A Design Technique for Evolving Web Services

Piotr Kaminski
University of Victoria
Dept. of Computer Science

Marin Litoiu
IBM Canada Ltd.
IBM Toronto Lab., CAS

Hausi Müller
University of Victoria
Dept. of Computer Science

Abstract
In this paper, we define the problem of simultaneously deploying multiple versions of a web service in the face of independently developed unsupervised clients. We then propose a solution in the form of a design technique called Chain of Adapters and argue that this approach strikes a good balance between the various requirements. We recount our experiences in automating the application of the technique and provide an initial analysis of the performance degradations it may occasion. The Chain of Adapters technique is particularly suitable for self-managed systems since it makes many version-related reconfiguration tasks safe, and thus subject to automation.

1. Introduction
Version management of deployed software has always been a tricky business. In this age of shortened development cycles, direct unsupervised links between independently developed applications, and increasingly self-managing systems, the complexity of evolving “live” applications is becoming a critical issue. In this paper, we explore the problem and propose a design technique that makes managing version evolution simpler—whether for human administrators or self-managing systems.

Since easing version management is an overly broad target, we focus specifically on versioning of web services—broadly understood as applications whose functionality is exposed to third-party clients over a network. Our goal is to permit the evolution of a service’s interface and implementation while remaining backwards-compatible with clients written to comply with previous versions. Section 2 lists all our requirements in detail and demonstrates why a number of common versioning strategies are inappropriate in this context.

Our solution, which we call Chain of Adapters and present in Section 3, is a design technique that can be applied by the service developer and imposes no requirements on clients or server infrastructure. While our solution is simple enough to be applied manually, in Section 4 we also describe a prototype tool we have built to automate some of its more repetitive aspects, and provide a preliminary performance analysis. The Chain of Adapters technique is well suited to deployment in self-managing systems since it affords the manager a larger number of safe configuration options.

Section 5 discusses related work, and Section 6 concludes with a summary and future research directions.

An earlier version of this paper appeared in the proceedings of the SEAMS 2006 ICSE workshop [16].

Copyright © 2006 Piotr Kaminski, Hausi Müller, and IBM Canada Ltd. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.
2. Requirements

In this section, we lay out precisely our interpretation of the version management problem in terms of the requirements that a solution would have to fulfill. To make the discussion more concrete, we also showcase a few standard approaches to solving the problem and explain how they satisfy (or fail to satisfy) the posited constraints. We illustrate the discussion with diagrams of sample web service configurations such as the one in Figure 1, which presents a basic single version arrangement that is the starting point for all approaches.

The underlying scenario we assume is as follows. A developer constructs a web service and makes it available at an advertised endpoint, while publishing its interface (e.g., in Web Services Description Language, WSDL [7]) and concomitant data type definitions (e.g., as XML Schema documents). One or more third-party clients start using the service, by binding the interface and data schemas into their application and connecting to the publicized endpoint (e.g., over SOAP [11]). The web service stores some information between invocations, and may share that information between clients (e.g., an auction once posted can be bid on by everyone).

The question we explore in this section is: what are the desirable properties of an evolving web service?

2.1 Backwards Compatibility

The fundamental requirement for evolution in cases where the service developer has no control over the clients—such as in our putative scenario—is to maintain strict backwards compatibility. Clients written to work with earlier versions of the web service must continue to function correctly even as the web service evolves, at least until support for the older version is formally withdrawn.

Trivial ways to satisfy this requirement (Figure 2) include: (a) supporting only the latest version—a tactic unlikely to please current customers, or (b) freezing its external interface at the first published version, which would effectively cause the feature set to stagnate and thus fail to attract new customers. Neither solution is particularly realistic unless exceptional circumstances prevail.

Note that we do not worry about forwards compatibility—that is, the ability of clients developed against a newer interface to work with older versions of the service. This only becomes a problem if the service’s interfaces are implemented independently at multiple endpoints, in which case a client written against v2 might find itself faced with a v1 interface after switching service providers. We believe that in this kind of scenario it is reasonable to devolve the burden of interacting with older versions of the service onto the clients’ developers.

2.2 Common Data Store

Another critical requirement is that a consistent, common service state must be exposed to all clients, from the oldest to the newest. For example, in an on-line banking web service, changes to an
account balance made by a new $v_2$ client deployed at the bank’s branches must be visible to an older $v_1$ client deployed on the customer’s home computer. Naturally, data related only to newly introduced features is exempt from this rule, since older clients would be unable to process it anyway.

This constraint immediately invalidates one of the simplest evolutionary strategies: keep each version of a service running as-is in isolation (Figure 3). This architecture is of interest only for stateless services, and even so suffers from other defects explored in the following section.

In practice, the most common approach is to arrange for all versions of a service to refer to a single database. This can introduce its own problems because, depending on the exact architecture adopted, it might become necessary to maintain a database schema that remains compatible with all versions of the service (Figure 4). Other acceptable solutions include periodic data synchronization between versions and other database-level tricks.

### 2.3 No Code Duplication

Beyond the functional requirements listed above, we also impose some software engineering requirements. A generally (though not universally) accepted principle is to avoid code duplication in software. While this tenet is typically expressed within the bounds of an application, we hold that it should also be applied across the versions of a service. Wherever possible, common code must be factored out to ease understanding and maintenance. This way, a bug detected and fixed in one version will be automatically fixed in all other versions (as applicable), helping to keep maintenance costs under control.

Note that both of the solutions proposed in Section 2.2 (Figures 3 and 4) break this rule by duplicating the entire codebase of the web service for each version.

### 2.4 Untangled Versions

Another important software engineering principle is that of encapsulation, which we adapt in this case to require that each piece of code be assigned to one or more versions of the service. Such a partitioning will permit the removal of dead code as versions of a service are withdrawn, reducing its complexity and avoiding the Lava Flow anti-pattern [5]. It is particularly valuable for self-managing systems, since it allows independent control over each version of the service without costly and complicated human intervention.

A typical design often observed in the wild that fails the tangling test is exhibited in Figure 5. The interface of each version incrementally extends that of the preceding one, and a single service
progressively accumulates the implementation of all these interfaces. This approach requires developer intervention to deprecate interface members and excise the corresponding implementation pieces from the codebase.

2.5 Unconstrained Evolution
Another important consideration is that the evolution of the service should be unconstrained by past versions, as much as possible. The developer should be allowed to refactor, redesign, and otherwise rethink both the service’s interface and its implementation without being shackled by previous decisions. This gives the service the best chance of avoiding a slide into design debt and becoming legacy software.

It is likely that this requirement is unachievable in practice, since the absolute need for backwards compatibility will almost always constrain the shape of the service. Nonetheless, it is a worthwhile ideal to strive for.

2.6 Visible Mechanism
Finally, we feel that it is important to expose to the service developers the mechanism by which backwards-compatible evolution is achieved. After all, backwards compatibility is an inherently tricky business with lots of special cases and exceptions, and sooner or later the developer will need to dig into the guts of the framework. The idea is to keep the framework simple and unobtrusively visible, rather than try to anticipate all possible scenarios with behind-the-scenes “magic”.

We are thus opposed to backwards-compatibility frameworks that reside in the web services engine (e.g., as handlers in the Axis architecture), or that require automated generation of large amounts of opaque code. Another approach that doesn’t pass muster is to extend the XML Schema used by WSDL interfaces; not only are such extensions limited by the existing contents of the schema (breaking the requirement of Section 2.5), but the details of the extension mechanism have proven very difficult to understand correctly [22].

3. Chain of Adapters
In this section, we first explain how to apply our proposed Chain of Adapters approach to web service evolution, and then evaluate it against the requirements of the previous section.

3.1 A Simple Design Technique
The Chain of Adapters is a design technique that is most easily explained by illustrating its application to a web service under development. Suppose that a first version of the service has just been completed; to enable the evolution of the service through the Chain of Adapters, the developer should then perform these steps:

1. Duplicate the interface of the web service into a different namespace. The resulting copy will then have the same members and data structures as the original, while having no formal relationship to its parent. Call this the v1 interface.

2. Create an implementation of the v1 interface that forwards all calls to the original endpoint and interface, translating the namespace of any data structures as necessary.

3. Publish and advertise the v1 interface endpoint as the stable first version of the web service.

The result (depicted in Figure 6) corresponds to the classical web service architecture with an additional delegation layer in the form of a pass-through adapter. The structure shown is both deployed externally and used internally for further development.1

Figure 6. Chain of Adapters structure after the first version is published

---

1 Why not deploy a snapshot of the web service just before creating the v1 interface and v1→v2 adapter? While these components serve no useful function in the deployed version, publishing the current interface as v1 would force it to change namespaces with each version, inconveniencing the service’s developers.
Once v1 is deployed and in use and development of the service turns towards a new version v2, the v1↔v2 adapter\(^2\) comes into its own. Whenever the web service is modified, a compensating modification is made in the adapter to maintain the contract of the v1 interface. For example:

- If the current interface is changed through the addition of a parameter to an existing operation, the adapter must be modified to provide a default value for this parameter when forwarding the call.

- If the definition of a data structure is changed, the adapter must translate from the old one to the new one (for in parameters) or from the new one to the old one (for out parameters and return values).

- If an operation is removed from the interface, it must be re-implemented in the adapter, in terms of the other operations available in the current interface.

- If the contract of an operation is changed, the adapter must either compensate for the difference or re-implement the operation according to its v1 contract as if it had been removed.

Note that the adapter does not need to be modified when a new operation is added to the interface, nor when new optional members are added to a data structure—both will be ignored by the default delegation and translation processes.

In this way, the adapter accumulates a record of the differences between v1 and (the upcoming) v2, expressed as compensating code fragments. When v2 is ready for release, the developer must perform these steps:

1. Duplicate the current interface into a separate namespace; the copy will be v2 of the interface.
2. Create an adapter for the v2 interface that delegates to the current endpoint and interface.
3. Retarget the v1↔v2 adapter to delegate to the v2 endpoint.
4. Publish and advertise the v2 interface endpoint as the stable second version of the web service.

Figure 7 shows the resulting structure, with a new pass-through v2↔v3 adapter, and a slightly fatter v1↔v2 adapter.

Development of the web service can now continue towards v3, with compensating code placed into the v2↔v3 adapter. The v1↔v2 adapter need never be touched again, since all future incompatibilities will be compensated for by the v2↔v3 and further downstream adapters. In fact, as the service grows older and the versions mount up, the only code that needs to be edited is the service’s current codebase and the most recent adapter.

By following the “freeze, adapt, and delegate” technique established above, the web service forms a Chain of Adapters supporting an arbitrary number of versions, as shown in Figure 8.

---

\(^2\) We use a two-headed arrow (i.e., v1↔v2, rather than v1→v2) to emphasize that the adapter may need to convert the v1 interface to v2 semantics as well as convert v2 results back to v1 form.
3.2 Trade-off Evaluation

We now evaluate our proposed design with respect to the requirements stated in Section 2.

Backwards compatibility is preserved—at least in theory—by publishing only frozen versions of the service’s interface, each at its own endpoint address. In practice, it is up to the developer, backed by the full power of the programming language, to ensure that the adapters compensate appropriately for changes that are not backwards-compatible. While the compiler will pick up any trivial signature mismatches, semantic incompatibilities will be harder to catch. Chances of success can be increased by having the adapter developed concurrently with the mainline code, and by judicious application of test suites frozen along with previous service versions. Nonetheless, the risk that changes to the web service will impact past versions is intrinsic to our approach, and may make it unsuitable in certain contexts. As a colleague quipped, “share the fixes, share the bugs”.

The requirements for a common data store and no code duplication are both fulfilled by the Chain of Adapters design, since there is only one central web service implementation for all the versions. At the same time, the code specific to the peculiarities of each version is encapsulated within a separate adapter, thus preventing version tangling. The resulting structure allows versions (and their code) to be withdrawn from service cleanly, as long as it is done in strict oldest-to-newest order. The evolution of services under this design is also mostly unconstrained. The interface and implementation can be changed in arbitrary ways, provided that there exists a way to implement the contract of the previous interface in terms of the new one. Small changes are usually easily accommodated; for larger ones, the developer has the full power of the underlying programming language at his disposal, rather than some limited declarative mapping dialect. Nonetheless, if the changes are fundamental, it may make more sense to consider the new version as an entirely new service rather than try to force fit it into the old interface.

Although on its face the evolution constraint presented above is not a very onerous limitation, since obsolete operations can simply be moved into the adapter, there is one important caveat. Any functionality that requires access to the data store must remain in the main web service implementation, or the obsolete data must be split off into a new adapter-specific database. We need more experience with the technique before we can determine if this will become a real problem in practice.

Finally, the delegation mechanism espoused by this design technique is both simple and fully exposed to the developer. While the initial pass-through adapter is amenable to code generation, the resultant code is straightforward (if repetitive) and easily understood and modified by the developer.

In summary, the Chain of Adapters design technique achieves a clear win on four out of the six requirements, and delivers a manageable compromise on the other two.

3.3 Tips and Tricks

The proposed design technique raises some additional concerns that we address here. For example, there arises the question of what to do when a bug is discovered in the service implementation code. If it is fixed in the current release, the fix will affect all versions and may break clients that have come to (unknowingly) rely on the bug or implemented workarounds for it. Unfortunately, there is no straightforward answer to this question, but the Chain of Adapters supports three options:

1. If it is preferable to fix the bug in all versions, a single fix in the current version will suffice.

2. If it is decided to let the bug stand in older versions, then the bug must be fixed in the current version and compensating code that replicates the buggy behaviour must be added to the $v_n \mapsto v_{n+1}$ adapter.

3. If it is decided to let the bug stand in older versions by default but to offer a bug-fixed release under the older interface, it is possible to proceed as in option 2 but also to make a copy of the original adapter chain and offer it at a new set of endpoints (see Figure 9), effectively implementing options 1 and 2 simultaneously. This approach is useful for clients that want to take advantage of the bug fix.
without upgrading to the latest version’s interface.

The last option must be exercised carefully to prevent a proliferation of adapter chains and endpoints; it would be best to limit deployment to one stable chain and one “bug-fixed” chain. Nonetheless, as development progresses over the years, the chains will grow in length and complexity thus impeding manageability and performance. The issue is offset by taking advantage of one of the strengths of the proposed design and limiting the number of supported versions.

If withdrawing older versions from service is not desirable and their performance starts to suffer because of a surfeit of delegations, it is possible to employ another trick. To demonstrate by example, consider a web service with five deployed versions where the performance of v1 and v2 has become unacceptable because of the overhead of forwarding calls through the rest of the chain. We can rewrite the v2 ↔ v3 adapter to target the newest v5 interface instead, folding in the compensations introduced in the v3 ↔ v4 and v4 ↔ v5 adapters. The new v2 ↔ v5 adapter now skips two links in the chain, reducing the overhead and improving performance (Figure 10). In general, it is possible to skip any number of links and introduce any number of jumps into the chain, but coalescing a bunch of old adapters is not an easy job and is best left for exceptional circumstances.

### 3.4 Self-configuration Scenarios

Though the Chain of Adapters technique is applied at the level of individual web services, its effects on self-configuration are mainly felt at the level of an entire multi-service application. Whether the self-reconfiguration is triggered by a fault-healing mechanism or by the availability of updated components that add functionality or improve the quality of service of the application, having the application’s component services implemented as Chains of Adapters can help ensure a seamless transition.
A basic requirement when reconfiguring applications is that the transition should take place with no visible discontinuity in the services offered. One way to fulfill this requirement is to perform these steps: (i) buffer all incoming requests, (ii) wait until all requests in progress are completed, (iii) replace the application with a new version, and (iv) resume the buffered requests by forwarding them to the new version of the application. Step (iii) has to be done within the boundaries of an ACID transaction, which offers the ability to roll back the change if the update is not successful. The disadvantage of this approach is that draining requests out of a whole application and synchronizing a transaction across its distributed web service components can take a long time, and lead to a service interruption that is glaringly obvious to the users.

The advantage of using an approach such as the Chain of Adapters that preserves backwards compatibility for each web service—even within an application—becomes apparent in these circumstances. Instead of upgrading the whole application (i.e., all its web services) simultaneously, we can upgrade the services one-by-one using the method described above, in many localized transactions that introduce much smaller discontinuities and are easier to roll back in case of failure. Consider Figure 11a, which shows version 1 of a deployed application made up version 1 of web services A, B, C and D; each box represents an entire web service, including its database and its whole chain of adapters. Figure 11b shows the reconfiguration in progress after two steps, where web services A and B have been replaced with newer versions in two small upgrade transactions. Note that services C and D remain at version 1, and still invoke the version 1 interfaces of A’ and B’; furthermore, the operation of service D is unimpeded during the replacement of services B and C. The reconfiguration process continues replacing web services in small, inconspicuous steps until the entire application has been brought up-to-date. In case of transaction failure, the upgrades can be rolled back individually in reverse order.

4. Prototype Implementation
We have built a prototype plug-in for the Eclipse Web Tools Platform (WTP) that automates the process of freezing and publishing a version of a WSDL/SOAP web service. It has proven invaluable for testing and refining the design’s concepts because of the bulky, work-intensive syntaxes of WSDL and XML Schema. Although the plug-in lacks support for all XML Schema features, it has successfully demonstrated that it is possible to reduce the developer’s workload while keeping the version management mechanisms visible.

The plug-in was originally developed against version 0.7 of WTP and later ported to version 1.5. We describe both experiences here, as the difficulties encountered reveal limitations of WTP and of our approach that may inform the design decisions of future implementers.

4.1 Version Structure
A key design decision was how to map frozen versions to Eclipse’s project structure. Luckily, WTP v0.7 provided a ready-made answer in the form of the “module” concept. WTP modules were an organizational unit that fit within an Eclipse project but could gather files from across multiple directories and had its own context root URL for deployment. All modules in a project were automatically aggregated into the prescribed directory structure and configuration files for deployment, making it easy to keep the versions
compartmentalized without littering the work-space with projects. The separate context root URLs also allowed us to keep the web service local name constant between versions, which can be an advantage come upgrade time for clients whose WSDL code generators embed this name in the service’s stubs [9].

A major change between WTP v0.7 and v1.5 was the withdrawal of the modules feature, which was due to incompatibility with the rest of the Eclipse framework. This forced us to reconsider the structural mapping of frozen versions; we entertained four approaches:

1. Put each frozen version into its own project.
2. Put all frozen versions into one separate project.
3. Put the most recent frozen version into the current project, and all other frozen versions into one separate project.
4. Put all frozen versions into subdirectories of the current project.

The first option affords the most flexibility, cleanly isolating each frozen version and allowing each to have its own context root URL yet share the web service local name, as was the case with modules. However, the developers would need to manage an ever-growing list of projects, and project configuration changes would need to be propagated manually across all versions. The second option eases those two concerns, but forces all frozen versions to share a context root URL and hence expose different service names. It also forces developers to keep the code for all the versions loaded even though they only need to modify the most recent one.

The third option is most intriguing as it separates code that needs to be edited (the current codebase and the most recent frozen version) from code that should not be touched. However, it would require the two projects to share the same context root (which could prove confusing), the services to have different names (as above), and code to be regularly moved between projects (which might upset some code repositories).

The last option keeps all the pieces in one project, which makes it easy to keep the configuration consistent, but shares the disadvantage of a common context root URL, and increases the risk of inadvertently introducing undesirable dependencies between the versions. As this option is internally closest to the previously used modules, we have implemented it for the sake of expediency. Still, the other approaches are worth exploring in the future, and with no clear winner, perhaps it is best to leave the decision up to the developer.

4.2 Implementation Quirks

Our initial implementation of the Evolution Assistant plug-in was difficult to write since WTP v0.7 has no public API and very limited developer information. WTP is an open source project so the source code was accessible, but as Microsoft has learned in Europe, access to source code cannot substitute for cogent documentation [2]. Making the matter worse were packages entirely generated from EMF (Eclipse Modeling Framework, http://www.eclipse.org/emf/) models with no hints to the data members’ meaning, and extensive use of indirection via reflection that made caller navigation mechanisms based on static analysis (widely available in contemporary IDEs) less than useful.

Certain WTP design decisions also made our job more difficult. The framework’s level of abstraction generally does not rise above that of individual files, so we had to settle on the simplifying assumption that a WSDL file stands in for a web service even though the relationship is really more complex. The framework also fails to surface many properties that the various web service runtimes have in common, which made it impossible to make our plug-in server runtime-agnostic. (It is currently specific to Apache Axis.)

Trying to remain runtime-agnostic was also the motivation for keeping the delegation calls at the SOAP level, rather than using the better performing direct Java™ method invocation on service instances. Since our attempt at agnosticism was defeated anyway, this is a design decision that we could reconsider in the future. However, we believe there is still value in staying out of the web services runtime’s internals, and the performance gain may be negligible as many servers already optimize local SOAP calls (see Section 4.3).

Getting back to WTP, another issue is that the web service code generation process and its options are tightly tied into a “wizard” style series of dialog boxes. This process is not easily invoked programmatically, and some of its user-selectable
options that we would like to reuse are not (and cannot be) recorded in the WSDL files. As a workaround, and based on sample WTP testing code, we developed our own “headless” wizard driver that can inject additional commands and probes into the hard-coded sequences, allowing us to extend them in a loosely-coupled aspect-like fashion. The downside is that our custom driver code partially replicates a core part of WTP’s guts, making it uncomfortably fragile.

WTP v1.5 brings with it a new Ant task for batch access to the web service code generation process. Unfortunately, we were unable to take advantage of this new feature, as it has no support for injecting commands into the process or capturing intermediate values. Instead, we rehabilitated our custom wizard driver, which as expected was rather thoroughly broken by the transition. With the driver running again, the injected commands proved resilient to changes in the hard-coded processes and required remarkably few alterations.

Finally, lest our criticism of WTP seem too exuberant, we must temper it with the fact that we could not have built our plug-in prototype without the framework’s support. Even with the flaws mentioned above, WTP is a valuable base for further research on web services.

4.3 Performance Analysis

As noted in Section 3.3, the potential performance degradations occasioned by a long chain of adapters may be a cause for concern. In this section, we look at two performance metrics that are affected by our technique: the latency and the scalability of the application. Latency is the time elapsed at the client side from when a web service method is invoked until a response is returned. Scalability is the capacity of the application to accommodate an increasing number of users without a degradation of the performance metrics.

In most scenarios, the chain of adapters will run on the same server and most probably within a process. Therefore, the overhead introduced in that case is low, and latency and scalability will not be affected much. When the adapters are deployed on different servers or run in different processes, the overhead can be higher, and ways to understand and mitigate the effect are required. While both latency and scalability depend on the particular implementation, the middleware, the message size and its complexity, the transport protocol, etc., there are general ways of mitigating the possible performance penalties.

Latency. Figure 12 shows the delays that make up the latency in an adapter invocation, in the general case of using SOAP over a network as the message-passing mechanism. The delays come from several sources besides the adapters.

Transport protocol. The role of the transport protocol is to deliver the SOAP messages across the web. The use of HTTP or SMTP as main transport protocols by SOAP implementation has many advantages, but it can also introduce unacceptable delays. It is important to know the delay introduced by the transport protocol.

SOAP runtime. This includes the creation or extraction of the SOAP envelope by the SOAP client or RPC Router, the conversion of the data types from language-specific formats to XML and vice versa, and the location and invocation of the service. The XML parser and its performance characteristics influence the latency to a high degree. The parsing of an XML tree to get the envelope, the body, and the header of the SOAP message is done in several passes, and it is likely to be time-consuming. Total parsing versus partial parsing, and validating versus non-validating parsing are factors that can profoundly affect the latency and scalability.

Figure 12. SOAP latency

---

3 Unfortunately, some commands must still run on the user interface thread even though we suppress all UI interactions.
layers will be in the order of milliseconds to tens of milliseconds [8, 17]. That might or might not be noticeable by the application, which, in web services environments, has a magnitude in the order of hundreds to thousands of milliseconds. However, the final decision about the number of versions supported at any given time should be taken based on measurements. Most of the web application servers support the Java Management Extension (JMX) [26], which can be used to measure the CPU demand of any web service.

Scalability. Figure 13 shows the response time seen by the clients of several versions of an application, under different deployments and under different request rates. The application is a 3-tier application that consists of the client, a web server that runs the web service, and the data object. Each tier resides on a separate server. The end-to-end response time of version v1, at a request rate of 0.001 requests/second, is 29 ms (which is a very fast response time, for a very simple application). After the deployment of three versions, with all the adapters in one process, the response time (“In Process” curve) remains very close to the initial response time (v1 line). If the adapters of the versions v2 and v3 are not deployed and optimized within the same process, then the end-to-end response time of the application can be affected, especially at higher request rates. Assuming a latency of 8 ms per adapter, the lines v2 and v3 show the response time seen by the user of older versions when the adapters are deployed in separate processes.

A good practice in deciding the deployment strategy and the number of versions supported is to do a performance engineering study, including measurements, modeling, and predictions in the design phase. Tools that support performance engineering studies [1] can help the decision process.

5. Related Work

Vinoski [28] and Stuckenholz [26] provide high-level overviews of the state of the art in middleware versioning. In this section, we will focus on efforts that are either specifically targeted at web services, or that bear a strong technical resemblance to our proposal.

Within the realm of web services, Ponnekanti and Fox’s work [25, 24] is the closest to ours, proposing to chain interface adapters to achieve compatibility. However, their ultimate aim is to use third-party adapters to match clients with independently developed web services rather than to include the development of such adapters in the web service evolutionary cycle.

Keeping web services backwards-compatible as they evolve has received a fair amount of attention from both researchers and practitioners. The prevailing theme is that while the problem has a long history, the web services standard specifications do not touch on it, and there is no widespread agreement on best practices for handling the issue.

Brown and Ellis [4] advocate having one service support multiple interfaces (cf. Figure 5 but without the inheritance) and advertising the fact through UDDI. Irani [13] covers the subject at a high level, and seems to advocate running multiple versions in parallel (cf. Figure 3) at a single endpoint, with a broker in the server engine dispatching calls appropriately. Kalali et al. [15] assume that clients can adapt automatically to changing interfaces if they are but notified that they have indeed changed.

More recently, Butek [6] codifies the kinds of minor changes that can be made to a WSDL file without disrupting existing clients. The discussion is notable for the tricky details of what is and

---

4 The measurements were taken on an IBM PC with Intel® Pentium® 2.0 GHz and 1 GB of memory running Windows® 2000. The web server and application server are the IBM WebSphere® Application Server 6.0 and the database is the IBM DB2® Universal Database™ (DB2 UDB) Version 8.2.

---

Figure 13. Scalability
is not allowed, especially in terms of schema changes. To avoid this complexity, the author suggests adding generic string-typed placeholder members to data structures that are expected to evolve. Overall, this “minor changes” approach trades off backwards compatibility and code and data sharing against tangled versions, severely constrained evolution, and a visible yet complex mechanism. For major changes, the author offers nothing beyond the usual isolated versions pattern (cf. Figure 3).

Endrei et al. [9] pick up where Butek left off, and delve further into dealing with major changes to web services. Taking a wider, business-level view, they recommend keeping no more than two concurrent versions of a service running, and envisage a transition time of three months. They also advocate toggling the versions between two service URLs, a practice that contradicts established principles of web architecture [14]. Together with their other statements, this recommendation implies that they are assuming a deployment environment much more controlled than the Internet at large. Technically speaking, they propose a few ways of modifying interfaces (by operation name, by service name, or by context root), and seem to assume that a new interface will require a new isolated implementation. The second section of the article presents a worked example that uses both the isolated versions (cf. Figure 3) and confounded versions (cf. Figure 5) approaches simultaneously, and amply demonstrates the dire need for automating the versioning process.

Another promising avenue of approach is to look more generally at the extensibility of XML languages. Most of the work is clustered around the W3C, with unfinished proposals that range from XML Schema extensibility details [19, 21, 22] to general versioning principles [23] such as “must understand” and “must ignore” rules. Wilde [29] has looked at applying some of these ideas to web services, along with additional declarative semantics to describe extensions to a service’s vocabulary. While some of these ideas look promising and may yet come to fruition, they are not yet distilled enough to be employed by web services developers without further research.

Finally, the present work was inspired by (and its name derived from) the Adapter and Chain of Responsibility design patterns from Gamma et al. [10]. We later discovered that a design technique essentially identical to Chain of Adapters had been suggested by Hallberg [12] for Haskell modules under the name “Eternal Compatibility in Theory”; we do not know whether his proposal was adopted by that community. Hallberg hints that the idea may have been floated much earlier by Stroustrup, and we also have anecdotal reports of the technique being used informally in other systems. If so, it may well be a design pattern just waiting to be discovered.

6. Conclusions

In this paper, we laid out our requirements for a solution to the web services version management problem, and illustrated a number of unsatisfactory yet popular approaches. We then presented our own solution called Chain of Adapters, a simple design technique that can be applied by the developer to achieve backwards compatibility. Our technique provides a good trade-off between satisfying the various requirements, with particular strength in the area of version untangling.

The Chain of Adapters can prove useful in self-configuration scenarios. By decomposing a long update/roll-back transaction into a sequence of independent smaller transactions, the response time is affected to a smaller degree and the end user will not notice a discontinuity in service.

Though we implemented an Eclipse plug-in that helps apply this technique to WSDL/SOAP web services and tested it on a few small applications, it is not clear whether the design would scale to large web services. Further evaluation along these lines is needed, as well as further research into independent re-inventions of this design technique in hope of an eventual promotion to a full-fledged design pattern. The effectiveness of this approach in self-configuration scenarios is also subject to further work.

We are also considering refining the plug-in to support more advanced version-related functions. Among others, we are looking at freezing test suites along with the web service interface to encourage regression testing and help verify semantic backwards compatibility. We also intend to investigate options for integrating interface freezing with version control systems, and automatically notifying clients of new versions via an Atom feed [20]. These improvements would
make the plug-in more practical, helping to spur adoption of our technique.

Acknowledgements
This research was supported by the IBM Centre for Advanced Studies (CAS), Toronto, the IBM Corporation, and the National Sciences and Engineering Research Council (NSERC) of Canada.

About the Authors
Piotr Kaminski is a PhD student at the University of Victoria, where he works with Dr. Müller’s research group on software engineering of autonomic systems. His research interests include software documentation, software development processes, reverse engineering, programming languages, and security. He has previously done research on knowledge representation and data integration for his Master’s thesis, and is currently working on autonomic design scribes, a new approach to recording the design of evolving software systems.

Dr. Marin Litoiu is a member of the Centre for Advanced Studies at the IBM Toronto Laboratory where he initiates and manages joint research projects between IBM and universities across the globe in the area of application development tools. Prior to joining IBM in 1997, he was a faculty member with the Department of Computers and Control Systems at the University Politehnica of Bucharest and held research visiting positions with Politechnic of Turin, Italy (1994 and 1995), and Politechnic University of Catalunia (Spain), and the European Center for Parallelism (1995). Dr. Litoiu’s other research interests include distributed objects; high-performance software design; performance modeling, performance evaluation, and capacity planning for distributed and real-time systems.

Dr. Hausi Müller is a Professor in the Department of Computer Science and Director of Software Engineering Programs at the University of Victoria, British Columbia, Canada. He is a Visiting Scientist with the Centre for Advanced Studies at the IBM Toronto Laboratory. He is a principal investigator and Chair of the Technical Steering Committee of CSER, a Canadian Consortium for Software Engineering Research. Together with his research group and in collaboration with IBM, he investigates methods, models, architectures, and techniques for autonomic computing applications. He also concentrates on building Adoption-Centric Software Engineering (ACSE) tools and on migrating legacy software to autonomic and network-centric platforms. Dr. Müller’s research interests include software engineering, software evolution, autonomic computing, adoption-centric software engineering, software architecture, software reverse engineering, software reengineering, program understanding, visualization, and software engineering tool evaluation.

References
[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.
Trademarks
IBM, DB2, DB2 Universal Database, and WebSphere are trademarks or registered trademarks of International Business
Machines Corporation in the United States, other countries, or both.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or
both.
Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.
Intel and Pentium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other coun-
tries.
Other company, product, or service names may be trademarks or service marks of others.