A Design for Adaptive Web Service Evolution

Piotr Kaminski
University of Victoria
Dept. of Computer Science
pkaminsk@cs.uvic.ca

Hausi Müller
University of Victoria
Dept. of Computer Science
hausi@cs.uvic.ca

Marin Litoiu
IBM Canada
IBM Toronto Laboratory CAS
marin@ca.ibm.com

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. 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.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *Patterns.*

General Terms
Design

Keywords
Web services; software evolution; backwards compatibility; design patterns.

1. INTRODUCTION
Version management of deployed software has always been a tricky business. In this age of foreshortened 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 it is simple enough to be applied manually, we also describe a prototype tool we have built to automate some of its more repetitive aspects. It is well suited to deployment in self-managing systems since it affords the manager a larger number of safe configuration options.

Section 4 discusses related work, and Section 5 concludes with a summary and future research directions.

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 Service Definition Language, WSDL [4]) and concomitant datatype 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 [6]). 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?

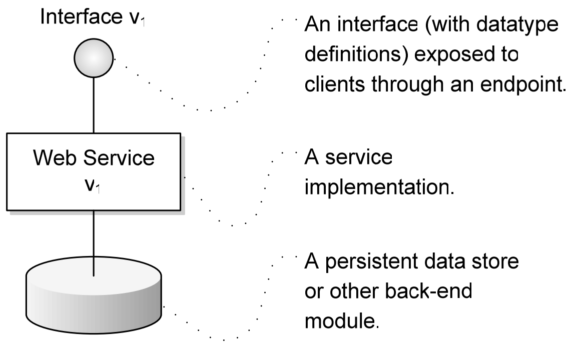
Interface v1
An interface (with datatype definitions) exposed to clients through an endpoint.
Web Service v1
A service implementation.
A persistent data store or other back-end module.

Figure 1. Single version of a simple web service

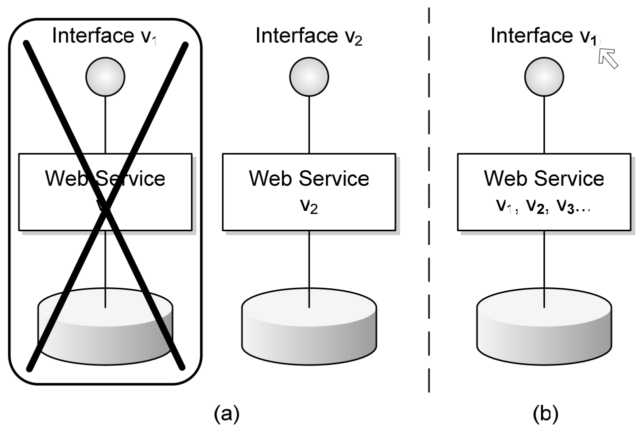

Figure 2. Trivial approaches to backwards compatibility: (a) only support the latest version, or (b) freeze the first published interface forever

2.1 Backwards Compatibility

The fundamental requirement for evolution in cases where the service developer has not 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 most recent release of the web service—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 v_2 might find itself faced with a v_1 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.

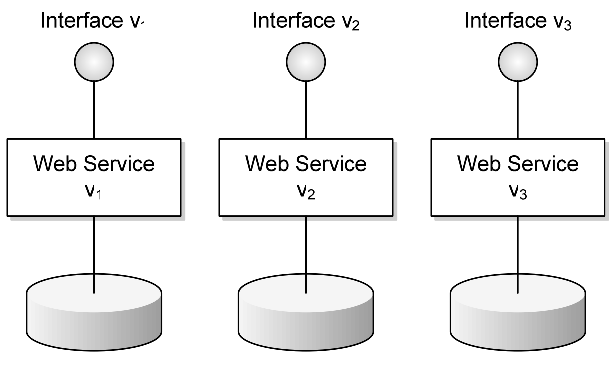

Figure 3. Multiple isolated versions of a web service

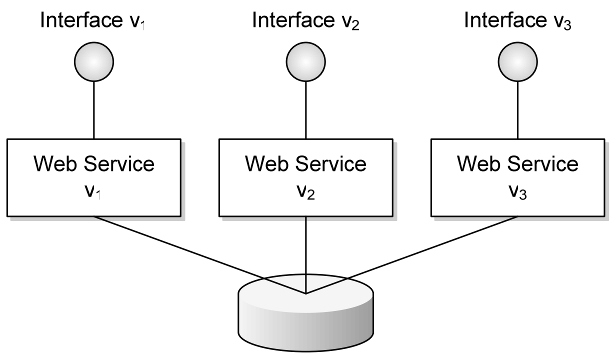

Figure 4. Multiple versions of a web service sharing a single database

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 since—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 typically expressed within the bounds of an application, we hold that this tenet 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

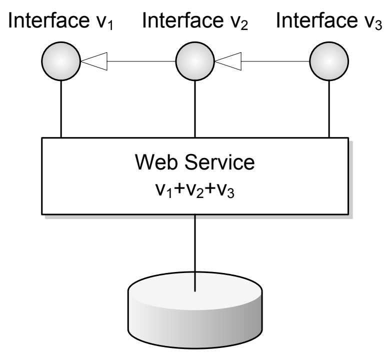

Figure 5. Incrementally extended interface with a single tangled implementation

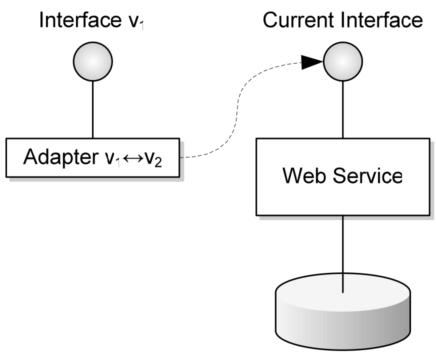

Figure 6. Chain of Adapters structure after the first version is published

service are withdrawn, reducing its complexity and avoiding the Lava Flow anti-pattern [3]. 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 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 the mechanism by which backwards compatible evolution is achieved to the service developers. 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 framework should be kept 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 service 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 [13].

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:

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 v_1 interface.
2. Create an implementation of the v_1 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 v_1 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]

Once v_1 is deployed and in use and development of the service turns towards a new version v_2, the $v_1 \leftrightarrow v_2$ adapter comes into its own. Whenever the web service is modified, a compensating modification is made in the adapter to maintain the contract of the v_1 interface. For example:

[1] Why not deploy a snapshot of the web service just before creating the v_1 interface and $v_1 \leftrightarrow v_2$ adapter? While these components seem to serve no useful function in the deployed version, publishing the current interface directly as v_1 would force it to change namespaces with each version, inconveniencing the service's developers.

- 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 v_1 contract as if though 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 v_1 and (the upcoming) v_2, expressed as compensating code fragments. When v_2 is ready for release, the developer must:

1. Duplicate the current interface into a separate namespace; the copy will be v_2 of the interface.
2. Create an adapter for the v_2 interface that delegates to the current endpoint and interface.
3. Retarget the $v_1 \leftrightarrow v_2$ adapter to delegate to the v_2 endpoint.
4. Publish and advertise the v_2 interface endpoint as the stable second version of the web service.

Figure 7 shows the resulting structure, with a new pass-through $v_2 \leftrightarrow v_3$ adapter, and a slightly fatter $v_1 \leftrightarrow v_2$ adapter.

Development of the web service can now continue towards v_3, with compensating code placed into the $v_2 \leftrightarrow v_3$ adapter. The $v_1 \leftrightarrow v_2$ adapter need never be touched again, since all future incompatibilities will be compensated for by the $v_2 \leftrightarrow v_3$ 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.

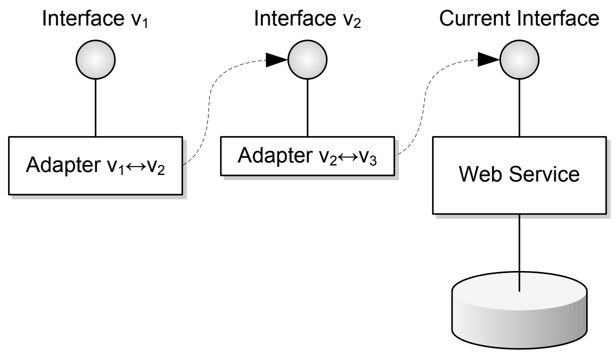
Interface v_1
Interface v_2
Current Interface
Adapter $v_1 \leftrightarrow v_2$
Adapter $v_2 \leftrightarrow v_3$
Web Service

Figure 7. Chain of Adapters structure after the second version is published

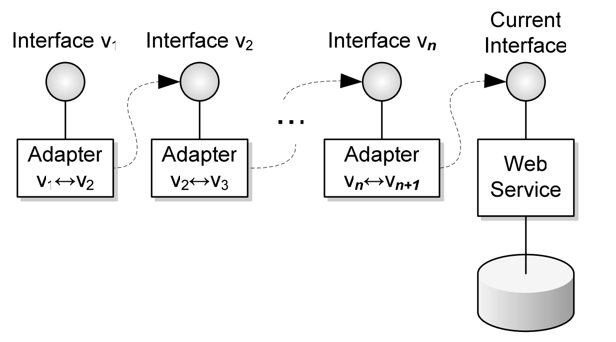
Interface v_1
Interface v_2
Interface v_n
Current Interface
Adapter $v_1 \leftrightarrow v_2$
Adapter $v_2 \leftrightarrow v_3$
...
Adapter $v_n \leftrightarrow v_{n+1}$
Web Service

Figure 8. Chain of Adapters structure after *n* versions have been published

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.

3.2 Tradeoff Evaluation

We now proceed to 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.

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. Although on its face this 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,

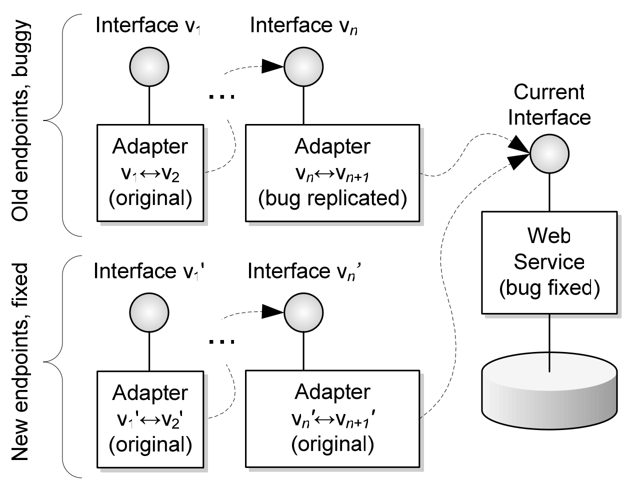

Figure 9. Maintaining both buggy and fixed interfaces to a web service to satisfy all clients

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 or implemented workarounds for the bug. 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 behavior must be added to the $v_n \leftrightarrow v_{n+1}$ 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 to also 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 due to 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 v_1 and v_2 has become unacceptable due to the overhead of forwarding calls through the rest of the chain. We can rewrite the $v_2 \leftrightarrow v_3$ adapter to instead target the newest v_5 interface, folding in the compensations introduced in the $v_3 \leftrightarrow v_4$ and $v_4 \leftrightarrow v_5$ adapters. The new $v_2 \leftrightarrow v_5$ 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 [10] to fulfill this requirement is to (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

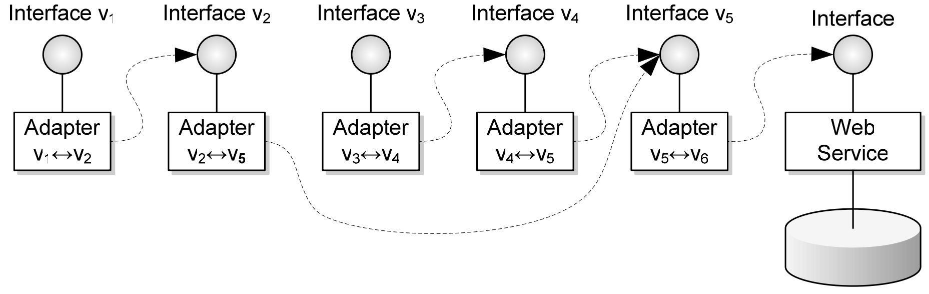

Figure 10. Skipping over links in the chain to reduce forwarding overhead

the boundaries of an ACID transaction [1], 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 service 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.

3.5 Tool Prototype

We have built a prototype plug-in for the Eclipse Web Tools Platform (WTP v0.7) 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 due to

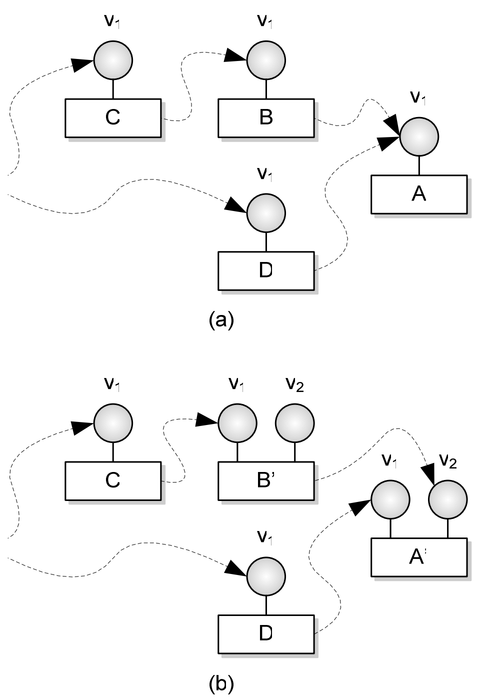

Figure 11. Reconfiguring an application piece-by-piece

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 functionality of the tool was not difficult to develop in principle, though some unexpected corner cases provided a few surprises—much like when attempting to automate apparently simple refactoring techniques. The main challenges in building the plug-in, though, came from integrating it deeply with the largely undocumented WTP v0.7 framework. WTP has since moved on to v1.0 and, while the documentation is now much improved, the internal models have shifted sufficiently to require a nearly complete rewrite of our plug-in.

In summary, the prototype has served us well in proving the fundamental viability of the Chain of Adapters design technique, and we anticipate that it should be fairly easy to implement similar tools for WTP v1.0 or other platforms.

4. RELATED WORK

Vinoski [18] and Stuckenholz [17] provide rather bleak overviews of the state of the art in middleware versioning. Within the realm of web services, Ponnekanti and Fox's work [15, 14] is the closest to ours, proposing to chain interface adapters to achieve compatibility. However, they focus on using third party adapters to match clients with independently developed web services rather than on including the development of such adapters in the web service evolutionary cycle. Brown and Ellis [2], on the other hand, advocate having one service support multiple interfaces (cf. Figure 5 but without the inheritance) and advertising the fact through UDDI. Irani [8] 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. Finally, Kalali et al. [9] assume that clients can adapt automatically to changing interfaces if they are but notified that they have indeed changed.

For web services defined using WSDL and XML Schema, another promising avenue of approach is to look specifically at the extensibility of XML languages. Most of the work is centered around the W3C, with unfinished proposals that range from XML Schema extensibility details [11, 12, 13] to general versioning principles [14] such as "must understand" and "must ignore" rules. Wilde [19] 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 service 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. [5]. We later discovered that a design technique essentially identical to Chain of Adapters had been suggested by Hallberg [7] 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 object-oriented systems. If so, it may well be a design pattern just waiting to be discovered.

5. CONCLUSIONS

In this paper, we laid out our requirements for a solution to the web service version management problem, and illustrated a number of unsatisfactory yet popular approaches. We then presented our own solution called Chain of Adapters, which is a simple design technique that can be applied by the developer to achieve backwards compatibility. Our technique provides a good tradeoff 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 won't 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.

6. ACKNOWLEDGEMENTS

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

7. REFERENCES

[1] Bernstein, P. and Newcomer, E. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[2] Brown, K. and Ellis, M. Best practices for Web services versioning. *IBM developerWorks*, Jan. 30, 2004. http://www-128.ibm.com/developerworks/webservices/library/ws-version/

[3] Brown, W. J., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, NY, 1998.

[4] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S., eds. *Web Services Description Language (WSDL) 1.1*. W3C Note, Mar. 15, 2001.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielson, H. F., eds. *SOAP Version 1.2 Part 1: Messaging Framework*. W3C Recommendation, June 24, 2003.

[7] Hallberg, S. M. Eternal Compatibility in Theory. *The Monad.Reader*, Issue 2, May 2005. http://www.haskell.org/tmrwiki/EternalCompatibilityInTheory

[8] Irani, R. Versioning of Web Services: Solving the Problem of Maintenance. *Web Services Architect*, Aug. 8, 2001. http://www.webservicesarchitect.com/content/articles/irani04.asp

[9] Kalali, B., Alencar, P., and Cowan, D. A Service-Oriented Monitoring Registry. In *Proceedings of the 2003 Conference of the Centre For Advanced Studies on Collaborative Research (CASCON 2003)* (Toronto, Ontario, Canada, Oct. 6-9, 2003). IBM Centre for Advanced Studies Conference. IBM Press, 107-121.

[10] Kramer, J. and Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. In *IEEE Transactions on Software Engineering*, 16, 11 (Nov. 1990), 1293-1306.

[11] Mendelsohn, N. *An Approach for Evolving XML Vocabularies Using XML Schema*. IBM Corporation, June 15, 2004. http://lists.w3.org/Archives/Public/www-tag/2004Aug/att-0010/NRMVersioningProposal.html

[12] Orchard, D. Extensibility, XML Vocabularies, and XML Schema. *O'Reilly xml.com*, Oct. 27, 2004. http://www.xml.com/pub/a/2004/10/27/extend.html

[13] Orchard, D. *Providing Compatible Schema Evolution*. Jan. 19, 2004. http://www.pacificspirit.com/Authoring/Compatibility/ProvidingCompatibleSchemaEvolution.html

[14] Orchard, D. and Walsh, N., eds. *Versioning XML Languages*. Proposed TAG Finding, Nov. 16, 2003. http://www.w3.org/2001/tag/doc/versioning.html

[15] Ponnekanti, S. R. and Fox A. Application-Service Interoperation without Standardized Service Interfaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)* (Dallas-Fort Worth, Texas, United States, March 23-26, 2003). IEEE, 2003, 30-37.

[16] Ponnekanti, S. R. and Fox, A. Interoperability Among Independently Evolving Web Services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware* (Toronto, Ontario, Canada, Oct. 18-22, 2004). ACM International Conference Proceeding Series, vol. 78. Springer-Verlag, New York, NY, 331-351.

[17] Stuckenholz, A. Component Evolution and Versioning State of the Art. *ACM SIGSOFT Software Engineering Notes*, *30*, 1 (Jan. 2005), 1-13.

[18] Vinoski, S. The More Things Change. *IEEE Internet Computing*, Jan/Feb 2004, 87-89.

[19] Wilde, E. Semantically Extensible Schemas for Web Service Evolution. In *Proceedings of the 2004 European Conference on Web Services (ECOWS '04)* (Erfurt, Germany, Sep. 27-30, 2004). Springer-Verlag, Lecture Notes in Computer Science, vol. 3250, 2004, 30-45.