/THEORY/IN/PRACTICE

SOA in Practice

The Art of Distributed System Design

Nicolai M. Josuttis

O'REILLY®

SOA in Practice
by Nicloai M. Josuttis

Copyright © 2007 Nicolai M. Josuttis. All rights reserved. Printed in the United States of America.

Published by O'Reilly Media, Inc. 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or c*orporate@oreilly.com*.

Editor: Simon St.Laurent
Production Editor: Sumita Mukherji
Copyeditor: Rachel Head
Proofreader: Nancy Reinhardt
Indexer: Tolman Creek Design
Cover Designer: Mike Kohnke
Interior Designer: Marcia Friedman
Illustrator: Jessamyn Read

Printing History:

August 2007: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *SOA in Practice* and related trade dress are trademarks of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

Java™ is a trademark of Sun Microsystems, Inc. .NET is a registered trademark of Microsoft Corporation.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52955-4
ISBN-13: 978-0-596-52955-0
[M]

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.

CHAPTER TWELVE

Versioning

WHEN YOU ESTABLISH SOA, YOU CAN'T EXPECT TO BE ABLE TO DO AND ANTICIPATE EVERYTHING AT once. Large distributed systems are never static. Requirements evolve, and new ones appear. Also, as you develop and implement services you constantly learn and improve, and you may want to apply your new knowledge to existing services. Therefore, you need the ability to update and grow.

As discussed in the previous chapter, SOA is a concept that has to deal with existing running solutions and services as well as new requirements that lead to the development of new solutions and services. Therefore, you need a mixture of maintenance and innovation.

This chapter will discuss how to deal with changes to services.

12.1 Versioning Requirements

There are two common reasons why services (and interfaces in general) have to be modified:

- When implementing new business processes you usually become aware of aspects you didn't know about or consider at design time, resulting in services (implementations as well as interfaces) needing to be modified.
- New or modified requirements may necessitate modifications of existing services.

Some people argue that with good design, the first case should not happen. One of my customers even deliberately introduced a process that made it difficult to make modifications once a service interface was specified, in an effort to force stability in the designs.

However, in practice, interfaces are no more stable than any other code. One reason is that people make mistakes (which you should be able to correct when you recognize them). In addition, these days we don't often have enough time for good designs. Finally, reality is always different from what we anticipate.

When you implement business processes, you learn. According to your new insights, you may decide to modify even interfaces. Otherwise, inappropriate or bad designs would remain in your system for years.

Now, given that services change during their development and when they are used in running systems, the question is how to deal with these changes. SOA is a concept for large distributed systems, and you can't require that all systems involved make corresponding modifications at the same time. You need migrations. In addition, you need processes that enable a testing department to test an older revision of a service while a new revision or a bug fix is already under development. You also need to allow service consumers to continue to use an older version of a service even after a new version becomes available.

In principle, there are two different requirements regarding the versioning of services:

- It must be possible to have multiple *versions* of a service running in the same runtime environment. For example, it must be possible for there to exist two versions of a service that returns customer data.
- It must be possible to have multiple *revisions* of a service under development. However, these different revisions don't have to be available in the same runtime environment.

The former is a business-driven requirement, because at runtime you will see that different versions of services are available. I will discuss this first, in Section 12.2. The latter requirement usually leads to the topic of configuration management. I will discuss this later, in Section 12.4.

12.2 Domain-Driven Versioning

Domain-driven versioning enables different versions of the same service to be run at the same time in the same runtime environment. That is, two consumers might call the same service using different interfaces. While one consumer uses a newer version of the service, another consumer might use an older version.

These are plenty of articles about how to deal with this issue, but these articles usually assume additional requirements that have to do with automatic updates of services. However, policies for automatic updates lead to additional complications and requirements.

Instead, I'd like to present a trivial versioning approach that has proven to be appropriate in all projects I have seen so far. Because it keeps things simple, there have to be good reasons not to follow this approach. So, we'll look at this option first, before briefly exploring some alternatives.

12.2.1 Trivial Domain-Driven Versioning

My trivial policy for domain-driven versioning is simply *not* to provide any technical support for versioning. That is, treat every modification of an existing service as (technically) a new service.

If you need to modify a service that returns customer data (say, `GetCustomerData()`), you simply introduce a new service that incorporates the modification. Of course, you should make it obvious that this new service is a successor of the other service, which you can easily do by naming the new service accordingly (e.g., `GetCustomerData_2()`). To avoid having a special rule for the first version of a service, you might choose to name the first service `GetCustomerData_1()` (see Figure 12-1). With this convention your service names will always have two parts: one that indicates what the service does and one that specifies the version number.

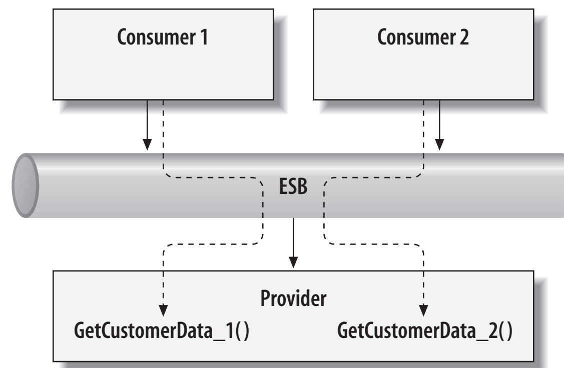

FIGURE 12-1. Two consumers calling two different service versions with trivial versioning

Of course, the costs of bringing a new service into existence are usually higher than those of modifying a service. For this reason, in practice I recommend a slightly relaxed rule for domain-driven versioning: from the moment a service is used in production, any modification that is not simply a bug fix should result in a new service. This rule has two important consequences:

- During development time any desired modifications can be made, and these modifications are not considered to result in new versions. Note, however, that from the moment a service is first used (e.g., in integration tests), the service provider should inform existing service consumers about any modifications and discuss them with the consumers (as should always be the case when a contract changes).

- At runtime, it is possible to fix bugs without creating new versions of the service (which would be more expensive). Of course, this implies that the service interface doesn't change. In practice this might lead to the problem that bug fixes sometimes turn out to be modifications, so that semantically new versions would have been more appropriate. But the price of creating new services for each bug fix is usually higher.

Note that I have not said anything about the question of whether or not the modifications are backward compatible. In fact, this policy applies even for backward-compatible modifications.

Why, you might wonder, shouldn't the provider be able to make changes to existing services without introducing new versions if these changes will not impact the existing consumers? There are two reasons. First, backward-compatible modifications often turn out not to be as "compatible" as expected. A typical example is when an additional attribute leads to longer running times, resulting in the (formal or informal) SLAs of the service being broken (see Section 13.4 for an example). In addition, any modification involves a risk. Introducing a modification as a new service gives you the chance to observe its runtime behavior for only a single consumer; all the other consumers can then switch to the new service when it is clear that everything works fine.

The second reason for not making even backward-compatible changes without introducing new versions is that these changes often result in modified data types. This issue is discussed later, in Section 12.3.

So, to sum it all up, after a service is brought into production, bug fixes are OK, backward-compatible modifications *should* result in new service versions, and incompatible changes *must* result in new versions. Note, however, that there is a gray area: a bug fix might actually be a modification (even a nonbackward-compatible modification). This versioning concept is a policy, not necessarily a law. Its purpose is to give service participants common guidelines so that the normal behavior is clear and intuitive. If in doubt, talk to each other.

The problem with this approach is obvious: there is a danger that you will end up with too many versions. Especially for services that return data (say, customer data), there is a risk that each new consumer will need additional data, so the original service will grow and grow. I've seen this happen. One company I know of that has hundreds of services in production (each version counts) started with the rule of having not more than three versions of the same service in production. In practice, they often wound up having as many as five (and more) versions running simultaneously.

To avoid having too many versions (which hinders code maintenance), you will occasionally need to take services out of production. As discussed in Section 11.2.2, this is usually done in two steps:

1. Declare an old service version as deprecated.
2. Withdraw the deprecated service.

Also bear in mind that in mission-critical systems (which SOA environments typically are), you can remove services only when they are no longer being used. For this reason, you need monitoring for this policy (see Section 5.4.7).

There is always a danger that some consumers will continue to use old, deprecated service versions. Remember that distributed systems have different owners, and that for consumers changing service versions incurs costs but may not result in any direct benefits. However, if old versions are not phased out, the entropy of the system as a whole will get worse and worse. Thus, you might need to escalate things organizationally, or the provider might need to offer consumers some incentive to switch to a new service version.

See Chapter 11 for more details about this topic from a service lifecycle point of view.

12.2.2 Nontrivial Domain-Driven Versioning

We've looked at the trivial versioning policy. What might a nontrivial policy look like?

There are a lot of possible answers to this question. In principle, the options include:

- Provide a mechanism that ensures services are forward compatible. That is, if your infrastructure allows you to provide a hook for future extensions, you can add extensions as needed and mark them as being optional. Existing service consumers will still be able to use the interface as they always have, and new consumers will be able to take advantage of the newer features.
- Introduce techniques that allow you to extend services in such a way that you can specify what should happen with consumers using the older interface. For example, your infrastructure might provide a mechanism to add new attributes, including specifying default values in case these attributes are not present.
- Provide a method of indirection so that different implementations are provided for different consumers. For example, a service broker might be able to determine which version of a service is provided for which consumer.

How these approaches are realized is a different question. As an example, you can use Web Services to deal with using different namespaces for different versions and/or use a UDDI registry (introduced in Chapter 16) as a broker that routes service requests differently. See, for example, [BrownEllis04] for more details about this approach.

12.3 Versioning of Data Types

When different versions of services exist, different versions of data types are also involved (at least, if the services use structured data types). Dealing with this issue can become a lot more difficult than just dealing with different service versions.

Say, for example, that a new attribute for a Post Office box is added to a service that returns data including an address. That is, the existing address type, which has the following attributes:

```
String street
String zipcode
String city
```

gets this new attribute:

```
String postbox
```

Because older versions of the service use the older address type and newer versions of the service use the newer address type, two different address types are in use in the same runtime environment (see Figure 12-2).

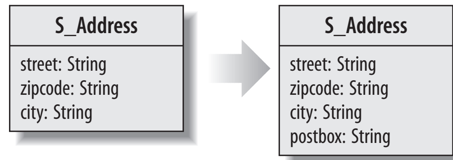

FIGURE 12-2. Two different versions of an address type

The question is how to deal with this fact. In principle, there are three possible options:

- Use different types for typed interfaces.
- Use the same types for typed interfaces.
- Use generic code so that type differences don't matter.

I will discuss these options now in detail.

12.3.1 Using Different Types for Different Versions of a Data Type

When using different types for different versions of a type, you might be tempted to simply apply the same rule I suggested in Section 12.2.1 for naming and distinguishing between the types. However, this situation is more complex, for a few reasons.

The first reason is that modified types lead to other modified types. That is, if a type is used by another type, the other type also changes. For example, if an address type is used by a type for lists of addresses, which is part of a type for customer data, which is used by a type for lists of customer data, a change of the inner address type changes all the other types (see Figure 12-3). As a consequence, you get many types.

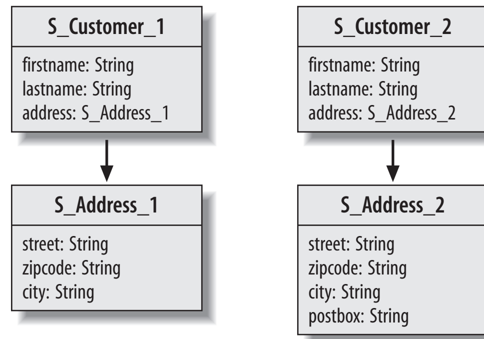

FIGURE 12-3. Modifying a versioned address type that is used by other types

The other reason for the increased complexity is that, in effect, services have different data types for the same kinds of information. In a programming language with type binding, the result of these type differences is that it is not possible to compare, copy, or assign the types as a whole; you have to program utility functions that compare, copy, or assign the different types element by element (ignoring or providing default values for attributes that are not in both versions of the data type). This has consequences for service providers and service consumers:

- As a service provider, you have to use different types for the same kind of information. You might do this by copying and pasting the code that implements the functionality for different types, by implementing functionality for one type and mapping the data to other types, or by using generic code (templates).
- As a service consumer, it might happen that you need a new service that uses the new version of a data type as well as an older service that still uses the old version of the type. Because these types differ, you will have to map data to deal with the same kind of information in both services.

The second point in particular has nasty consequences, because service consumers sooner or later will probably have to deal with different versions of the same type. To help you understand this problem, consider the following example. Say you have two different versions of a service that returns customer data. As discussed earlier, the address types are different: one consumer uses `S_Address_1` and the other uses `S_Address_2`. Now suppose you have another service, called `GetInvoiceData_1( )`; this service returns invoice data that includes customer data, and it also uses the newer address data type. As a result you get the situation illustrated in Figure 12-4. Note that all these services (including all versions) are used by some consumer(s).

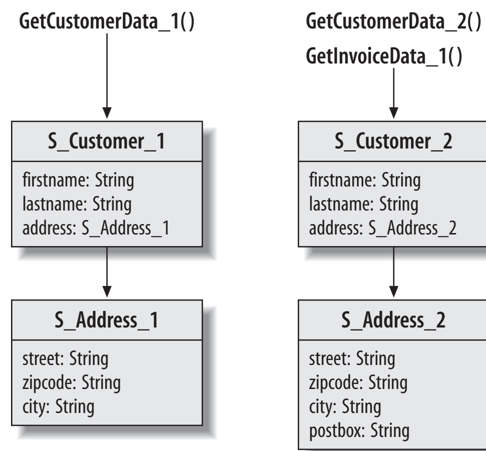

FIGURE 12-4. Different data types for different services

Now say that later an additional requirement is introduced for a consumer that specifies that the service returning invoice data should also return a tax number as part of the customer data. So, you introduce a new service called `GetInvoiceData_2()` that returns a new data type, `S_Customer_3`. As a result, you get the situation illustrated in Figure 12-5.

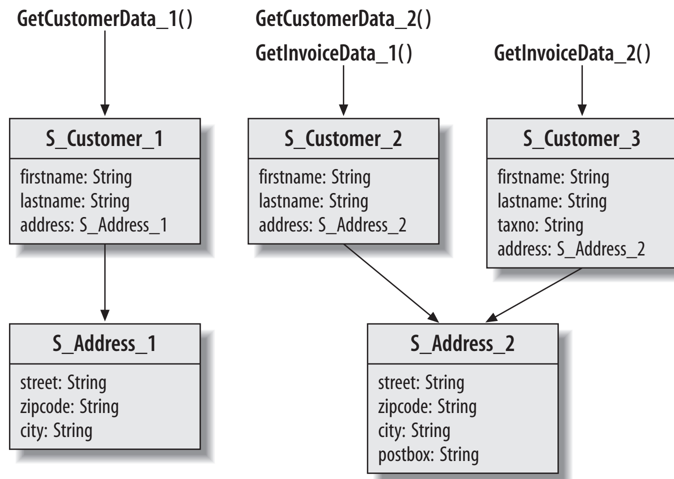

FIGURE 12-5. New attributes result in even more different data types for different services

Now, if a new consumer wants to use both services (`GetCustomerData()` and `GetInvoiceData()`), it will find itself in the strange position that two different data types are involved when it uses the latest versions of the two services. Of course, you can avoid this situation by always upgrading all services that use a certain data type when the data type changes for one of the services where it is used, but this leads to a lot of additional service versions.

Note that what is described here is a conceptual problem arising from the fact that you have to support different versions of APIs with structured data types. There are alternatives, some of which will be discussed in the following sections: you might not use structured types, you might not use typed APIs, or you might try to share different versions of a type inside a process. All of these alternatives have their own drawbacks, though.

12.3.2 Using the Same Type for Different Versions of a Data Type

When the same type is used for different versions of a type, this type must contain all the attributes of all the versions of the data type. All services will use the same type, but they will use only those attributes that are specified for them. This policy introduces three problems:

- You have to document which attributes are valid for which service versions. For complex data types or types used in different services, this can become very complicated.
- The data types of older services change over time. This means that these different versions are not binary compatible. As a result, you have to make sure that all libraries of a process are compiled with the same version of a data type. Thus, if a data type changes

for a new service and you need this new service, you have to recompile all existing code for all other code that used this data type. If this fails, very nasty runtime misbehavior will occur.

- If you validate input data according to your point of view, you have to make sure that additional attributes do not make your input invalid.

To illustrate, consider again the situation depicted in Figure 12-5. With this approach, the different service versions would all use the same data types, as shown in Figure 12-6.

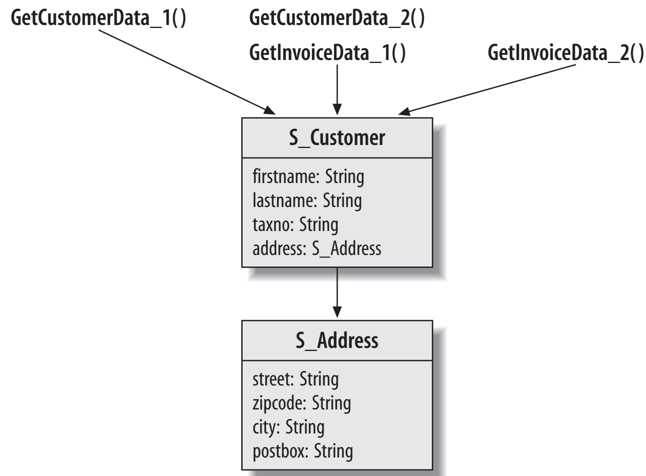

FIGURE 12-6. Different services with different versions sharing the same data types

However, for `GetCustomerData_1( )` the `taxno` and `postbox` attributes are ignored, and for `GetCustomerData_2( )` and `GetInvoiceData_1( )`, the `taxno` attribute is ignored. This, at least, must be documented (which is not easy, because when writing the documentation for `GetCustomerData_1( )` you don't know about future attributes, and when modifying `S_Address`, it might not be obvious where it is used).

In addition, if a process uses a library calling `GetInvoiceData_1( )` that is compiled for an old version of the data type and later adds a new library calling `GetInvoiceData_2( )` that is compiled for the new version of the data type, the problem of binary incompatibility arises. As a result, you might get bugs due to different interpretations of the same memory, which are one of the worst possible kinds of bugs to deal with.

Only if you can ensure that each consumer uses a consistent set of libraries (for example, by providing different library names for each consistent release of all service code) might this policy be appropriate.

12.3.3 Using Generic Data Types

A third option for dealing with different versions of data types is to use only generic code that is able to deal with any data type. In this case, the binary format of the approach with the same types doesn't matter. As a consequence, data types don't matter at compile time; all the processing happens at runtime.

To see the difference, consider what a static API to access the street of the address of a customer returned by `GetCustomerData_1( )` might look like:

```
S_Customer_1 custData;
S_Address_1 address;
String street;
input.setCustomerID(id);
custData = serviceAPI.getCustomerData_1(input);
address  = custData.getAddress( );
street   = address.getStreet( );
```

The same task implemented using a generic API might look as follows:

```
Data custData, address;
String street;
input.setValue("customerID",id);
custData = serviceAPI.getCustomerData_1(input);
address  = custData.getValue("address");
street   = address.getValueAsString("street");
```

Alternatively, the last two lines might be combined into one line:

```
street   = custData.getValueAsString("address.street");
```

This is definitely an option to consider. However, this currently seems to be a very uncommon approach. Especially in the Web Services context, all generators I know of (by default) generate typed APIs, introducing the versioning problems discussed earlier. One reason for this is that without typed APIs you lose the advantage of finding bugs at compile time. Whether or not the path of an attribute is correct is evaluated at runtime. Still, for large systems, this approach might pay off. Maybe it's not so common because typed interfaces seem to be so easy and intuitive (as long as you don't have large applications of the SOA concept). The prototypes probably look fine, and later on it might be too late to change things conceptually.

12.3.4 Summary of Versioning of Data Types

Hopefully, you understand now that versioning of data types is an issue for large distributed systems. I've discussed three options, which all have pros and cons (and, of course, a lot more could be said about all of these options). A fourth option would be to use flat lists of parameters rather than structured data types, but of course this doesn't work when coarse-grained services send around complex data.

As I mentioned previously, the option of using only generic data types is not often used in practice, and it can be hard to provide support for this approach on all platforms of a SOA system. Also, the option of sharing types for different versions is a very dangerous one,

because if your process does not ensure that all your libraries are consistent (which is difficult in distributed systems) it can result in very ugly bugs and undefined runtime behavior. In addition, this option reduces your ability to find bugs at compile time. Thus, the first option I presented is the most commonly used. This approach often results in complaints by ordinary programmers about providing such a silly versioning concept, where it is not even possible for providers to have consistent data types. But as we've seen, this is part of the price of distributed systems with different owners: you can't just switch to new services across the board.

As a service consumer, it is a good idea to make your code somehow independent from the versioning of the services called. You should use your own data types, which are mapped to the data types of the services when they are used. For this reason, service consumers usually should have a thin top layer that maps external data types to internal data types (which might or might not be combined with the mapping layer between service APIs and service protocols, as discussed in Section 5.3.3).

However, be careful, and try to keep things simple. You will discuss this approach when you realize that dealing with different versions is an issue. Don't make things too complex by trying to prepare for things that might happen in the future.

12.4 Configuration-Management-Driven Versioning

As introduced at the beginning of this chapter, by "configuration-management-driven versioning" I mean the requirement of having multiple revisions of a service under development in different runtime environments. This requirement leads to the topic of configuration-management tools such as version-control systems.

When you have a set of files and other artifacts that belong together, it must be possible to give them a common label so that you can deal with them as one group. That is, if you model a service, generate interfaces based on the model, implement services against these interfaces, compile libraries out of these implementations, and deploy the resulting libraries, you have to be able to find all the versions of the different artifacts that belong together.

If all the artifacts are files, this is easy. You can use any version control system that is able to label files and support the management of different versions of a file (including showing differences and merging multiple modifications).

If there are artifacts that are not files, you need related mechanisms. For example, if you have a service repository (see Chapter 17) that is implemented in a database, in the repository interface you need support for configuration management or corresponding organizational rules (such as having different repositories for different configurations).

Again, note that it is usually very important at least to be able to show the differences between two versions of an artifact. This might require you to have special tools or scripts (e.g., special database scripts, stored procedures, or Visual Basic scripts for a tool such as Rational Rose).

12.5 Versioning in Practice

It's important to be aware that requirements are likely to change more often than you expect, and also that difficult modifications (particularly those that are not mission-critical) have a tendency to be put off. This means that you may end up with more versions being rolled out more frequently than you expect, and that it may turn out to be harder to than you expect to phase out old versions. Recall the large SOA system I mentioned earlier, which expected to have a maximum of three versions of each service in the same runtime environment. The system had more than 30 service participants and more than 300 services, and there were regularly up to six different versions of individual services in production.

With these difficulties in mind, you should take the following recommendations into account.

12.5.1 Modifications Should Impact Only the Provider and Consumer(s)

Try to make modifying services as easy as possible. In particular, service modifications should impact only the service provider and the service consumer(s)—no one and nothing else. Note that central teams that design the architecture for distributed systems tend to impose more control than is necessary.

For example, your infrastructure might help you by checking service interfaces during the transfer of data. However, this means that each modification must also be deployed to the infrastructure component that does the runtime checking. This leads to more complicated processes and potential bottlenecks.

Thus, inside the SOA infrastructure should be as generic as possible with respect to domain-specific business functionality. Any specific processing that is influenced by different versions of services should affect only the endpoint of the infrastructure (e.g., libraries and proxies for the provider and consumer(s)).

12.5.2 Call Constraints Should be Considered

One of my customers has introduced a concept known as a "call constraint," which is a parameter that each service has that gives the consumer(s) the opportunity to signal a special context that might have an impact on the service implementation. The problem that prompted this innovation was that while services are intended to be (re)used by multiple consumers, in practice the granularity of services often differs because additional attributes can affect running times (see Chapter 13). In addition, it is not always clear at design time which service context will occur in practice, and when such a context might become critical. That is, you don't know ahead of time which attributes it will be critical to return.

The concept of a "call constraints" argument allows you to delay runtime optimizations until the moment when they become necessary. The provider and the consumer agree upon a special flag that the consumer can send inside the "call constraints" argument to

indicate that there is a need for special optimizations or a special behavior for that specific consumer. The provider can then implement this special behavior for the consumer without changing the service interface or breaking existing behavior.

This is a useful technique, but you should note some things:

- This is not a general mechanism that consumers can use to specify whether or not the provider should return a certain attribute; it is a formatless flag that both participants must agree upon and that is described in the semantic description (contract) of the service.
- Inside the implementation of a service, you usually have to pass this parameter as part of the "calling context."
- This is a practical approach for the dealing with the fact that when a service enters integration testing or production, some things may become evident that were not clear before. Using this flag can be a lot easier than modifying the service. However, don't do it too often—you might consider these flags as workarounds for future modifications (knowing that nothing remains as stable as a workaround).

For more details about call constraints, see Section 13.3.1.

12.6 Summary

- SOA requires a smooth migration strategy for new service versions.
- The best approach is to treat each modification of a service (in production) technically as a new service.
- If you have typed APIs for services, versioning of service types is also recommended (although there are alternatives).
- To avoid an explosion of versions, you have to introduce processes to deprecate and remove old service versions. This is one reason why it's useful to be able to monitor service calls.
- To become independent from versioning aspects of called services, service consumers usually should have a thin layer that maps external data types into internal data types.
- Service modifications should never affect anyone other than the service provider and consumer(s).
- To be "forward compatible," you might provide an attribute for upcoming call constraints. This enables consumers to signal special needs at runtime, without having to modify service signatures. However, be cautious about making everything generic, because this might introduce a lot of complexity and hidden dependencies.