Software Architectural Reuse Issues in Service-Oriented Architectures

Julie Street
George Mason University
jstreet1@gmu.edu

Hassan Gomaa
George Mason University
hgomaa@gmu.edu

Abstract

Many companies today are adopting service-oriented architecture (SOA). While SOA has the potential to offer significant benefits to companies, the architectural issues associated with this new architecture paradigm need to be explored. The goal of this paper is to explore software architectures and patterns for SOAs using Unified Modeling Language (UML). Specifically, this paper addresses issues on composing applications from reusable services, differentiating between several services that are similar but not identical, and establishing a common data model.

1. Introduction

Software architectures are the foundation for designing, communicating, and constructing complex software systems. The intent of software architectures is to illustrate, from many views, a software system's decomposition into the individual components, the communication paths, and the processing resources. Software architectures can be grouped according their style, which are also known as software architectural patterns. For example, the client/server architecture is a distributed architecture in which a server provides functionality to requesting clients [1].

A software architectural style, also known as a software architectural pattern, is a recurring software architecture reused in a variety of applications, thereby allowing large grained software reuse at the architectural level. For example, the client/server architecture is a solution to solving the problems associated with monolithic centralized software architectures. In a monolithic architecture a single computer is responsible for all computation, data storage, and user interface functionality. This does not lend itself to a large number of users because I/O overhead and context switching is a substantial burden on the necessary computational power. The client/server architecture, however, is able to overcome this problem by separating the user interface functionality (the client) from the computationally intensive functionality (the server). This enabled a few large servers to interface with multiple less computationally intensive clients. While the client/server architecture does solve the scalability problems associated with monolithic architectures, it has other problems, which have led to more software architectures styles and design techniques to be developed, as described here.

Today many companies are adopting service-oriented architecture (SOA). In fact, the Gartner Group estimates that IT professional services involving SOA will grow from $46 billion in 2003 to $261 billion by 2008 [2]. SOAs offer the potential to increase reuse, create new applications from old and new services, loosen technology coupling, and allow division of responsibility between business and technology staff [3]. However, there is nothing intrinsic about SOAs that guarantee these benefits. Because the IT market is moving so rapidly towards SOA, it is important that researchers examine the architectural issues associated with SOAs. Therefore this paper addresses the architectural issues using SOAs with platform-independent examples in UML. The paper uses the standard UML extension mechanism for stereotypes by providing SOA specific stereotypes to complement the stereotypes used by the COMET design method[1]. Specifically, this paper addresses issues on composing applications from reusable services, differentiating between several services that are similar but not identical, and establishing a common data model.

The structure of the paper is as follows: Section 2 discusses background information on service-oriented architectures (SOA). Section 3 describes related work. Section 4 describes architectural practices for composing applications from reusable services. Section 5 describes approaches for differentiating between several similar services. Section 6 addresses strategies for establishing a common data model. Section 7 describes how the concepts discussed can be mapped to Web Services. Finally, Section 8 summarizes the main conclusions from this paper.

2. Related Work

Service-oriented architecture (SOA) is a software architecture style that has recently gained in popularity because of its potential to maximize reuse, interoperability, and flexibility. SOA achieves the aforementioned benefits by dividing the software architecture into three main players: (1) service providers (2) service consumers, and (3) service brokers. To achieve a SOA, the players must interact in a specific fashion, as shown in Figure 1 [3].

1530-1605/08 $25.00 © 2008 IEEE

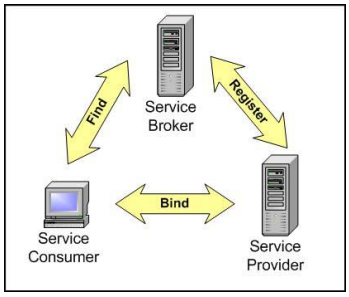

Figure 1. Basic SOA interactions [3]

A lot of research describes implementations or potential applications for SOA. For example, Gomaa and Saleh describe an approach for designing software product lines (SPL) with Web Services (an implementation of SOA) using UML. They show how Web Services can be exploited to achieve reusability in SPL development [4]. Thanh et al present an architectural framework for applying SOA to mobile services and communication [5], [6]. Wong-Bushby et al present a case study on re-architecting an application in SOA as well as lessons learned [7]. This paper is different from the above research because it addresses architectural differences in SOAs that are not specific to any one application, domain, or implementation.

Limited research has been conducted at the architectural level for SOA. Stal provides insight on SOA's core architecting principles using software patterns, which are not specific to any application [8]. Enderi et al demonstrate how IBM patterns for e-business applications can be applied to SOA [9]. Zirpins et al successfully demonstrate using design pattern for creating flexible coordination of services in SOAs [10]. This paper is different from the research mentioned above because it addresses issues beyond architectural patterns.

Some research has been conducted on approaches to designing SOAs with UML. Amir and Zeid propose the use of a UML Profile for SOAs based on Web Services. They describe some extensions to make UML more amenable for designing architecture with Web Services [11]. Baresi et al propose an approach using UML models of the architectural style of the platform and the application scenario and graph transformation analysis for SOA. They successfully show how this approach can be leveraged using a supply chain management system [12]. This paper describes software architectures and patterns for SOA using the UML notation. The paper uses the standard UML extension mechanism for stereotypes by providing SOA specific stereotypes such as <<service>> and <<coordinator>> objects and classes to complement the stereotypes used by the COMET design method, such as <<user interface>> and <<entity>>.

3. Composing a System From Reusable Services

Software reuse is a desired goal; however in practice it is difficult to achieve. One aspect of achieving reuse is using good architectural practices. The following subsections address various practices that should be employed to develop reusable services and systems.

3.1. Service with High-level Interfaces

Services with high-level interfaces are more reusable because providing an interface with a high level of granularity masks specialized or implementation specific methods, thereby enabling one service to be used by multiple applications. Any changes, additions, and removal of the implementation or specialized functions occur inside the service and are hidden from the user. This enables services with high-level interfaces to be adapted and reused more easily because the client application will not be affected by changes inside the service. The high-level functions can be designed using bridge, brokers, and multi-agents patterns that serve as intermediaries to route messages to the appropriate specific controllers [1], [8].

To illustrate this point consider a platform independent electronic travel reservation SOA system, which reserves and books flights and hotels for individuals via the Internet. A service can be used to create reservations, see Figure 2. The Reservation service provides three methods to reserve different type of rooms: reserveQueenRoom(in checkinDate, in checkoutDate, out message); reserveKingRoom(in checkinDate, in checkoutDate, out message); and reserveTwoBedRoom(in checkinDate, in checkoutDate, out message). The Reservation service uses an Availability service to check for room availability and to decrement the room count if rooms are available. If a room is available the Reservation service then creates a reservation record and notifies the client

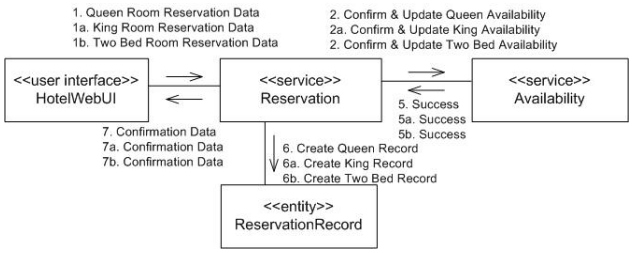

Figure 2. Low level functionality reserve room communication diagram

Suppose after the system is deployed, the hotel manager decides to offer new luxury suites. In this

architecture, the Reservation service could not be easily reused because the methods are very granular and designed for specific room types. To support the new room type, the Reservation service needs a new reserverSuite() method. This change will most likely impact the existing client functionality.

Now consider the same hotel reservation SOA system with high-level functionality, as shown in Figure 3. The Reservation service provides a generic high-level reservation interface with a single method: reserveRoom(in roomType, in checkinDate, in checkoutDate, out message), where the roomType parameter is used to designate the type of room. In this architecture, when the hotel manager adds a new luxury suite, Reservation service's method can be reused because it has high-level functionality. All the required changes occur inside the Reservation and Availability service and are hidden from the client application. Therefore existing client functionality will be not be impacted by the change.

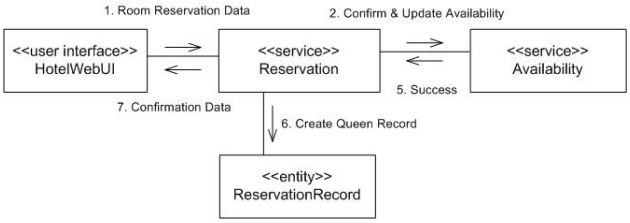

Figure 3. High-level functionality reserve room communication diagram

3.2. Process Separation

Another architectural strategy for creating applications from reusable services is to separate out the process logic from the business logic [3]. Process logic refers to the order in which steps are processed; such as a check for room availability must occur before a reservation can be made. Business logic refers to the business functionality, such as create a reservation record or check room availability. Separating out the process logic from the business logic means that business services should not have embedded process logic that ties it to a specific process. This enables business services to be reused in multiple processes because they are not bound to one process flow. The process flow can be implemented as controller services, which orchestrate the use of different business services. This enables flexible process flows that can be easily modified without affecting multiple business services. Subsequently, this also reduces coupling between business services. This is desirable for reuse because low coupling implies that one change will impact a smaller number of services [13]. Process separation can be implemented using Business Process Execution Language (BPEL) or in some cases session façade and multi-agent architectural patterns [1], [8]. Complex processes can even be composite controller services that leverage other controllers [14].

To illustrate process separation consider the Reserve Room Communication Diagram in Figure 3. In this design, the Reservation service first checks for availability using the Availability service. Then the Reservation service creates a reservation record. Now consider another hotel chain with a different reservation process is reusing this SOA. The new hotel creates reservations by checking for availability, verifying a credit card to hold the room, and finally creates a reservation record. The new hotel could not reuse the Reservation service because it will not allow credit card verification in between the availability check and record creation. If the original SOA were design with a separation between business and process logic, shown in Figure 4, then the new hotel could more easily reuse the SOA. This design is more reusable because the basic business logic is captured in the Availability and Reservations services. The process logic is controlled by the HotelXReservationCdr using BPEL.

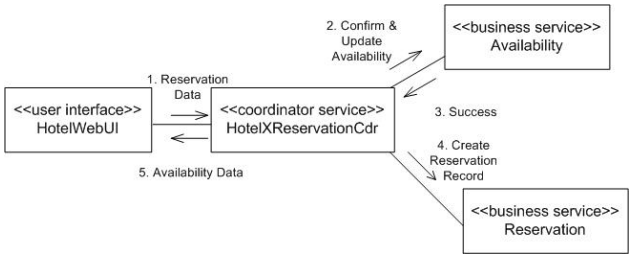

Figure 4. Business and process logic separation

In this architecture, the new hotel can easily create a new controller service with different process logic that first checks for availability, then verifies the credit card to hold room, and finally creates a reservation record. The new controller can reuse the Availability, Reservation, and Credit Card (assuming one already exists) business services because they do not have any of original hotel's processing logic embedded in them.

3.3. Multiple Message Exchange Patterns

As discussed in section 4.2, process separation creates a separation between the business and process logic, which enables business services to be reused in number of different controller services or applications. To ensure that services are well suited to support multiple processes, they should be designed to support multiple message exchange patterns. This will enable business services to be reused by a wider range of processing situations [3]. Common exchange patterns include asynchronous, asynchronous with call back, synchronous with reply, and synchronous with reply [1]. The message exchange patterns should be clearly described in the service's interface description.

To illustrate this example, consider the communication diagram (Figure 4) for processing a reservation. HotelXReservationCtlr does not need to wait for a reply from the Reservation service before sending HotelWebUI confirmation of the reservation. Suppose a new hotel trying to reuse this SOA service needs a response from the Reservation service to ensure the reservation was created. The Reservation service could not be reused because it only supports asynchronous communication. However, if the Reservation service were designed upfront to support multiple invocation styles, then it could be reused more easily.

3.4. Stateless Service

Stateless services facilitate reusability because they improve scalability, reliability, and load balancing in a distributed architecture [14]. However stateful services are still permitted in SOAs [15], [3]. A stateless service is where temporary data about previous requests is not maintained on the server. Instead the temporary data is stored on the client's resources and it is passed back to the server in every message. This means that the same server does not have to process successive requests from the same client to ensure correct operation. This improves scalability and load balancing because clients can be independent routed at each message. Reliability is improved because should a particular server go down, the state is not lost and the client would be unaffected.

To illustrate this point, consider the process where a user is trying to retrieve all their reservations. Figure 5 shows the internal implementation of a stateful Reservation service performing this business function. In a stateful interaction, the user's temporary data from previous interactions, such as the user id, is stored in a server object. The controller must retrieve this information from the server object to determine the reservations it needs to retrieve. This requires that the client be routed to the same server each time, to ensure their information can be accessed.

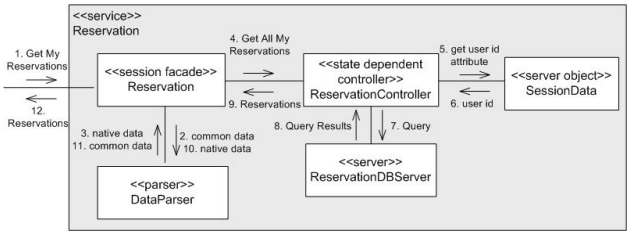

Figure 5 SOA stateful service

Figure 6 shows the same service implemented using a stateless interaction. In this interaction, the user's data is passed in the incoming message. Therefore, a server object is not required and the user can be routed to any server.

In summary, stateless services promote reusability and therefore should be used whenever possible. If there are scenarios where a stateless service cannot be used, it is best to architect a service that maintains state in a shared database, rather than a server object. This is more reusable because other services that access this database can also retrieve the state and service the customer. For example, in a banking service where the PIN is first validated and then a cash withdrawal is made, the valid PIN state could be stored in a customer transaction record prior to interacting with the customer and checking that the customer has enough funds to allow a cash withdrawal.

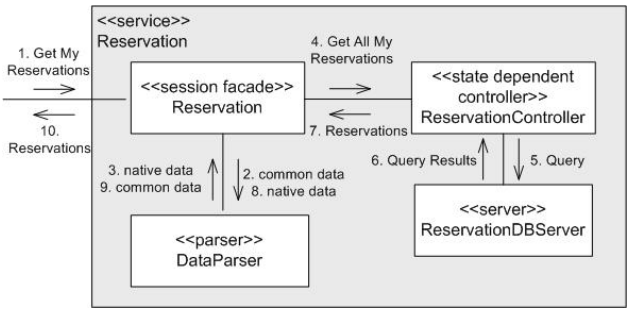

Figure 6. SOA stateless service

3.5. Anticipating Changes in Functionality and Design for Change

Another architectural strategy for creating reusable services is to consider possible future changes and different uses of the service. Designing for variability is useful because future functionality or different application uses can be considered within a single design. This helps ensure that the services implemented are well suited to support future functionality. Different techniques for modeling variability can be applied, such as software product line approaches and variation point models [16], [4], [17], [18]. These techniques can be applied to SOAs where the core functionality represents the current functionality to implement. The variable functionality represents SOA future functionality to consider. By considering future functionality upfront, the design can be adapted more easily to support the addition of new functionality.

To illustrate this point consider the travel reservation SOA. When the SOA development begins, the SOA may only support one hotel. Therefore the Reservation service is only required to create reservations for this one hotel and the Reservation service does need the hotel identification as input. However, suppose the hotel chain intends to expand its business to have multiple hotels in multiple locations. If the SOA was designed to anticipate future changes, the Reservation service could be initially

designed with a hotel identification number, so that future hotels could reuse it.

4. Differentiating Between Similar Services

As discussed in section 4, facilitating reuse begins with using good architectural practices. However, these practices do not provide a complete answer because they do not address domain issues. For example, what if there are two services that are similar, but not identical. What criteria should be used to decide if they should be deployed as one or two services? Also, if two similar services are deployed how can a client application determine which one to use? The following subsections address different approaches and strategies that can be taken to help answer these questions.

4.1 Parameterization

During service design, if there are two similar services, one approach that can be utilized to decide if they should be deployed as one or two services is parameterization. Parameterizations techniques are commonly used in software product lines for designing reusable components that can be configured for a specific product line member [4], [16]. Parameterization techniques can be leveraged in SOAs for determining whether two similar services should be combined or remain separate. If the functionality can be differentiated with parameters, then one service is sufficient. Otherwise two services should be used.

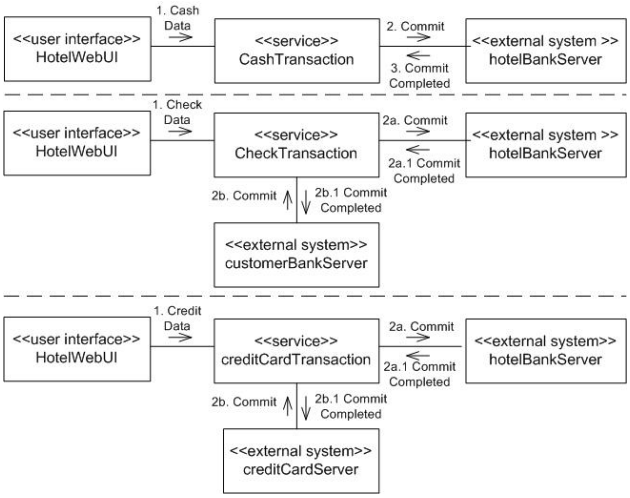

Figure 7. Multiple transaction service communication diagrams

To illustrate parameterization, consider making a transaction payment for a room. Transactions can be made via cash, check, or credit card. All three types of transactions perform electronic funds payment functions, where the transaction information is sent to the appropriate billing systems, as depicted in Figure 7. For check and credit card transactions, electronic funds are transferred from a customer bank server or credit card server to the hotel bank server using a two-phase commit protocol [1].

Each transaction type requires slightly different inputs. A cash transition requires the amount, the check requires amount, account number, bank, and check number, finally the credit transaction requires the credit card type, credit card number, expiration date, name on the credit card, and amount. All these inputs can be captured as parameters of a single electronic funds transaction service that provides the functionality of the three original services, as depicted in Figure 8. Depending on the transaction type, the electronic funds transaction service communicates with the appropriate external system servers, using a two-phase commit protocol for check and credit card transactions.

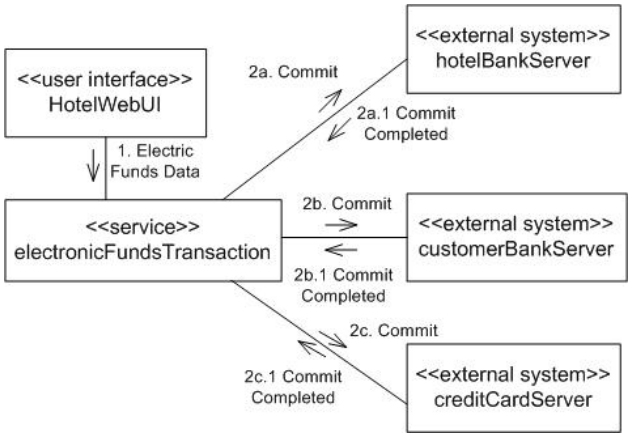

Figure 8. Parameterized single transaction service communication diagram

4.2 Enhanced Service Retrieval Searching

Differentiating between deployed similar services is a cognitive activity and requires human involvement. In SOAs, deployed services are registered with a service broker using a standard protocol, such as Universal, Description, Discovery, and Integration (UDDI), to enable searching and discovery of existing services for reuse. However, many retrieval mechanisms are keyword-based searches, therefore it is likely that many similar services will be returned [3]. This leaves the burden of choosing the correct service to the developer, which can be very time consuming. If finding a service takes too long, then a developer is more likely just to create a new redundant service, which reduces resue. One approach to help minimize the time spent on determining the right service is to enhance repository search capabilities to return most relevant results.

Yao and Etzkorn propose a methodology to enhance querying a UDDI repository using a combination of natural language processing, program understanding, and semantic representation techniques. Their approach will provide the user with a natural language interface to enable unrestricted and natural queries from the user. The search engine would then translate the queries into conceptual graphs, apply program understanding to translate services into conceptual graphs, and finally a semantic matchmaker would be able to determine the best matches. This approach has the potential to eliminate more unsuitable choice than keyword searches [19]. The reduced time spent searching for existing services will likely increase the amount of reuse.

Another approach that could be modified for SOAs, is a proactive information repository. Ye et al propose the idea of an active information delivery, in which a reuse repository sends component (or service) information to the user without an explicit user query requesting this information. An active information delivery repository stores user profiles that contain previous queries from the user and components they have reused. This information is then used to determine if new or existing components are potentially relevant to the user. If they are, the repository will automatically send it to the user. This has the potential eliminate the time developers spend querying repositories, and ensures the user is aware of the latest components they may find useful [20].

5. Impacts of a Common Data Model

In the past, it was common practice for organizations to develop their data models independent of each other. This resulted in data models for similar applications to be different. This creates data integration problems when applications are integrated. The use of a common data model between systems can facilitate interoperability and reusability in SOAs. While using a common data model does increase the reusability of a SOA, it does have some architectural implications. The subsections below address the architecture issues resulting from a common data model.

5.1. Data Translation

A common data model requires that applications use a common message format and send data in the common schema. This approach forces applications to translate the common data into its internal native data before it can call its internal methods. Although, this adds overhead time and resources to an application, a common data model does help create interoperability between systems and makes it easier to reuse services because universal data formats and schema are used, which eliminates the need for point-to-point interface formats and schemas.

Figure 9 illustrates this difference between the native and common data exchange process with a state chart of the Reservation service to process a reservation request, shown in Figure 3. The top of the Figure 9 shows the processing of a message exchanges using a common format and schema. Once a message is received the service leaves the Listening state and enters a Waiting for Decoded Message state. The decoding is required to translate the common data into the native platform data.

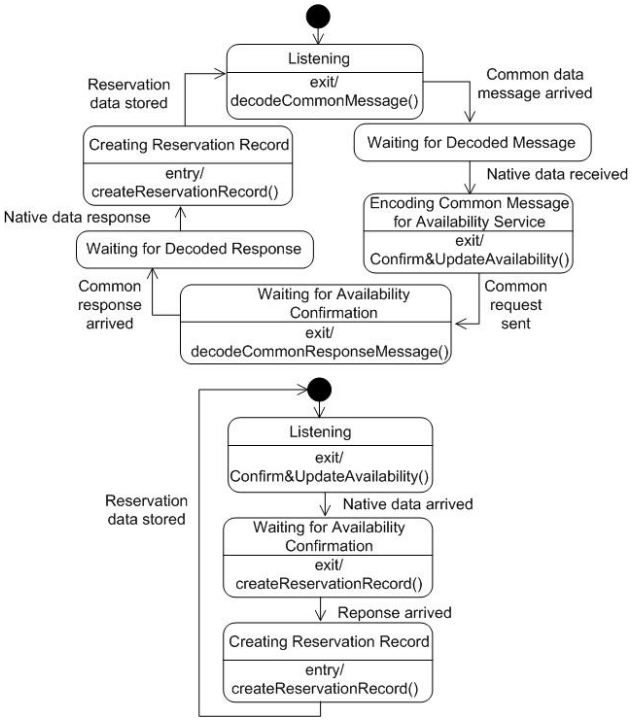

Figure 9. Service state machines with and without data translation

Once the message is translated, the Reservation service must confirm the availability of the room from the Availability service. This requires encoding a message into the common format, which is done in the next state. Once the encoding process is complete, the message is sent and the service goes into a Waiting for Availability Confirmation state. When the response is received, the service then must decode the response, which is done in the Waiting for Decoded Response state. Finally, if the availability is confirmed the service goes into a Creating Reservation Record state. Upon completing that process, the service sends the response back to the client and returns to the Listening state.

The bottom diagram of Figure 9 shows the traditional way in which messages are exchanged in the operating environment format. In this situation all message calls and data exchanges are in the application's native format.

Therefore the encoding and decoding states are not required, resulting in a more efficient process.

In summary, while a common data model does help create interoperability between systems and makes it easier to reuse services, it does come at a performance cost.

5.2. Complex Data Exchanges

Data exchanges with services require that data be translated into the interoperable common data model. However, complex data types may not easily translate into a platform neutral common data model, which inhibits SOA data exchanges[14]. For example, an image file is a common form of data, but it cannot be easily translated into an XML schema for SOAP messaging. There are several different ways to architect a system.

One approach that can be utilized when complex data types are involved is to send a location string. The location string, such as a Uniform Resource Locator (URL), can be sent as part of a XML schema, and the receiver can use the location to access the file directly [22]. If the data needs to be updated, this requires that the original data receiver provide an update method, where it receives a URL, accesses the updated file, and finally updates the original file. This type of architecture can limit reusability because any service involved in this interaction is required use non-standard approach for communicating that data. Non-standard approaches are less likely to be supported; therefore this can impact the reusability of this architecture.

Another approach to solving this problem is to use a SOA messaging protocol that supports non-XML data transfers. For example, Simple Object Access Protocol (SOAP) with Attachment (SwA) enables data to be transferred in its current format, such as a Joint Photographic Experts Group (jpg), with encoding mechanisms [14]. This approach has several advantages from the reusability perspective. First, it leverages this existing SOA infrastructure, which makes it easy to implement. Additionally, a standard SOA messaging protocol is more universal; therefore it is likely other components will support this protocol and will be able to reuse the component. Finally, an experiment comparing SwA with SOAP message using matrices showed that SwA was faster than SOAP for complex data types [21]. While better performance does not directly increase component's reusability, a faster component with the same functionality will probably be selected for reuse over a slower component.

In summary, many systems complex data types may not easily translate into a platform neutral common data model. In these this situations, alternative means of transmitting the data are required. Leveraging existing SOA standard protocols will most likely lead to better reusability.

6. Mapping SOA Concepts to Web Services

So far the previous sections in this paper discuss architectural reuse concepts from the platform independent level. This section goes a step further and begins to explain how these concepts can be mapped to the platform specific level using Web Services. This mapping step is essential to enable these concepts to be realized in SOAs.

To illustrate how mapping can be accomplished, consider Figure 8, a two phase commit protocol scenario for check and credit card transactions. In this scenario the electronicFundsTransaction is a Web Service provide a service to the HotelWebUI. The first element a Web Service needs is a Web Service Definition Language (WSDL) service definition, which contains the information necessary to understand and invoke the service. In the WSDL definition, an abstract description captures the service interface [14]. In this example, the interface captures the operation execute ElectronicFundsTransaction and the data required in the interface as the Electronic Funds Data input message. The concrete description of the WSDL definition contains the information necessary to create the physical connection to the Web Service [14]. After the WSDL definition is created it is can be stored in the appropriate UDDI repository.

The message received by the electronicFundsTransaction Web Service use SOAP [14]. In this example the message from the HotelWebUI will be in SOAP format. The different commit messages from the electronicFundsTransaction Web Service to the different external systems would be in external systems' native format. However, if the external systems were also Web Services, the electronicFundsTransaction Web Service would send SOAP messages as specified in their respective WSDL definitions. Additionally, the orchestration for the electronicFundsTransaction Web Service could be written using WS-BPEL.

In summary, this section highlights how the reused architectural concepts can be implemented as Web Services

7. Conclusions

The SOA paradigm has the potential to offer significant benefits to software systems development, maintenance, and reuse. However, many of SOA's

benefits are not guarantee just by implementing a SOA. Therefore it is necessary for researchers and developers to address the important software architecture and reuse issues prior to creating a SOA. This paper has described reusable software architectures and patterns for SOA using the UML notation and the standard UML extension mechanism for stereotypes to provide SOA specific stereotypes. Table 1 provides a summary of the SOA architectural reuse issues and the section numbers in which they are discussed in this paper. In summary, good architectural design practices and patterns are necessary for composing systems from reusable services.

Table 1. Summary of Reuse Issues

Category	Issue	Summary	No
Architecture	Interface design	High level interfaces are more reusable because they can cover multiple situations.	3.1
	Separation of logic	Separating business logic from process logic increases the reusability of business services	3.2
	Message exchange patterns	Services that support multiple exchange patterns can more easily be reused because they already support a variety of invocations.	3.3
	Storing service states	Stateless services increase stability and subsequently reusability	3.4
	Anticipating change	Designing for change is useful because future functionality or multiple applications can be considered within a single design, allowing developing of more reusable services.	3.5
Service differentiation	Parameterization	Analyze the use of parameterization to help determine if functions should be a single or multiple services	4.1
	Service retrieval	Improved repository retrieval services will help developers locate the right service more quickly making reuse easier	4.2
Data exchanges	Data translation	Common data model result in increased processing, which needs to be factored into software designs and performance requirements	5.1
	Complex data	Complex data that cannot be easily translated into a common data model needs an alternate transfer means	5.2

8. References

[1] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Third ed. Boston: Addison-Wesley Object Technology Series, 2000.

[2] M. Cantara, "Market Focus: Trends and Forecast for IT Professional Services fro Web Services and SOA, 2005 (Executive Summary)," The Gartner Group G00129458, June 28 2005.

[3] E. Newcomer and G. Lomow, *Understanding SOA with Web Services*. Upper Saddle River, NJ: Addison-Wesley, 2005.

[4] H. Gomaa and M. Saleh, "Software Product Line Engineering for Web Services and UML," in *3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, 2005.

[5] D. v. Thanh and I. Jørstad, "A Service-Oriented Architecture Framework for Mobile Services," in *Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/ELearning on Telecommunications Workshop*, Lisbon, Portugal, 2005.

[6] D. v. Thanh, S. Dustdar, and I. Jørstad, "Service Oriented Architecture Framework for Collaborative Services," in *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)*, Linkoping University, Sweden, 2005.

[7] I. Wong-Bushby, R. Egan, and C. Isaacson, "A Case Study in SOA and Re-Architecture at Company ABC," in *39th Hawaii International Conference on System Sciences*, Hawaii, 2006.

[8] M. Stal, "Using Architectural Patterns and Blueprints for Service-Oriented Architecture," in *IEEE SOFTWARE*, 2006, pp. 54-61.

[9] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: Service-Oriented Architecture and Web Services*: International Business Machines (IBM) Corporation, 2004.

[10] C. Zirpins, W. Lamersdorf, and T. Baier, "Flexible Coordination of Service Interaction Patterns," in *International Conference of Service Oriented Computing (ICSOC)*, New York, New York, USA., 2004.

[11] R. Amir and A. SZeid, "A UML Profile For Service Oritented Architecture," in *International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, Canada, 2004, pp. 192-193.

[12] L. Baresi, R. Heckel, S. Thone, and D. Varro, "Modeling and Validation of Service-Oriented Architectures: Application vs. Style," in *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Helsinki, Finland., 2003.

[13] M. E. Dunn and J. C. Knight, "Automating the Detection of Reusable Parts in Existing Software," in *15th international conference on Software Engineering*, Baltimore, Maryland, 1993, pp. 381 - 390.

[14] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Boston, MA: Prentice Hall PTR, 2005.

[15] D. J. N. Artus, "SOA realization: Service design principles." vol. 2006: IBM Developer Works, 2006.

[16] The Software Engineering Institute (SEI), "Feature-Oriented Domain Analysis." vol. 2006: Carnegie Mellon

University, 2004, pp. http://www.sei.cmu.edu/domain-engineering/FODA.html.
[17] D. L. Webber and H. Gomaa, "Modeling Variability with Variation Point Model," in *7th International Conference on Software Reuse*, Austin, Texas, 2002, pp. 109-122.
[18] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, First ed.: Addison-Wesley Object Technology Series,, 2004.
[19] H. Yao and L. SEtzkorn, "Towards A Semantic-based Approach for Software Reusable Component Classification and Retrieval," in *ACM 42nd Southeast Conference (ACMSE '04)*, Huntsville, Alabama, USA., 2004.
[20] Y. Ye, G. Fischer, and B. Reeves, "Integrating Active Information Delivery and Reuse Repository Systems," in *ACM SIGSoft 2000*, San Deigo, CA, USA, 2000.
[21] Y. Ying, Y. Huang, and D. W. Walker, "Using SOAP with Attachments for e-Science," in *UK e-Science All Hands Meeting*, Nottingham, UK, 2004.
[22] J. D. Mier, S. Vasireddy, A. Babber, and A. Mackman, "Improving .NET Application Performance and Scalability." vol. 2006: Microsoft Corporation, 2004.