Modeling Services using Contracts
Identifying Dependencies in Service–Oriented Architectures

Thibault Estier[1], Beat Michel[2], and Oliver Reinhard[3]

[1] HEC - Université de Lausanne
1015 Lausanne, Switzerland
thibault.estier@unil.ch
[2] Beat Michel Conseil en Informatique
1030 Bussigny, Switzerland
michel@beatmichel.ch
[3] Paranor AG
3046 Wahlendorf, Switzerland
oliver.reinhard@paranor.ch

Abstract. Design by contract is a well-established paradigm in software engineering. Bertrand Meyer first introduced the rigorous distinction between the responsibilities of service provider and service consumer for fine grain software artifacts (classes). This paper considers service contracts in the context of service-oriented architecture for complex enterprise information infrastructures. Identifying dependencies between applications with service contracts may help to master the complexity of numerous interconnected information systems and to ease evolution towards a service-oriented architecture. This paper proposes both a model and a methodology to systematically apply the notion of contract for structuring relationships and identifying dependencies between applications in a service oriented architecture.

1 Introduction

While enterprise-wide IT infrastructures face every day more complex challenges in managing and developing application interoperability, Service-Oriented Architectures (SOA) [4] gain progressively more and more success as an integrative paradigm. Reasoning about applications interfaces in terms of services seems a natural extrapolation of software engineering concepts for components assembly: a software component, or module, should expose a clearly defined set of operations and properties, should give strict conditions of usage, without exposing the details of how the component executes these operations. Decomposing software into modules was already introduced in 1972 by D. Parnas [2].

A clear separation between the purpose of a component (what) and its actual implementation (how) gives interesting properties (reusability of the component, low coupling between a component and its users, etc.). These are also desirable properties of

[4] While it is difficult to trace a unique origin of the term SOA, it seems to have appeared around 2000 in both IBM research papers and Gartner group reports, before being fixed by standards [1].

related applications in an IT infrastructure. This naturally promoted the idea of *unambiguous interface definitions* for information exchanges between applications. The SOA approach encourages the definition of exposed properties and operations available outside an application to be defined in terms of services. While this idea already appeared in distributed applications and with middleware buses (like DCOM or CORBA for instance), the recent apparition and success of web-oriented standards for information interchanges refuelled it (Web Services and XML encoding of information). Unfortunately it is generally difficult to handle an enterprise application as a "large software component": the definition of an appropriate and reusable interface for a large application is generally a complex task. The model proposed in this paper helps to better express the offered services of an application, by providing a complete and rigorous specification of each service.

Bertrand Meyer introduced in 1992 [3] the notion of *Design by Contract* to help formalizing the exposed behaviour of a software component, in terms of pre-conditions, post-conditions and invariants. One of the extensions of a service description proposed here includes definitions of pre- and post-conditions for each operations offered in a service. So we take the word *contract* for a service in a similar meaning, rather than the emphasis on legal involvement of both parties.

In this paper, we recognize the importance of web services for handling applications interoperability between organisations or in heterogenous contexts, but we focus on integration inside a given IT infrastructure (Enterprise Applications Integration), possibly but not necessarily implemented using web services. In the global map of an IT infrastructure, applications may offer services based on very different protocols and mechanisms, including traditional file transfers, publish/subscribe systems, etc. We propose a service description independent of the underlying mechanism used to make the service accessible. This description gives also important information about failures, failure signals, how they will propagate and when they may arise.

Related Work

Heckel & Lohmann recently proposed [4] to also adopt pre- and post-conditions for extending Web Services definitions. The behaviour of service operations is then given by UML collaboration diagrams transformation rules. Their proposition is to use these graphs for automatic testing of a web service.

Tosic, Pagurek and Patel [5] proposed a language called Web Service Offerings Language (WSOL), to extend a WSDL description with formal expressions of various constraints, including pre- and post-conditions, plus a notion of future-conditions. The language enables declaration of several business deployment characteristics of a service (like Quality of Service or Service Level Agreement). These definitions are oriented toward management, monitoring and measurement of deployed services, when the usage of a service available in a different organization implies some garanties for the application willing to use it.

In the domain of Web Services, several XML-based language extending WSDL [6] have been proposed: WSLA (Web Services Level Agreements) from IBM [7], and WSML (Web Service Management Language) coming from a proposition by HP [8]. Both propositions are also focused on web services and on formalization of service–level agreements.

In the second section of this paper, we propose and illustrate a complete concept of Service Contract, covering typical questions that a designer must handle when defining a service: orientation, scope, structure, semantics and quality. The third section sketches briefly a methodology applicable to migrating a traditional infrastructure to a service-oriented architecture, using systematically this service modeling approach.

2 Service Model and Service Contracts

A *service contract* is a mutual agreement between the provider of a service and the consumer (or consumers) of the service. Like a legal contract, a service contract defines the conditions and terms under which the provider and the consumers will collaborate. A service contract can be expressed using a *service model*. The model defines the *orientation* and type of dependencies which exist between parties, and defines four aspects of each service:

1. **Operations and Structures** – the service operations, their parameters and parameter types,
2. **Semantics and Scope** – the behavior of the service operations, returned results and side effects; do they overlap the information scope of other services,
3. **Failures** – when the consumer-provider interaction will not succeed,
4. **Quality** – what non-functional requirements does the service comply to.

While the service model defines the interaction between providers and consumers, an actual service contract based on the model identifies the actual provider and consumer(s) and is only established at deployment time or even at run time.

Quality of service, is also an important aspect of a service model, however, it is not the focus of this paper and is not further addressed. The other aspects are presented as sub-sections of this section.

Example illustrating the model presentation

We illustrate the various aspects of the model using an example: a DVD-rental business with an inventory of movies and carries zero or more DVD copies of each movie. The corresponding IT architecture consists of a server-based DVD-rental application and two types of user-interface clients: the internet-based home client supports customers reservation request; the staff client at the store is for reservation, rental and customer management by the staff.

Orientation – Establishing a service relation

The relationship between service provider and consumers is asymetric: consumers applications depends on the provider application. Service model dependencies may be:

- **information dependency** – the consumer needs *information* supplied by the service provider, in which case he addresses a *request for information* (RFI)
- **delegation dependency** – the consumer delegates part of its *processing* to the service provider, in which case he addresses a *request for processing* (RFP).

This distinction is orthogonal to that of different modes of interaction:

- **blocking mode** – the consumer remains suspended until the provider has finished processing its request, even if it has no information to return,
- **non-blocking mode** – the consumer may resume before the provider starts processing its request; sometimes the consumer may wish to retrieve a possible response or status later,
- **publish-subscribe mode** – the consumer subscribes to a certain type of information supplied by the provider. The consumer is later notified by the provider each time a corresponding information item is available.

Blocking and unblocking modes may be used for both RFI and RFP, while publish–subscribe generally implements an RFI. We will come back in section 3 on the fact that the service orientation cannot be derived simply from the data-flow direction. In our

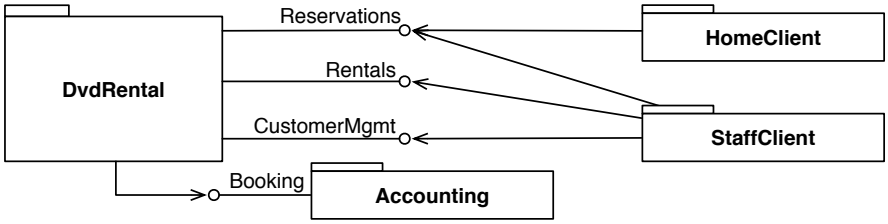

Fig. 1. Service contracts and roles

DVD-rental example, a *DvdRental* application is the service provider that owns and updates the rental information. The *HomeClient* acts as a service consumer making reservation requests from the user's home. The *StaffClient* is another service consumer, this time for the rental staff at the shop. Both client applications interact with the server in blocking mode (see Fig. 1).

Operations and Structures – Syntax

A service is a set of closely related service operations. Operations are closely related if they use the same parameter types and access or modify the same persistent information. Each operation has a signature (name, formal parameters, failure signals).

The closure of all types of all parameters of all operations belonging to a given service is the service type model (STM). The STM is part of the service model.

Semantics and Scope – About producing effects

The core type model (CTM) is the minimal set of types, attributes and associations required to specify the service operations. Sharing a CTM between services operations allows to specify how the side effects of one service affects another service.

Instances of CTM types have identity and persistent state, they represent the state between operation invocations of the services sharing the CTM. They are generally

not passed around as operation parameters, and they are not accessible to service consumers. Fig. 2 shows the CTM for the DVD-rental application.

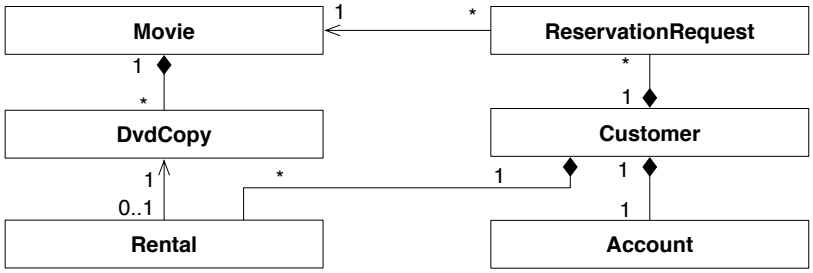

Fig. 2. Core-type model for DVD-rental application

The semantics (the behaviour) of a service operation is defined over both its parameters and the CTM. The scope of a service operation includes all the CTM types and attributes referenced by this operation's specification.

This approach goes one step beyond classical *design by contract* [3] where the method scope is normally the class itself. In class contracts, pre- and postconditions are specified over the method parameters and over the instance variables. For service operations, pre- and post-conditions are specified over both the operation parameters and the type instances of the CTM, as suggested by Cheesman and Daniels [9]. The pre- and post-conditions specifications are declarative. Ideally a predicate language such as OCL [10] is used for formal specification, but natural language making strict use of the type models and the associations between types yields good results.

Failures – About not producing effects

The design-by-contract paradigm is radical and unambiguous: $preconditions \wedge operations \Rightarrow postconditions$. This postulate implies that the service provider is free to do anything if preconditions are not satisfied – including rendering all of its managed information corrupt. But service providers have a second crucial mission: ensure integrity and consistency of owned information. The service model declares how preconditions failures will be handled:

- *Checked preconditions* – the service model declares all preconditions of the operation as checked by the provider and defines failure signals which are raised in the negative case. The service consumer can safely rely on the provider check.
- *Unchecked preconditions* – the service model explicitly declares the effect of a given service operation as undefined if its preconditions are not satisfied. The service consumer can only expect a correct result if it guarantees the precondition.

In practice, unchecked preconditions ideally suit RFIs, whereas checked preconditions are a natural match for RFPs. In some cases, RFIs may also require checked preconditions, if the returned information depends crucially from parameters provided and from state informations in the provider.

3 Towards a methodology for Service Oriented Architecture migration

Motivation for a SOA migration

The perspective of this paper with respect to SOA is that of the evolution of existing large enterprise wide IT-infrastructure, as opposed to the implementation of a SOA from scratch. What we have in mind is a company with multiple historically grown applications which size may vary from an application for employees-car park and a fully fledged ERP.

Most applications will not operate in isolation but will exchange information. Data exchange mechanisms may use any type of technologies like: file transfer, data replication, socket communication, messaging, CORBA or SOAP. Migration to a SOA then means implementing data exchange between applications as service relationships.

Although the final result will raise applications equipped with service interfaces, there is more to be done for a SOA-Migration. It should start with a reverse engineering and modelling effort before re-engineering the application landscape.

Architectural principles and migration process

A successfull SOA-Migration should be guided by following architectural principles:

- *Separation of concerns*: service relationships must be based on cleaarly established responsabilities of each participating application.
- *Loose coupling*: service relationships necessarily couple the service consumer to the producer. But coupling should be loose where this is possible.
- *Reusability*: This is the minimal requirement for services. It should be a requirement even when a service is supposed to be used only by one consumer. Reusability means that the producer should not expect specificities of the consumer, other than conditions specified in the service model.

Separation of concerns implies that for each application, all the supplied services be specified. These are not only the services for other applications but also those supplied to business processes, typically through user interfaces. We call the latter *business services*.

We suggest the following steps when applying these principles to a SOA-Migration:

1. Identify the scope of the migration effort and identify all data flows between applications in the scope.
2. Determine, clarify and optimize dependencies between applications, starting from existing data flows.
3. Formally specify the service contracts as explained in section 2.
4. Choose the most appropriate technology for implementing each service model.
5. Implement and deploy service.

Separation of concerns is an important aspect of step 2. In fact the direction of dependency is **not** immediately given by the direction of the data flow. A data flow from application A to B may as well be the interpreted as an information dependency of B from A than as a delegation dependency of A from B. Suppose a file of customer transactions being transferred from *DvdRental* to *Accounting*. Does this represent an information dependency of *Accounting* on *DvdRental* or rather a delegation dependency of *DvdRental* on *Accounting*? It may well be that John in charge of *DvdRental* thinks that the Accounting application needs his data to do their accounting, whereas Mary in charge of Accounting argues that DvdRental needs her processing to have their accounts settled.
How a dependency is modelled at this step is an important decision for the separation of concerns. In our example (see Fig. 3) one model would mean that *DvdRental* provides the business service of handling and billing rental events including book keeping, but the latter is delegated to accountability. In this perspective *DvdRental* is delegation dependent from *Accounting*. An other way is to consider *Accounting* as providing book keeping services to the business. To do so it needs information from *DvdRental* and thus is now information dependent from this application.

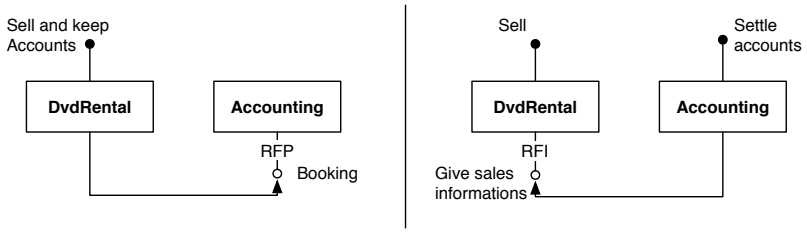

Fig. 3. Orientation: two different ways to associate applications by service

In step 3 loose coupling may be achieved by avoiding blocking RPC each time it makes sense. The alternative to blocking RPC depends on the kind of service dependency: RFI or RFP. In the case of RFI, an RPC mechanism may be replaced by a publish subscribe protocol, where the service provider notifies events to the service consumer. In the case of RFP loosest coupling may be achieved when the service provider does not return any information about the success of the processing. This means that the service contract garantees processing for all data delivered that conforms to the pre-conditions and all possible failures are handled by the service provider. This requires careful specification of the service contract.
It appears that the case where delegation dependency is realized with a service that does not return any success confirmation looks very similar to a publish/subscribe type information dependency (in reverse direction). Thus, one might suggest that a typical pattern consists in replacing a blocking delegation dependency by an publish/subscribe type information dependency in reverse direction. This may well be interpreted as a argument for event driven service architecture where autonomous applications with

clear responsibilities, providing well defined business services, publish events that are used by others.
Choosing the most appropriate technical implementation should be a consequence of the design decisions discussed so far, of available technologies and feasablity in a given environment. Web services is definitely not the only way of implementig SOA. Message oriented middleware (possibly combined with SOAP) may be an excellent solution for RFP but also to implement publish/subscribe style RFI. However we think that even file transfer may still have its role to play in a service architecture.

4 Conclusion

Service models and contracts encourage designers and maintainers of a Service–Oriented Architecture to make explicit descriptions of their services functionalities. The model proposed in section 2 shows that this may go well beyond definition of services signatures and parameters types. Analysis of interfaces between applications in terms of service contracts may help to master crucial issues of enterprise-wide systems integration. While migrating an architecture towards a SOA approach, we recommend a process where service models and contracts are used as a formalism organize at best the overall dependencies between existing applications before making any change in the technology used for the implementation of services.

Further research need being made on migration projects of different scales to measure the impact of this approach and its effect on time, when evolution of information systems become necessary without missing out on the benefits of existing and running services. The hypothesis we would like to verify is that this approach lowers the effort and cost of evolution of an IT infrastructure.

References

1. OASIS: Reference model for service oriented architectures. Working draft, OASIS open group (2005)
2. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12) (1972) 1053–1058
3. Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10) (1992) 40–51
4. Heckel, R., Lohmann, M.: Towards contract-based testing of web services. ENTCS **82**(6) (2004)
5. Tosic, V., Pagurek, B., Patel, K.: WSOL - a language for the formal specification of classes of service for web services. In Zhang, L.J., ed.: ICWS, CSREA Press (2003) 375–381
6. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL), http://www.w3.org/TR/wsdl (2001)
7. Keller, A., Ludwig, H.: The WSLA framework: Specifying and monitoring service level agreements for web services. J. Network Syst. Manage. **11**(1) (2003)
8. Sahai, A., Durante, A., Machiraju, V.: Towards automated SLA management for web services. Technical report, Software Technology Laboratory, HP Laboratories Palo Alto (2002)
9. Cheesman, J., Daniels, J.: UML Components, A Simple Process for Specifying Component-Based Software. Addison-Wesley (2001)
10. OMG: Unified modeling language (UML), version 2.0, http://www.omg.org/technology/documents/formal/uml.htm (2004)