A Comparative Study of Interface Design Approaches for Service-Oriented Software

Hui Ming Teo and Wan M.N. Wan Kadir
Software Engineering Department
Faculty of Computer Science and Information Systems,
Universiti Teknologi Malaysia,
81300 UTM Skudai, Johor, Malaysia.
hmt@teohuiming.name, wnasir@fsksm.utm.my

Abstract

In the service-oriented software environment, interactions between components are highly dependent on the exposed service interfaces. Therefore, designing an appropriate service interface is essential. In this paper, we aim to perform a comparative evaluation on three different approaches to service interface design, i.e. method-centric, message-centric and resource-centric. The evaluation is peformed systematically based on a list of selected evaluation criteria. It is expected that the evaluation results may assist software architects to understand the differences between approaches and adopt the approaches wisely in the service interface design.

1. Introduction

Back to year 2000 in a software maintenance and evolution roadmap [1], Bennet and Rajlich envisioned a "software as a service" (SaaS) model that aims to improve software evolution by allowing users to assemble a set of services for use on demand [2]. The model allows dynamic service substitution whereby, unsatisfied service components can be disengaged easily and replaced immediately with new ones found in an open marketplace. Besides SaaS, a similar model, termed Service-oriented Architecture (SOA) is progressively being pushed as an alternative to traditional Electronic Data Interchange (EDI) for business-to-business interactions [3] and promoted as an integration solution for enterprise applications [4].

Despite the differences of usage, service-oriented models share a common challenging environment that is, to handle the interactions among a large number of heterogeneous components. Due to the complexity of such environment, it is essential for service-oriented software to raise the abstraction by shielding more implementation details behind well-defined interfaces. Interactions are allowed only through the service interfaces. Thus, an appropriate interface design is important.

This paper starts with describing a common service-oriented software environment in section 2, followed by a brief description on the three service interface design approaches in section 3. The evaluation criteria and the comparative evaluation are discussed in section 4, whilst related work is presented in section 5. The conclusion and further work are summarized in section 6.

2. Service-oriented software environment

Service-oriented software paradigm has its own unique environment, which will directly influence the design of software. First, a precondition to allow dynamic service substitutability is to have an open marketplace with a large number of services offered by competing vendors. *Software interoperation and integration issues* cannot be avoided when users need to assemble different services developed independently into one software within a short period (termed as ultra-late binding [5]). Second, different services are often distributed across wide geographical locations, which leads to *issues in distributed computing* [6]. Finally, in contrast with the distributed environment within single trust boundary, where single authority is able to coordinate all participating components for software changes in a controlled manner, the service-oriented environment often needs to scale up to involve world-wide participating organizations from different trust boundaries, thus a centralized *maintenance and evolution* process becomes impractical.

The three issues described above represent a generalized view of the service-oriented environment.

To design a service that will operate well in such environment, architects should understand potential problems raised by each issue and tackle them appropriately in the design. However, depends on the actual deployment environment, the significant impact of each issue may be vary. At one end of the spectrum, the actual environment may consist of a few services that are developed and maintained by single vendor and operate in a local area network within a small organization. At the other end, we expect an open marketplace with many services developed independently to compete and interact with each other in an Internet-scale distributed environment. Based on the actual deployment environment, architects may choose to ignore issues with little impact in order to achieve an optimized design.

3. Service interface design

Service providers offer services by exposing a set of well-defined service interfaces to consumers. Service interface is the only communication point between both parties. It is important to design service interfaces that match with the nature of service-oriented environment. In this paper, we discuss three service interface design approaches currently used in practice, i.e. *method-centric*, *message-centric*, and *resource-centric*. This section provides a brief description of each approach.

3.1. Method-centric approach

Similar to Remote Procedure Call (RPC) that allows a program to call procedures located on other machines, method-centric approach exposes service functionality via a set of distinct procedure calls. Consumers invoke an application-specific operation on a service endpoint with input arguments and optionally expecting for a return value. Current implementations can be found in several public web-based service applications. For example, Google exposes its services as three operations: `doGoogleSearch`, `doGetCachedPage` and `doSpellingSuggestion` [7]. Most earlier service-oriented research [8-12] are centered on this approach. In some literatures, it is termed control-centric [13], programmatic approach [14] or RPC-style.

3.2. Message-centric approach

In message-centric approach, instead of invoking function calls, consumers consume a service by exchanging application-specific messages with a service endpoint. Service providers define a set of schemas (e.g. XML document schema) for messages that will be used in the interactions. Several online business standards (e.g. RosettaNet [15] and OpenTravel Alliance (OTA) [16]) have adopted this approach. For example, a purchase order can be submitted to a RosettaNet-compliant purchase service by sending a `PurchaseOrderRequest` message and receiving a `PurchaseOrderConfirmation` message as response. In contrast with method-centric interface that exposes multiple operations, message-centric interface has only a `processThis` operation to receive various messages for processing. Since there is only one operation, it is therefore implicit [17].

3.3. Resource-centric approach

Resource-centric is also known as content-centric [13], Representational State Transfer (REST) style [18] or constrained interface [19] in different literatures. The most distinct difference from other approaches is that, it applies a *uniform interface* constraint [18] to restrict the interaction. Everything that consumers interact with is always a resource, thus a set of messages can be used uniformly to interact with all resources across different domains. The most notable implementation of this approach is the World Wide Web (WWW), a world-wide deployed distributed document-based system [20]. It uses a set of standard general-purpose messages in Hypertext Transfer Protocol (HTTP) [21] to allow consumers to interact with resources. For instance, a HTTP GET message can be used to retrieve a web page, as well as a bookmark list at del.ico.us bookmark service [22] or even an object stored in Amazon S3 storage service [23].

4. Comparative evaluation of service interface design approaches

In this section, we systematically analyze and evaluate each service interface design approach based on a set of evaluation criteria.

4.1 The evaluation criteria

It is important to realize that designing a service interface is different from designing a library-based API interface for local running applications. There are simply more issues to concern due to the nature of service-oriented environment. We have selected a list of design concern issues as the evaluation criteria, and then group them into three design concern areas:

Table 1. Summary of evaluation criteria

Criteria	Brief Description
Design for developers	
Abstraction model	the underlying abstraction model used by a service interface
Granularity	the degree of business-level interaction details described by a service interface
Design for maintenance	
Integration	the ease of interface standardization and software integration tasks within and across domain
Evolution	the cost of changing service interfaces over time
Design for distributed environment	
Latency	to handle the network latency in a distributed environment
Partial failure	to handle the partial failure nature of distributed environment
Sync/async interaction	to model business-level asynchronous interaction in service interface

- **Design for developers.** Developers who build a service implementation or software that consume the service need to understand the abstraction model underlying a particular service interface. Each interface design approach may adopt a different model. Each model provides a different way to think about a domain problem and a different way to model the service abstractions. Understanding each model is crucial for architects to construct appropriate service abstractions and to enforce certain design practices tailored to the adopted model. It is also important for service-oriented application framework developers to design frameworks that match to the nature of each abstraction model. In this area, we consider two criteria: *abstraction model* and *granularity*.
- **Design for maintenance.** As mentioned earlier, the integration issue cannot be avoided in service-oriented software environment, thus it should be taken care right from the design. For long-term maintenance, since the communication between service providers and consumers are heavily relying on service interfaces, changes on interfaces may break the binding between both parties. Coordinating interface changes in an Internet-scale distributed environment without centralized maintenance is non-trivial. Periodically changes that each time invalidates the previous interface contract may discourage consumers from binding to a service. Therefore, the interface evolvability should be taken care as well. We consider both *integration* and *evolution* as the criteria in this area.
- **Design for distributed environment.** Early distributed computing research has warned about the negative impact of hiding the nature of distributed environment from programmers [6]. Architects should realize the distributed computing is a part of the service-oriented environment and should handle related issues accordingly in the interface design. In this paper, we include three criteria: *latency*, *partial failure,* and *synchronous/asynchronous interaction*.

The criteria presented here is not an exhaustive list. However, they are enough to cover a wider range of concern areas and able to reveal the differences of three design approaches from various perspectives.

4.2 The comparative evaluation

4.2.1 Abstraction model. The method-centric approach adopts a similar abstraction model of structured programming paradigm. *A method-centric interface can be viewed as a list of procedure calls.* Each procedure call is identified by its signature consists of an operation name, a list of input arguments and optional return value. As a result, method-centric interfaces can be mapped naturally to existing interface definition languages (e.g. OMG IDL [24]) and programming languages. This may help to retain features directly from earlier programming paradigms, e.g. the static type checking at compilation time. Furthermore, the established design techniques (e.g. structured design [25] and data engineering [26]) may continue to be applicable in the service design. In the abstraction model, business interaction semantics are encoded into the procedure signatures. It means that by sending a `doGoogleSearch` call to Google service, a consumer is expecting to receive a web page search results, and not other results.

The message-centric approach focus on defining a set of message formats for message-exchange interactions. *A message-centric interface can be viewed as a collection of messages*. Based on the implementations such as RosettaNet and OTA, interestingly, we observe a common method used by architects to identify and design messages. To illustrate this, given a method-centric interface `doGoogleSearch`, the common way is to split the procedure call into two different messages: a `GoogleSearchRequest` message that contains all input arguments and a `GoogleSearchResults` message with the returned search results. A message can be as simple as an argument list or as complex as a hierarchical data structure. Business interaction semantics are encoded into each message. For instance, by sending a `GoogleSearchRequest` message, the sender can safely assume that a `GoogleSearchResults` message will be returned.

The resource-centric approach constrains the consumers to always, and only interact with resources. Thus, *a resource-centric interface can be viewed as a collection of resources*. Two object-oriented (OO) concepts, i.e. *polymorphism* [27] and *programming to interface* [28] may help to explain the model. Using the former, all resources should be derived from a single abstract class `Resource`, thus they can be accessed uniformly through operations defined in the abstract class. Based on the latter, all resources should implement a same `Resource` interface. Users can then interact with them through a same set of operations regardless of their actual types.

The resource-centric model has been implemented in the Web architecture, where HTTP/1.1 provides a standard set of general-purpose messages for resource manipulation. To further understand the model, we borrow the OO concept of *identity/state/behavior* [27], whereby: each resource has an *identity* (URI); it may have a *state* similar to an object's variables; a resource's state is retrieved or updated through HTTP GET or PUT message respectively; data is sent to a resource via POST message for encapsulated processing (*behavior*). Interestingly, Venner's observation [29] in OO design paradigm shows that most object's operations can be generalized into three basic types: state-view, state-change and utility, which are semantically equivalent to HTTP GET, PUT and POST messages respectively. Booch [27] also suggests a similar classification (i.e. selector, modifier and free subprogram). From this insight, we may suggest that a resource is approximate to a *generalized* object with a small set of general but sufficient operations for common OO interactions. A more concrete model proposed by Baker can be found in [30].

Rather to have application-specific business semantics, HTTP messages are encoded with general resource manipulation semantics (e.g. retrieve/modify resource state, create new resource). Thus, the messages can be reused to manipulate various resources across applications. For example, we send a HTTP GET message to the Google Search resource [31] to retrieve a search results; by changing the target resource's identifier, the GET message can be sent to the *Bucket* resource at Amazon S3 service [23] to retrieve a list of objects stored in that bucket. Similar to OO, resources are differentiated by their identifiers.

4.2.2 Granularity. In the context of service interface, granularity often refers to the degree of business-level interaction details described by an interface. Raising the abstraction level often results a coarse-grained granularity. For example, a service may provide an interface to accept a purchase order submission (includes all purchase items) as a whole unit in single interaction. In contrast, a fine-grained service interface may accept a purchase order submission through multiple interactions (e.g. by accepting each purchase item separately). Since granularity represents the business-level interaction abstraction, it is important to align it with consumer needs and application domain requirements. In other words, the granularity level should be a design choice made by architects to match the business requirements, and not because of the adoption of certain interface design approach. Therefore, it should not become a primary reason to choose between interface design approaches.

However, architects should be aware of the cost of network latency in actual deployment environment since most interactions will be done through network links. Besides, it is worth to note here, as pointed out by Feuerlicht [14], if without careful design, the message-centric approach may produce *over*-coarse-grained interfaces, which results in complex message structures. A better message design method may be needed to cope with message schema maintainability issues.

4.2.3 Integration. Standardization on communication point is an important step to promote interoperable interaction and integration among a wide range of heterogeneous components [32]. It has been identified that it is more difficult to standardize *control-centric* interfaces than *content-centric* interfaces [8, 13].

The method-centric interface is control-centric. A service is allowed to expose a variety of arbitrary application-specific operations, which are often vendor-specific. For example, Google search engine currently provides a `doGoogleSearch` operation, while other search engines will certainly provide similar operations with different names. Thus, in order to substitute services dynamically, consumers are required to know every vender-specific operation available in the marketplace. This approach does not scale well for integration tasks within a domain as well as across domain when the number of services grows. Early service-oriented research on integration and interoperation has focus on workarounds to reduce the interface variation by using an interface adapter framework [8], introducing a generalized abstract service interface layer on top of services [9, 11], and mapping *is-a* hierarchical relationship between similar services [33].

In current practice, message-centric interfaces are often developed and governed by a few standard business alliances (e.g. OTA [16]) and used within their own domain. Open and standard message formats help in promoting interoperable communications.

However, since the current practice of standardization in message-centric is towards domain-specific, the messages are often not usable outside a specific interaction context. Thus, it does not ease the integration tasks across domain. Section 4.2.4 will continue to elaborate more on this particular issue.

The resource-centric interface is towards content-centric. Its generic resource interaction model provides a unified data-centric model to simplify overall integration tasks [13]. Interface variation is minimized by enforcing standardization on message semantics, i.e. HTTP. General-purpose message semantics in HTTP encourage further adoptions within and across domain. The resource-centric has been adopted by several ubiquitous computing research [34-36] as a middleware integration platform for a wide range of portable computing devices.

4.2.4 Evolution. In method-centric approach, each service procedure call is identified via its signature, thus suffers from tight coupling between the operation name and input arguments. A slight change in input arguments will alter the procedure signature and immediately break the interface contract. Interface evolvability may be improved by separating input arguments from the signature and representing them with a more flexible format (e.g. RDF document).

It is common to see huge and complex XML message schema specifications in message-centric interfaces. Feuerlicht suggests the complexity and redundancy of message data structures increase data coupling and reduce message evolvability [14]. Besides, we also notice that many message schemas are hardly reusable outside a specific interaction context. Possibly, it is due to the tight coupling between the context-specific protocol message and its document payload. To illustrate this: RosettaNet's Cluster 3 "Order Management" PIP3A4 Specification [15] defines a `PurchaseOrderRequest` message standard specifically for the buyer to submit an order to the seller to purchase desired items. In this case, the message is context-specific thus limiting its reusability outside the buyer-seller purchase order submission scenario. An alternative design is to separate out a standalone `PurchaseOrder` document from the `PurchaseOrderRequest` message and standardize the document format for public adoption. Compare to a standardized `PurchaseOrderRequest` message, a standardized `PurchaseOrder` document may be reuse across domain.

In contrast to message-centric, resource-centric decouples the document payload from HTTP protocol message. This simple separation introduces significant advantages. First, both protocol message and document can now evolve independently. Second, a standalone document can be reuse in different contexts across domain. For instance, a HTTP POST message with `PurchaseOrder` document as payload can be sent to a printing service to print out the document, as well as to a purchase service to submit the purchase order. Besides, a simple technique similar to OO method overloading may be used in resource-centric design to encourage evolvability. For instance, a purchase service may initially accept only one type of `PurchaseOrder` document via a POST message. Over time, it may overload the POST message to accept new variants without rejecting the earlier one. This may allow a service to upgrade gradually without breaking the earlier interface contract.

4.2.5 Latency. The resource-centric model is often associated with HTTP, a network-based application protocol. HTTP provides a caching feature to handle network latency. Thus, architects may leverage this feature and decide the caching strategy at the design time. On the other hand, method-centric and message-centric approaches are not associated with a particular network-based application protocol. Since the network latency problem is unavoidable in the service deployment environment, architects should know how to handle the problem in the chosen communication platform. Although some may argue that latency is more a deployment issue, it is an advantage to consider it earlier to ensure services are designed with performance concern from ground up. Hiding or ignoring the latency issue may consider harmful in a distributed computing environment [6].

4.2.6 Partial failure. Partial failure is a central reality of distributed computing environment [6]. Due to the distribution of computation tasks among a set of physically separated components, one component either machine or network link may fail while the others continue. In an Internet-scale network, it is impossible to tell precisely where the failure has taken place. This leads to the question: can a buyer send an identical purchase order request twice or more, when fail to receive a response?

In the resource-centric model, every interaction can be generalized into simple resource state manipulation. Sending a request to a resource is simply an action to change or retrieve the resource's state. When dealing with partial failure, this model allows a sender to retrieve the latest resource's state to verify whether the previous state change attempt has been successful before deciding to resend the request. In addition, each message in HTTP is given a label (e.g. safe, idempotent [21]) to suggest its usage and possible effects when dealing with identical requests. It aids architects to make appropriate design decision, for example HTTP PUT may be preferred over POST when possible, due to its idempotent nature that allows clients to resend identical requests without breaking the intended semantics. Both method-centric and message-centric approach may utilize this feature by choosing HTTP as the communication protocol in deployment. An alternative is to migrate the responsibility to a separate reliable-messaging layer (e.g. WS-Reliability [37]) underneath all interacting components.

4.2.7 Synchronous/asynchronous interaction. A service may require certain business process interactions to be asynchronous. For example, when a purchase order is submitted to the seller, the purchase order may be put into pending for days before the final confirmation (or rejection) returned to the buyer.

At interface level, we often view a procedure call as synchronous [38] because in method-centric approach, each business process function is often modeled as single procedure call. For instance, when a `submitPurchaseOrder` operation is designed to return the final confirmation results, it does not make much sense to invoke the procedure call and maintain the connection for days before the results returned. Therefore, method-centric model may not be as natural in representing asynchronous interactions. In local computing, passing a callback function reference address along with the invocation will allow asynchronous call. To apply this concept in a large-scale distributed environment, standardization on the reference address schema is essential to ensure the callbacks are identifiable and accessible globally.

By decoupling the request and response into two separate messages, message-centric is often viewed as a better choice for asynchronous interaction. A pair of separated request/response messages are designed to represents a single business process function. For example, a purchase order submission function can be modeled as two separate messages: a `PurchaseOrderRequest` message contains all purchase items details and a `PurchaseOrderConfirmation` message with the order confirmation. Due to the partial failure nature in the deployment environment, architects often need to design an extra `Acknowledged` message to be sent back immediately from the receiver to the sender whenever a request received.

By not hiding the network existence at interface level, the HTTP specification provides a similar `Acknowledged` message mentioned above (HTTP 202 Accepted [21]). Business-level asynchronous interactions can be achieved with two separate client-server requests: a buyer sends a purchase order (*first* request) to the seller along with a special resource identifier (e.g. a URI) and the seller returns a "HTTP Accepted" response. Later, the seller sends an order confirmation results (*second* request) to the special resource supplied in the first request.

5. Related work

An earlier service interface design comparison done by Henkel and Zdrakovic can be found in [19]. They presented a brief discussion about the differences of the three interfaces, without providing detailed evaluation. In a service-oriented design methodology proposed by Papazoglou and Heuvel [38], they suggested the difference between method-centric and message-centric interface is merely the interaction model (synchronous/asynchronous), thus architects may simply decide to adopt one of the approaches based on the preferred interaction model. Feuerlicht further suggested that decision can be delayed until the implementation stage [14]. We suspect the assertions were made due to the lack of understanding on the abstraction model underlying each approach. From our evaluation, it is clear that there are simply more issues to be considered during the design of service interface.

6. Conclusion and further work

The paper presents a comparative evaluation on three service interface approaches, i.e. method-centric, message-centric and resource-centric. It first explains common issues found in the service-oriented environment. Based on this environment, a list of evaluation criteria has been selected and grouped into three design concern areas. The list is then used in the analysis and evaluation of each design approach.

From the evaluation, we notice the method-centric abstraction model can be mapped naturally to existing programming languages, thus several established design techniques may be applied easily. On the other hand, the uniform interface constraint in resource-centric approach simplifies the integration tasks. While the message-centric interface allows asynchronous data exchange, its maintainability problems should be taken care during the design. This paper does not provide quantitative data to insist that certain design approach is superior over another. It is more important for architects to first, be aware of the common issues in service-oriented environment, then selectively adopt or hybrid desired design approaches and decide the associated technology platforms tailored to the actual deployment environment. Further research interest may focus on service interface design techniques from both interoperability and maintainability perspectives.

7. References

[1] K. Bennett and V. Rajlich, "Software maintenance and Evolution: A Roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. Limerick, Ireland: ACM Press, 2000, pp. 73-87.

[2] K. Bennett, M. Munro, N. Gold, P. J. Layzell, D. Budgen, and O. P. Brereton, "An Architectural Model for Service-Based Software with Ultra-Rapid Evolution," in *Proceedings of ICSM'01*. Florence: IEEE Computer Society Press, 2001, pp. 292-300.

[3] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid, "Business-to-Business Interactions: Issues and Enabling Technologies," *The VLDB Journal*, vol. 12, pp. 59-85, 2003.

[4] M. Acharya, A. Kulkarni, R. Kuppili, R. Mani, N. More, S. Narayanan, P. Patel, K. W. Schuelke, and S. N. Subramanian, "SOA in the Real World - Experiences," in *Service-Oriented Computing - ICSOC 2005, Third International Conference*, vol. 3826, *Lecture Notes in Computer Science*. Amsterdam, The Netherlands: Springer, 2005, pp. 437-449.

[5] M. Turner, D. Budgen, and P. Brereton, "Turning Software into a Service," *Computer*, vol. 36, pp. 38-44, 2003.

[6] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A Note on Distributed Computing," Sun Microsystems Labs SMLI TR-94-29, November 1994.

[7] Google, "Google SOAP Search API," 2006; http://www.google.com/apis/.

[8] S. R. Ponnekanti and A. Fox, "Application-Service Interoperation without Standardized Service Interfaces," in *In IEEE International Conference on Pervasive Computing and Communications (PerCom '03)*. Fort Worth, Texas, 2003, pp. 30-37.

[9] B. Verheecke and M. A. Cibrán, "AOP for Dynamic Configuration and Management of Web Services," in *The International Conference on Web Services - Europe*, 2003.

[10] P. Henderson and J. Yang, "Reusable Web Services," in *Proceedings, 8th International Conference on Software Reuse*, 2004, pp. 185-194.

[11] L. Melloul and A. Fox, "Reusable Functional Composition Patterns for Web Services " in *IEEE International Conference on Web Services (ICWS'04)*, 2004, pp. 498.

[12] M. Turner, F. Zhu, I. Kotsiopoulos, M. Russell, D. Budgen, K. Bennett, O. P. Brereton, J. Keane, P. J. Layzell, and M. Rigby, "Using Web Service Technologies to create an Information Broker: An Experience Report," presented at 26th International Conference on Software Engineering (ICSE'04), 2004.

[13] R. T. Fielding, "JSR 170 Overview: Standardizing the Content Repository Interface," Day Management AG (www.day.com), Switzerland 13 March 2005.

[14] G. Feuerlicht, "Application of Data Engineering Techniques to Design of Message Structures for Web Services," in *Proceedings of the First International Workshop on Design of Service-Oriented Application (WDSOA'05)*. Amsterdam, The Netherlands: IBM Research Division, RC23819, 2005.

[15] RosettaNet, "RosettaNet Standards," 2006; http://www.rosettanet.org/.

[16] OTA, "The OpenTravel Alliance," 2006; http://www.opentravel.org/.

[17] M. Baker, "Towards truly document oriented Web services," 18 July 2005; http://www.coactus.com/blog/2005/07/towards-truly-document-oriented-web-services/.

[18] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[19] M. Henkel and J. Zdrakovic, "Approaches to Service Interface Design," in *Proceedings of the Web Service Interoperability Workshop, First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'2005)*. Geneva, Switzerland: Hermes Science Publisher, 2005.

[20] S. A. Tanenbaum and V. M. Steen, *Distributed Systems - Principles and Paradigms*. Upper Saddle River, New Jersey: Prentice-Hall, Inc, 2002, pp. 647-677.

[21] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, IETF RFC 2616, HTTP Working Group, June 1999.

[22] del.icio.us, "del.icio.us online bookmark system API," 2006; http://del.icio.us/help/api/.

[23] Amazon, "Amazon Web Services: Amazon S3," 1 Mar. 2006; http://docs.amazonwebservices.com/AmazonS3/2006-03-01/.

[24] OMG, *Common Object Request Broker Architecture (CORBA) Core Specification*, Version 3.0.3, Mar. 2004; http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[25] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and*

Systems Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1979.

[26] G. Feuerlicht, "Data Engineering Approach to Design of Web Services," in *Lecture Notes in Computer Science*, 2005, pp. 766-767.

[27] G. Booch, *Object-Oriented Analysis and Design: with applications*, Second Edition ed. United States of America: Addison Wesley Longman, Inc, 1994, pp. 81-143.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. United State of Ameria: Addison-Wesley Publishing Company, 1994, pp. 17.

[29] B. Venners, "Designing Fields and Methods: How to Keep Fields and Methods Decoupled," Mar. 1998; http://www.artima.com/designtechniques/coupling3.html.

[30] M. Baker, *An Abstract Model for HTTP Resource State*, Internet-Draft draft-baker-http-resource-state-model.txt, IETF, Nov. 2001; http://www.markbaker.ca/2001/09/ draft-baker-http-resource-state-model-01.txt.

[31] Google, "Google Search," 2006; http://www.google.com/search.

[32] G. Feuerlicht and S. Meesathit, "Design Method for Interoperable Service Interfaces," in *Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA: ACM Press, 2004.

[33] S. R. Ponnekanti and A. Fox, "Interoperability among Independently Evolving Web Services," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Toronto, Canada: Springer-Verlag New York, Inc., 2004.

[34] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Shchettino, B. Serra, and M. Spasojevic, "People, Places, Things: Web Presence for the Real World," *ACM MONET (Mobile Networks & Applications Journal)*, vol. 7, 2000.

[35] J. Barton, T. Kindberg, H. Dai, N. B. Priyantha, and F. A. Ali, "Sensor-enchanced Mobile Web Clients: An XForms Approach," presented at Proceedings of the 12th International Conference on World Wide Web (WWW'03), 2003.

[36] I. Radusch, S. Steglich, and S. Arbanowski, "Evolvable Services for the Ubiquitous Information Society based on Cooperating Objects Platforms," presented at Proceedings of IEEE International Conference on Systems, Man and Cybernetics (IEEE SMC 2004), The Hague, Netherlands, 2004.

[37] OASIS, *Web Services Reliability Messaging TC (WS-Reliability 1.1)*, OASIS Standard, 15 Nov. 2004; http://docs.oasis-open.org/wsrm/ws-reliability/v1.1.

[38] M. P. Papazoglou and W. J. van den Heuvel, "Service-Oriented Design and Development Methodology," *to appear in Int'l Journal of Web Engineering and Technology (IJWET)*, 2006.