RC22184 (W0109-054) September 25, 2001
Computer Science

IBM Research Report

Software Engineering for Web Services: A Focus on Separation of Concerns

Brent Hailpern, Peri L. Tarr
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Software Engineering for Web Services: A Focus on Separation of Concerns

Authors:
Brent Hailpern (bth@watson.ibm.com) and
Peri Tarr (tarr@watson.ibm.com)
IBM Thomas J. Watson Research Center

1. Introduction

XML has started a revolution on the World Wide Web, moving it from static formatted content to dynamic, self-describing information with real semantics. The addition of semantics is only the first step to providing real applications, known as "web services," to Internet users. The first generation of web-service infrastructure is already under development, and it will allow one web service to issue a request to another and to register/describe/find a service to use (e.g., SOAP [1], WSDL [2], UDDI [3]), thus creating interconnected sets of cooperating web service components.

Eventually, web services will grow beyond the new distributed computing infrastructure of SOAP, WSDL, UDDI, etc., to become *electronic utilities* (eUtilities) that are delivered to end users over the Internet. EUtilities represent a critical new application domain in electronic commerce. Like traditional utilities, such as telephone and electricity, web services will be metered and customers will pay for their use of the eUtility. The terms of use (called *service level agreements*, or SLAs) of the eUtilities will include functionality, availability, performance, resources, and reliability. These terms of use may vary from customer to customer, and they may change over time. This, in turn, necessitates dynamic monitoring of eUtilities, dynamic control over SLAs, and the ability to respond quickly to changing customer needs and available resources. Providing these important eUtility capabilities impose some challenging requirements on the design, development, deployment, and evolution of web services. SLAs represent a legal description of a service—not simply in terms of its *functional* interface, but also in terms of performance, payment, legal consequences of non-compliance, and levels of support and access. Because SLAs are legal agreements, it must be possible to *monitor* web services to verify that the services being provided conform to those that were negotiated, and to redress any conformance failure immediately.

As soon as web services must satisfy serious, demanding SLAs, developers will have to confront the task of designing, building, configuring, deploying, and managing software that must meet, and continue to meet, constraints beyond the simple functional correctness with which we have been struggling for 50 years. Web service applications must include both the conventional APIs, and also interfaces for metering, performance, manageability, auditability, replication, and reliability. These interfaces, though distinct, reflect capabilities that must interact with one another in deep and profound ways. Supporting the description, realization, integration, and separate evolution of these interfaces will be a major challenge facing eUtility engineers.

To increase the complexity even further, we note that web services must be dynamically configurable. Just as the telephone company depends on the electric utility in providing its phone services, so future web service applications will depend on web eUtilities provided by others. So the environment in which this software runs will be subject to change without notice – both in terms of the underlying eUtilities and in terms of the number and kind of users (where some of those users may be higher-level web services). Other services on which a given eUtility depends may disappear or may change in some way that affects the eUtility, and it must be able to detect and respond to such changes whenever possible. When it is not possible, they must be able to degrade gracefully, since organizations may rely on them for critical parts of their business.

Our research is based on, but goes beyond, notions of Aspect-Oriented Software Development (AOSD) and Multidimensional Separation of Concerns (MDSOC) [4], applied to the software engineering of web services. AOSD (and notably, Hyper/J™ [5] and AspectJ™ [6]) has, up to this point, focused predominantly on the "programming" part of the software development lifecycle. In particular, aspect-oriented approaches support the identification and creation of "crosscutting" concerns that affect many modules throughout an object-oriented system. By modularizing aspects of a system that do not conveniently fit the dominant hierarchy (e.g., inheritance), one can avoid duplication and error in implementation and isolate changes during maintenance. These various aspects are then woven into the base code to form a new program. The "base code" itself imposes a dominant structure on the software and on the aspects, which must be written with knowledge of the semantics and structure of the base code. MDSOC supports aspect-oriented development, but it also supports the reconciliation and integration of separately developed class hierarchies, eliminating the distinction between "base code" and "aspect code." Within the development of web services, we expect to see the need for both aspects (AOSD) and for multiple domain models (MDSOC). Aspects will be useful in the definition of particular web services, for example interfaces for one particular subset of customers of the eUtility. However, particularly because different web services that must ultimately work together will be developed independently (and will be able to run as standalone applications), by different organizations, we do not believe the "base" vs. "aspect" distinction will be appropriate to support the composition of different web services. MDSOC will be needed to achieve this, and to permit the separation of interacting concerns, such as a monitoring service from one supplier along with a multileveled-performance service from another supplier.

Our approach to the engineering of web services is to identify a set of key concerns—both functional and non-functional, such as performance, manageability, and monitoring—implement the software to realize and address these concerns, and to integrate some set of these concerns together—possibly dynamically—to produce web services that can be configured differently to meet the needs of different customers. We expect that performance management, for example, will impact many portions of an eUtility (what level of resources are available for a user or class of users, who has legal requirements that must be met, and so on). Hence, collecting those issues together in a concern can serve both as a design, implementation, and evolution tactic. Dependence on remote services can also be modularized as another dimension, so that the underlying application can be prepared to switch between a variety of competing (or failing) suppliers – this concern, then, becomes a configuration/deployment tactic. Providing run-time interfaces during execution for monitoring/control will meet the legal needs without cluttering up the functional APIs. Finally, maintenance will enjoy the usual simplification that having a multidimensional separation of concerns provides.

We will discuss the application of MDSOC and AOSD to the domain of web services and, perhaps more importantly, explore how these areas must grow to support all of the software lifecycle in order to be a solution for the wealth of challenges that web services will present to the software engineers of the future.

2. Issues in building and deploying eUtility applications

To motivate the issues we are addressing, we begin with a description of our view of an object-oriented web service (OOWS). As shown in Figure 1, we view an OOWS as a software component that is built on top of a highly reliable infrastructure and possibly other OOWS, providing such services as those listed above. Like other kinds of components, OOWSs provide functional interfaces (APIs), and each API supports some set of capabilities. Unlike other components, however, OOWSs must be dynamically controllable by SLAs, so that they can change their behavior and characteristics in response to changes in SLAs. Thus, they must also provide *management interfaces*, which permit control over such attributes as performance, reliability, monitoring, metering, level of service, and potentially, the set of capabilities the OOWS provides and the particular components on which it is built. Because SLAs may change at any time, it must be possible to control a service's functional and non-functional properties at any time during the software lifecycle—build-time, integration-time, or runtime.

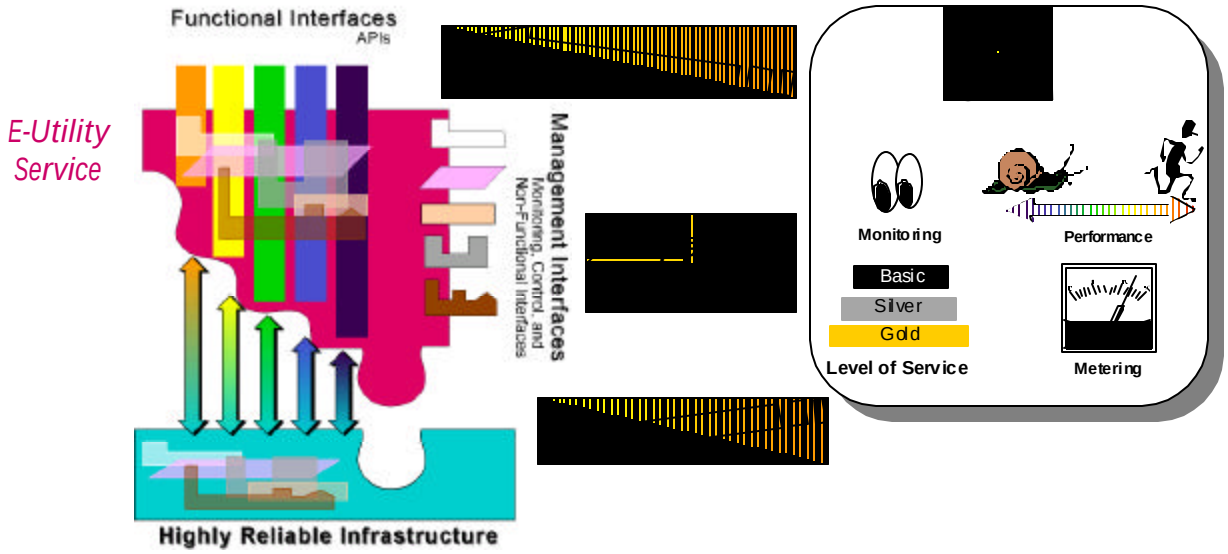

Figure 1: Web Services (eUtilities), SLAs, and Interaction Among Functional, Management, and Infrastructure Interfaces

Interaction among functional, management, and infrastructure APIs: web service applications challenge the software engineering models that have dominated application development for traditional clients and server platforms. In particular, note that traditional applications are specified and implemented in terms of a single set of functional requirements. These requirements are generally uniform for all end users; thus, these applications define one or more *functional* interfaces (APIs). A web service (and the components that comprise it) also addresses traditional functional requirements, but each customer also negotiates for the particular functionalities, levels of service, performance and reliability that they will obtain from the service, and they will pay for their use of the service. Thus, a web service must define one or more functional APIs (possibly different ones for different customers), but it must also present interfaces that permit control over performance, reliability, metering, and level of service. These interfaces must interact with one another in some deep and fundamental ways, as suggested by Figure 1. For example, obtaining high performance may necessitate the use of different communication protocols, search algorithms, and concurrency control models, or it may preclude the use of certain functions that cannot be made to achieve that performance. The definition and support for such deeply interacting interfaces presents a significant challenge to service engineering and deployment.

MDSOC and AOSD technologies may help to achieve these goals. They facilitate the separation and integration of many types of concerns, and could, therefore, help OOWS builders to achieve decompositions that permit pluggable, mix-and-match functional and management capabilities.

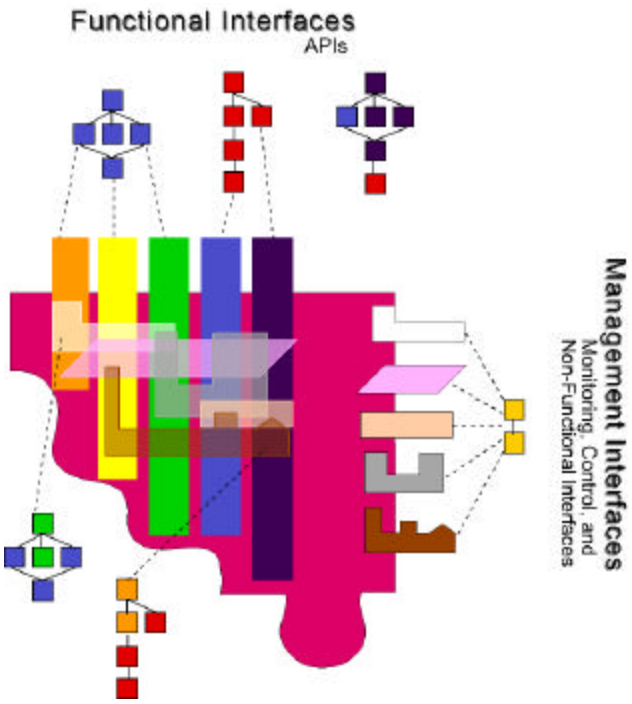

Figure 2: Different capabilities are implemented using different domain models and different class hierarchies. These hierarchies are reconciled and integrated when they are composed.

Figure 2 suggests two key characteristics of web services. One is that capabilities embodied in management interfaces may crosscut capabilities present in APIs, making them good targets of aspect-oriented technologies. The other characteristic is that some functional and management capabilities may require their own class and interface hierarchy to implement the necessary domain model, and that these different hierarchies must be integrated as the capabilities are combined into a particular OOWS. In fact, the same issue arises when different OOWSs are composed together to help build a new service. MDSOC supports this, while other kinds of aspect technologies do not. This is likely to be a critical problem for OOWS developers, who do not have control over the domain models used in either underlying eUtilities. For this reason, we believe the MDSOC approach, which permits the reconciliation and integration of somewhat different domain models, is more appropriate for engineering web services than other AOSD approaches.

While both MDSOC and AOSD technologies are promising, they do not, at present, address some critical issues in the interaction of functional and management interfaces:

?? Semantic mismatch: A semantic mismatch will occur when two or more capabilities have mutually incompatible assumptions or requirements. For example, a client may negotiate for a level of service that cannot be attained with a given concurrency control mechanism or with the particular strategy employed to implement some feature. For OOWSs, it is necessary to be able to identify semantic mismatches *before* they bite a client. Thus, for example, it is important to be able to tell that a particular set of requirements in a client's SLA cannot be satisfied simultaneously, or that doing so will necessitate new components, algorithms, or infrastructure, and this must be determined before the capabilities are promised to the client.

The problem of semantic mismatch is not a new one—it can occur in any piece of software. In many cases, however, semantic mismatches caused by compositional approaches, like MDSOC and AOSD, cannot be identified until late in the software engineering process—typically at runtime. This is far too late for the web services domain, where SLAs represent legal contracts with customers, and where SLAs may change dynamically. It is imperative to be able to detect semantic mismatches early enough to fix them, or to prevent agreement on an unsatisfiable SLA. Neither MDSOC nor AOSD currently

addresses semantic mismatch in any form, yet because they can modify an encapsulated module, they have even more potential to introduce semantic mismatches than standard OO mechanisms.

?? Interference: Interference is a common kind of semantic mismatch problem in concurrent software, but there is real potential for interference to occur in OOWSs between and within the management and functionality interfaces. It happens when an action taken by one part of the software interacts with another's actions in an undesirable manner, causing an effect that does not occur when the actions occur independently. An example of interference can be found when messaging and transactions are present together. To allow a transaction manager to preserve atomicity and serializability, messages should not generally be sent until a transaction commits. A messaging capability could easily send messages independently, thus interfering with the ability of the transaction manager to satisfy its requirements.

   The problem of interference, while present in other areas of software engineering, is, like other semantic mismatches, particularly severe when using AOSD mechanisms, because such mechanisms integrate separately encapsulated, separately tested entities. It is not possible to detect all kinds of potential interference by inspecting just the code, because the composition operations themselves add logic when integrating the individual concerns or aspects. The interference problem, while prevalent and potentially devastating (particularly in an OOWS context), is not one that is addressed in current work on AOSD. Without reasonable mechanisms for doing so, the technology may be more harmful than helpful in the engineering of OOWSs.

?? Unpredictability: As the discussions of semantic mismatch and interference suggest, it is not always possible to determine, *a priori*, what a piece of composed software will look like. Unlike non-compositional paradigms, where one does not get behaviors that one did not directly program (excluding problems like concurrency errors), developers using compositional paradigms will encounter circumstances where their composed software behaves unexpectedly and unpredictably. This is, in part, because compositors add logic, as noted earlier, but it is also because compositors break existing encapsulations to integrate concerns. The developers of those encapsulations made certain assumptions about the behavior of the module, and those assumptions are easy to violate when code from new concerns is interposed within an existing module. The unpredictable effects of composition tend not to be found until they manifest as erroneous behavior at runtime. This will not be acceptable in an OOWS context.

Verifiable and controllable conformance to SLAs: Each customer's contract for a service is described and governed by an SLA (see Figure 1). It must, therefore, be possible to determine whether the appropriate SLA(s) is/are being satisfied, to determine how and why it is not being satisfied, and to control the service to bring it into compliance with the SLA(s). This suggests again the need for programmatic interfaces to control metering, performance, and other crosscutting, non-functional capabilities. It also potentially requires dynamic addition, removal, and replacement of capabilities, which could be supported eventually by AOSD and MDSOC but is not at present[1]. The interaction between these management capabilities and the functional capabilities is again apparent here, since an OOWS may not provide certain functionality or performance for users who request lower levels of service. Finally, our assumption that OOWSs cooperate with the ultra-reliable infrastructure to satisfy SLAs requires some interfaces to support that cooperation.

Building services when the "components" are neither local nor locally controllable: As with all software, it is reasonable to assume that "composite" OOWSs may be built—i.e., where one service depends on other (lower level) services, each with their own SLAs. Hence, the designer/builder of a service may—indeed, should—spend more effort in integrating services than in constructing new ones from scratch. But this designer/builder is confronted with a body of services (comparable to traditional

[1] Some AOSD mechanisms can support degree of dynamic addition, but we are not aware of any that support dynamic removal, and only MDSOC supports the on-demand remodularization capability needed to identify and remove parts that were not originally encapsulated as concerns.

components) that may not be under his/her direct control, since some else may own the service and may be delivering it over the network. Thus, such a designer/builder will not be able to use traditional software engineering methodologies in developing composite services, since these methodologies depend on the centralized control and static deployment assumptions that clearly do not hold.

Note that the assumption in most of AOSD of a distinguished base vs. aspect is inconsistent with the composition of independent web services. MDSOC does not have this limitation, but most existing techniques assume that the compositor has control over all the software that is to be composed. Thus, here again, these approaches will have to grow to accommodate the requirements of OOWSs.

Inevitable, unpredictable, dynamic changes and need to limit the impact of such changes: Any given OOWS may change or become unavailable without notice, thus potentially causing a cascaded impact on all the services that depend upon it. Services are subject to significant reliability and availability constraints, so the change or loss of an underlying service cannot, in general, be allowed to crash or impede other services. Thus, services must come with integration and runtime support systems that can identify service change or loss and respond to it. This likely includes, though is not limited to, the ability to handle real-time version control, configuration management and "hot swapping" of bits and pieces of services, and the ability to upgrade and degrade gracefully. Note again that changes and unavailability affects both the functional *and* non-functional (management) aspects of a dependent service (and the legal implications of such a change). The version control and configuration management problem is considerably more complex than its traditional build-time analogue. At build-time, a version control or configuration management system need only be concerned with choosing a set of modules to put together, but at runtime, it must be possible to replace much smaller-grained pieces, to ensure the lowest impact of change on the modified service (the larger the pieces that are replaced, the larger the potential impact on other parts of the service, and on any services that depend on it). Advanced configuration management approaches, like [7], may be of use here.

SLAs are software, too: SLAs themselves must be considered a key component in the software engineering of services. In particular, they both define and configure the functional and non-functional aspects of a service. They are, therefore, a specification for the service, and it must be possible to ensure that they are satisfied both statically and dynamically. If SLAs are to be monitored and their requirements satisfied, they must be treated as software artifacts in their own right—perhaps as declarative specifications of services, or as some kind of operational semantics. In either case, they must have their own build/integrate/test/deploy cycle, which must tie in directly with the capacity planning, design, and architecture of the OOWS, the monitoring/execution/control of the service, and the compliance checking and reporting to the provider and the end-user of the service. Given the competitive nature of the first generation of OOWS-like vendors (ASPs), one can expect rapid evolution of SLAs (at least in terms of cost of service options provided). Hence these SLA "programs" will have to accommodate change during execution.

3. Conclusions and Future Work

Object-oriented web services represent a critical new application domain in electronic commerce. Like more traditional utilities, such as electricity and telephony, eUtilities will be metered services, and customers pay for their use of the service. Customers negotiate the terms of the service provided, such as the functionality they require, the performance, reliability, resources, etc. These terms are represented using service level agreements (SLAs). The need to control the terms of service—probably dynamically—and to respond to changing customer needs and available resources imposes some challenging requirements on the design, development, deployment, and evolution of OOWSs.

As we have noted, traditional software engineering methodologies and programming paradigms are completely inadequate to permit the engineering and deployment of these critical new applications. Advanced separation of concerns paradigms and technologies, like AOSD and MDSOC, overcome some of these limitations, but they are not, at this time, sufficiently powerful or predictable to enable the engineering of OOWSs and, indeed, their limitations may render them of more harm than help at present.

Considerably more research is needed to push these paradigms to the point where they will satisfy the requirements of this challenging new application domain.

References

[1] SOAP web site, http://www.w3.org/TR/SOAP.

[2] WSDL web site, http://www.w3.org/TR/wsdl.

[3] UDDI web site, http://www.uddi.org/about.html.

[4] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns." In Proc. ICSE 21, May 1999.

[5] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and the Hyperspace Approach." In *Proc. Symp. Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2001.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. "An Overview of AspectJ." In Proc. ECOOP 2001, June 2001.

[7] M.C. Chu-Carroll and S. Sprenkle. "Coven: brewing better collaboration through software configuration management." Proc. 8th International Symposium on Foundations of Software Engineering, 2000.