Application of Data Engineering Techniques to Design of Message Structures for Web Services

George Feuerlicht

Faculty of Information Technology,
University of Technology, Sydney,
P.O. Box 123 Broadway, Sydney, NSW 2007, Australia
jiri@it.uts.edu.au

Abstract. Unlike components, Web Services are primarily intended for inter-enterprise applications that use Internet as the underlying transport mechanism, and consequently are subject to a different set of design considerations than intra-enterprise applications. Most practitioners recommend the use of coarse-grained, message-oriented Web Service that minimize the number of messages and avoid the need to maintain state information between invocations. In this paper we argue that excessive use of coarse-grained, document-centric message structures results in poor reuse and undesirable interdependencies between services. We describe a design approach that provides a framework for designing message payloads for service-oriented applications. We treat the problem of message design from a data engineering perspective and apply data normalization techniques to the design of service interfaces. We consider the impact of increasing message granularity on cohesion and coupling of service operations and discuss the associated design tradeoffs.

1 Introduction

Web Services are used mainly in inter-enterprise applications that rely on the Internet as the underlying transport mechanism, and consequently are subject to a different set of design considerations than intra-enterprise applications. Internet-scale distributed applications must be able to deal with unreliable connections, provider site failures, network latency, and trust issues. Most practitioners recommend the use of coarse-grained (i.e. services with aggregate data structures), message-oriented Web Services on the basis that coarse-grained Web Services generate fewer SOAP messages and therefore have lower communication overheads and less opportunity for failure [1]. Another motivation for using messages with aggregated message payloads is to avoid the need to maintain state information between individual requests. Proponents of the REST (Representational State Transfer) model [2] argue that Internet-scale distributed applications should rely entirely on stateless HTTP-based communications with coarse-grained XML message payloads. Given slow response times, high latencies, and poor reliability of the present Internet environment, performance is clearly an important consideration. However the design of Web Services should not be driven by performance-related objectives alone. The choice between fine-grained and

coarse-grained services is a choice between two extreme design options, each with different impact on performance as well as software engineering properties of service-oriented applications. Granularity (i.e. level of aggregation) of Web Services determines the scope of functionality that a given service (or service operation) implements, and is a key determinant of reusability and maintainability of service-oriented applications. While coarse-grained services achieve performance advantages by reducing the number of network interactions required to implement a given business function, when considered from the perspective of software engineering they suffer from a number of significant drawbacks, including limited reuse and poor maintainability. Coarse-grained Web Services are characterized by complex message structures that arise from designing message payloads to include all the information needed to perform the corresponding business function (e.g. airline booking). Complexity of messages is further increased by embedding business rules and constraints within the message data structures [3]. To illustrate this point consider, for example, the OTA (OpenTravel Alliance, www.opentravel.org/) flight booking business process. OTA defines request/response message pairs for individual business processes, and implements the flight booking business process using the OTA_AirBookRQ/OTA_AirBookRS message pair. The flight booking request document OTA_AirBookRQ (www.opentravel.org/downloads/2002B_XSD/OTA_AirBookRQ.xsd) is a complex, aggregate document that contains a large number of data elements (many optional) including flight booking, itinerary, traveler and payment details. The underlying assumption is that all of the information is available at the time of booking the flight, and that the airline tickets are paid for when the flight is booked. In practice, however flight booking and payment are often performed separately resulting in duplication of information and potential for data inconsistencies. The complexity and redundancy of message data structures makes it difficult to evolve the specification without producing undesirable side-effects that invalidate existing applications. Message payloads that externalize complex data structures result in high levels of data coupling and interdependencies between services, violating a fundamental design objective for distributed applications (i.e. minimization of coupling). Decomposing the flight booking request into separate, lower-granularity operations (e.g. flight enquiry, flight booking, payment, etc.) leads to simplification of the interface, improved flexibility, and potential for reuse (e.g. payment operation can be reused in another context, e.g. car rental, or a hotel room booking).

From a software engineering perspective, service interfaces need to be designed to maximize cohesion and minimize coupling [4], [5]. Maximization of cohesion refers to the requirement for methods to implement a single conceptual task and is closely related to reusability and maintainability of application components. High level of cohesion produces orthogonal services and improves the stability of the application as modifications can be typically confined to a specific service, or service operation. Minimization of coupling (i.e. interdependencies between services), results in improved ability to accommodate change. Applying these principles to service design leads to improved clarity of the interfaces, reduction in undesirable side effects, and improved flexibility of applications [6], [7]. Such requirements tend to favor finer-granularity services, and therefore conflict with performance considera-

tions. Balancing performance and software engineering considerations involves design tradeoffs and requires good understanding of the impact of service aggregation on cohesion and coupling of service-oriented applications.

In this paper we consider the problem of designing message structures for service-oriented applications from a data engineering perspective, applying data normalization rules to service interface parameters. In the following section (section 2) we briefly review research literature dealing with the design of Web Services applications. We then describe a framework for the design of message structures for service-oriented applications (section 3) and use this framework to design a set normalized interfaces (section 4). We then evaluate the impact of increasing message granularity on cohesion and coupling (section 5). In the concluding section (section 6) we summarize the benefits of the proposed design framework.

2 RELATED WORK

Web Services design approaches can be broadly classified into methodologies based on object-oriented design [8], [9], [10], [11], [12], methods for transformation of industry domain specifications [13], and business process transformation approaches [4], [14], [15], [16]. For example, Papazoglou and Yang [4] describe a design methodology that gives a set of service design guidelines based on the principles of minimizing coupling and maximizing cohesion to ensure that the resulting services are self-contained, modular, extendable and reusable. The methodology produces definition of WSDL Web Service interfaces and WSFL service flow models, and also includes non-functional service design guidelines that relate to service provisioning strategies and service policy management models. Web Services design is an active research area and while there is some agreement about the basic design principles there are no widely accepted design methodologies that can guide designers of Web Services applications. The focus of this paper is on the design of message data structures that form the basis of interaction between services and determine the software engineering properties of service-oriented applications.

3 DESIGN FRAMEWORK

From an architectural point of view service-oriented applications can be considered at different levels of abstraction. From one perspective they can be regarded as distributed systems that use message interchange as the basic communication mechanism, i.e. messages are regarded as the key artifacts of service-oriented applications. Message-oriented approaches and Message-Oriented Middleware (MOMs) have been used extensively in the context of Enterprise Application Integration (EAI) for the implementation of loosely-coupled, asynchronous applications. Alternatively, service-oriented applications can be viewed as programmatic environments that use procedure calls to execute local and remote procedures (RPCs). RPC-based programming environment is typically (but, not necessarily) used to implement synchronous,

tightly-coupled applications. We exploit this duality between messages and procedures and describe a design framework that leverages object-oriented design principles and data engineering techniques for the design of message structures for service-oriented applications.

3.1 Procedures vs. Messages

Procedures typically implement well-defined functions and use simple data parameters. However, it is possible to pass complex objects (e.g. XML documents) as procedure parameters, in effect using RPCs to interchange documents. Given this programmatic perspective, the interface contract is the signature of the corresponding procedure call (service operation), for example:

FlightEnquiry(INPUT: OriginLoc, DestinationLoc, DepartureDate,
OUTPUT: FlightNumber)

Given the message-oriented, document-centric perspective, message payloads (i.e. XML documents within SOAP envelopes) define the interface contract. For example, the XML schemas of the messages OTA_AirBookRQ and OTA_AirBookRS constitute the interface contract and specify the method signature as:

BookFlight(INPUT: OTA_Air_BookRQ,
OUTPUT: OTA_Air_BookRS)

Importantly, these abstractions are independent of the physical implementation of Web Services application that the designer may eventually choose. So that adopting the programmatic perspective during the design stage does not imply that the implementation of services will be based on synchronous RPCs. It is, for example, possible to conduct the design using the programmatic perspective and adopt the document style, asynchronous Web Service implementation. We regard decisions about the implementation style (i.e. binding style, RPC or document) and interaction model (i.e. synchronous or asynchronous, stateful or stateless) as orthogonal concerns to the task of designing the service interface, and defer such decisions to the implementation stage of the systems development process. This separation of concerns allows focus on interface design without introducing implementation dependent constraints during early design stages. From the design point of view, taking the document-centric perspective makes it difficult to reason about design tradeoffs associated with different message design strategies (e.g. level of message aggregation). However, changing the level of abstraction from messaging to programmatic interactions and regarding the messages structures as service interfaces makes it possible to apply well-established program design techniques to Web Services message payloads.

3.2 Design Principles

Using the programmatic perspective, the task of designing interfaces for service-oriented applications is conceptually similar to design of methods for object-oriented applications. The guiding principles for interface design include orthogonality (i.e. each interface should define a distinct function), maximization of method cohesion

and minimization of method coupling. Cohesion and coupling have been studied extensively in the context of structured and object-oriented programming [17], [18]. Myers [19] defined module cohesion as a degree of interaction within programming modules and coupling as the degree of interaction between programming modules, and classified both measures according to type. According to Myers, the highest levels of cohesion are Informational (all functions within a module share the same data) and Functional cohesion (module performs a single function). Minimal (i.e. the most desirable) types of coupling are Stamp coupling, where modules use data structures as parameters, and Data coupling where individual data elements are used as parameters. Thus the combination of Functional cohesion and Data coupling produces the most desirable situation from the point of view of reuse and maintainability. To achieve the highest level of cohesion the designer must ensure that service operations use the same data structures (i.e. Informational cohesion) and that each service operation (i.e. method) implements a well-defined, atomic task (Functional cohesion). Importantly, high level of method cohesion leads to orthogonality as functional overlap is minimized, or eliminated altogether. The requirement for data coupling dictates that interfaces consist of individual data parameters rather than complex data structures. Furthermore, using individual data parameters for interface specification rather than coupling via complex data structures (i.e. Stamp coupling) enables the application of data engineering techniques to minimize interdependencies between service operations, as described in the following sections.

3.3 Design Steps

The definition of service interfaces involves specification of operations and corresponding input and output parameters. This task is similar to designing method signatures in the context of object-oriented design, and involves identifying suitable candidate methods that are progressively refined to produce a set of well-defined service interfaces [5], [20]. The design framework consists of three design stages: initial design of service interfaces, refining interface design using interface normalization, and finalizing design by adjusting interface granularity. We base our design examples on the OTA the airline availability request/response messages: OTA_Air_AvailRQ/OTA_Air_AvailRS and booking request message pair: OTA_Air_BookRQ/ OTA_Air_BookRS.

Decomposition of the Flight Booking business function can be achieved by modeling the interaction between a travel agent and an airline using a Sequence Diagram. Each step in the Sequence Diagram dialog produces a Request/Response message pair and corresponds to an elementary business function [5]. Alternatively, elementary business functions can be identified as leaf functions in a business function hierarchy [21]. The resulting service interfaces correspond to elementary business functions as illustrated by the FlightEnquiry interface below:

FlightEnquiry(INPUT: OriginLocation, DestinationLocation, DepartureDate, OUTPUT: FlightNumber, DepartureAirport, ArrivalAirport, DepartureTime, ArrivalDate, ArrivalTime)

We can now apply interface normalization to detect extraneous interface parameters that can be removed in order to minimize data coupling between interfaces, and at the same time improve the cohesion of the operations.

4. Interface Normalization

Normalized data structures have been used extensively in database design [22]; we use the same principles here in order to minimize data coupling of service interfaces. Data coupling involves two or more interfaces being coupled via interface parameters, i.e. output parameters of one interface match input parameters of another. Removing data parameter interdependencies for the input and output parameter sets will ensure that both parameter sets are minimal (i.e. do not contain redundant parameters). We classify service operations according to type into query (i.e. operations that return data in output parameters given a query specified using input parameters) and update operations (i.e. operations that update data given update operation specified using input parameters), and formulate the following interface design rules [5]:

Rule 1: *Input parameters of query and update operations should form a minimal set, i.e. individual data parameters must be mutually independent.*
Rule 2: *Output parameters of query and update operations should form a minimal set, i.e. individual data parameters must be mutually independent.*
Rule 3: *Output parameters of query operations must be fully functionally dependent on input parameters.*

We regard the interfaces of query operation as relations where the input parameter set corresponds to the relation key, and the output parameter set are the non-key attributes. Output parameters of normalized interfaces are fully functionally dependent on the input parameter set, i.e. the interface parameters form a BCNF (Boyce-Codd Normal Form) relation. This ensures that parameters are used as data, not as control parameters and avoids Control coupling that involves using interface parameters to control the execution of the method [19]. Normalization of interfaces of query operations also ensures mutual independence of interfaces parameters for both input and output parameter sets (i.e. input and output parameter sets are minimal). Update operations, in general, do not exhibit functional dependencies between input and output parameters. However, both input and output parameters sets should be minimized by removing redundant data parameters, to avoid unnecessary data coupling. Now, assuming the functional dependencies below we can produce a set of to normalized interfaces:

FD1: {OriginLocation, DestinationLocation, DepartureDate → FlightNumber}
FD2: {FlightNumber → DepartureAirport, DepartureTime, ArrivalAirport, ArrivalTime}
FD3: {FlightNumber, DepartureDate → ArrivalDate}
FD4: {FlightNumber, DepartureDate, CabinType → Quantity}

FD5: {FlightNumber, DepartureDate, CabinType → BasicFareCode, BasicFare}

Query Operations:

FlightEnquiry(INPUT: OriginLocation, DestinationLocation, DepartureDate,
OUTPUT: FlightNumber)
ScheduleEnquiry(INPUT: FlightNumber,
OUTPUT: DepartureAirport DepartureTime, ArrivalAirport, ArrivalTime)
ArrivalEnquiry(INPUT: FlightNumber, DepartureDate,
OUTPUT: ArrivalDate)
SeatEnquiry(INPUT: FlightNumber, DepartureDate, CabinType,
OUTPUT: Quantity)
PriceEnquiry(INPUT: FlightNumber, DepartureDate, CabinType,
OUTPUT: FareBasisCode, BaseFare)

Update Operations:

BookFlight(INPUT: FlightNumber, DepartureDate, CabinType, TravelerName,
OUTPUT: BookingReferenceID)
SeatingRequest(INPUT: BookingReferenceID, SeatPreference,
OUTPUT: BookingReferenceID)
MealRequest(INPUT: BookingReferenceID, MealPreference,
OUTPUT: MealType)

We can verify that the interfaces are fully normalized by noting that all input parameters for the enquiry operations are determinants (i.e. right-hand side of functional dependencies) satisfying the condition for BCNF Normal Form [23]. In addition to minimizing coupling, the effect of interface normalization is to maximize cohesion as resulting interfaces implement atomic operations.

5. Finalizing Design

The above analysis leads to normalized service interfaces and results in fine-granularity operations. While this may be theoretically appealing, the associated increase in the number of runtime calls and complexity of the interaction dialogue makes this approach difficult to implement in practice given the existing low-reliability and slow response time Internet infrastructure. Finding an optimal level of granularity for Web Services and individual service operations requires further examination.

5.1 Adjusting Granularity of Interfaces

We can use the normalization framework introduced in section 4 to understand the impact of aggregating interfaces. For example, the query operations SeatEnquiry and

PriceEnquiry share common input parameters FlightNumber, DepartureDate, CabineType. Combining the two interfaces produces a composite operation SeatPriceEnquiry:

SeatPriceEnquiry(INPUT: FlightNumber, DepartureDate, CabinType,
OUTPUT: Quantity, FareBasisCode, BaseFare)

This clearly leads to loss of cohesion as the resulting operation no longer implements a single atomic task, and in situations where it is used to perform a partial enquiry (e.g. seat availability enquiry only) the operation returns values that are not used by the application. Applying the normalization framework, this lack of cohesion is reflected by a partial functional dependency between the input and output parameter sets of the SeatPriceEnquiry interface (i.e. the loss of *full* functional dependence). This tradeoff can be justified in this instance on the basis that both operations are frequently performed together, and that the benefits of reduced number of operations and runtime procedure calls outweighs the loss of cohesion. Similar considerations apply to update request operations. For example SeatingRequest and MealRequest can be combined into a composite operation SeatingMealRequest:

SeatingMealRequest(INPUT:BookingReferenceID, SeatPreference, MealPreference,
OUTPUT: SeatNumber, MealType)

This time, a partial request, e.g. seating request only, produces non-homogeneity with MealPreference and MealType left undefined.

5.2 Implementation Style and Interaction Model

Following decisions about the appropriate level of aggregation, the final design stage involves decisions about the implementation style (i.e. binding style, RPC or document) and interaction model (i.e. synchronous or asynchronous, stateful or stateless). Adopting the document-centric (message-oriented) approach the resulting interface definitions are transformed into document-style WSDL specifications. Alternatively, the resulting interfaces can be mapped directly into Web Services operations using the RPC binding style [3]. Detailed discussion of such implementation issues is outside the scope of this paper.

6. Conclusions

We presented a design methodology for Web Services that applies data engineering principles to the design of message structures of service-oriented applications. The design approach relies on the principles of orthogonality, maximizing method cohesion, and minimizing method coupling, and uses data normalization techniques to avoid externalization of redundant data parameters. While we have argued that excessive use of coarse-grained, document-centric message structures results in poor reuse

and undesirable interdependencies between services, we do not advocate fine granularity services as a universal solutions. Equally, this paper does not represent argument for any specific Web Services implementation style (i.e. RPC or document style), as such decisions need to be made in the context of specific application requirements and taking into account the implementation environment. The main benefits of the proposed design framework is that it facilitates making informed decisions about the level of granularity of service operations based on normalization of the underlying message structures. As shown in section 5, composite operations can be constructed from operations with fully normalized interface messages by combining operations based on the properties of interface parameters. The impact of the resulting message aggregation on cohesion and coupling of service-oriented applications can be evaluated using the normalization framework, so that the designer can determine the most appropriate message design for a particular set of requirements.

8. REFERENCES

[1]Huhns, Michael N. and Munindar P. Singh, "Service-Oriented Computing: Key Concepts and Principles," IEEE Internet Computing, vol. 9, no. 1, 2005, pp. 75-81.

[2] Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures, PhD Dissertation, 2000, Available on: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[3] Feuerlicht, G. Implementing Service Interfaces for e-Business Applications. In Proceedings of the Second Workshop on e-Business (WeB 2003). Seattle, USA, December 2003. ISSN: 1617-9846.

[4]Papazoglou, M.P. and Yang, J. (2002), Design methodology for Web services and business processes. In Proceedings of the 3rd VLDB-TES workshop (Hong Kong, August, 2002). Springer, pages 54-64.

[5] Feuerlicht, G, Designing Service-Oriented e-Business Applications using Data Engineering Techniques, The Third Workshop on e-Business, in conjunction with ICIS 2004, December 11, 2004, Washington D.C., USA, ISBN:957-01-9161-9

[6] Venners, B. (1998) Introduction to Design Techniques. Available on: http://www.javaworld.com/javaworld/jw-02-1998/jw-02-techniques.html, February, 1998.

[7]Venners, B. (2002) API Design: The Object. Available on: http://www.artima.com/apidesign/object.html, April 26, 2002.

[8]Ambler, S.W. (2002) Deriving Web Services from UML models, Part 1: Establishing the process. Available on: http://www-106.ibm.com/developerworks/webservices/library/ws-uml1/

[9]Levi, K. and A. Arsanjani (2002) A goal-driven approach to enterprise component identification and specification. Communications of the ACM. Vol. 45:(10). (2002) 45 - 52

[10]Luo, M. et al. 2005, Service-Oriented Business Transformation in the Retail Industry Part 1: Apply SOA to Integrate Package Solutions and Legacy Systems [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-retail1/ [Accessed 15 April 2005].

[11] Meyer, B. Object-oriented Software Construction. 2nd ed. Prentice Hall, Upper Saddle River, N.J., 1997.

[12] Smith, R. Modeling in the Service Oriented Architecture, 2003. http://archive.devx.com/javasr/articles/smith1/smith1-1.asp.

[13] Masud, S. RosettaNet-based Web Services, Part 2: BPEL4WS and RosettaNet, 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-rose2/.
[14] Leymann, F. Web Services Flow Language (WSFL 1.0), 2001.http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.p
[15] Radeka, K. Designing a Web Services Project for Maximum Value: the 90 Day Challenge. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications archive (OOPSLA 2002) Practitioners Reports, Seattle, Washington, November 2002. ACM Press New York, NY, USA. ISBN:1-58113-471-1.
[16] Stevens, M. Multi-Grained Services. http://www.developer.com/design/article.php/1142661, May 21, 2002.
[17] Yourdon, E. and Constantine, L.. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Englewood Cliffs, N.J., 1979.
[18] Stevens, W.P., Myers, G.J., and Constantine, L.L., Structured Design, IBM SYSTEMS JOURNAL, VOL38, NOS2&3, 1999
[19] Myers, G.J.: Composite Structured Design. 1978, Van Nostrand Reinhold, ISBN 0-442-80584-5, 175 pages
[20] Feuerlicht, G., Design of Service Interfaces for e-Business Applications using Data Normalization Techniques, Journal of Information Systems and e-Business Management, Springer-Verlag GmbH, 26 July 2005, pages 1-14, ISS:1617-98
[21] Feuerlicht, G. and S. Meesathit. Design Framework for Interoperable Service Interfaces. In Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 299-307, New York, NY, USA, November 2004. ACM Press. ISBN:1-58113-871-7.
[22] Codd, E.F (1971). Normalized Data Structure: A Brief Tutorial. In Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control (San Diego, California, November 11-12, 1971). ACM, 1971, 1-17.
[23] Date, C. J. Fagin, R. (1992) Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases, ACM Transactions on Database Systems (TODS) Volume 17, Issue 3 (September 1992) Pages: 465 - 476, 1992, SSN:0362-5915