCo is deployed on every remote host where joinpoints are to be specified. Also, in order to cope with network failures and package losses, a reliable communication protocol such as WS-Reliable Messaging [BBC+05] is needed.

Secondly, aspect advices can also be made distributed, meaning the advice is executed on one or more remote hosts. For instance, the distributed monitoring aspect with measurement points in the service could trigger an advice running on the service side to notify the provider in case of slow downs in the service performance. Clearly, these kinds of distributed AOP solutions create a tighter coupling between Web services and clients, which was initially what we wanted to resolve. However in an intranet solution, where services are deployed in a Enterprise Service Bus (ESB), AOP can be very suited to modularise concerns that crosscut service boundaries. Another scenario where distributed advices are useful is replication: in a large setup, multiple instances of the WSML may be required to serve a wide range of clients. In that case, management, selection and monitoring aspects may need to be synchronised with each other.

[1] In that code example, this issue does not have any ramifications: the monitored properties are added to each Web service instance on the meta-level through the `IntroduceHook`. Only those Web services that are actually monitored through the second Monitorhook on the functional level, will have actual values stored for those properties.

7.7 Related Work

A lot of research is going on in the Web service context and numerous vendors are currently working on dedicated Web service management platforms. However, most of these approaches focus on the server-side management of Web services. They allow developers to build and deploy Web services and also provide management capabilities such as load balancing, concurrency, monitoring, error handling, etc. Our approach provides support for the client applications that want to integrate and manage different third-party Web services.

A wide range of application servers is currently being extended with aspect-oriented support. As such, crosscutting concerns can be more easily implemented and deployed on the server. *Java Aspect Components (JAC)* [PDS+04] provides distributed and dynamic AOP programming. *AspectWerkz* [Boner04] has been integrated in various application servers. *JBoss AOP* [BB03] is a Java-based aspect oriented framework that can be used in any programming environment or integrated in the JBoss application server. *Lasagna* [Truy04] is aspect-oriented middleware for context-sensitive and dynamic customisation of distributed services. *PRISMA* [PRJL04] is a conceptual model which enables the description of distributed software architecture by combining CBSE and AOSD. The *PROgrammable extenSions of sErvices system (PROSE)* [PGA02] is a dynamic weaving tool that allows inserting and withdrawing aspects to and from running applications. *MIDleware Adaptive Services (MIDAS)* [PAG03] is a system based on PROSE that allows applications to self-organise into spontaneous information systems, but without relying on a fixed infrastructure. *Spring* [JHA+05] is a layered Java/J2EE application framework. A central focus of Spring is to allow for reusable business and data access objects that are not tied to specific J2EE services. Such objects can be reused across J2EE environments (web or EJB), standalone applications, test environments, etc. without any hassle. An extensive evaluation of these approaches, including the WSML, can be found in [LPP+05]. This report compares these approaches on three criteria, *flexibility*, *reliability* and *performance*. Flexibility is the ability of a platform to accommodate large scale customisation, configuration and extension both statically and dynamically; reliability is the ability to eliminate inconsistencies that might be inadvertently introduced by such customisation; and performance is the ability of a platform to perform its tasks with adequate speed and minimum consumption of resources.

At the time this research started, the idea of applying AOP concepts explicitly in the context of Web services concerns was quite innovative and thus not many approaches had been proposed focusing on this combination. Arsanjani et al. [AHM+03] have also identified the suitability of AOP to modularise the heterogeneous concerns involved in Web services. They refer to approaches like AspectJ and the Hyper/J which, in contrast to JAsCo, only allow static aspect weaving. As a challenge for the Web service research, they also identify the need to create software that meets constraints beyond simple functional correctness, in order to satisfy service level agreements. Thus, Web services need to include both conventional functional interfaces and non-functional interfaces that permit control over performance, reliability, availability, metering, auditability and level of service. Recently, a specific Aspect-Oriented Framework for Web Services [AoF4WS] supporting on-demand context-sensitive security has been presented in [MMN+06].AoF4WS argues that flexible security schemes are needed in many Web services applications and that security mecha-

nisms need to be customised to the continuously changing requirements of Web services. The AoF4WS uses aspect-oriented programming and frames. Aspects provide flexibility to the framework, and frames adjust aspects to specific requirements.

Filman et al. [FBL+02] propose dynamic injectors to introduce aspects into a given component configuration. They incorporate dynamic injectors into OIF (Object Infrastructure Framework), a CORBA-based system for distributed applications. Technically, OIF generates enhanced CORBA stubs and skeletons that are able to incorporate one or more dynamic injectors. This allows for the dynamic injection of independent behaviours on both sides of the communication path between system components, novel communication channels among injectors and between such injectors and the application itself. The main purpose of OIF is to enforce *ilities*, such as like reliability, availability, responsiveness, performance, security, and manageability in the system. In WSML, we could support similar dynamic injectors by implementing a dynamic SOAP extension handler as well. This would present the advantage of integrating better with the existing Web Services infrastructure and would not require dynamic byte-code weaving (as JAsCo). Adopting an AOP approach, however, allows us to exploit an expressive language for selecting joinpoints and executing advices, which would need to be implemented manually with the SOAP handler approach. Another major advantage of AOP is that the joinpoints can identify any kind of method call or execution, even those within the client application and the WSML framework.

Aspects are also employed in other middleware approaches. DADO [WJD03] exploits aspects to add security, performance monitoring, and caching examples to CORBA based applications. Duclos et. al. [DEM02] shows how aspects can be used to provide security, transactional semantics, and object persistence to applications using a CORBA Component Model. It is also worth mentioning that the use of aspect-like mechanisms for security and transactions, particularly, is not without controversy [Gar03, KG02, WTM+04].

7.8 Conclusions

In this chapter we have extended our framework with support for various client-side service management concerns. Examples of concerns that were modularised in aspects have been given, including billing, caching and broadcasting. One aspect encapsulates one reusable concern. A library of reusable aspects that offer a wide variety of management concerns is envisioned: these aspects can be instantiated in a concrete deployment context by means of a separate connector. In the next chapter, we discuss how this step can be automated, hiding away any aspect-oriented details from the administrator of the system.

Using aspects to implement the management concerns has the same advantage as using aspects for selection policies and monitoring concerns: each concern is treated as a first-class identity, cleanly modularised in a single logic unit even though a concern might be triggered by various triggering points and may have an effect on multiple places in the system. Non-invasive runtime enforcement of the concerns becomes possible when a dynamic AOP technology is employed. Also, context passing becomes much easier as triggering points can be set up at those places where the context needs to be retrieved without having to change any code.

Using the expressivity of an AOP pointcut language, fine-grained control over where a management concern needs to be deployed, becomes possible. Concerns can be deployed at the service type, composition and individual service level. Additional conditions can be added by sub-classing aspects, or by externalising these conditions as business rules. In case several concerns interfere with each other, their interaction can be made explicit through various aspect combination strategies. At the moment these strategies need to be provided manually, meaning that if they are not put into place, interference can occur. Finally, aspects are also suited to deal with distributed concerns, where either or both the joinpoints and the advices are distributed. Distributed joinpoints and advices are useful in scenarios for distributed monitoring, logging, load-balancing, replication, etc. As such, AOP can be very suited to modularise concerns that crosscut service boundaries.

Chapter 8

Development and Deployment of a Prototype

Abstract This chapter starts with an overview on how an implementation can be made for the WSML aspects modularising service redirection, selection and client-side management concerns. Several options including automatic aspect code generation through semantic matchmaking, high-level service composition specifications and aspect libraries are discussed. Next, a prototype of the WSML framework, implemented as a proof-of-concept in Java and JAsCo, is presented. Its architecture, the provided tool support and the realise development quality attributes are discussed. As a case study, the WSML prototype has been integrated with the Service Enabling Platform (SEP) of Alcatel Bell, a provisioning system for broadband Internet applications.

8.1 Introduction

In the previous three chapters it was shown how aspects are used to realise dynamic service integration, compositions, selection and client-side management. Writing, compiling and deploying these aspects for Web service concerns is a process that can be largely automated. As a proof-of-concept, a prototype implementing the proposed WSML mediation framework has been developed in Java and JAsCo. This is the main topic of this chapter. First, we discuss in the next section how a client application such as the *Travel Agent Application* can benefit and make use of all three aspect categories at once. In section 8.3 we answer the question how an aspect implementation can be obtained. We discuss the options of manual implementation, automatic code generation and the usage of aspect templates. The WSML plays an important role in this process as it is the framework containing, managing and enforcing these aspects in the client. In section 8.4 we discuss the prototype of the WSML, implemented in Java and JAsCo and the realised and envisioned tool support for our framework. In that section the role of the WSML in the development process of the client will be discussed as well and we list how the overall pursued WSML development quality attributes have been achieved. In section 8.5 we discuss how the WSML prototype has been deployed on the Services Enabling Platform (SEP) to enable third-party service provisioning in a broadband context. This was done for a mid-term and end-term review demonstrator of the IWT Mosaic project in cooperation with Alcatel Bell.

8.2 Travel Agent Example

Each of the three previous chapters focusses on a particular subset of service related concerns a client application has to deal with when integrating with one or more Web services: service integration and composition was discussed in chapter 5, service selection in chapter 6 and additional management concerns in chapter 7. In an actual setup, all of these concerns will come together, as shown in an example in Figure 8.1. The figure depicts the travel agent client on the left side, requesting holiday information to a *HolidayServiceType*. These client requests are either redirected to one of the available *HolidayRedirectionAspects* that communicate with *HolidayServices*, or they are redirected to a *HolidayCompositionAspect*. A *HolidaySelectionAspect* is employed to make that selection. Additionally, a *HolidayFallbackAspect* is used to capture any exceptions that arise somewhere in the service invocation process, and to redirect the client request to another service. A *HolidayMonitoringAspect* monitors the speed of the individual *HolidayServices*.

The *HolidayCompositionAspect* composes *Hotel-*, *Flight-* and *CarWebServices* into a composition. This is done by referring to dedicated service types instead of hard-wiring concrete services into the composition. The *HotelServiceType* is equipped with a *CachingAspect* to store service results, and a *LoggingAspect* to keep a log of all communication with *HotelServices*. Finally, the *CarServiceType* has another *SelectionAspect* specified on it, to select the most appropriate *CarService* for any request. Note also that the client can also directly invoke the *HotelServiceType* and the *FlightServiceType* if that is necessary.

The various aspects, as shown in the example above, modularise a wide variety of service related concerns. They are either very specific aspects, such as service redirection

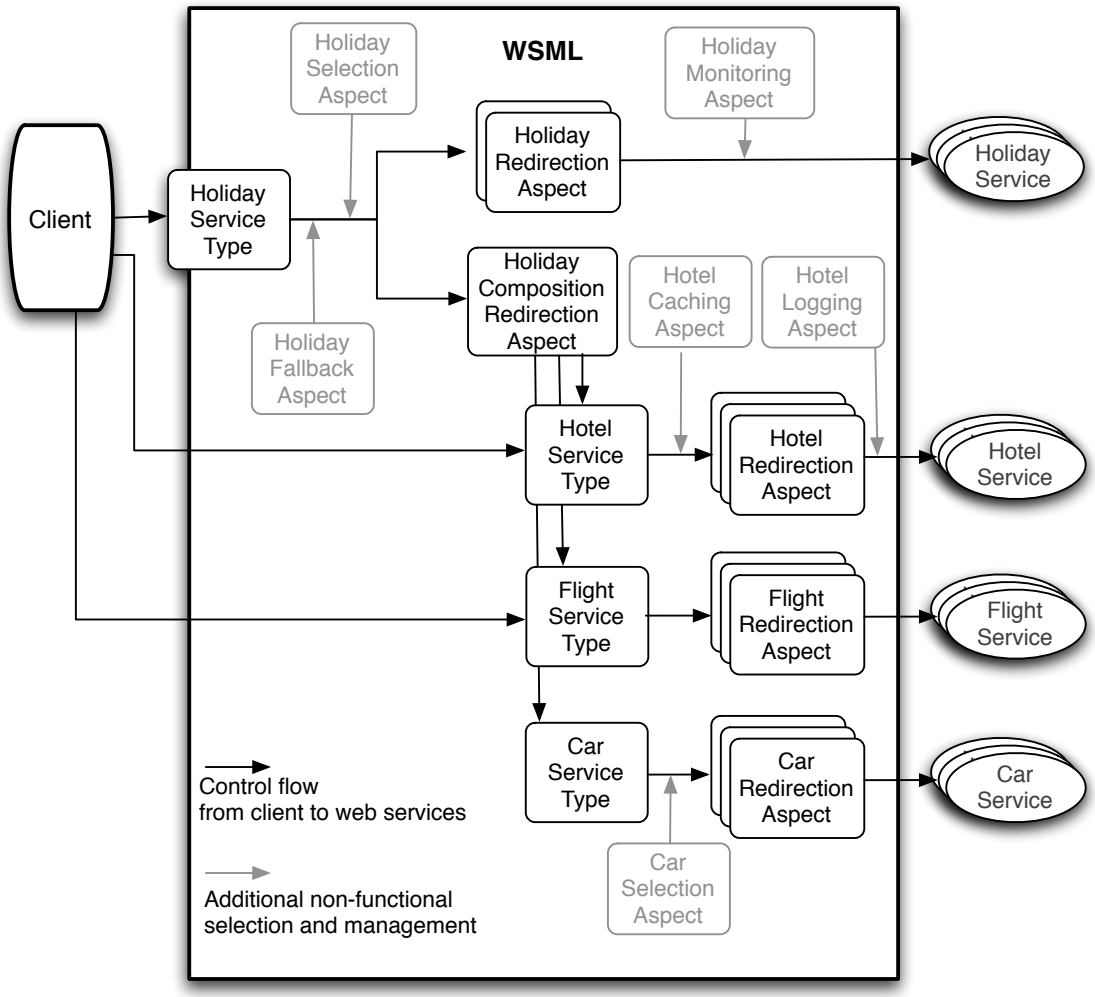

Figure 8.1: Possible Scenario for Travel Agent Application

aspects that modularise communication details for concrete services, or they are reusable aspects, containing for instance some generic caching logic. All these aspects reside in the WSML framework. Its main purpose is to enforce all the service related concerns for one or more clients. For the administrator of the WSML the usage of aspects as an implementation technology of the WSML should be hidden and enforcing a concern will be done on administration level rather than on implementation level. Before discussing the WSML framework and its prototype, we give an overview of the various approaches to implement the WSML aspects.

8.3 Implementation of WSML Aspects

8.3.1 Overview

We distinguish three possible scenarios to obtain aspect implementations and their corresponding deployment descriptors, i.e. the connectors:

Manual Aspect Implementation

A first approach is to implement all service related aspects manually alongside the client code. The most rudimentary tool support needed is a dedicated aspect compiler and a tool to weave in the aspects in the base code. However, more advanced tool support is desired as writing and deploying aspects can become complex. First of all, the aspect code as shown in the previous chapters are over-simplifications to focus on the AOP concepts explained. But the reality is that the actual code is more complex. For instance, the caching aspect shown in Chapter 7, Code fragment 7.4, contains 28 lines, while the caching aspect as deployed in the WSML prototype contains 4 times as many lines. A second issue is debugging. Traditional runtime debugging tools referring to a specific line in the applications' code will refer to the wrong line as the base implementation has changed by the aspects woven into the code. An example of dedicated AOP tool support is the JAsCo plugin for the Eclipse IDE. This tool allows for the generation of aspect and connector skeleton code and doing in-depth analysis of the deployment of the aspects: for instance it is possible to do introspection on a joinpoint to see which aspects are deployed on it. Reducing aspect implementation efforts can be done in two ways: automatic aspect code generation, and aspect reuse through configuration of aspect templates.

Automatic Aspect Generation

Complete automatic generation of aspects and connectors is only possible in specific cases. For instance, automatically generating a completely functional caching aspect out of higher-level cache specification is a research topic on its own, and is something that is the subject of Model Driven Architecture (MDA) approaches. On the other hand, a higher-level description of a mapping between a service type and a web service, or even a service composition can be automatically translated into a service redirection aspect and a corresponding connector (as shown in Chapter 5). Automatic code generation of service redirection aspects can be realised in many ways, as we will discuss further in subsection 8.3.3 where we present our approach on doing automatic service mappings by semantically annotating both service types and Web service interfaces. Specifying high-level service

compositions in UML and making a translation into aspects is discussed in section 8.3.4.

Configuration of Aspect Templates

A third option is to provide a generic set of reusable aspects, implementing concerns such as caching, selection policies, logging and monitoring, and instantiate those aspects in a specific context by supplying additional configuration details, e.g. deploying a cache of a certain size, with a specific expiration time on a concrete Web service. By accompanying an aspect implementation with a deployment descriptor, automatic code generation for connectors can be done to deploy the aspect in a specific context. We discuss our approach to realise a template library for service selection and management concerns in subsection 8.3.5. But first, we discuss the most rudimentary form of code generation, i.e. generation of aspect skeletons.

8.3.2 Aspect Skeleton Generation

As hinted above, for service redirection aspects, we opt for automatic generation of the aspect code and their corresponding connectors. Remember from Chapter 5, that a redirection aspect links a service type with a Web service. This was depicted in detail in Figure 5.6. To generate an aspect implementation, the following is needed:

- **Web Service Proxy:** to communicate with the Web service, the aspect will reference a proxy to that service. The proxy can be generated using a WSDL2Java tool, as discussed in section 3.3.2.1
- **Service Mapping:** A hook is generated for each service type method. Each of the hooks contains a constructor and an around advice, where the code to invoke the Web service resides. This advice contains some pre-processing, the invocations, post-processing and a return statement. The pre- and post-processing are optional.
- **Service Composition:** In case of a composition, the advices will contain invocations to multiple web services and/or service types and contain a workflow on how to pass on results from one service to another and/or how to combine all service results in one single return statement.
- **Conditional Binding:** An optional statement can be used to determine the applicability of a service.

Given a service type specification and a URL to a WSDL file, it is possible to generate the skeleton of the aspect and the connector. The advice bodies will be missing from the generated code, as these contain the mappings or the composition. Generating code for the actual service mapping is more difficult as it requires semantic understanding to map a client request to a service invocation. To generate a composition, we somehow need to have a composition specification. The easiest way is to provide the mapping or the composition as a Java implementation: an aspect skeleton is automatically generated, and the service mapping or composition is provided by a programmer in Java. Alternatively, the mapping or composition could be provided as a series of *statements* that can be translated into Java

code. A statement can be a web service invocation, a service type invocation, the definition of a variable or a custom statement (i.e. hand-written JAsCo code). Based on this series of statements, automatic code generation of the aspects and connectors can be done.

8.3.3 Semantic Matchmaking for Service Mappings

Completely automating the service mapping is only possible when both the service type and the Web services are semantically annotated. A semantic description of both parties helps in determining whether a `bookHotel` method in a *HotelServiceType* means the same as a `makeRoomReservation` method in a *HotelService*. As recognised by the Semantic Web community [BFD99], the WSDL service descriptions are not expressive enough and thus it is required to enrich these descriptions with semantic information [MPM+05]. We have done experiments by adopting domain-specific ontologies, which constitute the base for the documentation of the semantics of both service types and Web services. We have chosen ontologies defined in the OWL language [BSP+01] since it supplies web service providers with a core set of mark-up language constructs for describing the properties and capabilities of their web services in unambiguous, computer-interpretable form. This language is suggested for standardisation by the W3C. Domain specific ontologies contain the definition of all the concepts in the applicable domain (for instance the *travel agent domain*). Once the domain ontology is defined, matchmaking is done as follows:

1. Determining Compatibility Between Service Types and Web Services

The service types requests and the web services operations are mapped to the concepts in the common ontology to define their semantics: operation, inputs, outputs, effects and preconditions are linked to their meanings represented as concepts in the shared ontology. The concepts to which each element maps can be related, belonging to the same hierarchy of concepts or can be unrelated, belonging to unrelated hierarchies. In OWL-S, *Service Profiles* are used to describe service requests as well as service advertisements. Service types are mapped to the service request part of the Service Profile and concrete Web services are mapped to service advertisements. The enriched service descriptions serve as a base to perform compatibility checks between the service requests, i.e. service types, and the service advertisements, i.e. concrete web services. Note that if the providers of the descriptions used different travel agent ontologies, a mapping would be needed to specify their correspondence.

2. Determining Compatibility Between Service Types and Web Services

At service integration time (i.e. when a new service is integrated in the WSML for a given service type), the compatibility between that service type and the new Web service is determined. A service type request can be fulfilled by a web service invocation if they have the same semantics. This is determined by checking the links to the shared ontology and the relation between the concepts in the ontology itself. For instance, if the service operations and the service type requests point to the same concepts in the ontology, the

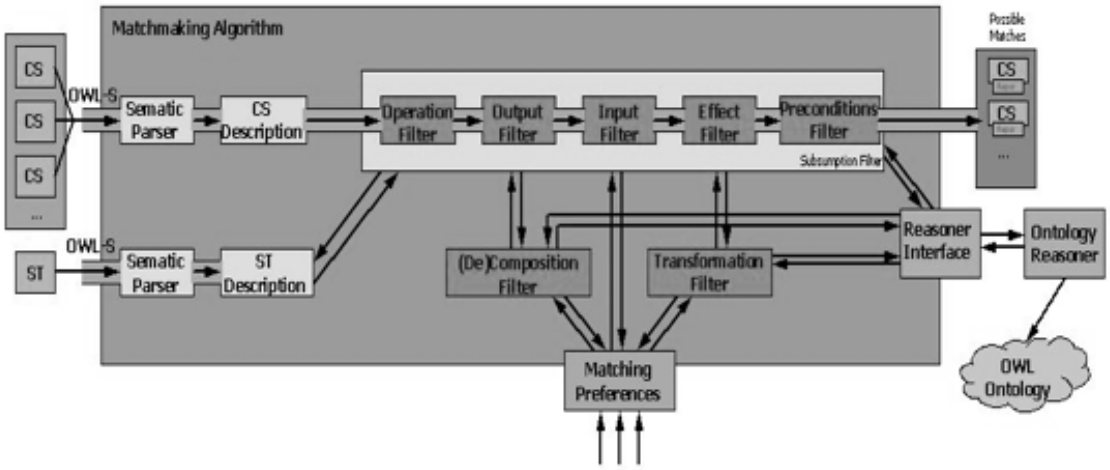

Figure 8.2: WSML Matchmaking Algorithm

service is compatible with the service type. Otherwise, the relations between the concepts to which the service operations and the service type requests are mapped need to be analysed further: three different degrees of matching can occur as identified in [Ver04]. The degree of matching is based on the definition of *subsumption* of ontological concepts. *Concept A* subsumes another *concept B* if the extension of B is a subset of that of A. This means that the logical constraints defined in the term of the *concept B* logically implies those of the more general *concept A*. For example, the concept Building subsumes the concept Hotel, in the travel agent ontology. The degrees of matching between a service type request and a web service are listed in Table 8.1. Additional semantic incompatibilities that may occur between arguments are listed too.

The algorithm for automatic discovery implemented as part of the WSML considers these differences and a set of configuration parameters to decide whether a given service or composition can fulfil the functionality requested by the client application through the corresponding service type: for each request of the service type a matching operation is searched in the web service based on the semantics of the operation. For the closest match, the input and output parameters, preconditions and effects are compared. This process is based on the matching algorithm described by Paolucci et al. [PKP+02]. Figure 8.2 illustrates the algorithm as part of the WSML. On the left the OWL-S documentation of three available web services and one service type is fed into the algorithm. For each method of the service type and the web service, filters on the operation, output, input, effect and preconditions will eliminate incompatible services based on subsume, plugin or exact matchmaking. If necessary, a (de)composition and transformation filter will also try to suggest a possible match. The algorithm gives as output all possible matches. The algorithm is configurable to enable or disable any of the filters, and to specify the allowed distance between concepts in the ontology. For our experiments, the RACER [HM01] ontology reasoner was used.

Table 8.1: Degrees of Mismatching between Service Types and Web Services

Name	Description	Example
Subsumption	Any of the ontological concepts, to which the operations, inputs and outputs of the service type are mapped, subsumes the concepts mapped to the web service invocation.	The *HotelServiceType* requests a hotel description, but the web service returns a building description.
Plug-in	Any of the ontological concepts for the operations, inputs and outputs of the web service, subsumes the concepts mapped to the service type request.	The *HotelServiceType* requests a hotel description, but the web service only returns Bed&Breakfast descriptions.
Exact match	The elements are mapped to exactly the same concept in the ontology. They are completely equivalent.	
Composition	The input or output provided is different, but semantically equivalent. Some elements can be composed to form the required argument on the other side.	The *HotelServiceType* checks hotel available by providing a begin and end date, while a service takes in a begin date and number of nights.
Decomposition	The number of arguments required is different but semantically equivalent. An argument can be decomposed to obtain the required arguments that the service is expecting.	The *HotelServiceType* books a hotel by providing an instance of a Hotel class, while a service takes in a String containing a hotel name.
Transformation	The representation ontology can be used to convert input and output parameters that match to the exact same ontological concept, but are represented differently.	A Date object provided by the *HotelServiceType* needs to be decomposed into 3 strings representing the day, month and year

3. Generating Glue Code

Determining the compatibility between a Web service and the service type ensures that the service can be used to fulfil the functionality expressed in the service type. In case of an exact mapping, there is no problem and the service mapping will only consist of a mapping between the service type methods and Web service operations. In case of Subsumption or Plugin mismatching it is important to realise that the service might not exactly return a compatible result, so glue code is needed to filter out incompatible results (e.g. if hotel descriptions are requested, all building descriptions should be filtered out in case of Subsumption). In case composition, decomposition or transformation of the input or output is required, specific glue code will be needed. This glue code is part of the service mapping and must be included in the service redirection aspects. These results have been published in [CVS+04b], but fully automating glue code is part of future work.

8.3.4 High-level Service Composition Specification

Similar to service mappings, it is possible to specify a service composition in Java, or as a series of statements, and translate this specification into an aspect. However, Java is a general-purpose language, lacking specific composition constructs, while specifying a composition as a series of statements may lack the required expressivity to specify complex compositions, including conditional loops, parallel branching, etc. There are dedicated composition and workflow languages for Web services available, such as WS-BPEL [ACD+03], Web Services Conversation Language (WSCL) [BBB+02], Yet Another Workflow Language (YAWL) [AH05]. In [Rol05], it was researched how the Unified Modeling Language (UML) [FS97] is suited to specify service compositions and how a translation to WSML aspects can be made. UML is the de-facto standard for modelling software applications. Several diagrams can be specified at various levels of detail: high-level models do not contain any implementation details, while implementation models are more detailed, containing artefacts targeted at a specific implementation language. Implementation models map to a specific language such as Java, C++, or in this case, WSML aspects.

A Web service composition is a special kind of business process where the different partners in the process are Web services. While standard UML can represent business processes, it does not have specific provisions for Web services, such as service invocations, messages and process instance IDs. In [Rol05] an extension of UML is proposed by specifying a new profile, i.e. a collection of related extensions, targeted towards a specific application domain. Profiles respect the semantics of UML constructs, but extend them or add additional constraints. The proposed extension is targeted at expressing several composition constructs:

- **Static web service structure:** class diagrams (extended with stereotypes, tagged values and constraints) represent the composition interface and the interfaces of its partners. These interfaces are based on service types.
- **Web service interactions:** activity diagrams model web service interactions. Three kinds of communication can occur between clients and compositions or between compositions and partners: receive, return and invoke.

  - *Receive*: client that invokes an operation on the composition. This happens transparently by invoking the service type.
  - *Return*: composition that returns a value to the client. A return action must follow a receive action, and only if the operation of the receive action does not have a return type of void.
  - *Invoke*: composition that invokes an operation on a partner.

- **Control flow:** activity diagrams provide constructs for sequential, conditional, and parallel execution. Loops can be modelled implicitly, by drawing a path that starts and ends at A.

- **Event handling:** both message-based and time-based events are supported. They are represented as a stereotyped conditional statement. Each branch must start with a *receive* or an *acceptTime* action. The branch with the event that is triggered first gets executed.

- **Exception handling:** the exception handling mechanism is based on try/catch blocks in classical programming languages. UML 2.0 activity diagrams fully support this. The *throwException* action specifies which kind of exception to throw. Exception handling activities catch errors thrown by an action or a nested activity. If no handler is defined for an activity, exceptions propagate to the enclosing nesting level.

- **Transaction management:** long-running transactions are supported through compensation handlers. A compensation handler can be associated with a nested transaction. It is the responsibility of the parent transaction to invoke the compensation handler.

- **Process composability:** activities can be nested. A nested activity can contain actions (which cannot be decomposed any further) or other activities. Nested activities can be invoked by the *callNested* action. This can be compared to invoking a function in a classical programming language.

- **State:** compositions can define variables that are globally accessible to all activities. Both static and instance-level variables can be declared. The assign action assigns values to variables.

- **Composition instance identity:** each receive action (except for the one that starts the composition) contains a correlation expression, which extracts identifying data from an incoming message.

A mapping is provided to translate a service composition specified in the high-level UML model into the WSML aspects that were discussed in section 5.6. At the moment, this mapping is done manually.

8.3.5 Aspect Template Library

Code generation for the aspects for service selection (Chapter 6) and management concerns (Chapter 7), is more difficult. Complete automatic aspect code generation requires a model or higher description of the specific concern, an approach taken in Model Driven Architecture (MDA). Through a series of refinements, such a model can be translated into a code for a specific platform or technology. This is outside the scope of this dissertation. In our approach, we opt to provide a library of reusable aspects that can be configured for a particular context. Aspect implementations of a variety of selection and management concerns are provided, and connector code for a specific context is automatically generated. For instance, a connector for a generic caching aspect can be generated by specifying for which concrete service invocations caching is required and what the expiration time of the cache is. In our approach, an aspect implementation is accompanied by an *XML deployment descriptor* for a particular concern. This descriptor file contains additional information required to deploy the concern and describes which additional properties need to be specified. Otherwise said, it contains all information required to generate a connector. In the current prototype, expert knowledge about the WSML is needed to implement additional aspects and add them to the library. A possible approach to facilitate this, might be to provide WSML API's or to specify design patterns for the aspects. This is subject of further research.

As an example, consider the caching mechanism discussed in Chapter 7, section 7.2.2. A library of several caching variants is made available as templates: for example, the library can contain a generic *CachingAspect* and a *ConditionalCachingAspect* variant that only caches when a certain condition is fulfilled, e.g. only to do caching if the Web service speed drops below a certain threshold. Next, a deployment descriptor is made for each aspect, as illustrated in Code fragment 8.1 for the *ConditionalCachingAspect*. The descriptor specifies general information (lines 2 to 5), where each aspect hook needs to be deployed (lines 7 to 17), and which additional parameters values need to be specified when the aspect is instantiated (lines 19 to 30).

In most cases, the deployment of one or more hooks will depend on the parameter values specified at aspect instantiation time. For instance, caching when a *HotelWebServiceA* is invoked will require that the CacheHook hooks on the method
" * HotelWebServiceA.*(*)". For this purpose, keywords (indicated with the % sign) are introduced. For instance, the %invocation% keyword in line 14 indicates that this keyword needs to be replaced by the signature of a concrete service type, service composition or Web service invocation. This signature can be composed using the parameters in line 20 to 22. The other parameters (lines 23 to 29) are used to pass along to the aspect constructor.

When a concern needs to be deployed, it suffices to specify an instance in an *XML configuration descriptor*, as shown in Code fragment 8.2. Besides a name and an indication of which template is instantiated, it suffices to provide a value for the set of parameters to generate the connector. In this example, a *HotelDescriptionCache* is setup for the method `getHotels` of the *HotelServiceType*. With this approach, the usage of aspects as an implementation technology is hidden away.

Note that also for the specification and deployment of the actual service types, service compositions and Web services, dedicated XML configuration files can be specified. This

```
<template
    name="ConditionalCaching"
    description="return cacheresults"
    package="templates.caching"
    priority="4">

    <hooks>
        <hook
            name = "ConditionalCacheHook"
            init="true"
            aspectFactory="perobject">
            <methods>
                <method
                            signature="%invocation%"/>
                </methods>
        </hook>
    </hooks>

    <parameters>
        <parameter name="type"/>
        <parameter name="name/>
        <parameter name="operation"/>
        <parameter name="expireTime"/>
        <parameter name="alwaysCache"/>
        <parameter name="property"/>
        <parameter name="type "/>
        <parameter name="minimum"/>
        <parameter name="maximum" />
        <parameter name="ifUndefined"/>
    </parameters>
</template>
```

Code fragment 8.1: XML Deployment Descriptor for a Caching Aspect

```
<templateInstance
    name="HotelDescriptionCache"
    template="ConditionalCaching">

    <parameters>
        <type=ServiceType>
        <name=HotelServiceType>
        <operation=getHotels>
        <expireTime="20"/>
        <alwaysCache="true"/>
        <property="executionTime"/>
        <type="Float"/>
        <minimum="10"/>
        <maximum="45"/>
        <ifUndefined="true"/>
    </parameters>
</templateInstance>
```

Code fragment 8.2: XML Configuration Descriptor for an instance of a Caching Aspect

way, the whole configuration of the WSML can be done through these files. Additionally, a specific "snap-shot" of the running WMSL can be made persistent by saving the complete configuration back to XML files. As such, the configuration files are XML representations of the various entities in the WSML.

8.4 Stakeholders

Due to the organisational defragmentation of a SOA, there may be multiple parties involved in dealing with the WSML. Typically, the WSML will be part of the client environment, as the WSML will act as a service agent for the client, taking care of all service related concerns. At the moment the client is developed, the required service types must be configured, so that the client be programmed against this set of service types. When the client is deployed together with the WSML and the client invokes a service type, the default behaviour of that service type (see Section 5.1) will be executed. This behaviour can be changed by further configuring the WSML: i.e. by deploying redirection, selection and management aspects. Ideally, the use of aspect-oriented techniques must be hidden from the WSML administrator. Figure 8.3 shows our approach for the realisation of redirection aspects. Either a mapping or a composition is specified as a series of higher-level statements, or semantic matchmaking is done based on the semantic description of a service type and Web service. Based on this process, an XML Descriptor (see Section 8.3.5) is generated which can be used to generate, compile and deploy redirection aspects and connectors. The WSML administrator involved in this process only needs to know the service type specification and the WSDL description of the Web service and optionally its semantic description. The prototype (see Section 8.5) of the WSML supports this scenario and illustrates that the aspect implementation can be completely hidden from the administrator. Note that the stakeholder involved in this process is not necessarily part of the client environment. For instance, in an intranet environment, one WSML may be used to service multiple clients, and it may act as the sole gateway for communication with external third-party Web services. In that case, the administrator of the WSML will be involved in looking up compatible services, specifying mappings and/or compositions and configuring the WSML accordingly, without having in-depth knowledge about the individual client applications on the intranet.

For the selection and management concerns, a different approach is followed as doing automatic code generation of these aspects is outside the scope of this dissertation. As mentioned before, for these aspects we propose a reusable and extensible aspect library. While instantiating these aspects can be done at the administration level, similar to the redirection aspects, the process of extending the library is more difficult, as it requires more in-depth knowledge of the WSML implementation. This is depicted in Figure 8.4. As we will discuss in Future Work (see Section 9.2), this proces can be optimized by facilitating the implementation efforts for the developer by providing additional tool support and API's. Another possibility is to research the automatic generation of aspect code. The stakeholder involved in deploying the selection policies and management concerns may need business knowledge about the client (e.g. to know which services to prefer) and possibly the network and service environment (e.g. to know wich kinds of management concerns to enforce). In the next section, we discuss the concrete implementation of the WSML

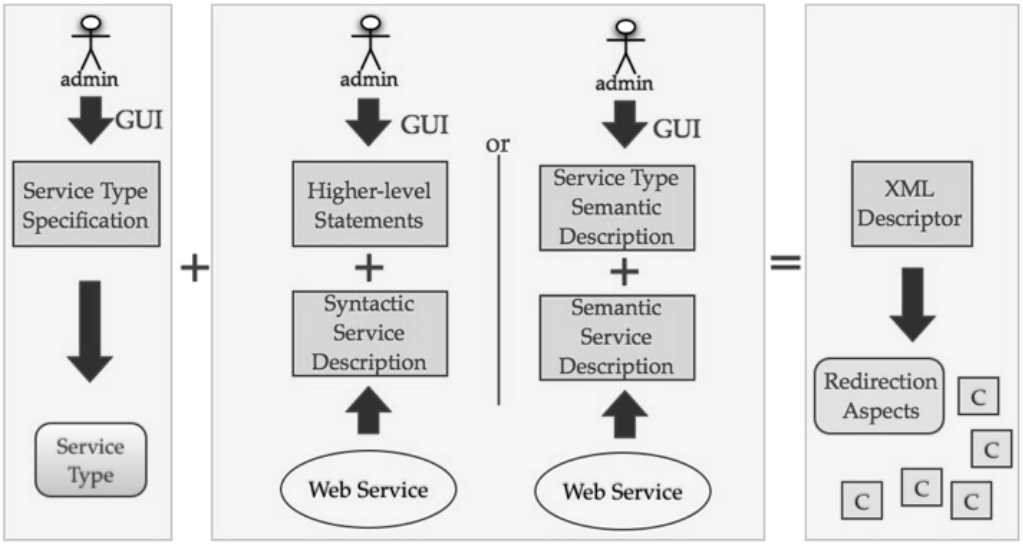

Figure 8.3: Implementing and Deploying Redirection Aspects

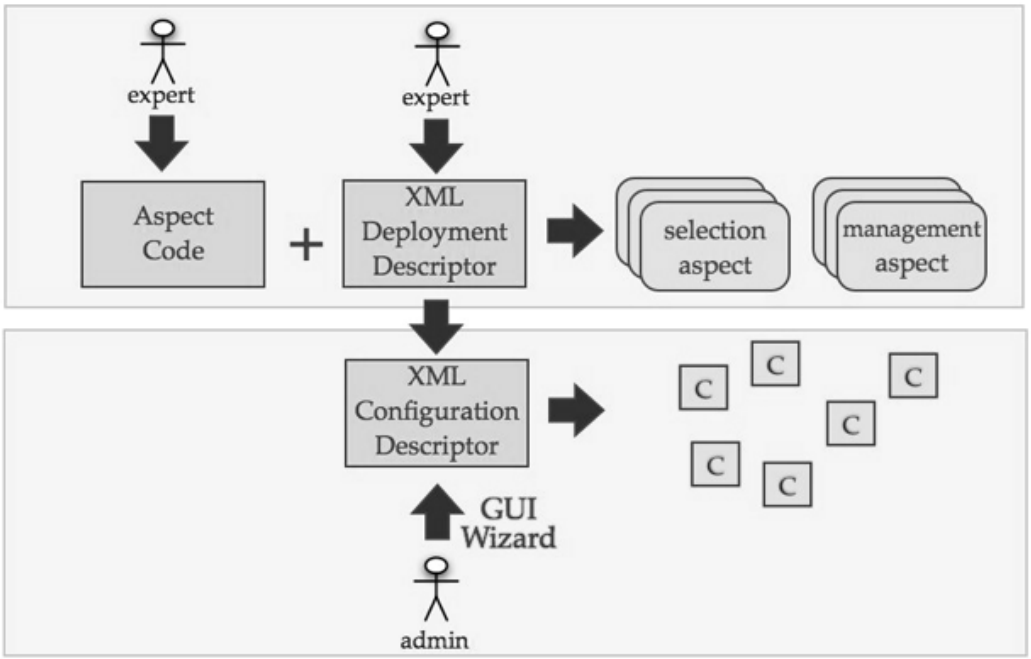

Figure 8.4: Implementing and Deploying Selection and Management Aspects

supporting automatic generation of redirection aspects, and the usage of a reusable library for selection and management aspects.

8.5 Prototype

8.5.1 Overview

A fully functional prototype of the WSML has been developed in the context of the IWT[1] Mosaic project and is available at [Ver03]. This prototype is implemented in Java and uses JAsCo for implementing the aspects. The WSML is the supporting framework for the various aspects discussed in the previous chapters. A key requirement for the WSML was hiding for the end-user of the system that aspects are used as an implementation technology. The WSML offers support in the various stages of development, deployment and runtime of the client application:

1. **Development**: During the development stage the client application is implemented and any service functionality required needs to be specified as service types. A service type can be integrated in the client, either as a local class, or remotely, as a web service. Dedicated tool support is available for this purpose.

2. **Deployment**: At deployment time, the client application is deployed together with the WSML. When running locally, the client and the WSML run on the same virtual machine. When running remotely, the WSML runs as a server, independently of the client(s). A pool of available web services can be registered with the WSML through XML configuration descriptors and linked to one or more service types. Additionally, compositions, selection policies and client-side management concerns can be configured.

3. **Runtime**: at runtime, any client requests on the service types are transparently redirected to available web services or service compositions. All specified selection policies and management concerns are enforced non-invasively. Tool support is available at runtime to monitor and administer the system.

8.5.2 Design

Figure 8.5 shows a conceptual UML class diagram of the WSML architecture. Every entity shown on the diagram is part of the WSML, except the client and Web service, which are depicted in grey. On the right hand side the functional level is shown, with from top the bottom first the redirection mechanism, then the service selection mechanism and finally, at the bottom, the client-side management concerns[2]. The left hand side of the picture shows the meta-level, a collection of classes in the WSML that *create and represent* the entities at

[1] IWT: Institute for the Innovation of Science and Technology in Flanders" ("Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen" in Dutch)

[2] While we could have used a UML class notation for every entity at the right-hand side, we have opted to keep the graphical notation of the aspects and connectors as used in the previous pictures in this dissertation.

the functional level. The classes on the left-hand side are concrete resident WSML classes. The entities on the right hand side are generated, compiled, loaded and instantiated at runtime by instantiating a class on the left-hand side.

At the moment a new instance of the `ServiceType` class on the left hand side is created, a new template class `<T>ServiceType` on the right-hand side is generated and compiled. Then, client(s) can instantiate that service type class and start invoking it to request service functionality. But first, Web services need to be registered. Any WSDL-exposed Web service can be registered, by instantiating a new `WebService` instance on the left hand side. This will generate a proxy on the right hand side. The proxy is the local representative of the remote Web service. Next, service mappings can be specified to map a service type interface to the service interface or multiple services can be composed in a composition for a service type. Either way, a service redirection aspect will be generated and compiled out of this, and a set of corresponding connectors will be generated too. As explained in Chapter 5, section, 5.2.7, n+1 connectors will be present with n being the number of requests in the service type. The connectors hook on the service type(s) the mapping or composition is specified for. As a result, the redirection aspect will be triggered as soon the service type is invoked by the client and the Web service(s) will be invoked through their proxies. Any meta-data for the service types, mappings, compositions and Web services, including their descriptions and properties are stored in the class instances on the left hand side.

Each instance of the `SelectionPolicyTemplate` class on the left hand side represents a `SelectionPolicyAspect` on the right hand side. When a new instance is created, a corresponding aspect is compiled. As we noted before, there is no automatic generation of this kind of aspects done in the current prototype. All instances of the `SelectionPolicyTemplate` class make up the library of available selection policy templates. Deploying a template in a concrete context is done through the `SelectionPolicyInstance` class. At the moment a new instance of this class is created, a new connector is generated and deployed. This connector instantiates the corresponding `SelectionPolicyAspect` and hooks on whatever Object in the system needed to enforce the policy. When the aspect is triggered, it will enable, disable and/or make a new ranking of the connectors of the redirection aspects and as such qualify, disqualify and prioritise the available Web services.

For the management concerns, a similar approach is followed as for the selection policies. Each `ManagementConcernTemplate` on the left hand side represents a `ManagementConcernAspect` on the right hand side. A deployment of a template in a concrete context is realised by instantiating the `ManagementConcernInstance` class, which will generate an appropriate connector. This connector instantiates the `ManagementConcernAspect` and hooks on any possible Object(s) in the system. The effect of a `ManagementConcernAspect` is not shown, as there is no visualisation possible as the effect is unrestricted.

From this explanation it follows that configuring and administering the WSML is done by instantiating the classes on the left-hand side and by manipulating these class instances. As depicted on the top left corner of Figure 8.5 this can be done by various means, including administration services, an administration console and an XML parser. We discuss these management tools in more detail, using Figure 8.6, which shows the complete layered architecture of the WSML prototype, based on Java and JAsCo. The picture shows the five

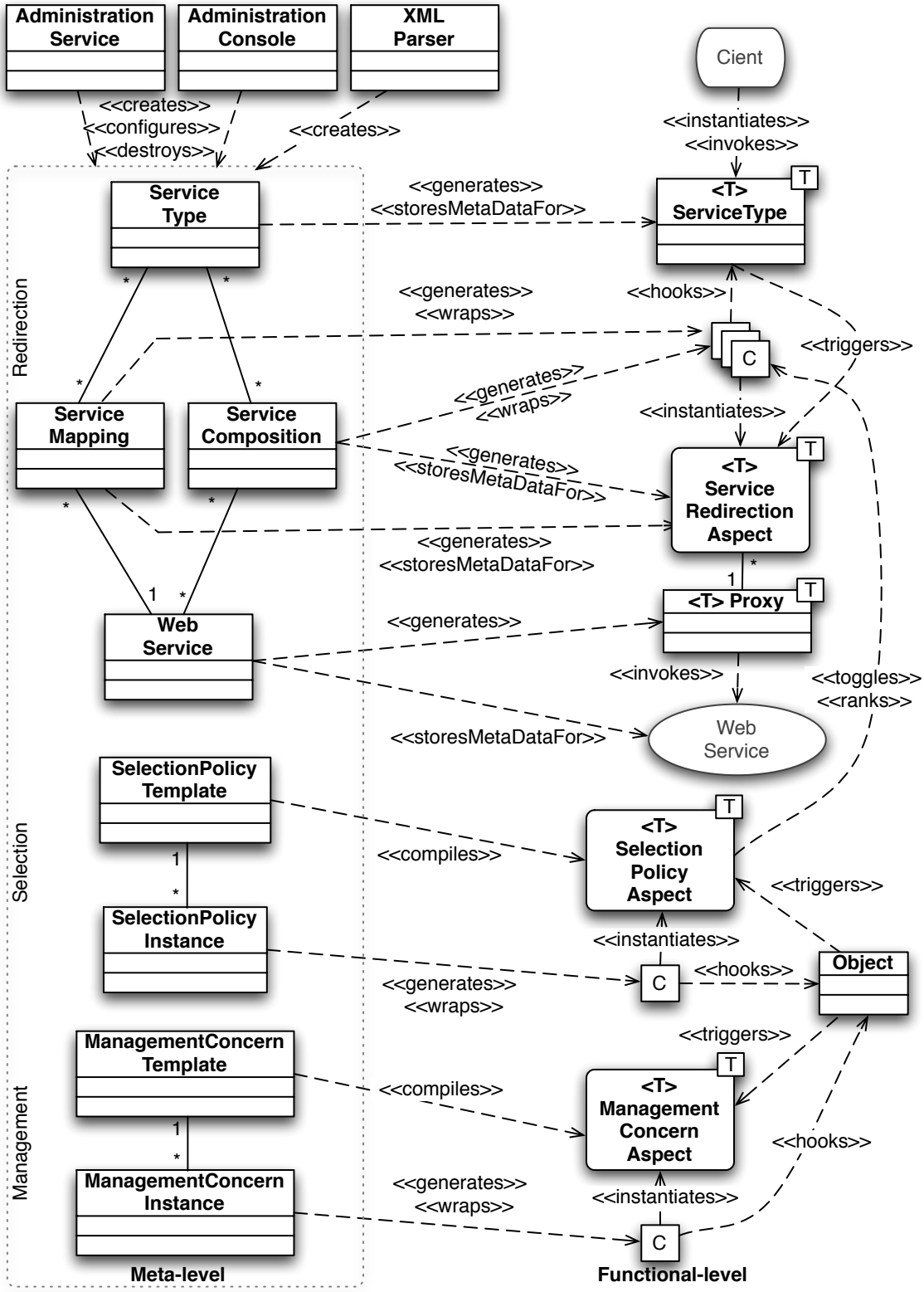

Figure 8.5: Conceptual UML Class Diagram of the WSML

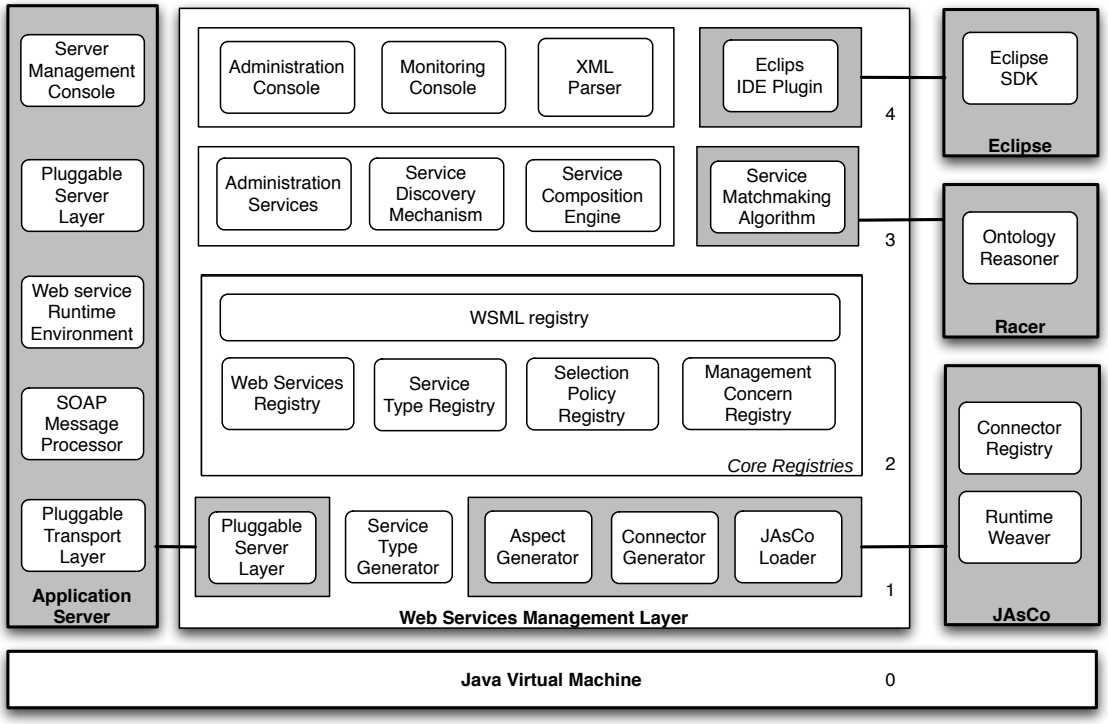

Figure 8.6: WSML Layered Architecture

layers of the architecture; the WSML is depicted in the middle, running on a Java Virtual Machine (level 0).

Level 1: Underlying technologies

JAsCo: JAsCo, depicted on the right hand side, is used as the aspect-oriented extension of Java. It is plugged into the WSML using a dedicated loader used to compile aspects and connectors, load and unload aspects and combination strategies, temporarily enable and disable connectors, and do inspection of the connector registry. Two other JAsCo related entities in the WSML are the aspect and connector generators used for automatic code generation.

Application Server: on the left hand side of Figure 8.6, the application server resides. It is used to deploy service types as web services when running in server mode, and to generate the necessary client proxies for remote web services. In the WSML prototype, the Systinet Server for Java [Sys05] is used, a platform-independent server for creating, deploying and managing Web services in Java applications. A pluggable server layer is used to connect the WSML with the server, so other servers can be used too.

Level 2: The WSML Core

WSML Core Registries: Four registries contain the deployed service types, selection policies, management concerns and an internal representation of the registered Web services as shown on the left-hand side of the conceptual UML-class diagram of Figure 8.3. The

WSML registry is the core of the WSML, containing links to each of the registries, the server, the console, etc.

Level 3: Additional Service Functionalities

Service Discovery Mechanism: A discovery mechanism can be used to find available web services on the network, for instance by querying a UDDI registry, or maintaining its own repository.

Description Logic Reasoner: In order to realise automatic mapping between service types and web services, a semantic description is required of both sides. Concepts are stored in an ontology and a matchmaking algorithm determines the compatibility between a given service type and a service by querying the ontology using a reasoner, such as Racer.

Service Workflow Engine: A dedicated engine, for instance based on a business process language such as WS-BPEL is used to compose services together in an advanced manner by specifying complex workflows. This engine exposes the composition as a Web service that can be registered for a service type.

Administration Services: The WSML can be administered through a set of administration Web services. These services can be invoked from within the client, other applications, the Eclipse plugin, the administration console , other applications or even the third-party Web services (e.g. to announce unavailability, versioning information or changing QoS, etc.).

Level 4: Management Tools

Consoles: The administration console is a web-based tool to administer the WSML. The GUI is used to specify new service types, register third-party Web services, specify mappings between service types and Web services, compose services together, and enforce selection policies and management concerns. Two versions are available: a .NET version, developed in Visual Studio.NET and implemented in ASP.NET and C#, and a Java version, implemented with Java Servlets. Additionally, a console is available to monitor the running system. A screen-shot of the WSML Monitoring Console is shown in Figure 8.7.

XML Parser: As mentioned before, the WSML can also be configured through XML deployment and configuration descriptors. This is particularly useful for persistent deployment of a specific configuration. Service types, web service mappings, compositions, policies and management concerns can all be specified in XML descriptors and loaded at deployment time or runtime.

Ecplise Plugin: a plugin for the Eclipse IDE allows implementing a client application in close cooperation with the WSML. Service types can be specified through wizards in Eclipse, integrated in the client and the client can be deployed together with the WSML, both locally and remotely.

8.5.3 Realisation of Quality Development Attributes

With our approach, we realise the development attributes of the WSML defined in Chapter 4, section 4.3, as follows:

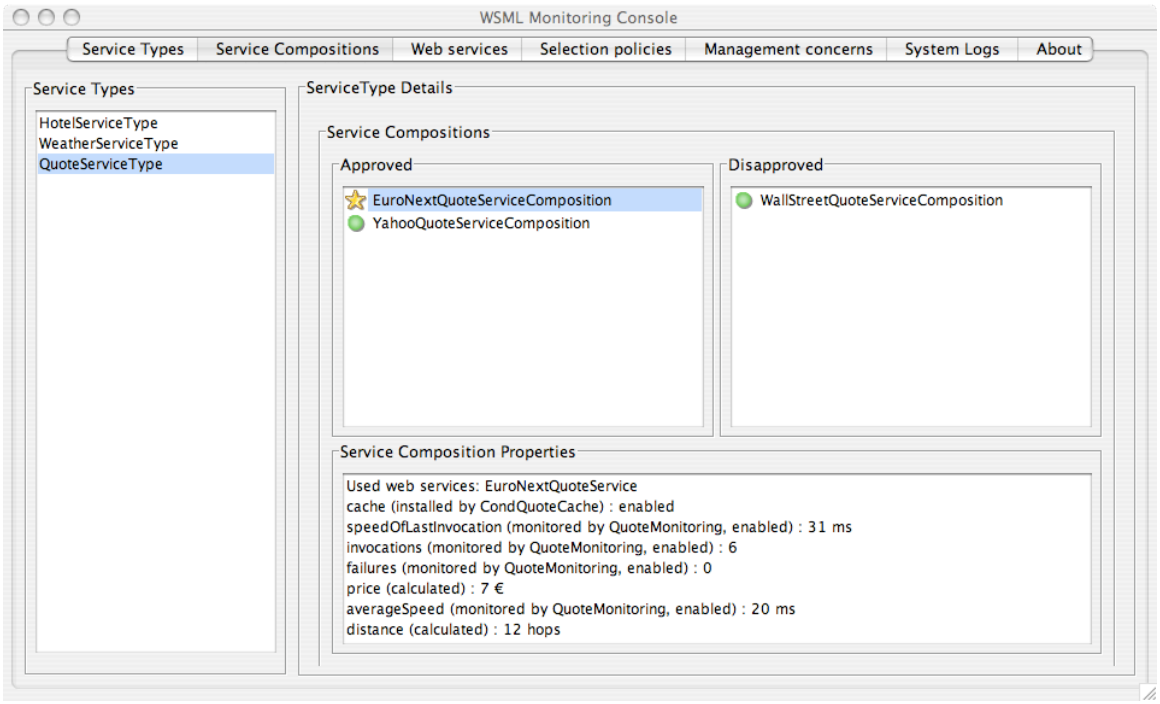

Figure 8.7: WSML Monitoring Console

- **Web Services standards compliance:** The WSML approach fully supports existing Web services standards. First of all, communication with the surrounding quadrants of the WSML environment takes place using Web service technology:

- Both third-party Web services and management Web services that have a WSDL-described interface can be integrated in the WSML: a WSDL2Java tool analyses the WSDL-file and generates a proxy which is being addressed from within the redirection aspects.
- Communication with the client occurs with service types, which can be deployed as WSDL-described Web services towards the client(s).
- Administrative WSML tasks can be done through dedicated administrative Web services which can be straightforwardly integrated in third-party applications or services.

Furthermore, additional concerns, enforced by a third-party service provider, that adopts a WS*-standard, can be implemented in the client using dedicated management aspects. New standards and specification could be supported through the extensible management template library.

- Performance and scalability: runtime flexibility and configurability is typically a trade-off against performance of the system. The WSML offers dynamic binding of Web services and runtime integration of unanticipated services. Furthermore, the binding mechanism takes a set of selection policies into account and enforces additional client-side management concerns when the service is invoked. As we discussed in Chapter 4, section 4.5.4.2,

the infrastructure of JAsCo consists of a runtime weaver. Instead of inserting traps at every possible place where an aspect possibly will trigger, a highly optimised code fragment is directly inserted into the target joinpoints. This code fragment directly invokes all applicable advices in the correct sequence and thus avoids the indirection through the JAsCo run-time infrastructure. As a result, triggering a dynamically inserted aspect does not cause any considerable overhead than a regular method invocation. In [VS04] performance tests of JAsCo indicated that it performed similar to a static weaving approach as AspectJ. Therefore, we can conclude that the level of indirection created by the redirection aspects is completely negligible considering the time consuming process of (de)serialising service requests and sending SOAP messages over the network.

Furthermore, the WSML can enhance performance, for instance by monitoring service execution time and pro-actively selecting the best performing services. Pro-active, meaning this selection process takes place even before the client makes a service request. Other approaches to deal with performance issues are hot-swapping, exception handling, broadcasting and caching, all concerns that have been modularised in the WSML by aspects and that can be used to optimise the functional flow from the client to the service and back. As triggering an aspect is as computationally expensive as a simple method invocation, scalability of the system will not be limited by the WSML or the JAsCo implementation, but rather by how many requests an as Web service deployed service type can handle, which is determined by the application server, the VM and the hardware platform.

Another facet of performance is the deployment time of the framework. As deploying the WSML requires an extensive amount of code generation and compilation of Java classes and JAsCo aspects, it can be a very lengthy process to deploy the framework with large configuration files. To reduce the deployment time, the WSML will analyse at deployment time the loaded configuration files and compare them with configuration files that were used in previous runs and not re-generate and recompile classes and/or aspects if this is not necessary.

The other development attributes, being reusability, configurability and extensibility are mainly realised because of the high-level of *modularisation* of the code, achieved through the adoption of an aspect-oriented approach.

- Reusability: the WSML has been designed to contain no context specific code: the basic framework is completely reusable and is intended to be instantiated for a specific context. For this purpose, it will automatically generate and deploy service types, service redirection aspects and connectors. Additionally, a library of generic aspects implementing service selection policies and management concerns is available, which can be instantiated through a configuration process.

- **Configurability:** the WSML is completely configurable at the administrative level. First, at deployment time, all specified XML configuration files for service type, compositions, services, selection policies and management concerns are loaded. At runtime, the configuration can be altered through administration services, a web-based console, and additional XML files.

- **Extensibility:** the framework is mainly extensible through the addition of aspects and combination strategies. By adding new aspects and complementing deployment descriptors,

unanticipated concerns and policies can be added to the framework. Aspect inheritance and aspectual polymorphism are useful here.

8.5.4 Synergy between WMSL Research and JAsCo Research

While applying AOP in the context of SOAs offers several advantages to realise the requirements and development attributes of the WSML, there was also feedback loop towards the AOP community. The WSML research served as a useful complex and dynamic testing environment for different AOP features. Several required features were identified during the realisation of the WSML and were incorporated in later JAsCo implementations. An overview was published in [AOSD05-2]. First of all, we have identified various scenario's where deploying aspects on aspects was needed, for instance in the case of selection aspects (see section 6.5.1.3). Also, a new feature was needed that allows for the re-invocation of the original method that triggered an aspect around chain, for instance for fallback aspects (see section 5.2.5). For this purpose, the JAsCo language was extended with a new method `invokeAgain`, in order to allow for invocation of the original method that triggered the aspect chain all over again. AspectWerkz has recently identified the need for such a feature and added the `invokeOriginalMethod` construct, which allows skipping the rest of the advices and calling the original method directly [DFS04]. An identified disadvantage of this feature is that it is not always clear to understand its effect, i.e., which method will be the one invoked again, especially in the case of having aspects deployed on other aspects. When looking in isolation to the fallback aspect, it is not straightforward to identify the method that originated the aspect chaining. Thus, analysability of aspects is reduced since knowledge about "where" the aspects are hooking is needed to understand the behaviour of the aspect itself.

A problem with JAsCo after advices was that, by default, these advices were not triggered when the joinpoint caused an exception. However, the monitoring aspect (see section 6.4.2.2) counting the number of failed service invocations needs to be triggered after an exception has been thrown. Without a dedicated after throwing advice, this should be implemented with an awkward and counterintuitive around advice that continues with the execution of the original joinpoint and has a *try catch* block around it. This was changed and after advices are now triggered even when an exception is thrown in the joinpoint, similar to AspectJ after advices. Also dedicated after throwing and after returning advices were added, allowing for more control over when an after advice should be triggered. Similarly around returning and around throwing advices were added. This made the implementation of several WSML aspects much more elegant and intuitive.

An important issue arose as aspects can be temporarily enabled and disabled with a dynamic AOP approach. In a complex setup as the WMSL we encountered several situations where this enabling or disabling required the execution of some behaviour belonging to the crosscutting concern. For instance, disabling a service selection aspect requires re-approving all Web services that were previously disapproved. For this purpose, the JAsCo runtime environment now throws an event whenever an aspect is added, removed, enabled or disabled. Inside the aspect, additional behaviour can be specified that needs to be executed in these cases.

Another identified feature that was added to JAsCo are the complement keyword for stateful aspects (Chapter 5, section 5.4.5); connector priorities (Chapter 6, section 6.4.3.4) and connector combination strategies (Chapter 7, section 7.3). These features are included in the JAsCo language in order to improve modularity of crosscutting concerns and evolvability of the overall solution. They allow the addition and removal of aspects independently of each other in a much easier way while it is still possible to express aspect interactions in a modular way.

Finally, we improved the aspect runtime environment as the JAsCo connector registry looks for new connectors with specified intervals, allowing for easy loading and removal of aspects at runtime. However, in the context of the WSML, a tighter control on the addition and removal of connectors is needed. The WSML is an aspect-oriented layer on top of the JAsCo runtime environment that controls the compilation and instantiation of aspects and the automatic generation and compilation of the connectors. Therefore, we decided to disable the hot-deployment of connectors and replace it by a mechanism that delegates the control of loading and unloading connectors to the WSML by means of the use of the Connector Registry API. This mechanism also had the advantage that it avoids the possibility of loading illegitimate connectors and aspects into the system. Before, it was possible to load malicious aspects that were not fully tested and that could crash the whole system. With the new mechanism, only aspects that were thoroughly tested and registered in the WSML could be instantiated.

An aspect causing a crash of the system has far-reaching consequences. Not only did this result in downtime of the server, but as mentioned in section 8.4.3, it also takes a long time to reboot the system and to recompile and reload all aspects and connectors., in a small scenario with 10 connectors instantiating 7 aspects, the loading time of the WSML was more than 3 minutes on a Pentium 4 with 1GB of RAM. Therefore, a caching mechanism is provided as part of the WSML in order to reuse already compiled connector and aspect classes. This reduced the loading time by a factor of 10. To avoid that aspects would bring the system down, all aspects were implemented following a coding convention. This allows for the detection of the aspects that cause exceptions and immediately removing them from the runtime environment. The incorporation of these improvements to the JAsCo runtime environment contributed to achieving a more robust implementation of the WSML and enhancing evolvability of the overall solution.

8.6 WSML Deployment on SEP

Validation of the approach presented in this dissertation is not simple. Ideally, two implementations of an extensive service-oriented architecture are to be realised by two separate development teams. One team uses traditional software engineering approaches and current Web service technologies, while the other team uses the WSML approach based on dynamic AOP. Then, both efforts should be compared with each other in terms of various criteria such as number of code lines, the development time, the quality, the flexibility, configurability and extensibility of the resulting implementation, etc. This is not a realistic approach in the context of a PhD dissertation. As an alternative, we have opted to deploy the WSML in an existing real-life SOA and have made a running implementation of the

integrated platform, which offers more capabilities than the standalone platform.

The existing platform has been developed by Alcatel Bell, a telecommunication company, providing solutions to telecommunication carriers, Internet Service Providers (ISP) and enterprises for delivery of broadband applications. According to Alcatel, there is no single killer application for broadband Internet. Rather, it is an enabling technology for accessing a plethora of different service capabilities offered by several *content providers*. Each of these providers needs a set of capabilities such as user accounting, billing, profiling, etc. However, the current situation in the use of broadband Internet shows that service capabilities are implemented from scratch by each service provider, increasing the effort of developing service applications. Each service provider uses their own systems and standards, making it difficult for *network providers* to accommodate to the different approaches employed by each service provider. This places a heavy burden on the network providers since they need to provide enough infrastructures to be able to integrate with all these different systems.

As a consequence, Alcatel observed the need and a market for a service and network management framework that would facilitate the adoption of service capabilities. They developed the Service Enabling Platform (SEP), a service-provisioning platform targeted for broadband Internet. By gathering all enabling applications into one SEP, costs will be reduced and services will have a reduced time-to-market. The SEP needs to be flexible to adapt to changes in the business environment, support reusability and configurability of service capabilities, allow for customisation and rapidly fine-tuning of service offerings and provide added value in the provisioning of services. For this purpose, it combines an extensive set of recent technologies such as enterprise JavaBeans, radius servers, session resource brokers, network management tools, etc. Web service technology is adopted to enable the communication among the different internal components of the SEP and also to integrate different capabilities such as billing, accounting and personalized advertisements with third-party Web services that deliver content.

Figure 8.8 shows the SEP deployed in a content delivery architecture. End users have access to the broadband network via a SEP client. This SEP client communicates over an ASDL connection to a Broadband Access Server (BAS), which connects to a variety of online services such as ISPs, corporate networks and content providers. The SEP hosts a set of management services and offers connectivity to third party services through the Open Services Platform (OSP). However, the Web services techniques used during the development of the SEP suffered from the issues identified in Chapter 3. Especially integrating unanticipated Web services and crosscutting management concerns became a hindrance for the further development, robustness and maintainability of the system. These are the main motivations for the integration of the WSML framework. With the WSML integrated in the SEP it becomes much easier to integrate and select new Web services in the SEP framework and to enforce various management concerns that crosscut service boundaries. To illustrate the benefits of the WSML in the context of the SEP, consider *Video on Demand (VoD) content services*. The basis version of a VoD seems straightforward: a user gets access to the network, logs in on the VoD service, browses through a selection of movies, selects one, pays for it and watches the streamed content. However, nearly every step in this scenario has numerous variations. A simple look at classical payment schemes (see section 7.2.1) reveals that even the billing of every step could be done in lots of different ways. Also all kinds of security schemes (single logon, two-phase commit ...) exist in every different step

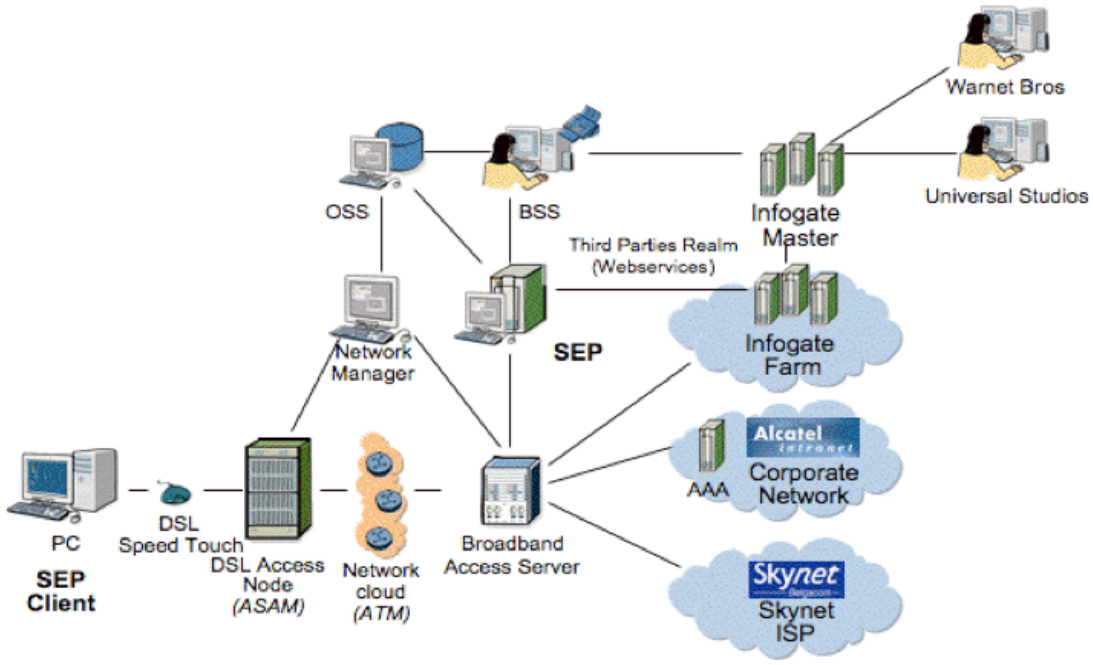

Figure 8.8: Alcatel Bell Service Enabling Platform (SEP)

as well as different streaming possibilities for the video (real time, staggered ...). Additionally, advertisement mechanisms, network monitoring, user accounting, etc. are needed too. In short, many VoD services are possible. However, this does not mean that these services should all be built from scratch. The SEP-WSML platform was used to set up a demonstrator to indicate that VoD service providers can more easily integrate existing services into the network infrastructure, and that they can create new services more easily using existing components and flexible configurations provided by the SEP and the WSML.

Figure 8.9 shows the complete architecture of the integrated platform. Users can watch streaming video content on a television connected to a broadband ADSL network. Several VoD content providers offer the content. In one of the demonstrators we have developed, the demonstrator hosts the SEP and the WSML at the network provider and the VoD systems at several content providers. The SEP is a WSML client, and the VoD systems offer Web services for requesting, streaming, and paying for videos. End users can play streaming media on a television connected to a setup box, prepaying for the product with their mobile phone account. In this distributed setup with multiple partners, billing, accounting, dynamic bandwidth allocation, etc. become complex tasks.

In the described scenario, a user first logs on to the SEP via his or her television set, which is connected to the ASDL network via a modem. Other service providers can interface with the authentication interface of the SEP (exposed as a web service) to reuse these credentials for their own service allowing a single log on for the users of their service. The SEP keeps a profile for every user to present him or her with the services he uses. The user can connect to a VoD service and select a movie. Every request from the user to the VoD service is intercepted by the WSML, so that the network provider and content provider

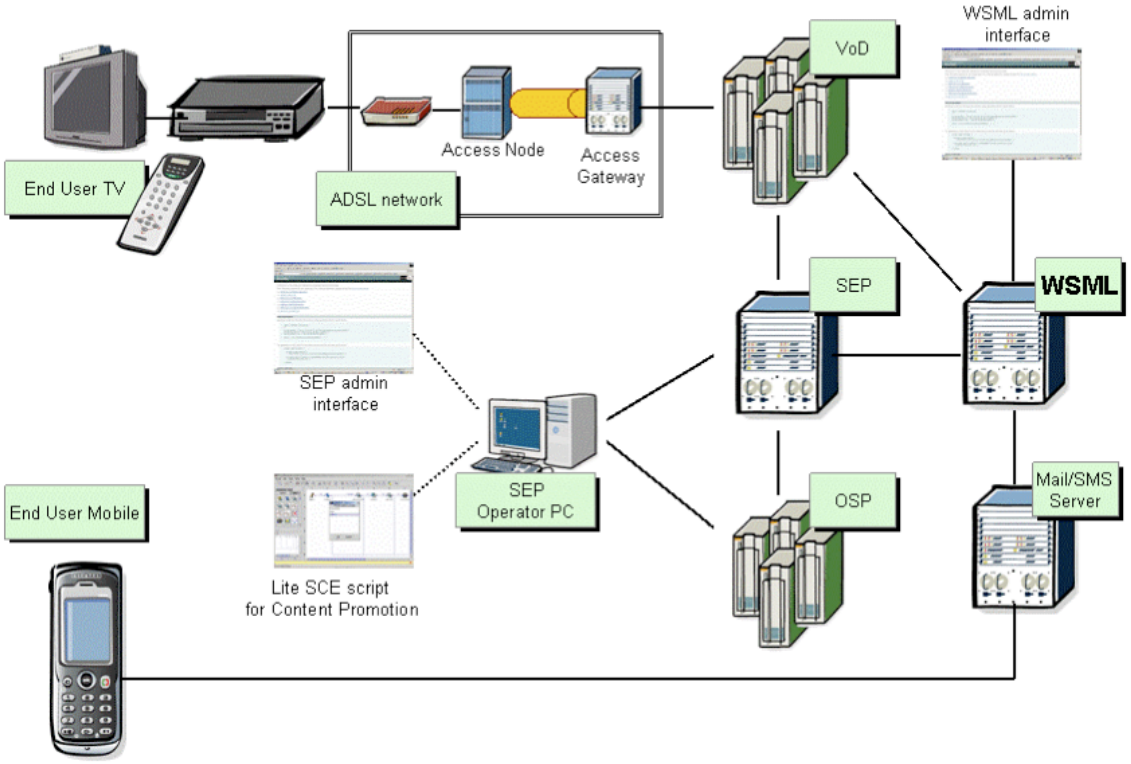

Figure 8.9: WSML-SEP integrated platform

can script the WSML to enforce load balancing, service selection, fail safe systems, Q0S, etc.. . .

In one scenario, the WSML applies pre-paid billing before the selected movie is streamed to the end-user. Depending on the customer's subscription status (bronze, silver, or gold), the product bought, the VoD content provider, and the mobile phone operator, a specific billing process is initiated through a dedicated SMS server. When the customer has confirmed the payment with his mobile phone, the movie will start playing. The original setup was not conceived with this mobile billing schema in mind. The VoD systems are proprietary and could not be modified. Therefore, the WSML was deployed to intercept all client requests from the SEP to the VoD and vice-versa to enforce the billing and promotional actions through dedicated aspects. The billing aspects calculate the correct price and request a customer payment through a dedicated SMS server. Additionally, these aspects retrieve additional context information (such as the customer's subscription status) from other services deployed on the SEP (such as the accounting service). Once, the customer has paid, the blocked call to the VoD is released through service redirection aspects. The billing concern continues with distributing a percentage of the paid amount to each partner (i.e. the network provider, the content provider, etc.). Furthermore, temporal promotions can apply to the billing, including fee reductions, bonuses with purchases, price reductions on other products, and so on. For instance, an aspect adds a call to the rating engine of the network provider whenever a bandwidth upgrade is granted for a given user and another one blocks every 10th call to the rating engine resulting in a free bandwidth upgrade every

10th time. Additionally to the billing and the promotions, optional movie-related advertisement mails and/or SMS messages are sent to the customer. In a next step, the video needs to be streamed to the user. The VoD service provider uses the SEP platform (using the exposed web service interface) to ask for a given bandwidth based on the properties of the selected movie. Users can get bandwidth based on their subscription, increase their bandwidth temporarily (the price for this could be included in the price of the movie or could be charged separately by the network provider, ...), ask for a reduced quality view of the movie, etc. In this scenario, the VoD becomes the client and the WSML redirects requests to the appropriate service deployed on the SEP.

These examples indicate the main advantages of the integration of the WSML into the SEP: the WMSL allows the flexible integration of orthogonal non-functional management concerns, and enhances the functionality of the capabilities offered by the SEP. All concerns can be deployed in a non-invasive way, without requiring any changes to the proprietary VoD systems. The described demonstrator uses the WSML's redirection aspects to intercept and redirect calls to the appropriate VoD systems and mobile phone operators. It implements the payment schemes and promotional offers through a set of WSML management aspects. In a typical setup with three VoD systems, two mobile operators, five billing schemes, and three promotional actions, WSML deploys about 25 aspects (including five redirection aspects, five selection aspects, and 15 management aspects), containing around 2,500 lines of aspect code.

8.7 Discussion on AOP in Enterprise Service Bus (ESB)

The SEP case study illustrates the usability of the WSML framework in a complex service oriented architecture. It is interesting to note that the WSML acts as a client for the SEP, redirecting calls to the VoD servers, the billing and accounting services deployed on the OSP, and the SMS servers of the mobile providers. The other way around is also possible: the VoD acts a client of the WSML redirecting calls to the billing and accounting services of the OSP. In this setup, the line between clients and services is more vague, and the WSML acts as the communication hub: an Enterprise Service Bus (ESB). An ESB is software infrastructure that simplifies the integration and flexible reuse of business components using a SOA. As such, the WSML is an aspect-oriented extension of a ESB, offering the capability to dynamically connect, mediate and control the communication between services.

As a concluding overview, Figure 8.10 shows the three possible kinds of applications of AOP in the SOA context (1) AOP at the client side where service related concerns are modularised in aspects. Joinpoints reside in the WSML and the client. (2) Evidently, AOP can also be applied inside service implementations. As a Web service is a regular software application (with a published API for its clients), aspects can be applied locally to help in achieving better separation of concerns. As the service implementation is hidden away from its clients in a black-box fashion, this does not alter anything in the service behaviour; from outside it can be considered an unimportant technical implementation detail. However, in [BM04] an AOP-extension to WSDL is suggested. As such, the public service interface is made AOP aware, and clients could remotely add and remove pre-approved aspects in the service implementation. (3) The third applicability of aspects involves deploying the

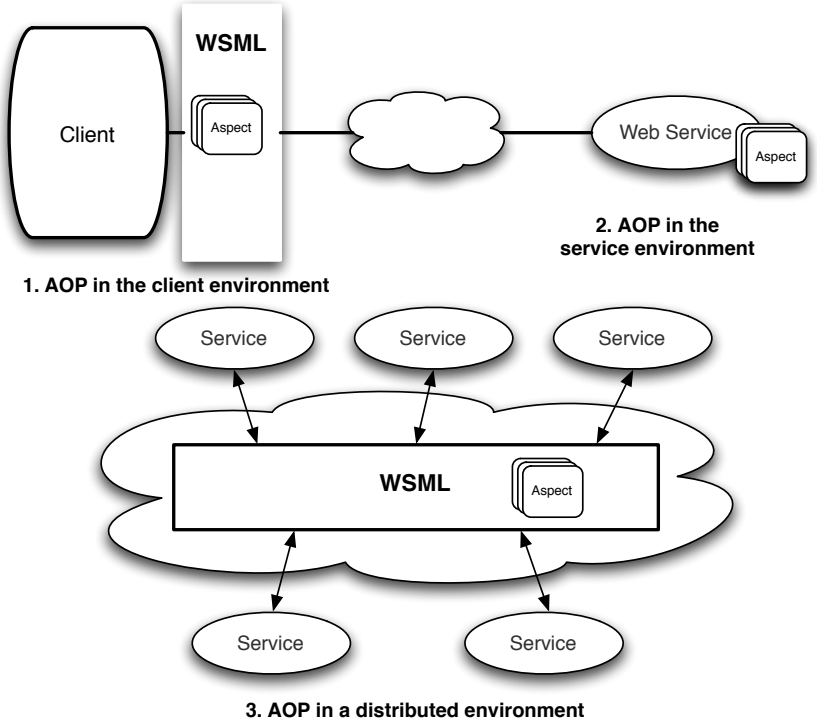

Figure 8.10: Applicability of AOP in SOA

aspect in a distributed fashion, meaning the underlying AOP technology requires distributed advice support and/or the aspect deployment requires remote pointcut constructors. Note that this kind of aspects cannot be deployed on third-party services that are not under the same control as the rest of the SOA as they require to be AOP-enabled, something that does not meet the original premises of this dissertation, stating that services belong to different providers. Nevertheless, in a controlled environment, distributed AOP can have many advantages to enforce concerns that crosscut the service boundaries.

8.8 Conclusions

In this chapter we have discussed how we can obtain implementation for the various service related aspects present in the WSML. Various options, including generation of aspect skeletons, full generation of service redirection aspects based on semantic web matchmaking and high-level service compositions specifications have been presented. For selection and management aspects, the option to provide them as reusable templates is chosen. Using dedicated XML deployment and configuration descriptors, these aspects can be instantiated in a concrete context, while the use of aspects as an implementation technology is hidden away. As such, instantiating and configuring the WSML framework can be done at the administration level, rather than on the implementation level.

A prototype of the WSML has been implemented in Java and JAsCo. It provides support

during development time to specify service types and include them in a client application using an Eclipse IDE plugin. Next, at deployment time, the WSML can be deployed together with the client, or run as a dedicated server. Using the XML descriptors, the WSML is configured and web services and service compositions are registered for the service types. Additional selection policies and management concerns are deployed accordingly. This configuration can be changed at runtime through a dedicated web-based administration console and services.

Finally, the WSML prototype has been integrated in a Service Enabling Platform (SEP) of Alcatel Bell. The SEP is a service-provisioning platform targeted for broadband Internet and communicates with a variety of management Web services and third-party content providing Web services. In a integrated demonstrator, the WSML is used to redirect client requests to Video on Demand (VoD) systems, while taking care of billing, accounting, promotional discounts and advertisements. The original implementations were unable to support these scenario's without having to change the code, but with the WSML these concerns were enforced non-invasively. Furthermore, using the service redirection mechanism of the WSML could redirect requests easily to the appropriate VoD or billing service without having to change any code in the SEP.

Chapter 9

Conclusions

Abstract In this chapter, the conclusions of this dissertation are presented. First, the ideas and work presented in this dissertation are summarised stressing on our contributions. Finally, directions for future research are discussed.

9.1 Summary and Contributions

The goal of this dissertation was to present and implement a mediation framework for the dynamic integration, composition, selection and client-side management of Web services.

Web service technology is an open standards-based mechanism for communication over a network. Web services are simple, self-contained applications that perform functions, from simple requests to complicated business processes. XML is used to encode all communications to a Web service. Client applications can easily integrate an existing Web service and communicate with it over a network, regardless of the hardware or platform used on either side of the wire. As Web services enable computer-to-computer communications in a heterogeneous environment, they are ideally suited for the Internet.

A Service-Oriented Architecture (SOA) is a term used for a type of application that integrates and relies on a number of remote Web services. We have observed that the implementation and deployment of a SOA integrating several third-party Web services, poses a variety of challenges, especially if those services are to be reached over an unpredictable network such as the Internet. We have identified these challenges and categorised them as events occurring in a *dynamic service environment*. These events result in a number of requirements for the service integration, selection and client-side management process of the Web services in the client applications. The current approaches dealing with these processes however, do not provide the needed runtime flexibility to deal with these events. Web services get hard-wired in the client, or only provide limited flexibility to integrate functionally equivalent services. The current dynamic service binding mechanisms do not offer explicit support for key requirements such as compositional mismatches, hot-swapping and service compositions. Explicit code needs to be added to the client application to deal with these issues. Not only does this code end up tangled and scattered in the client at every place where service functionality is required, it is also impossible to deal with unanticipated events.

If client applications want to take into account non-functional properties of Web services in order to select the most optimal service for a given request, we observe that there is no support available for this process. Current service registries only offer key-based searches to look-up functional compatible services. There is no way to explicitly specify selection policies based on the Quality-of-Service of the available services, or to take into account other criteria such as the client context, the client requests towards the services or to do quality control of the returned results, without explicitly providing code to deal with these concerns in the client at compile time. However, the notion of which service is most appropriate is based on the business requirements of the client, and may vary over time. A flexible way of specifying and enforcing service selection policies at runtime is needed.

Finally, we observe that invoking third-party services is far more complicated than making a local method invocation. A variety of concerns needs to be dealt with, including exception handling, billing, logging, service monitoring, authentication, etc. All of these concerns will need to be reflected in the code of the client. Again, current approaches require that code dealing with these concerns is provided explicitly and in advance, leading to tangled and scattered code. Furthermore, service providers may change their policies at any time requiring adaptations in the client. Current dynamic approaches provide limited

flexibility to deal with these issues, especially when the deployment of the concern is not limited to the message handling level. In that case, code has to be added explicitly in the client.

Our approach to tackle these issues, is the Web Services Management Layer (WSML), which is placed in between the Web services environment and the client. The WSML acts as a mediation layer to deal with all service related events, leaving the client untouched. Our approach is not intended to merely fix the flaws in the current Web service stack and the technologies built around it. Even with improved standard mechanisms and more robust services technologies, the services environment will remain heterogeneous, due to its fragmented nature. The fact that service clients need to co-evolve with the Web services they integrate will remain in the future as it is practically impossible to eradicate incompatibilities between two independent parties through standardisation. Instead of relying completely on standardisation it is a good design practice to assume incompatibilities will arise, and to put mechanisms into place to easily resolve them. The WSML is intended to be a generic framework offering exactly these capabilities. Next, we will list the contributions of this dissertation.

The observation that dynamic service environments impose a range of requirements on the client application and require a high level of flexibility in the client, and the observation that current approaches fail in providing adequate solutions to offer these runtime flexibility while realising better separation of service related concerns, lead to the first contribution:

> ***Contribution 1*** *- A requirements analysis for the integration, selection and management of Web services in dynamic service environments and an analysis of current approaches and state-of-the art tools [CV03], [VCJ03].*

To address the shortcomings of existing Web service integration approaches in dynamic service environments, we proposed an architectural framework for the mediation of Web services in client applications. An abstraction layer, called Web Services Management Layer (WSML), is placed in between the client and the world of Web services dealing with all service related concerns in a transparent way for the client. To modularise every cross-cutting concern, while providing the ability to deploy the concerns non-invasively, we have proposed to use *Aspect-Oriented Programming (AOP)*. Service communication and composition details, selection policies and service management concerns are all ideal candidates to be modularised in aspects. As the WSML requires a lot of runtime flexibility, we have opted to use a dynamic AOP approach, which allows for the hot deployment of aspects. This results in our second contribution:

> ***Contribution 2*** *- An architecture for a reusable service mediation framework targeted at dynamic service environments; adopting aspect-oriented design principles to implement this framework and the employment of dynamic AOP technologies for the flexible and dynamic configuration of the framework [VCJ03], [VCV+04], [CVV+06].*

The primary goal of the WSML framework was to provide a dynamic integration mechanism for services. For this purpose, *service types* were introduced. Clients communicate

with service types, which are a generic description of some service functionality without referencing concrete services. At runtime, client requests on a service type are redirected to an available service. This mechanism is based on dynamic AOP: all service communication details and composition details are modularised in *service redirection aspects*. By manipulating the around advice chains of the deployed aspects, services are dynamically bound to the client. This brings us to the third contribution:

> ***Contribution 3*** *- A dynamic service integration mechanism for Web services, based on service types and service redirection aspects, with support for hot-swapping, multiple and conditional service binding, compositional mismatches, service compositions, synchronous and asynchronous communication and conversational messaging.[VCV+04], [VJ05]*

A second goal of the WSML framework was to offer explicit support for selection policies specified by the client to guide the service integration process. Selection policies can specify the expected Quality-of-Service of the available services or take other criteria such as the client or service context into account. To enforce these policies we have proposed to modularise each policy in a *service selection aspect*. As such, the concerns are nicely modularised and can be deployed non-invasively so that the client code remains untouched. For policies that take the service behaviour into account we have set up measuring points using *service monitoring aspects*. Using aspects, unanticipated policies can be enforced in a running system whenever the client changes its selection policies. This leads to the fourth contribution:

> ***Contribution 4*** *- A service selection mechanism to guide the runtime selection of services based on the client or service context, and on non-functional and behavioural service properties by employing selection and monitoring aspects [VCJ04], [CV05]*

The third goal of the WSML framework was to support the enforcement of a wide variety of management concerns that are either specified by the service provider, or by the client itself. As the enforcement of these concerns may require crosscutting code while they evolve over time, we have encapsulated them in *service management aspects*. We have made implementations of various concerns in a reusable manner while dealing with possible feature interaction of the concerns by employing aspect combination strategies and connector priorities. This results in the following contribution:

> ***Contribution 5*** *- A client-side service management mechanism for the enforcement of various service and client-side driven concerns using service management aspects [VC04], [CVV+06].*

To obtain an implementation of the WSML aspects, we have investigated various approaches including aspect skeleton code generation, the usage of aspect template libraries for selection and management concerns and the full generation of redirection aspects using

semantic service documentation or high-level UML composition specifications. These approaches have been realised in a prototype, implemented in Java and JAsCo. The WSML prototype offers support at development time of client applications via its integration in the Eclipse IDE; at deployment time via deployment descriptors and at runtime via administration and monitoring console and administration services. This leads to the sixth contribution:

> ***Contribution 6*** *- A prototype of the WSML developed in Java and JAsCo and running on a production scale server. Tool support is realised for the development of service-oriented applications in Eclipse, a state-of-the-art IDE. Runtime configuration of the WSML is done through web-based administration interfaces and an XML-based configuration language [CVS+04a], [CVV+06]*

As a concrete case, the WSML has been deployed on the Service Enabling Platform (SEP) an open telecom platform for broadband service delivery of Alcatel Bell. The WSML was used in a Video-on-Demand scenario to take care of the billing procedures and the redirection to the appropriate content and mobile phone providers, which brings us to our final contribution:

> ***Contribution 7*** *- The successful deployment of the WSML on a Service Enabling Platform (SEP), an open telecom platform for broadband service delivery of Alcatel Bell to facilitate the integration with different content and mobile phone providers [VC05], [VVJ06].*

9.2 Future Work

9.2.1 High-Level Business Rules Language

As we have discussed in this dissertation, a lot of the decisions to deploy and configure the WSML are taken at the business level, for instance the specification of selection policies and the deployment of specific client-side management concerns. We have proposed a framework where each of these concerns is decoupled from the rest of the application. As a result, we have obtained a large amount of runtime flexibility and configurability. However, which aspects are to be deployed and how they are to be configured is done manually as aspect instances are created at runtime, e.g. through XML configuration descriptors. Specific aspects can however partially automate this task (for instance, a caching aspect can be triggered only when the service becomes too slow, something that is measured by a monitoring aspect), but even then, these aspects need to be deployed manually. Also, if a selection policy reasons about a service property that is not available in the system, this property has to be added manually in the system, or automatically through an additional monitoring aspect or another polling or notification mechanism.

The decisions when an aspect needs to be deployed are essentially business rules and the properties or features these rules reason about are low-level domain specific concepts. By externalising these business rules and by introducing a high-level domain model that allows

for their specification in terms of domain concepts, the instantiation of the WSML aspects can become more automated. Following a Model-Driven Engineering (MDE) approach, high-level rules and their connections in the WSML could be automatically translated into code. This approach has been proposed recently in [CDJ06]. The novelty of this approach is the use of AOP for mapping the domain model to implementation. In [CDJ06] it is shown how a high-level business rule language can be used to express and enforce the dynamic business rules that guide the customisation of the WSML: existing rules are externalised and new rules can be easily added, enhancing the adaptability of the WSML. This approach has three clear advantages: first, at runtime the varying conditions that guide the different selection, integration and management tasks offered by the WSML; second, we can non-invasively extend the core functionality of the WSML, abstracting from its technical complexity; third, it is possible to add rules that refer to runtime service properties that were not foreseen in the existing WSML implementation.

9.2.2 Usability Analysis

As discussed in Chapter 8, section 8.6, validation of the WSML approach is not trivial. Demonstrating or proofing the *usability* of our approach can de done through various criteria, including measuring its *flexibility, operability* and *end-user attractiveness*. In this dissertation, we have demonstrated the operability of our approach by developing a working prototype and deploying it in a complex real-world distributed setup. The supported flexibility of the prototype is determined by the set of events we support based on the requirements analysis made in Chapter 3. However, this does not guarantee that the level of flexibility offered is high enough to withstand any changes or evolutions in an actual third party service environment. A possible setup to measure the level of flexibility is to determine a set of representative third party Web services available on the Internet, integrate them through our approach in a client and then monitor if the system eventually breaks over time or not. Measuring the quality of the approach with respect to end-user friendliness could be done in a similar manner. Groups of test-users could be assigned in an experiment to use the WSML and its support tools. Note that these tools, including the WSML plugin for the Eclipse IDE, were designed and implemented as a proof-of-concept and should therefore be further tested and debugged before being deployed in such experiments. Another interesting criterium is the level of abstraction provided in our approach to hide away the use of aspect technology for the administrative level of WSML configuration. If the runtime flexibility and capabilities of the WSML turn out to be too limited, one will have to resort to the manual implementation of additional aspects, which obviously requires more expert skills. It can be assumed that this risk can be reduced by either extending the available aspect library before deployment time or by facilitating the implementation efforts for the developer. This can be done by providing additional tool support and API's or by improving the automatic generation of aspect code, the later being a research topic on its own.

9.2.3 Process Description Language

The service redirection aspects in the WSML contain service communication details or composition details. They are either implemented manually (possibly after a step of aspect skeleton generation), or they are generated out of mapping containing a series of statements or out of a high-level UML composition specification. Completely automated code generation is pursued using semantically annotated Web services and service types. Interesting future work is the further investigation of the automated determination of the compatibility between the service types and the concrete Web services, and the automatic generation of glue code, even if this involves the specification of (simple) service compositions. For complex compositions involving complete workflows or business processes, the automated mapping to executable code is another research track. One of the goals of Wit-case, the follow-up IWT-project of the Mosaic project, which involved the WSML, is to realise a service creation environment (SCE). This SCE focuses on the visualisation of service compositions, while offering support for crosscutting concerns through a dedicated aspect language, called Padus [BVJ+06].

9.2.4 Performance Modelling

One of the weak points of Web services technology is performance. For the developer of the client application it becomes a problem to make assumptions about the performance, as certain factors such as the Internet and third-party services fall outside his or her control. As a result, it is very difficult to make performance guarantees to the customer. The selection policies introduced in this dissertation are a first step to alleviate this problem as the WSML will only communicate with Web services that provide the desired Quality-of-Service. However, in order to to be able to specify selection policies that make sense for more advanced applications, one needs more advanced performance modelling. There are specific formalism available for performance modelling, such as the Layered Queuing Network models (LQN). More research is needed to apply these models to the complex case of applications where services are dynamically integrated and where the performance can be optimised by selecting other functionally equivalent services. Another possible research track is to specify high-level descriptions of the application, as done in [VDGD05] and to generate automatically performance models for it. From those performance models, early indications of the system performance can be extracted and validated at runtime.

9.2.5 Distributed WSML

In this dissertation, the WSML is primarily focussed on dealing with service related concerns from the client perspective while the Web services remain under control of the third-party service providers. The WSML mediates between the client and the services and *adapts* to the services. However, aspects can also play an important role in a more controlled environment where a set of services are deployed in a SOA and each service is under the same organisational control. In that case, truly *distributed aspects* can be deployed for a variety of concerns and scenarios. An example already discussed in this dissertation is distributed monitoring. As mentioned above, SOAs are frequently performance-critical. One way to

measure performance consists in setting up distributed measurement points as part of the SOA for each sub-processes, network communication and the involved Web services. Based on this, bottlenecks can be analysed and actions undertaken. For instance, calls to slow services can be distributed over multiple semantically equivalent services, network traffic on congested networks can be optimised by installing caching functionality and service invocations that take a long time can be executed in advance.

Another possible scenario are decentralized compositions. In approaches such as WS-BPEL and WSML, the compositions are centralised in one place, possibly causing a bottleneck. In a distributed approach, the different parts of the service composition become decentralized and each node deals with a particular subset of the business process at different locations within the distributed system. The nodes communicate directly with each other to transfer data and control, instead of relying on a central coordinator. In that case, the management concerns such as exception handling and logging also need to become distributed. In [NSVV06] we have recently proposed a distributed version of the WSML, where a dedicated distributed AOP language, Aspects with Explicit Distribution (AWED), is used for modularising concerns that need to be enforced on multiple hosts.

Appendix

A prototype of the Web Services Management Layer (WSML) has been developed in Java. It is available for download at http://ssel.vub.ac.be/wsml. The WSML runs on any machine with the Java 2 1.5 Runtime Environment installed. The system has been tested on various platforms, including Windows, Linux and MacOS X. The WSML relies on various technologies:

- **JAsCo:** the WSML uses JAsCo technology to compile and deploy aspects. The latest version of JAsCo is included in the WSML and automatically installed with the WSML prototype. *http://ssel.vub.ac.be/jasco*
- **Jetty:** Jetty is an open-source Java HTTP Server and Servlet Container used for the web-based administration console used to configure the WSML at runtime. The dynamic content of the console is generated using a combination of XSL Transformations and XML. Jetty is automatically installed with WSML. *http://jetty.mortbay.org*
- **Systinet Server:** When running as a server, the WSML relies on the Systinet Server for Java, an industry scale application server, for the deployment of the service types as Web services. Systinet Server is also used for the generation of client-side proxies of remote Web services and for the deployment of the administration services and test services. The Systinet Server for Java can be downloaded from the Systinet website. *http://www.systinet.com*
- **.NET:** for cross-platform testing, some test services are developed in C# and deployed on Microsoft Internet Information Services (IIS). An example scenario is included in the WSML prototype, with both Java and .NET clients and Web services. There is also a .NET version of the WSML administration console available. The .NET framework is an optional requirement. *http://msdn.microsoft.com/netframework/*

Installing the WSML prototype is done in four steps:

1. Make sure Java 2 1.5 Runtime environment is installed. The `bin` folder of the Java environment must be added to the system PATH environment variable.
2. Install the Systinet Server for Java. Add `JAVA_HOME` as a new environment variable. It should point to the installation directory of Java. Optionally you can add the /bin folder to the PATH environment variable.

3. Unzip the WSML.zip file to a folder. This will be the installation folder of the WSML. Run `setup.bat` or `setup.sh` located in the folder of the WSML to start the setup procedure. Follow the instructions of the installation wizard.

4. When the setup procedure completes, the WSML framework is installed. It can be started by running `WSML.bat` or `WSML.sh` in the WSML folder.

There are various options to configure the WSML. The easiest way is to use the WSML administration console, which is available at `http://localhost:8080` when the WSML is running. The console can be used to specify and deploy service types, Web services, service compositions, selection policies and management concerns. Every configuration is automatically saved as XML configuration files for persistent deployment. The XML configuration files can also be written and manipulated directly. They are located in the `XMLconfiguration` subfolder of the WSML. Alternatively, the WSML can enable a set of administration Web services that can be invoked from the client application, third-party Web services are other applications. More information is available in the included readme document.

Bibliography

[ABH+02] Ankolekar, A., Burstein, M., Hobbs, J.R. et al. *DAML-S: Web Service Description for the Semantic Web*. Proceedings of First International Semantic Web Conference (Sardinia, Italy, June 2002), ISWC'02, Lecture Notes in Computer Science, Volume 2342, pp.348-363.

[ACD+03] Andrews, T. , Francisco Curbera, F., Dholakia, H. et al. *Business Process Execution Language for Web Services (BPEL4WS), Specification version 1.1*. International Business Machines, Siebel Systems, May 2003. http://www.ibm.com/Developer Works/library/ws-bpel/

[ADH+02] Atkinson, B., Della-Libera, G., Hada, S., et al. *Web Services Security (WS-Security), Specification version 1.0*. International Business Machines Corporation, Microsoft Corporation, VeriSign, April 2002. ftp://www6.software.ibm.com/software/developer/library/ws-secure.pdf

[AH05] van der Aalst, W.M.P. and ter Hofstede A.H.M. *YAWL: Yet Another Workflow Language*. Information Systems, Elsevier, Volume 30, Issue 4, June 2005, pp. 245-275.

[AHM+03] Arsanjani, A., Hailpern, B., Martin, J. et al. *Web Services Promises and Compromises*. ACM Queue, Volume 1, Issue 1, ACM Press, New York, USA, March 2003, pp. 48-58.

[Ana05] Ananiev, A.S. *Implement a Web service that deals with complex XML document*, International Business Machines Developer Works, November 2005. http://www-128.ibm.com/Developer Works/webservices/library/ws-complexml.html

[Apa02] Apache Software Foundation, *Apache Axis Web Services Framework*. 2002, home page at http://ws.apache.org/axis/

[Apa03] Apache Software Foundation, *Apache Web Services Invocation Framework (WSIF)*. 2003. home page at http://ws.apache.org/wsif/

[BA01] Bergmans, L. and Aksit, M. *Composing Crosscutting Concerns Using Composition Filters*. Communications of the ACM, Volume 44, Issue 10, ACM Press, New York, USA, October 2001, pp. 51-57.

[Ban02] Ban, B. *JGroups, a toolkit for reliable multicast communication*, 2002, home page at http://www.jgroups.org/

[BB03] Burke, B. and Brock, A. *The Aspect Oriented Programming and JBoss tutorial*, OnJava, O'Reilly Media, May 2003. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html

[BBB+02] Banerji, A., Bartolini, C., Beringer, D. et al. *Web Services Conversation Language (WSCL), Specification version 1.0.* W3C Note, w3.org, March 2002.

[BBC+05] Bilorusets, R.. Box, D., Cabrera, L.F. et al. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging) Specification.* W3C Member Submission W3C February 2005.

[BBD+05] de Bruijn, J., Bussler, C., Domingue, J. et al. *Web Service Modeling Ontology (WSMO).* W3C Member Submission, June 2005.

[BCC+04a] Box, D., Christensen, E., Curberaet, F. et al. *Web Services Addressing (WS-Addressing) Specification.* W3C Member Submission, August 2004.

[BCC+04b] Bellwood, T., Capell, S., Clement, L. et al. *Universal Discovery, Discovery, and Integration (UDDI) Specification version 3.02,*October 2004.

[BCC+04c] Box, D., Cabrera, L., Critchley, C. et al. *Web Services Eventing (WS-Eventing)*, W3C Member Submission, August 2004.

[BCH+03] Box, D., Curbera, F., Hondo, M. et al. *Web services policy framework (WS-Policy), Specification version 1.1.* International Business Machines, Microsoft Corporation, et.al., May 2003.

[BFD99] Berners-Lee, T., Fischetti, M. and Dertouzos, T. M. *Weaving the Web.* HarperCollins, New York, NY, USA, first edition, 1999.

[BHMO04] Bockisch, C., Haupt, M., Mezini, M., and Ostermann, K. *Virtual machine support for dynamic join points.* In Proceedings of the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004). AOSD '04. ACM Press, New York, NY, USA, pp. 83-92.

[BM04] Baligand, F. and Monfort, V. *A concrete solution for web services adaptability using policies and aspects.* Proceedings of International Conference on Service-Oriented Computing (New York, NY, USA, November 2004), ICSOC 2004, pp. 134-142.

[Boner04] Bonér, J. *What are the key issues for commercial AOP use – how does AspectWerkz address them.* Proceedings of the Third International Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, ACM Press, New York, NY, 2004, pp. 5-6.

[BSP+01] Burstein, M,., Sycara, K., Paolucci, M., et al. OWSL-S, web ontology language. 2001, home page at http://www.daml.org/services/owl-s/

[BSV06] Navarro, L.D.B., Südholt, M., Vanderperren, W., et al. *Explicitly distributed AOP using AWED.* Proceedings of the Fifth International Conference on Aspect-Oriented Software Development (Bonn, Germany, March 2006), AOSD'06, ACM Press, New York, Npp. 51-62.

[Burn03] Burner, M. *The Deliberate Revolution*, ACM Queue, Volume 1, Issue 1, pp. 28-37, ACM Press, New York, USA, March 2003.

[Bus00] The Business Rules Group. *Defining Business Rules: What Are They Really?* home page at http://www.businessrulesgroup.org/, July 2000.

[Butek05] Butek, R. *Which style of WSDL should I use?* Internation Business Machines Developer Works, May 2005, http://www-128.ibm.com/Developer Works/webservices/library/ws-whichwsdl

[BV85] Brans, J.P. and Vincke, P. *A Preference Ranking Organisation Method: (The PROMETHEE Method for Multiple Criteria Decision-Making*). Management Science, Volume 31, Issue 6, June 1985, pp. 647-665.

[BVJ06] Braem, M., Verlaenen, K., Joncheere, N., et al. *Isolating Process-Level Concerns Using Padus.* Proceedings of the fourth International Conference on Business Process Management (Vienna, Austria, September 2006), BPM 2006, LNCS Springer-Verlag. (to appear)

[BW03] Balke, W.-T., Wagner, M. *Towards Personalized Selection of Web Services.* Proceedings of the twelfth International World Wide Web Conference (Budapest, Hungary, May 2003), WWW '03, ACM Press, New York, NY, USA, 2003.

[CBR03] Colyer, A., Blair, G. and Rashid, A. *Managing Complexity in Middleware.* Proceedings of the at Workshop on Aspects, Components and Patterns for Infrastructure Software at The Second International Conference on Aspect-Oriented Software Development (Boston, MA, March 2003), AOSD'03.

[CCF+05] Cabrerra, L.F., Copeland, G., Feingold, M., et al. *Web Services Atomic Transaction (WS-AtomicTransaction) Specification version 1.0,* August 2005, available at ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf

[CCMW03] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. *Web Services Description Language (WSDL) Specification version 1.2.* W3C Technical Documents, W3C Web Services Activity, March 2003.

[CDJ03] Cibrán, M. A., D'Hondt, M., Jonckers, V. *Aspect-Oriented Programming for Connecting Business Rules*. Proceedings of the sixth International Conference on Business Information Systems (Colorado Springs, CO, June 2003), BIS'03.

[CDJ06] Cibrán, M. A., D'Hondt, M. and Jonckers, V. *Mapping high-level business rules to and through aspects.* journal L'Object, Hèrmes (to appear 2006).

[CDS03] Cibrán, M. A., D'Hondt, M., Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo for Linking Business Rules to Object-Oriented Software.* Proceedings of International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (Rio De Janeiro, Brazil, June 2003), CSITeA'03, pp. 1-7.

[CE05] Cottenier, T. and Elrad, T. *Dynamic and decentralized service composition with contextual aspect-sensitive services.* Proceedings of the First International Conference on Web Information Systems and Technologies (Miami, USA, May 2005), WEBIST 2006.

[CFF+04] Czajkowski, K., Ferguson, D.F., Foster, I. et al. *The Web Services Resource Framework (WSRF).* Specification version 1.0, The Globus Alliance, August 2004.

[Chin03] Chinnici, R., *Java(TM) API for XML-based Remote Procedure Call (JAX-RPC) Specification.* Sun Microsystems, October 2003, pp. 167.

[Chung91] Chung, L. *Representation and utilization of non-functional requirements for information system design.* Proceedings of the third International conference on Advanced information systems engineering (Trondheim, Norway, 1991), CAiSE '91, Lecture Notes in Computer Science , Volume 498, Springer-Verlag, New York, NY, 1991, pp. 5-30.

[CIJ+00] Casati, F., Ilnicki, S., Jin, L et al. *Adaptive and Dynamic Service Composition in eFlow.* Proceedings of the third International conference on Advanced information systems engineering (Stockholm, Sweden, June 2000), CAiSE 2000, Lecture Notes in Computer Science, Volume 1789, Springer, New York, NY, 2000, pp. 13-31.

[CJ01] Chakraborty, D. and Joshi, A. *Dynamic Service Composition: State of the Art and Research Directions*" Technical Report TR-CS-01-19, Department of Computer Schience and Electrical Engineering, University of Maryland, 2001.

[CM04] Charfi, A and Mezini, M. Aspect-Oriented Web Service Composition with AO4BPEL" Proceedings of the European Conference on Web Services (Växjo, Sweden, 2004), ECOWS'04, Lecture Notes in Computer Science, Volume 3250, Springer, New York, NY, 2004.

[CSH+04] Cibrán, M. A., Suvée, D., D'Hondt, M., Vanderperren, W. and Jonckers, V., *Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming.* Proceedings of the fifth Argentine Symposium on Software Engineering (Córdoba, Argentina, September 2004), ASSE 2004.

[CV03] Cibrán, M. A. and Verheecke, B. *Modularizing Web Services Management with AOP*, Proceedings of the First European Workshop on Object-Orientation and Web Services at the seventeenth European Conference on Object-Oriented Programming, (Darmstadt, Germany, July 2003), ECOOP'03.

[CV05] Cibrán, M. A. and Verheecke, B. *Dynamic Business Rules for Web Service Composition.* Proceedings of the Second Dynamic Aspects Workshop (DAW05), the fourth International Conference on Aspect-Oriented Software Development (Chicago, Il, USA, March 2005), AOSD'05, RIACS Technical Report 05.01, 2005, pp. 13-18.

[CVJ03] Cibrán, M. A., Verheecke, B. and Jonckers, V. *Modularizing Client-Side Web Service Management Aspects.* Proceeding of the Second Nordic Conference on Web Services (Växjo, Sweden, November 2003), NCWS'03, Vajxo University Press, Series: Mathematical Modelling in Physics, Engineering and Cognitive Sciences, Volume 8, pp. 1-12.

[CVS+04a] Cibrán, M. A., Verheecke, B., Suvee, D. and Vanderperren, W. *A web services management layer for the selection, integration and management of web services.* Formal Research Demo at the eighteenth European Conference on Object-Oriented Programming (Oslo, Noway, June 2004), ECOOP 2004.

[CVS+04b] Cibrán, M. A., Verheecke, B., Suvee, D., Vanderperren, W. and Jonckers, V. *Automatic Service Discovery and Integration using Semantic Descriptions in the Web Services Management Layer.* Proceedings of Third Nordic Conference on Web Services (Växjo, Sweden, November 2004), NCWS'04, Journal of Mathematical modelling in Physics, Engineering and Cognitive Sciences, Volume 11, 2004, pp.79-89.

[CVV+06] Cibrán, M. A., Verheecke, B., Vanderperren, W., Suvee, D. and Jonckers, V. *Aspect-Oriented Programming for Dynamic Web Service Selection, Configuration and Management.* World Wide Web Journal (WWWJ), Springer, 2006. (to appear).

[DEM02] Duclos, D., Estublier, J. and Morat. P. *Describing and using non functional aspects in component based applications.* Proceedings of the first international Conference on Aspect-Oriented Software Development (Enschede, The Netherlands, April 2002), AOSD'02, ACM Press, New York, NY, USA, 2002, pp. 65-75.

[Deu91] Deutch, P. J. *The Eight Fallacies of Distributed Computing.* defined at Sun Microsystems Labs, 1991-92, http://today.java.net/jag/Fallacies.html

[DFS02] Douence, R., Fradet, P. and Südholt, M. "*A framework for the detection and resolution of aspect interactions.* The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (Pittsburgh, PA, USA, October 2002), GPCE'02, Lecture Notes in Computer Science, Volume 2487, Springer-Verlag, 2002, pp. 173–188.

[DFS04] Douence, R., Fradet, P. and Südholt, M. *Composition, Reuse and Interaction Analysis of Stateful Aspects.* Proceedings of the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, ACM Press, New York, NY, 2004, pp. 141-150.

[DJ04] D'Hondt, M., Jonckers, V. *Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge*. Proceedings of the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, ACM Press, New York, NY, 2004, pp. 132-140.

[Dun02] Dunham, M.H. *Data Mining, Introductory and Advanced Topics*. Prentice Hall, Upper Saddle River, New Jersey, first edition, August 2002, pp. 336.

[DVS05] De Fraine, B., Vanderperren, W., Suvee, D. et al. *Jumping Aspects Revisited*. Proceedings of the Second Dynamic Aspects Workshop (DAW05), Research Institute for Advanced Computer Science, Technical Report 05.01, workshop at the fourth International Conference on Aspect-Oriented Software Development (Chicago, Il, USA, March 2005), AOSD'05, pp. 77-86.

[EFB01] Elrad, T., Filman, R.E. and Bader, A. *Aspect-oriented programming: Introduction*, Communications of the ACM, Volume 44, Issue 10, pp, 29-32, ACM Press, New York, USA, October 2001.

[EH99] Edmond. D. and Hofstede, A, *Achieving Workflow Adaptability by means of Reflection*. ACM SIGGROUP bulletin, Volume 20, Issue 3, December 1999, pp. 10.

[EMP05] Erradi, A., Maheshwari, P., Padmanabhuni, S. *Towards a Policy-Driven Framework for Adaptive Web Services Composition* Proceedings of the first International IEEE Conference on Next Generation Web Services Practices (Seoul, South Korea, August 2005), NWeSP 2005, IEEE Press, pp. 33-38.

[FB02] Fensel, D. and Bussler, C. "*The Web Service Modeling Framework (WSMF)*", Electronic Commerce Research and Applications, Volume 1, Issue 2, 2002, pp. 113-137.

[FBL+02] Filman, R.E., Barrett, S. Lee D. et al. *Inserting Ilities by Controlling Communications*. Communications of the ACM, Volume 45, Issue 1, January 2002, pp. 116-122.

[FFG04] Foster, I., Frey, J., Graham, S. et al., *Modeling stateful resource with Web services* Whitepaper, International Business Machines Developer Works, January 2004.

[FGM+99] Fielding, R., Gettys, J., Mogul, J. et al. *Hypertext Transfer Protocol – HTTP/1.1*. Internet RFC 2616 , W3C, October 1999, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[FGWP01] Feng, N., Gang, A., White, T., and Pagurek, B. *Dynamic Evolution of Network Management Software by Software Hot-Swapping* Proceedings of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (Seattle, WA, USA, May 2001), IM 2001, pp. 63-76.

[FICA04] Filman, R.E., Elrad, T., Clarke S. and Aksit, M. *Aspected-Oriented Software Development*. Addison Wesley Professional, October 2004, pp. 800.

[FR03] Fleury, M. and Reverbel, F. "*The JBoss Extensible Server*", Proceedings of the fourth Middleware International Conference (Rio de Janeiro, Brazil, January 2003), Lecture Notes in Computer Science, Volume 2672, Springer, 2003.

[FS97] Fowler, M. and Scott K. *UML distilled: applying the standard object modeling language*. Addison-Wesley Longman Ltd. Essex, UK, 1997.

[FS02] Farias, A. and Südholt, M. "*On components with explicit protocols satisfying a notion of correctness by construction*. International Symposium on Distributed Objects and Applications 2002, (Irvine, CA, USA, October 2002), DOA'02, Lecture Notes in Computer Science, Volume 2519, Springer, 2002.

[FW04] Fallside, D.C., and Walmsley., P. *XML Schema Part 0: Primer, Second Edition*. W3C Recommendation Document, October 2004.

[Gar03] Garfinkel. T. *Traps and pitfalls: Practical problems in system call interposition based security tools*. Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS), February 2003, pp. 163-176.

[GGT03] Gibbs, K., Goodman, B. and Torres, E. *Create Web services using Apache Axis and Castor*. Technical Article, International Business Machines Developer Works, September 2003, http://www-128.ibm.com/DeveloperWorks/webservices/library/ws-castor/

[GHJ95] Gamma, E., Helm, R., Johnson, R. et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, first edition, January 1995, pp. 395.

[GHM+03] Gudgin, M., Hadley, M., Mendelsohn, N., et al. "*SOAP Version 1.2 Part 1: Messaging Framework*", W3C Recommendation, June 2003.

[GKPG+05] Grosof, B. N. , Kabbaj, Y. , Poon, T. C. , Ghande, M. et al., "*Semantic Web Enabling Technology (SWEET)*", http://ebusiness.mit.edu/bgrosof/

[GNYW01] Gu, X., Nahrstedt, K., Yuan, W., Wichadakul, D. and Xu, D., *An xml-based quality of service enabling language for the web*. Technical report, Champaign, IL, USA, 2001.

[Hall01] von Halle, B. *Business Rules Applied Building Better Systems Using the Business Rules Approach*. Wiley, first edition, September 2001, pp. 464.

[HHJ+99] Heinl, P., Horn, S., Jablonski, S. et al. "*A comprehensive approach to flexibility in workflow management systems*. Proceedings of the International Conference on Work activities Coordination and Collaboration (San Francisco, California, USA, February 1999), WACC '99, ACM Press, 1999, pp. 79-88.

[HHK02] Hofreiter, B., Huemer, C., Klas, W. *ebXML: status, research issues, and obstacles*. Proceedings of the Twelfth International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems (San Diego, CA, USA, February 2002), RIDE-2EC 2002, pp. 7-16.

[HM00] Hendler, J. and McGuinness, D. *The DARPA Agent Markup Language*. IEEE Intelligent Systems, Volume 15, Issue 6, November/December 2000, pp. 72–73.

[HM01] Haarslev V. and Möller, R. *RACER system description*. Proceedings of International Joint Conference on Automated Reasoning (Siena, June 2001), IJCAR'01, Lecture Notes in Computer Science, Volume 2083, Springer, 2001, pp. 701.

[JHA+05] Johnson, R. , Hoeller, J. , Arendsen, A. et al. "*Spring- Java/J2EE Application Framework*". Reference Documentation Version 1.2.8, http://www.springframework.org/docs/reference/index.html

[JRM04] Jaeger, M.C., Rojec-Goldmann, G. and Muhl, G. *QoS Aggregation for Web Service Composition using Workflow Patterns*. Proceedings of Eighth IEEE International Enterprise Distributed Object Computing Conference (Monterey, California, USA , September 2004), EDOC'04, pp. 149-159.

[KB04] Karastoyanova, D. and Buchmann, A. *Extending Web Service Flow Models to Provide for Adaptability*", Proceedings of Workshop on Web Services and Service-Oriented Architecture Best Practices and Patterns, at the Nineteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (Vancouver, Canada, October 2004), OOPSLA 2004.

[KL03] Keller, A. and Ludwig, H. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management, Volume 11, Number 1, Springer, Netherlands, March 2003, pp. 57-81.

[KLM+97] Kiczales, G., Lamping J., Mendhekar, A., et al. *Aspect-Oriented Programming*. Proceedings of the eleventh European Conference on Object-Oriented Programming (Jyväskylä, Finland, June 1997), ECOOP 1997, Lecture Notes in Computer Science, Volume 1241, Springer, 1997, pp.220-242.

[Kirt00] Kirtland, M. *The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework*. MSDN Magazine, Volume 15, Number 9, September 2000.

[KG02] Kienzle J., and Guerraoui, R. *Aop: Does it make sense? the case of concurrency and failures*. Proceedings of the sixteenthEuropean Conference on Object-Oriented Programming (Málaga, Spain, June 2002), ECOOP 2002, Lecture Notes in Computer Science, Volume 2374, Springer, 2002.

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J. et al. *An overview of AspectJ*. Proceedings of the fifteenth European Conference on Object-Oriented Programming (Budapest, Hungary, June 2001), ECOOP'01, Lecture Notes in Computer Science, Volume 2072, Springer, 2001.

[KK04] Keidl, M. and Kemper, A. *Towards context-aware adaptable web services*. Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers & Posters (New York, NY, USA, May 2004). WWW Alt. '04. ACM Press, New York, NY, pp. 55-65.

[KR03] Kleijnen, S. and Raju, S. *An Open Web Services Arcitecture.* ACM Queue, Volume 1, Issue 1, ACM Press, New York, USA, March 2003, pp. 38-46.

[KRRS96] Kappel, G., Rausch-Schott, S. , Retschitzegger, W. and Sakkinen, M. *From rules to rule patterns.* Proceedings of the International Conference on Advanced Information Systems Engineering, (Heraklion, Crete, Greece, May 1996), CAiSE'96, Lecture Notes in Computer Science, Volume 1080, Springer, 1996, pp. 99-115.

[LH89] Lieberherr, K. and Holland, I. *Assuring Good Style for Object-Oriented Programs.* IEEE Software, Volume 6, Issue 5, September 1989, pp. 38-48.

[Lind05] Lindsay, G. *Increase Your Application's Reach Using WSDL to Combine Multiple Web Services.* MSDN Magazine, March 2005. http://msdn.microsoft.com/msdnmag/issues/05/03/WSDL/default.aspx

[LNZ04] Liu, Y., Ngu, A.H. and Zeng, L.Z. *QoS computation and policing in dynamic web service selection.* Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers & Posters (New York, NY, USA, May 2004). WWW Alt. '04. ACM Press, New York, NY, 66-73.

[LOO01] Lieberherr, K., Orleans, D. and Ovlinger, J. *Aspect-oriented programming with adaptive methods.* Communications of the ACM, Volume 44, Issue 10, ACM Press, New York, USA, October 2001, pp. 39-41.

[LPP+05] Loughran, N. ,Parlavantzas, N., Pinto, M., Fernández, L.F. et al. *Survey of aspect-oriented middleware research.* Lancaster University, Lancaster, AOSD-Europe Deliverable D8, AOSD-Europe-ULANC-10, June 2005, pp. 115.

[LS05] Loughran, S., Smith, E. *Rethinking the Java SOAP Stack.* Proceedings of the IEEE International Conference on Web Services (Orlando, FLA, USA, July 2005), ICWS'o5, IEEE, 2005.

[LSS05] Lohmann, D., Schroder-Preikschat, W. and Spinczyk, O. *Functional and Non-Functional Properties in a Family of Embedded Operating Systems.* Proceedings of the tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (Sedona, Arizona, USA, February 2005) Words 2005, IEEE, pp. 413-420.

[LVV03] Lämmel, R., Visser, E. and Visser, J. *Strategic Programming Meets Adaptive Programming.* In Proceedings of The Second International Conference on Aspect-Oriented Software Development (Boston, MA, March 2003), AOSD'03, ACM Press, New York, NY, USA, 2003, pp. 168-177.

[MMN+06] Mostéfaoui, G.K., Maamar, Z., Narendra, N. C. et al. *Decoupling Security Concerns in Web Services Using Aspects*, Proceedings of the 3th International Conference on Information Technology (2006), ITNG'06, pp. 20-27.

[MN02] Mani, A. and Nagarajan A., *Understanding quality of service for Web services*, International Business Machines Developer Works, January 2002. http://www-128.ibm.com/Developer Works/library/ws-quality.html

[MO03] Mezini, M. and Ostermann, K. *Conquering Aspects with Caesar*. Proceedings of the 2nd international Conference on Aspect-Oriented Software Development (Boston, MA, USA, March 2003). AOSD'03. ACM Press, New York, NY, USA, 2003, pp. 90-99.

[MPM+05] Martin, D., Paolucci, M., McIlraith, S. et al. *Bringing Semantics to Web Services: The OWL-S Approach*. Proceedings of First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) at the IEEE International Conference on Web Services (San Diego, CA, USA, July 2004), ICWS'04, Lecture Notes in Computer Science, Volume 3387, Springer, 2004.

[Ms03] Microsoft. *Microsoft Visual Studio.NET 2003*. home page at http://msdn.microsoft.com/vstudio/previous/2003/

[MS04] Maximilien, E.M., Singh, M.P. *A framework and ontology for dynamic Web services selection*. Internet Computing, IEEE, Volume 8, Issue 5, September 2004, pp. 84-93.

[MSDN04] MSDN Network. *Application Integration*. MSDN Academic Alliance, Microsoft MainFunction, July 2004.

[MTSM03] McGovern, J., Tyagi, S., Stevens, M. and Mathew, S. *Java Web Services Architecture*. Morgan Kaufmann, July 2003, pp. 831.

[Ngh02] Nghiem, A *IT Web Services: A Roadmap for the Enterprise*. Prentice Hall PTR., October 2002, pp. 336

[NST04] Nishizawa, M., Shiba, S. and Tatsubori, M. *Remote pointcut, a language construct for distributed AOP*. Proceedings of the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, ACM Press, New York, NY, 2004, pp. 7-15.

[NSVV06] Navarro, L.D.B., südholt, M., Vanderperren, W. Verheecke, B. *Modularization of distributed web services using AWED*. Submitted to the eight International Symposium on Distributed Objects and Applications (Montpellier, France, October 2006), DOA06.

[OT01] Ossher, H. and Tarr, P. *Using multidimensional separation of concerns to (re)shape evolving software*. Communications of the ACM, Volume 44, Issue 10, ACM Press, New York, USA, October 2001. pp. 43-50.

[PAG03] Popovici, A., Alonso, G. and Gross, T. *Spontaneous Container Services*. Proceedings of the seventeenthEuropean Conference on Object-Oriented Programming, (Darmstadt, Germany, July 2003), ECOOP'03.

[Parn72] Parnas, D.L. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, Volume 15, Issue. 12, ACM Press, New York, USA, May 1972, pp. 1053-1058

[PDS+04] Pawlak, R., Duchien, L., Seinturier, L. et al. *JAC: An Aspect-based Distributed Dynamic Framework*. Journal Software Practice and Experience (SPE), Volume 34, Issue 12, Wiley, October 2004, pp. 1119-1148.

[PG03] Papazoglou, M. P. and Georgakopoulos, D. *Service Oriented Computing*. Communications of the ACM, Volume 46, Issue 10, ACM Press, New York, USA, October 2003. pp. 25-28.

[PGA02] Popovici, A., Gross, T. and Alonso, G. *Dynamic weaving for aspect-oriented programming.* Proceedings of the first international Conference on Aspect-Oriented Software Development (Enschede, The Netherlands, April 2002), AOSD'02, ACM Press, New York, NY, USA, 2002, pp. 141-147.

[PKP+02] Paolucci, M., Kawamura, T., Payne, T.R. and Sycara K. *Semantic Matching of Web Services Capabilities*. Proceedings of First International Semantic Web Conference (Sardinia, Italy, June 2002), ISWC'02, Lecture Notes in Computer Science, Volume 2342.

[PRJL04] Pérez, J., Ramos, I., Jaén, J. and Letelier, P. "*PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*", Proceedings of the International Conference on Quality Software (Dallas, Texas, USA, November, 2003), QSIC'03, pp. 59-66.

[PSC+01] Pulvermüller, E., Speck, A., Coplien, J.O., et al. *Proceedings of the Workshop on Feature Interaction in Composed Systems* at the fifteenth European Conference on Object-Oriented Programming (Budapest, Hungary, June 2001), ECOOP'01, Technical Report No. 2001-14, 2001.

[PSDF01] Pawlak, R., Seinturier, L. Duchien, L. and Florin, G. *JAC: A flexible solution for aspect-oriented programming in Java*. Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Kyoto, Japan, September 2001). Reflection'01, Lecture Notes In Computer Science, Volume 2192. Springer-Verlag, London, pp. 1-24.

[PSL03] Patel, C., Supekar, K. and Lee, Y. "*A QoS Oriented Framework for Adaptive Management of Web Service based Workflows*, Database and Expert Systems 2003 conference (Prague, Czech Republic, September 2003) DEXA'03, Lecture Notes In Computer Science, Volume 2736, Springer, pp. 826-835.

[PV04] Peer, J., Vukovic, M. *A Propopsal for a Semantic Web Service Description Format*. Proceedings of the Second European Conference on Web Services (Erfurt, Germany, September 2004), ECOW'04), Lecture Notes In Computer Science, Volume 3250, Springe, 2004, pp. 285-299.

[Port03] Portier, B. *Invoking Web services with Java clients, A look at Web services clients in the J2SE and J2EE environment*. Internet Business Machines Developer Works, November 2003.

[PWWR03] Parastatidis, S., Webber, J., Watson, P. and Rischbeck, T., *A Grid Application Framework based on Web Services Specifications and Practices*. Technical

Report, North East Regional e-Science Centre, University of Newcastle, August 2003.

[Raj98] Raj, G.S. *A Detailed Comparison of CORBA, DCOM and Java/RMI.* September 1998, available at http://my.execpc.com/∼gopalan/misc/compare.html

[Ran03] Ran, S., " *A Model for Web Services Discovery with QoS.* ACM SIGecom Exchanges, Volume 4, Issue 1 ACM Press, New York, NY, USA, 2003, pp. 1-10.

[RDR00] Rouvellou, I., Degenaro, L., Rasmus, K., et al. "*Extending business objects with business rules*", Proceedings of Technology of Object-Oriented Languages (St-Malo, France, June 2000), TOOLS'o1, pp. 238-249.

[RKF92] Rosenberry, W., Kenney, D. and Fisher, G. *Understanding Distributed Computing Environment (DCE).* O'Reilly & Associates, Inc. Sebastopol, CA, USA, First edition, October 1992, pp. 258.

[Rol05] Roloux, M. *Specifying and deploying web service compositions using dynamic aspects.* Diploma thesis, Vrije Universiteit Brussel, Belgium, June 2005.

[RP94] Reynolds, J. and Postel, J. *Assigned Numbers.* STD 2, RFC 1700, October 1994.

[RS05] Ritter, J., Stussak C. *Stub and Skeleton Generation for a Single-Sign-On Web Service supporting Dynamic Objects.* Poster Session at the Third IEEE European Conference on Web Services (Växjo, Sweden, November 2005), ECOWS 2005.

[SA03] Sumra, R. and Arulazi, D. *Quality of Service for Web Services-Demystification, Limitations, and Best Practices.* International Business Machines Developer Works, March 2003.

[San03] Patil, S. and Newcomer, E. *ebXML and Web Services.* IEEE Internet Computing, Volume 07, Issue 3, , IEEE, May/June 2003, pp. 74-82.

[SDV06] uve, D., De Fraine, B. and Vanderperren, W. *A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development.* To be Published in Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (Västerås, Sweden, June 2006), CBSE 2006, in Lecture Notes In Computer Science, 2006.

[SFCV+05] Suvee, D., De Fraine, B., Cibrán, M., Verheecke, et al. W. *Evaluating FuseJ as a Web Service Composition Language.* Proceedings of the third European Conference on Web Services (Vaxjo, Sweden, November 2005), ECOWS'05, IEEE Computer Society, 2005, pp.25-35.

[SGHS05] Singh, S., Grundy, J., Hosking, J and Sun, J. *An architecture for developing aspect-oriented web services*:, Proceedings of the third European Conference on Web Services (Växjo, Sweden, November 2005), ECOWS'05, IEEE Computer Society, 2005, pp. 72-82.

[Szyp01] Szyperski, C. *Components and Web Services.* Beyond Objects column, Software Development, Volume 9, Issue 8, August 2001.

[STP04] Sandoz, P., Triglia, A., and Pericas-Geertsen, S. *Fast Infoset*. Technical Articles, Sun Developer Network, June 2004.

[SVJ03] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*. Proceedings of the 2nd international Conference on Aspect-Oriented Software Development (Boston, MA, USA, March 2003). AOSD'03. ACM Press, New York, NY, USA, 2003, pp. 21-29.

[Sun05] Sun, "*Java 2 Platform Standard Edition 5.0 Overview*", 2005, home page at http://java.sun.com/j2se/1.5.0/docs/guide/

[Sys05] Systinet, "*Systinet Server for Java 6.0 Primer*", White Paper, Systinet, 2005, pp. 166.

[Tane88] Tanenbaum, A. S. *Computer Networks*. Prentice Hall Professional Technical Reference, first edition, 1988, pp. 628.

[TBE05] Taher, L., Basha, R. and El Khatib, H. *QoS Information & Computation (QoS-IC) Framework for QoS-Based Discovery of Web Services*. Upgrade Journal, CEPIS, Volume 6, Issue 4, August 2005, pp. 55-66.

[TOH+99] Tarr, P., Ossher, H., Harrison, W. et al. *N degrees of separation: Multi-dimensional separation of concerns*. Proceedings of the twenty-first International Conference on Software Engineering (Los Angeles, CA, USA, May 1999), ICSE'99, IEEE Computer Society, 1999, pp. 107-119.

[TPP03] Tosic, V., Pagurek, B. and Patel, K. "*WSOL – A Language for the Formal Specification of Classes of Service for Web Services*," Proceedings of the International Conference on Web Services (Las Vegas, USA, June 2003), ICWS'03.

[Truy04] Truyen, D. *Dynamic and Context-Sensitive Composition in Distributed Systems.* PhD Thesis, K.U.Leuven, Belgium, November 2004.

[Tyagi04] Tyagi, S. *Patterns and Strategies for Building Document-Based Web Services*. Sun Developer Network (SDN), September 2004.

[Van04] Vanderperren, W. *Combining Aspect-Oriented and Component-Based Software Engineering*. Ph.D. Thesis, Vrije Universiteit Brussel, Belgium, 2004.

[VC04] Verheecke, B. and Cibrán, M. A. *Dynamic Aspects for Web Service Management*. Proceedings of the Dynamic Aspect Workshop (DAW'04) at the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, RIACS Technical Report 04.01, 2004, pp. 146-152.

[VC05] Verheecke, B. and Cibrán, M. A. *Dynamic Aspects in Large Scale Distributed Applications: An Experience report*. Proceedings of the Software engineering Properties of Languages for Aspect Technologies Workshop (SPLAT'05) at the fourth International Conference on Aspect-Oriented Software Development (Chicago, Il, USA, March 2005), AOSD'05.

[VCJ03] Verheecke, B., Cibrán, M. A. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web services in Client-Applications*. Proceedings of the International Conference on Web Services (Erfurt, Germany, September 2003), ICWS'o3-Europe, Lecture Notes In Computer Science, Volume 2853, Springer, 2003, pp. 137-151.

[VCJ04] Verheecke, B., Cibrán, M. A. and Jonckers, V. *Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection*. Proceedings of the European Conference on Web Services (Växjo, Sweden, 2004), ECOWS'04, Lecture Notes in Computer Science, Volume 3250, Springer, New York, NY, 2004, pp,15-29.

[VCV+04] Verheecke, B., Cibrán, M. A., Vanderperren, W., Suvee, D. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web Services*. International Journal of Web Services Research (JWSR), Volume 1, Issue 3, July-Sept 2004, pp. 25-41.

[VDGD05] Verdickt, T., Dhoedt, B., Gielen, F. and Demeester, P. *Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models*. IEEE Transactions on Software Engineering, volume 31, issue 8, August 2005, pp. 695-711.

[Ver03] Verheecke, B. *Web Services Management Layer (WSML)*. 2003, home page at http://ssel.vub.ac.be/wsml

[Ver04] Vermeir, D. *Discovery and Selection of Web Services through Semantic Annotations*. Diploma thesis, Vrije Universiteit Brussel, Belgium, June 2003.

[Vid97] Videira Lopes, C. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[Vin97] Vinoski, S. *CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments*. Communications Magazine, Volume 35, Issue 2,? IEEE, February 1997, pp. 46-55.

[VJ05] Verheecke, B. and Jonckers, V. *Stateful Aspects for Conversational Messaging with Stateful Web Services*. Proceedings of the IEEE Next Generation Web Services Practices (Seoul, South-Korea, August 2005), NWeSP'05, IEEE Press, 2005, pp. 363-370.

[Vogels03] Vogels, W. *Web Services are not Distributed Objects*. IEEE Internet Computing, Volume 7, Issue 6, November 2003, pp. 59-66.

[VSC+04] Vanderperren, W., Suvee, D., Cibrán, M. A. and Verheecke, B. *Automatic runtime evolution of web services with JAsCo dynamic AOP*. Software Evolution and Aspect-Oriented Programming Symposium, Ghent, Belgium, May 2004.

[VSCV+05] Vanderperren, W., Suvee, D., Cibrán, M. A., Verheecke, B. et al. *Adaptive Programming in JAsCo*. Proceedings of the fourth International Conference on Aspect-Oriented Software Development (Chicago, Il, USA, March 2005), AOSD'05, pp. 75-86.

[VSV+04] Vanderperren, W., Suvee, D., Verheecke, B. and Cibrán, M. A. *JAsCo & WSML: AOP for Component-Based Software Engineering applied to a Web Services Management Layer*. Formal Research Demo at the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04.

[VSV+05] Vanderperren, W., Suvee, D., Verheecke, B., et al. *Automatic Feature Interaction Analysis in PacoSuite*. Journal of Systemics, Cybernetics and Informatics (JSCI), January 2005.

[VVJ06] Verheecke, B., Vanderperren, W. and Jonckers, V. *Unraveling Crosscutting Concerns in Web Services Middleware*. In IEEE Software journal, Volume 23, Issue 1, January 2006. pp.42-50.

[VVSV03] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V. *JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services*. Proceeding of the Second Nordic Conference on Web Services (Växjo, Sweden, November 2003), NCWS'03, Vajxo University Press, Series: Mathematical Modelling in Physics, Engineering and Cognitive Sciences, Volume 8, pp. 125-136.

[VS04] Vanderperren, W. and Suvee, D. *Optimizing JAsCo dynamic AOP through HotSwap and Jutta*. Proceedings of the Dynamic Aspect Workshop (DAW'04) at the third international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 2004), AOSD'04, RIACS Technical Report 04.01, 2004, pp. 120-134.

[VSCD05] Vanderperren, W., Suvee, D., Cibrán, M. A. , De Fraine, B. *Stateful Aspects in JAsCo*. Proceedings of Software Composition 2005, LNCS, (Edinburgh, UK, April 2005), SC'05, Lecture Notes in Computer Science, Volume 3628, Springer, New York, NY, 2004, pp.167-182.

[VSJ03] Vanderperren, W., Suvee, S., Jonckers, V. *Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo*. Proceedings of the 2003 Net.ObjectDays International Conference (Erfurt, Germany, September 2003), NODe'03, 2003, pp. 36-50.

[W3C04] World Wide Web Consortium. *Web Services Glossary*. W3C Working Group Note, February 2004, available at http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211.

[Wal98] Waldo, J. *Remote procedure calls and Java Remote Method Invocation. Concurrency*. IEEE Parallel & Distributed Technology, Volume 6, Issue 3, July 1998, pp. 5-7.

[WKL03] Wu, P., Krishnamurthi, S. and Lieberherr, K. *Traversing Recursive Object Structures: The Functional Visitor in Demeter*. Proceedings of Software Engineering Properties of Languages for Aspect Technologies Workshop (SPLAT'o3) at The Second International Conference on Aspect-Oriented Software Development (Boston, MA, March 2003), AOSD'03.

[WJD03] Wohlstadter, E., Jackson, S. and Devanbu. P. *Dado: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems.* Proceedings of Twenty-fifth International Conference on Software Engineering (Portland, Oregon, USA, May 2003), ICSE'03, 2003, pp. 174.

[WTM+04] Wohlstadter, E., Tai, S. Mikalsen, T. et al. *GlueQoS: middleware to sweeten quality-of-service policy interactions.* Proceedings of the twenty-sixth International Conference on Software Engineering, (Edinburgh, UK, May 2004), ICSE 2004, pp. 189- 199.

[Za03] P. Zave. *An experiment in feature engineering.* Monographs In Computer Science, Springer-Verlag, New York, New York, NY, USA, pp. 353-377.

[ZJ03] Zhang, C. and Jacobsen, H.A. "Refactoring Middleware With Aspects," IEEE Transactions on Parallel and Distributed Systems, Volume 14,Issue 11,November 2003. pp. 1058-1073.