GENERIC WEB SERVICES
Extending Service Scope while Preserving Backwards Compatibility

Vadym Borovskiy, Juergen Mueller, Oleksandr Panchenko and Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering, Germany

Keywords: Generic web service, Generic interface, Web service evolution, Signature relaxation, Service extensibility, Service compatibility.

Abstract: In this article the challenge of extending the functionality of a Web service while guaranteeing backwards compatibility with old client applications is addressed. The authors contribute with a new interface design technique called "Generic Web Services". Using the technique service providers can extend the scope of Web services without breaking compatibility with existing clients. The goal is achieved by applying signature relaxation and interface balancing techniques to a current Web service interface. Furthermore, the authors discuss the advantages and disadvantages of generic Web services and give an example that applies the aforementioned techniques to a Web service from SAP Enterprise Services Workplace.

1 NEED FOR FLEXIBILITY

Modern businesses operate in an ever changing environment. Fast adaptation to the environment is therefore of vital importance. Since most of activities in a company are supported by IT-systems, the prerequisite of being flexible holds true for the systems too. For service-oriented systems the challenge of flexibility can be interpreted in two ways: *(i)* ability to substitute different services if they provide the same functionality to allow provider independence; *(ii)* ability to substitute different versions of the same service to allow the extension of services.

The former interpretation leads to standardization of services. Provider independence is achieved by employing common standards during the development of services (Krafzig et al., 2004). For this reason SOA assumes the usage of agreed-upon open source protocols (e.g. HTTP, SOAP, XML) (Erl, 2005). The standardized communication and data representation protocols significantly lower the effort of switching service providers. However, the effort is still considerable, since different providers most likely use different business protocols (Ryu et al., 2007). This means that the same business entities (e.g. customer, sales order, address) and business processes (e.g. purchasing, invoicing, billing) from an application domain are represented differently. This in turn creates a formidable barrier to switching between service providers. The Figure 1 illustrates the situation of such incompatibility. A client cannot switch its provider by simply redirecting calls to another one because the interfaces of *ChargeCreditCard* Web service offered by others are different.

To overcome this problem service providers must adhere to not only the agreed-upon technical, but also the business protocols. This creates the need of common business process standards. The standardization allows trading partners to conduct electronic commerce in a mutually understood way - both syntactically and semantically (Damodaran, 2004).

The most notable initiatives in the area of business process standardization are shown by RosettaNet[1], OASIS[2] and UN/CEFACT[3]. All three are non-profit consortia trying to establish common e-business standards to facilitate the automation of cross-organizational transactions. By involving key industry players and stakeholders, conducting extensive surveys and verifying all prospect standards in real-life business situations the consortia specify the business processes and associated documents for data exchange.

The second interpretation of the flexibility challenge leads to Web service evolution management. Any successful software product inevitably goes through a series of changes during its lifecycle

[1]www.rosettanet.org
[2]www.oasis-open.org
[3]www.unece.org/cefact/

Borovskiy V., Müller J., Panchenko O. and Zeier A. (2009).
GENERIC WEB SERVICES - Extending Service Scope while Preserving Backwards Compatibility.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 127-132
DOI: 10.5220/0002242001270132
Copyright © SciTePress

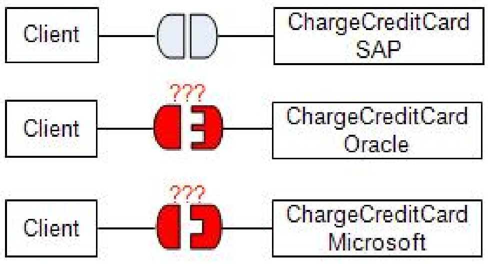

Figure 1: Incompatible service interfaces

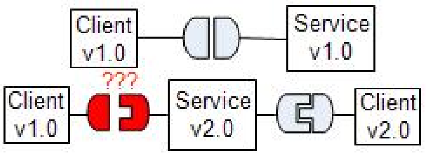

Figure 2: Incompatible service versions

(Brooks, 1987). Service providers must constantly revise their portfolios in order to follow customers' needs and add features to their services. While adding new functionality to services, providers must ensure that none of existing client applications faces incompatibilities caused by functionality updates.

In general, the challenge is to ensure the substitutability of service versions, i.e. correct functioning of all ongoing client applications relying on the old version of a service after the version has been substituted with a new one. Because none of currently available design methodologies guaranties preserving backwards compatibility mismatches between Web services and client applications will likely to happen over the lifecycle of services (see Figure 2). Therefore service providers must develop techniques helping to avoid or at least minimize the problem.

In this article we concentrate on the service evolution management and aim at finding a way of extending a Web service with new functionality while preserving backwards compatibility. We propose a new concept of *generic Web services (GSW)*. In its essence the concept is an interface design technique that allows service providers to extend the functionality of services without breaking backwards compatibility.

The rest of the article is structured as follows. The section 2 defines the concept of generic Web services with the help of the notion of *signature relaxation*, which is introduced in the subsection 2.1. The subsection 2.3 discusses the advantages and disadvantages of GWS. An example of GWS is presented in the subsection 2.2. The subsection 2.4 addresses the challenge of defining appropriate granularity level of an interface. Related work is discussed in the section 3. The section 4 concludes the article and outlines the future directions of the research.

1.1 What Causes Incompatibility

When two parties exchange information they mutually follow rules prescribed by their communication protocol (Hohpe and Woolf, 2004). The more rigid the rules the more efficient the communication can be. Each party assumes that its counterpart follows these rules. However, many assumptions make the communication sensitive to any kind of change. Thus, incompatibility can be understood as a mismatch in the assumptions communicating parties make about each other and the common protocol.

The next question to answer is why incompatibilities occur. In Web service architecture the roots of incompatibilities can be found on the message level. Messaging is the core of communication mechanism in service-oriented systems (Dahan, 2006). To work with a service client applications need only the address of the service and the XML schema of request and response messages. This information forms the description of a service and is included in a *WSDL* file that is published on the Web.

Successful communication assumes that the client and the service exchange with messages of specific content and format. This assumption in turn creates a dependency between the client and the service. If the assumption is broken, the communication is jeopardized and incompatibilities are likely to occur. The reason for this possible failure is that the client will produce messages according to the older version of the communication contract. The operation calls deserialized from these messages will not match the signature of the service's operations.

In this paper we present a new concept of *generic Web services* that brings service providers closer to the solution of incompatibility problem. In its essence the concept is an interface design technique that allows service providers to extend the functionality of Web services while ensuring backwards compatibility.

2 GENERIC WEB SERVICES

Service providers face the challenge of extending services with new features while preserving backwards compatibility. To avoid incompatibility a provider should prevent changes to the interfaces of their services. On the other hand adding new functionality may require changing the interfaces. How to freeze an interface and add features at the same time? To answer this question we introduce the notion of generic Web services.

We define a generic Web service through the notion of signature relaxation as follows: *A generic Web service is a Web service having at least one operation with relaxed signature*. Signature relaxation, in turn, is the reduction of an operation's signature rigidity. In other words, the semantics of an operation with relaxed signature is not statically defined at design time but determined dynamically at runtime through the values of input parameters. The next subsection explains the concept in detail.

2.1 Signature Relaxation

A Web service is a set of operations accessible via specific end point. Each service operation is characterized with a signature. A signature is a collection of an operation's name and two sets of types, instances of which are accepted and returned as input and output parameters respectively(Gamma et al., 1995).

By analogy with (Gamma et al., 1995) the definition of a service's interface can be derived. The set of all signatures defined by a service's operations is known as the interface to the service. A service's interface defines the complete set of request messages that can be sent to the service. To identify and distinguish interfaces developers use types. A type is a name used to denote a particular interface (Gamma et al., 1995).

The signature of an operation is not only a syntactic characteristic but also a semantic one. Semantics of an operation can be defined as the implied meaning of the operation and is used to define its role in a system. The relationship between the semantic and syntactic aspects of signature comes from the way software is developed. To simplify code comprehension and maintenance developers try to come up with signatures (especially with names) that are self-describing and easy to understand. Thus, the semantics of an operation greatly influences the operation's signature.

The signature of an operation imposes restrictions upon the types and values of the operation's parameters. This characteristic we will call the *rigidity* of an operation's signature. Having defined the term rigidity we would like to come back to the term signature relaxation. It can be defined as the loosing of signature rigidity (i.e. the loosing of restrictions that the signature places on the types and values of parameters). Thus, signature relaxation is the extension of parameters' connotation and the set of their possible values.

Signature relaxation can be achieved by introducing a special sort of input parameters that will be used to define the meaning of other parameters. In this case the later parameters become controlled parameters and the former ones - controlling or identity parameters (they identify the semantics of other parameters).

We will call an operation with relaxed signature a generic operation. A generic interface, in turn, is an interface that has at least one generic operation. A generic Web service is a service with at least one generic operation.

The next question to answer is how one can determine if a given operation is generic. A generic operation can be recognized via its identity parameters. Therefore, one should carefully examine the parameters of the operation. If there is at least one parameter that defines or extends the semantics of another, then the operation is generic. If an operation does not have identity parameters it is conceived to be fine-grained.

2.2 Using GWS in Practice

In this section we present an example of how an interface could be relaxed and transformed to a generic one. This is demonstrated with the set of operations that were taken from *Material Management* SAP ES Workplace service interface:

1. Find Material by GTIN[4]
2. Find Material by ID and Description
3. Find Material by Search Text
4. Find Material By Customer Information
5. Find Material by Elements

The operations perform the same kind of functionality. They all search the backend system for materials. Each operation, however, uses different criteria. The operation 5 has the most advanced functionality. It supports the criteria of all other operations. The rest of the operations perform specific search based on criteria that are reflected in the name of a corresponding operation.

One may conclude that the operation 5 is the most generic and the rest operations are fine-grained ones. This, however, is not true, because the operation has no identity parameters. As can be seen all search criteria are hard coded as the operation's input parameters[5].

```
Find   Material by Elements (
       string MaterialID,
       string MaterialDescription,
       string GTIN (StandardID),
       string MerchandiseTypeCode,
       string TypeCode,...)
```

[4]Global Trade Identification Number

[5]Other operations follow the same principle, but have fewer parameters.

Since no operation has identity parameters, the semantics of all parameters is defined at design time. Therefore, none of the operations has relaxed signature.

As will be mentioned in the subsection 2.3 such an interface will have difficulties with extending functionality. To support more search criteria SAP must either add a new fine-grained search operation or add more parameters to the operation 5. In both cases changing the signature of the interface is required. This limitation could be overcome if the interface was generic.

To turn this interface into a generic one we must relax the signature of at least one operation. The best candidate for relaxation is *Find Material by Elements*. The method can be relaxed in three ways.

1. *Find Material by Elements (*
   string attribute,
   string text)

   In this case the search can be performed only on a single criterion.
2. *Find Material by Elements (*
   string[] attribute,
   string[] text)

   This version of the operation searches on all attributes that a client specifies at runtime. All the criteria are used with a default predicate, for example *and*. A client application must guarantee that both input arrays are of equal size, otherwise a runtime error will be reported.
3. *Find Material by Elements (*
   string[] attribute,
   string[] text,
   predicate[] predicate)

   This is the most relaxed version of the suggested ones, because every *text[i]* parameter is controlled by two identity parameters *attribute[i]* and *predicate[i]*. As in the previous case a client application ensures the consistency of the input parameters.

The next question is which of the three versions to include in the interface. Of course, all of them can be included. However, this is not necessary. We can include only the last two versions or even only the last one. The rest of the operations could be represented as overloads of the selected master methods.

There is also one interesting detail to be mentioned. The number of input parameters of *Find Material by Elements* has been reduced. Now a client application only supplies the attributes on which search must be performed, whereas in the non-relaxed version all parameters must be supplied no matter if they are used or not.

2.3 Advantages and Disadvantages of GWS

Over time a service provider needs to add new features to services they offer. Not always do new features fit into the existing interfaces of fine-grained services. In this situation the provider has several options.

Firstly, the provider can introduce another version of the service being changed. In this case old clients will continue working with the previously designed version and will not face incompatibility. Clients willing to utilize new functionality will need to work with the new version of the service. In this scenario the service provider will need to employ a versioning mechanism and maintain all versions of the service. The disadvantage of this approach is the increase in maintenance costs.

Secondly, the provider can change service without maintaining backwards compatibility. In this scenario old clients will need to migrate to the new version of the service. Thus, the clients will incur additional costs.

Thirdly, the provider may introduce an adapter or a converter that will compensate the difference between the versions of the service. This approach is described in detail in (Kaminski et al., 2006) and (Borovskiy et al., 2008).

If a service has generic interface adding new features does not require changing the interface[6]. Adding new features is supported by design via introducing new values of identity parameters. An important detail is that adding a new feature does not require changing the signature of the operation. This is exactly the value of generic interfaces. Service architects can greatly benefit from the fact that a generic operation can add functionality by simply extending the domain[7] of its identity parameters. Thus, generic interfaces allow adding new functionality while keeping the signature stable and preserving backwards compatibility.

The downside of GWS is the ambiguity of controlled parameters. The more vales an identity parameter has, the greater the ambiguity of a parameter which it controls. This can result in less understandable service interface.

2.4 Defining Right Granularity Level

Because of the ambiguity of controlled parameters generic Web services are more difficult to use than

[6]At least that often as in case of a fine-grained service.

[7]The set of all possible values of a parameter is called the domain of the parameter.

fine-grained ones. The more generic an operation is, the harder it is to understand its semantics at design time. Therefore service providers must balance services' granularity and the ease of use. This creates a dilemma: flexibility versus usability. Services must be generic enough to allow adding new features, but not too ambiguous and confusing to service subscribers.

In general, service providers can include to interfaces as many operations as they want with any granularity level each. However, this will create several problems at the later stages of services' lifecycle. Firstly, the interface can grow. There can be several operations that perform the same functionality, but with slight difference. In such situations users can be confused and will not know which operations they should call. Secondly, once an operation has been included to an interface, a service provider cannot change (or even depreciate) it easily, since there can be clients using it.

Thus, it is in the interests of service providers to keep their interfaces as slim as possible. At the same time the interfaces must be convenient to use, meaning that the number of parameters must be reasonable and the purpose of each one must be not difficult to understand.

When there are not many operations (as in the example of the previous subsection) service architects can decide on their own (without any formal method) what granularity level each operation in an interface should have. In case there is a high number of operations a formal approach is needed. Such a technique we will call *interface balancing*, since it balances an interface's granularity level and its usability. For now we have not found a sound formal method of defining the appropriate degree of relaxation of an interface. This will be one of the future directions of the research.

3 RELATED WORK

In (Erlikh, 2000) Erlikh estimated that 90% of software costs are evolution costs. The importance of the evolution requires a systematic approach of managing an evolving software system. This is the task of configuration management discipline (Zeller, 1997).

The most frequently used approach in the area of software evolution is versioning. Versioning is used to distinguish different versions of components and libraries that are simultaneously running at the same machine (Sommerville, 2007). The way how a version is identified and which characteristics are included into the computation of version identifier are defined by a particular versioning model (Conradi and Westfechtel, 1998).

A number of versioning methods has been practically implemented. None of them, however, has solved the challenge of consistent software evolution (Stuckenholz, 2005). Moreover, versioning is not the mechanism of incompatibility resolution and does not facilitate software substitutability. It is rather a way to make software changes detectable from client applications. To figure out if two versions of the same component are substitutable an approach offered in (Lobo et al., 2005) could be used.

The works of Ponnekanti and Fox (Ponnekanti and Fox, 2004), Hohpe and Woolf (Hohpe and Woolf, 2004) and Kaminski, Litoiu and Mueller (Kaminski et al., 2006) address the incompatibility problem. (Kaminski et al., 2006) suggests to pass calls of older clients through a chain of adapters that compensates the difference between the versions of a service in terms of other operations available in the newer version. This is a powerful solution, but it is limited to the service side and might result in a serious performance hit in case of long chain. (Ponnekanti and Fox, 2004) suggests a similar technique that reconciles incompatibility inside a client-side proxy. Instead of a standard proxy a ”smart” proxy that bridges the gap between the older client and the newer version must be used. The approach is limited to the client side and requires changing the older application. (Hohpe and Woolf, 2004) presents message conversion as a pattern of enterprise integration. The work is fairly abstract without any implementation guidance.

From the business protocol standpoint the evolution of a Web service is described in (Ryu et al., 2007). The article suggests an approach to manage the protocol instances running according to the old protocol version. Firstly, a protocol manager selects the active instances that can migrate to the new protocol. This is done by analyzing the protocol itself (static analysis) and each individual instance of the protocol (dynamic analysis). All migrateable instances can be safely switched to the newer protocol version. Secondly, for non-migrateable instances an adapter must be developed. In case the development of an adapter is not feasible an individual temporary protocol must be introduced to the instance to meet new requirements without cancelling the ongoing instance.

4 CONCLUSIONS

Over the lifecycle of a Web service its provider will want to change it in order to keep the one up to date with customers' needs. This creates a challenge of maintaining the integrity of old client applications using the service. In case a change to a service happens the provider must ensure backwards compatibility between the new and the old service versions.

This article contributes with the notion of *generic Web service* defined with the help of terms *signature relaxation* and *identity parameter*. Based on these notions an interface design technique was suggested. Using the technique service provides can design generic Web services which have more stable and extensible APIs. In short the technique includes three main steps: *(i) Signature relaxation* - a service architect finds operations that must be relaxed and relaxes them through the introduction of appropriate identity parameters; *(ii) Interface balancing* - the architect selects the minimal number of operations effectively representing the functionality of the interface; *(ii) Operation overloading* - the service architect can represent excluded operations as overloads of included ones.

The biggest value of generic Web services is that they allow for adding functionality by simply extending the domain of their identity parameters and without changing the signature of operations. The downside of generic Web services is the ambiguity of controlled parameters.

The future research will concentrate on defining a formal method of computing the granularity level of operations. Furthermore, the theoretical results of the research will be validated on a broader set of industrial Web services.

REFERENCES

Borovskiy, V., Zeier, A., Karstens, J., and Roggenkemper, H. U. (2008). Resolving incompatibility during the evolution of web services with message conversion.

Brooks, F. (1987). No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4).

Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282.

Dahan, U. (2006). Autonomous services and enterprise entity aggregation. *The Architecture Journal*, (8):10–15.

Damodaran, S. (2004). B2b integration over the internet with xml - rosettanet successes and challenges. In *Proceedings of the 13th international World Wide Web conference*, pages 188 – 195.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Kaminski, P., Litoiu, M., and Mueller, H. (2006). A design technique for evolving web services. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*.

Krafzig, D., Banke, K., and Slama, D. (2004). *Enterprise SOA: Service Oriented Architecture Best Practices*. Prentice Hall.

Lobo, A. E., Guerra, P., Filho, F. C., and Rubira, C. (2005). A systematic approach for the evolution of reusable software components. In *ECOOP'2005 Workshop on Architecture-Centric Evolution (Glasgow, UK, 25-29th July 2005)*.

Ponnekanti, S. R. and Fox, A. (2004). Interoperability among independently evolving web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, volume 78, pages 331 – 351.

Ryu, S. H., Saint-Paul, R., Benatallah, B., and Casati, F. (2007). A framework for managing the evolution of business protocols in web services. In *Proceedings of the 4th Asia-Pacific Conference on Comceptual Modelling*, volume 67, pages 49 – 59.

Sommerville, I. (2007). *Software Engineering*. Addison-Wesley, 8 edition.

Stuckenholz, A. (2005). Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes*, 30(1).

Zeller, A. (1997). *Configuration Management with Version Sets - a Unified Software Versioning Model and its Applications*. PhD thesis, Technische Universitaet Braunschweig.