# 12 Versionierung

Große verteilte Systeme unterliegen ständigen Änderungen. Es tauchen neue Anforderungen auf, einzelne Teile müssen ersetzt oder überarbeitet werden, und es kommt vor, dass Lösungen nicht mehr benötigt werden. Dies alles geschieht nur in Teilen des Gesamtsystems. Eine SOA-Landschaft muss also verändert werden können, ohne dass davon mehr Systeme als notwendig betroffen sind.

Konzeptionell muss SOA gleichzeitig Veränderung als auch Stabilität berücksichtigen. Während existierende Services für die dazugehörigen Geschäftsprozesse weiterlaufen müssen, kommen immer wieder neue Services und neue Service-Versionen hinzu. Diese Anforderung kann nur mit einem Migrations- und Versionskonzept umgesetzt werden, auf das in diesem Kapitel näher eingegangen wird.

## 12.1 Anforderungen an Versionierung

Grundsätzlich gibt es zwei unterschiedliche Beweggründe, um Services (und Schnittstellen im Allgemeinen) zu verändern:

1. Es gibt neue oder geänderte Anforderungen für bereits in Betrieb befindliche Services.
2. Bei der Umsetzung einer Anforderung stellt sich heraus, dass das initiale Design eines Service geändert werden muss.

Manche Menschen argumentieren, dass der zweite Grund ein Zeichen für schlechtes Design ist. Einer meiner Kunden hat es deshalb besonders schwierig gemacht, einmal festgelegte Service-Schnittstellen zu ändern. In der Praxis sind Schnittstellen aber nicht wirklich stabiler als jeder andere Code. Ein Grund hierfür ist, dass beim Design Fehler gemacht werden. Diese mögen vermeidbar sein, die Frage ist nur, ob der Aufwand für ihre Vermeidung (also für ein »perfektes« Design) nicht zu hoch ist. Hinzu kommt nämlich, dass sich Anforderungen im Laufe einer Realisierung ändern. Dabei handelt es sich um Änderungen, die man bei noch so gutem Design nicht vermeiden kann. Tatsache ist, dass wir heutzutage schlichtweg nicht genug Zeit für gutes Design haben und sich die Wirklichkeit immer etwas anders herausstellt, als man meint. Das ist eigentlich auch gar kein Problem, denn entscheidend ist, dass eine Schnittstelle bei der Inbetriebnahme

passt und funktioniert. Ob die Schnittstelle vorher schon stabil sein muss, ist einzig und alleine eine Frage unserer Flexibilität und unserer Prozesse.

Solange ein Service noch nicht in Betrieb ist, kann man von dem, was man bei der Implementierung über Anforderungen und Lösungsansätze lernt, noch profitieren. Tut man das nicht, sorgt man dafür, dass schlechtes Design über Jahre verwendet wird und einen Einfluss auf alle Systeme hat.

Wenn man also davon ausgeht, dass sich Services sowohl während der Entwicklung als auch im Betrieb ändern können bzw. müssen, stellt sich die Frage, wie man mit diesen Änderungen umgeht. SOA ist ein Konzept für große verteilte Systeme, und man kann nicht verlangen, dass jede Änderung an einem Service gleichzeitig an allen Systemen durchgeführt wird, die diesen Service bereits nutzen. Und wenn Nutzer keinen Vorteil von dem Umstieg auf eine neue Version haben, kann es passieren, dass diese Nutzer über Jahre alte Versionen weiterverwenden. Hinzu kommt die Anforderung, dass man, um parallel arbeiten zu können, bei der Entwicklung mehrere Versionen gleichzeitig bearbeitet. Während eine Änderung noch im Test ist, wird die nächste bereits eingebaut.

Prinzipiell kann man daher zwischen zwei konzeptionell unterschiedlichen Anforderungen zur Versionierung von Services unterscheiden:

- Die eine Anforderung besteht darin, mehrere Versionen eines Service gleichzeitig in *derselben* Laufzeitumgebung zu betreiben. Darunter fällt, dass ein Nutzer noch eine alte Version eines Service aufruft, während ein anderer Nutzer schon eine neue Version des Service verwendet.
- Eine andere Anforderung besteht darin, dass unterschiedliche Versionen eines Service zwar gleichzeitig, aber in *verschiedenen* Umgebung verwendet werden. Darunter fällt der Sachverhalt, dass beispielsweise in der Entwicklung bereits ein Bug-Fix ist, der später eine zu dem Zeitpunkt in Betrieb befindliche Version eines Service ersetzt.

Bei der ersten handelt es sich um eine fachliche Anforderung zur Versionierung, denn in diesem Fall werden aus fachlicher Sicht verschiedene Versionen, die aus mehr oder weniger guten Gründen ein unterschiedliches Verhalten aufweisen, parallel betrieben. Auf diese Anforderung werde ich unter der Bezeichnung »fachlich getriebene Versionierung« (oder »domänengetriebene Versionierung«) im nächsten Abschnitt eingehen.

Die zweite Anforderung führt zum Thema Konfigurationsmanagement, also das Verwalten zusammengehörender Versionen und Artefakte. Dies wird in Abschnitt 12.4 auf Seite 191 erläutert.

## 12.2 Fachlich getriebene Versionierung

Fachliche getriebene Versionierung führt zu der Möglichkeit, mehrere Versionen eines Service gleichzeitig in derselben Laufzeitumgebung zu betreiben. Zwei Nut-

zer rufen dabei prinzipiell denselben Service auf, was aber zu unterschiedlichem Verhalten führt und/oder über eine unterschiedliche Schnittstelle laufen kann. Dabei müssen die Nutzer beim Aufruf auf irgendeine Art und Weise angeben, welche Version gemeint ist.

Es gibt zahlreiche Artikel zu diesem Thema, die sich aber vor allem mit der Frage beschäftigen, wie man die Verwendung unterschiedlicher Versionen möglichst elegant verbergen kann. Dahinter steckt die Idee, dass man von einer Änderung am besten gar nicht betroffen ist, was natürlich nur möglich ist, wenn die Änderung rückwärtskompatibel ist.

Im Idealfall ist man gar nicht betroffen, und es wird automatisch auf eine neue Version geschwenkt. Dies birgt allerdings das Risiko, dass sich ein Verhalten (und sei es nur das Laufzeitverhalten) unbemerkt ändert. Schon aus diesem Grund stellt sich die Frage, ob es nicht besser ist, ein im produktiven Betrieb befindliches Verhalten von neuen Anforderungen und Versionen erst einmal völlig zu entkoppeln, bis der Nutzer explizit auf die neue Version umschwenkt. Diese Vorgehensweise nenne ich »triviale fachliche Versionierung« und werde sie im Folgenden diskutieren. Nach meiner Meinung und Erfahrung muss es gute Gründe geben, von diesem Ansatz abzuweichen (»keep it simple«).

### 12.2.1 Triviale fachliche Versionierung

Die »triviale fachliche Versionierung« besteht aus einem vergleichsweise einfachen Ansatz: Es gibt im Grunde keinerlei technische Unterstützung für eine Versionierung. Jede Änderung an einem existierenden Service wird (technisch) als neuer Service betrachtet.

Soll zum Beispiel ein Service wie `lieferKundendaten()` geändert werden, führt man einfach einen neuen Service ein, der diese Änderung umsetzt. Natürlich sollte man deutlich machen, dass es sich fachlich um eine neue Version des existierenden Service handelt, was durch geeignete Namenskonventionen bequem erreicht werden kann. So kann man die geänderte Version in dem Fall einfach `lieferKundendaten_2()` nennen. Um eine Sonderbehandlung der ersten Version zu vermeiden, kann man dabei die erste Version gleich `lieferKundendaten_1()` nennen (siehe Abbildung 12-1). Mit diese Konvention besteht ein Servicename aus zwei Teilen: Ein Teil beschreibt, was der Service macht, und ein Teil spezifiziert die Version.

Natürlich hat diese Strategie den Nachteil, dass man bei jeder Änderung technisch einen neuen Service einführt. Man sollte diese Strategie deshalb auf den Fall beschränken, dass der Service bereits in Betrieb gegangen ist (und von mindestens einem Nutzer verwendet wird).

Eine derartige Strategie hat zwei wichtige Konsequenzen:

- Bevor ein Service in Betrieb geht (also während Entwicklung, Integration und Test), kann ein Service noch geändert werden, ohne dass dies zu einer neuen fachlichen Version eines Service führt (eine Unterscheidung aus Sicht

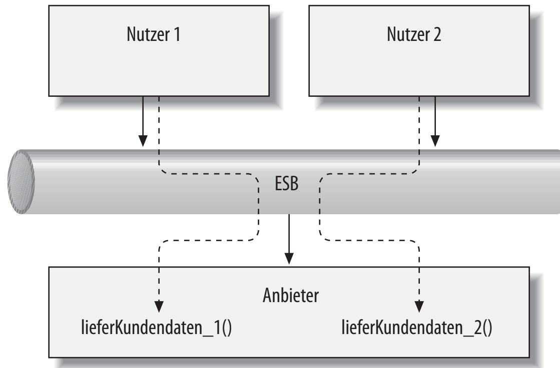


*Abbildung 12-1: Zwei Nutzer eines Services mit trivialer Versionierung*

des Konfigurationsmanagements kann dabei aber notwendig sein, siehe Abschnitt 12.4 auf Seite 191). Das bedeutet allerdings nicht, dass ein Anbieter machen kann, was er will. Spätestens wenn Services integriert sind, müssen die Änderungen mit den (potenziellen) Nutzern natürlich abgesprochen werden (dies sollte selbstverständlich sein).

- Sobald sich ein Service in Betrieb befindet, dürfen nur noch Fehler behoben werden (wobei die Schnittstelle stabil bleiben muss). Alles andere sollte technisch zu einem neuen Service führen.

Natürlich gibt es eine Grauzone zwischen Bug-Fixes und Änderungen, und mancher Bug-Fix kann sich im Nachhinein als Änderung herausstellen. Hier geht es aber erst einmal um die prinzipielle Strategie für Änderungen. Im Zweifelsfall sollte man miteinander reden. Und man kann natürlich immer noch absprechen, dass man eine Änderung aus pragmatischen Gründen als Bug-Fix durchführt. Wenn das allerdings ständig passiert, sollte man diese Strategie natürlich einmal hinterfragen.

Man beachte, dass ich bei dem ganzen Thema bisher nicht erörtert habe, ob die Änderungen rückwärtskompatibel sind. Tatsächlich spielt Rückwärtskompatibilität bei dieser Strategie keine Rolle. Diese Regelungen sollten grundsätzlich für *alle* Änderungen gelten.

Warum sollte es nicht möglich sein, angebotene Services rückwärtskompatibel zu erweitern? Dies hat mehrere Gründe:

- Rückwärtskompatible Änderungen stellen sich in der Praxis oft als doch nicht ganz so »kompatibel« heraus. Ein typisches Beispiel ist, wenn die Einführung eines neuen Attributs zu mehr Laufzeit führt, sodass (formale oder informelle) Service-Level-Agreements des Service nicht mehr eingehalten werden (siehe Abschnitt 13.5 auf Seite 209 für ein konkretes Beispiel).
- Jede Änderung birgt ein Risiko in sich. (»Was ist neu an dieser Version? Neue Bugs!«) Kann man die neue Version von der alten explizit unterscheiden,

kann man erst einmal abwarten und sehen, wie sich die neue Version in der Praxis bewährt, und dann auf die neue Version umsteigen.

- Sofern eine Änderung Datentypen betrifft, kann eine an sich kompatible Änderung aus technischen Gründen binärinkompatibel werden. Überspitzt formuliert gibt es so etwas wie Rückwärtskompatibilität dann überhaupt nicht. Auf dieses Thema wird in Abschnitt 12.3 auf Seite 184 eingegangen.

Das Problem bei dieser Strategie liegt auf der Hand: Es besteht die Gefahr, dass es jede Menge Versionen eines Service gibt. Vor allem bei Services, die Daten liefern (zum Beispiel alle Daten eines Kunden), passiert es sehr schnell, dass jeder neue Nutzer ein weiteres Attribut benötigt, und der Service wächst und wächst. Ich habe teilweise SOA-Landschaften gesehen, bei denen mehr als sechs Services parallel in Betrieb waren.

Um dies zu vermeiden (und die Wartbarkeit von Code zu verbessern), sollte man natürlich veraltete Services jeweils außer Betrieb nehmen. Wie in Abschnitt 11.2.2 auf Seite 175 diskutiert, erfolgt dies typischerweise in zwei Schritten:

1. Eine Service-Version wird als »veraltet« (englisch: deprecated) gekennzeichnet.
2. Der veraltete Service wird wirklich außer Betrieb genommen, wenn er nicht mehr aufgerufen wird.

Dabei besteht natürlich immer die Gefahr, dass Nutzer nicht auf eine neue Version umstellen wollen. Sie haben ja nicht unbedingt einen Nutzen davon, wenn die Erweiterungen für andere erfolgten, und die Systeme haben andere Prioritäten, als fremden Systemen zu helfen, Code wartbarer zu machen. Wenn alte Versionen deshalb aber im System bleiben, wird sich die »Entropie« der Software immer mehr verschlechtern. Dies muss man dann organisatorisch eskalieren oder der Anbieter muss in irgendeiner Form für den Aufwand des Nutzers aufkommen (siehe Abschnitt 11.2.2 auf Seite 175 für mehr Details zu diesem Thema aus Sicht des Lebenszyklus von Services).

### 12.2.2 Nichttriviale fachliche Versionierung

Wir haben bisher die triviale fachliche Versionierung diskutiert, bei der jede neue Version technisch ein neuer Service ist. Wie mag nun eine nichttriviale fachliche Versionierung aussehen?

Auf diese Frage gibt es eine ganze Reihe von möglichen Antworten:

- Es kann ein Mechanismus etabliert werden, der Services auf zukünftige Erweiterungen vorbereitet. Die Infrastruktur kann dazu einen Erweiterungspunkt anbieten, an dem bei Bedarf neue optionale Attribute eingeklinkt werden können. Existierende Nutzer sollten davon dann (auch technisch)

nicht betroffen sein, während neue Nutzer die neuen Attribute einfach verwenden können.

- Es können technische Mittel bereitgestellt werden, um bei Erweiterungen neue Service-Versionen auf alte Service-Versionen abzubilden, sodass ein Nutzer zwar den alten Service aufrufen kann, dieser aber auf den neuen Service abgebildet wird. Ein derartiges Mapping kann beim Anbieter, im ESB oder beim Nutzer erfolgen.
- Es kann eine Indirektion eingebaut werden, mit der unterschiedliche Implementierungen unter dem gleichen Namen unterschiedlichen Nutzern zur Verfügung gestellt werden. Irgendwo muss dann konfiguriert werden, welcher Service-Aufruf bei welcher Implementierung landet.

Wie diese Ansätze in der Praxis aussehen, ist eine andere Frage. So kann bei Web-Services zum Beispiel eine UDDI-Registry (siehe Abschnitt 16.2.4 auf Seite 270) als Broker verwenden werden, der Aufrufe unterschiedlich routet (siehe [*BrownEllis04*] für Details zu diesem Ansatz).

> **Übrigens:** Oft wird empfohlen, für unterschiedliche Versionen eines Service unterschiedliche »Namensbereiche« (englisch: namespaces) zu verwenden. Dieser Ansatz ist allerdings nicht geeignet, weil Namensbereiche in sich geschlossene Versionen umfassen sollten, was in der Praxis aber zum Problem wird, wenn Services gemeinsame Datentypen verwenden. Entweder hat dann jeder Service und jeder Datentyp einen eigenen Namensbereich oder bei einer Änderung sind alle Datentypen und Services, die diese Datentypen verwenden, von der Änderung betroffen.

## 12.3 Versionierung von Datentypen

Die bisherigen Erläuterungen betrafen das harmlose Problem bei der Versionierung. Sofern Services strukturierte Datentypen verwenden, muss auch eine Versionierung dieser Datentypen betrachtet werden. Und hier werden wir sehen, dass wir im Grunde nur die Auswahl zwischen Pest und Cholera haben, was ich an einem einfachen Beispiel erläutern werde.

Nehmen wir an, wir haben einen Service, der unter anderem Adressen verwendet oder zurückliefert. Und stellen wir uns nun vor, wir haben in einem ersten Ansatz Adressen mit folgenden drei Attributen modelliert:

```
String strasse
String plz
String ort
```

Nehmen wir nun an, es gibt die zusätzliche Anforderung, dass Adressen auch Postfächer enthalten können (natürlich wird niemand bei Adressen das Postfach

vergessen, aber das Beispiel zeigt nur exemplarisch, was passiert, wenn man eine Datenstruktur um ein Attribut erweitern will):

```
String postfach
```

Wenn nun beide Versionen des Service genutzt werden, werden auch zwei Versionen des Adress-Datentyps in der gleichen Laufzeitumgebung verwendet (siehe Abbildung 12-2).

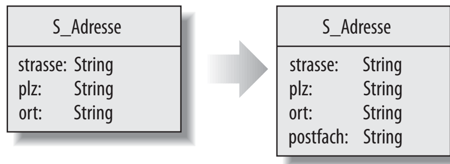


*Abbildung 12-2: Zwei verschiedene Versionen eines Adress-Datentyps*

Die Frage ist nun, wie man mit diesem Sachverhalt umgeht. Prinzipiell gibt es drei Möglichkeiten:

- Typisierte Schnittstellen mit unterschiedlichen Datentypen verwenden.
- Typisierte Schnittstellen mit dem gleichen Datentyp verwenden.
- Generische Schnittstellen verwenden, sodass die Unterschiede keine Rolle spielen.

Diese Optionen werde ich nun im Detail diskutieren.

### 12.3.1 Verschiedene Datentypen für verschiedene Datentypversionen

In Anlehnung an die von mir empfohlene Versionierungsstrategie von Services kann man einfach die Regel anwenden, dass jede Version eines Datentyps technisch ein neuer Datentyp ist. In diesem Fall zieht dieser Ansatz allerdings etwas größere Konsequenzen nach sich, was ich nachfolgend erläutern möchte.

Zunächst kann eine Änderung eines Datentyps dazu führen, dass auch andere Datentypen geändert werden müssen, weil diese diesen Datentyp verwenden. Wenn der Adress-Typ zum Beispiel von einem Datentyp verwendet wird, der eine Adresse als Attribut enthält, muss dieser Datentyp angepasst werden (siehe Abbildung 12-3). Entsprechend pflanzt sich die Änderung immer weiter fort, wenn Adressen von Adress-Listen, diese von Kundenlisten usw. verwendet werden. So kann eine Änderung an einem Datentyp zu einer ganzen Reihe von neuen Datentypen führen.

Zum anderen kann diese Strategie dazu führen, dass verschiedene Services, die fachlich die gleichen Daten liefern, dafür unterschiedliche Datentypen verwenden. In Programmiersprachen mit Typbindung führt dies dazu, dass man diese Daten(-typen) dann nicht mehr als Ganzes vergleichen oder zuweisen kann.

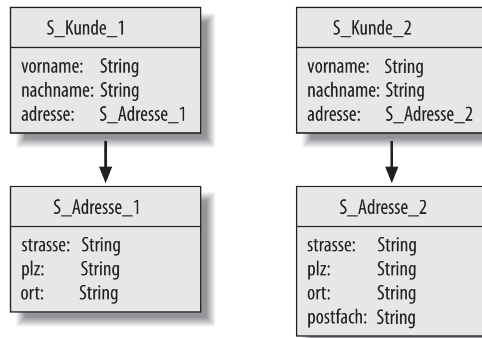


*Abbildung 12-3: Ein versionierter Adress-Datentyp, der von anderen Datentypen verwendet wird*

Stattdessen müssen Hilfsfunktionen eingeführt werden, die diese Operationen attributweise durchführen (und dabei hinzugekommene Attribute ignorieren oder mit Defaultwerten versehen). Dies hat sowohl für den Anbieter als auch für die Nutzer Konsequenzen:

- Als Service-Anbieter muss man für die gleiche Art von Information verschiedene Datentypen verwalten. Dies kann man dadurch erreichen, dass man Code für verschiedene Datentypen kopiert und getrennt implementiert oder einmal implementiert und auf die anderen Datentypen abbildet oder generischen Code (Templates) verwendet.
- Als Service-Nutzer kann es passieren, dass man bei dem einen Service eine Version mit der alten Datenstruktur und bei dem anderen Service eine Version mit der neuen Datenstruktur verwenden muss. Da die Datentypen nicht gleich sind, braucht man die gerade beschriebenen Hilfsfunktionen zum Vergleichen und Zuweisen der Daten aus den beiden Services.

Den letzten Punkt möchte ich noch einmal am Beispiel erläutern, denn er hat wirklich hässliche Auswirkungen.

Nehmen wir an, wir haben zwei verschiedene Versionen eines Service, wie `lieferKundendaten_1()` und `lieferKundendaten_2()`, die beide die in Abbildung 12-3 gelisteten Datentypen verwenden. Nun stellen wir als Service-Anbieter einen ganz neuen Service, `lieferRechnungsdaten_1()`, zur Verfügung, der mit den Rechnungsdaten auch Kundendaten liefert. Es bietet sich also an, den gleichen Datentyp und dabei am besten auch gleich die neueste Version dieses Datentyps zu verwenden. Nun haben wir technisch drei Services, die zum Teil gemeinsame Datentypen verwenden und alle von verschiedenen Nutzern gebraucht werden (siehe Abbildung 12-4).

Nehmen wir nun an, es kommt die zusätzliche Anforderung, dass der Service, der Rechnungsdaten liefert, bei den Kundendaten auch die Steuernummer mitliefern soll. Wir führen also eine neue Service-Version, `lieferRechnungsdaten_2()`,

ein, da die bisherigen Services weiterhin wie gehabt funktionieren sollen. Dieser Service liefert nun den neuen Datentyp `S_Kunde_3` mit dem neuen Attribut für die Steuernummer. Es ergibt sich die in Abbildung 12-5 dargestellte Situation.

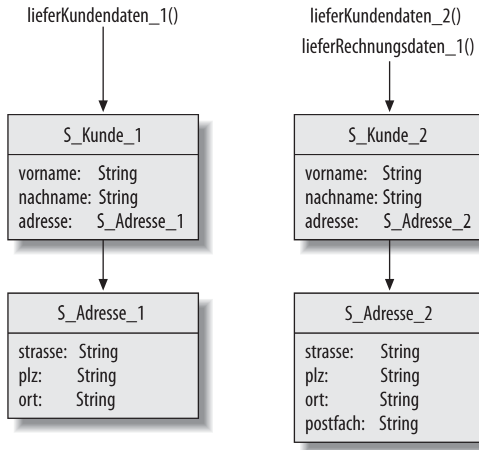


*Abbildung 12-4: Verschiedene Datentypen für verschiedene Services*

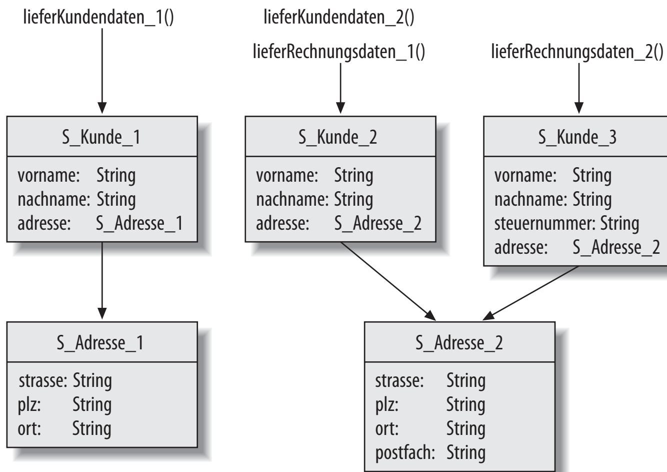


*Abbildung 12-5: Neue Attribute führen zu noch mehr Datentypen*

Wenn nun ein ganz neuer Nutzer kommt und sowohl Kunden- als auch Rechnungsdaten benötigt und wenn er dabei die jeweils neueste Version nutzen will bzw. soll, dann ergibt sich nun die seltsame Situation, dass beide Services im Prinzip die gleichen Daten liefern, diese aber technisch unterschiedliche Datentypen verwenden und somit inkompatibel sind. Der Nutzer könnte die Kundendaten aus beiden Serviceaufrufen nicht direkt vergleichen oder zuweisen. Man

kann den Vorwurf schon hören: »Wieso seid Ihr noch nicht mal in der Lage, für die von Euch angebotenen Services einheitliche Datentypen anzubieten?«

Natürlich ließe sich das Problem dadurch vermeiden, dass man bei jeder Änderung eines Datentyps für alle Services, die diesen Datentyp verwenden, eine neue Version einführt. Dies führt aber zu noch mehr Service-Versionen, als wir ohnehin schon haben. Man kommt vom Regen in die Traufe.

Das hier beschriebene Problem ist kein SOA-spezifisches Problem. Es ist ein grundsätzliches konzeptionelles Problem, das dadurch verursacht wird, dass man zwischen unterschiedlichen Versionen von strukturierten Datentypen sanft migrieren muss. Dazu gibt es Alternativen, die allerdings nicht unbedingt besser sind: Man kann auf strukturierte Datentypen oder Typbindung verzichten, grundsätzlich keine typisierten Schnittstellen verwenden, versuchen, für verschiedene fachliche Versionen ausgetauschter Daten doch den gleichen technischen Datentyp zu verwenden, und so weiter. Jede dieser Alternativen hat allerdings auch ihre Nachteile, auf die ich nachfolgend noch eingehen werde.

### 12.3.2 Der gleiche Datentyp für verschiedene Datentypversionen

Lassen Sie uns zum Vergleich einmal die Alternative betrachten, dass wir für alle fachlichen Versionen eines Datentyps den gleichen technischen Datentyp verwenden. Dieser muss dann natürlich alle Attribute enthalten. Von den Anbietern und Nutzern werden dann nur diejenigen Attribute gefüllt oder verwendet, die bei der jeweiligen Service-Version spezifiziert sind.

Dieser Ansatz führt zu drei Problemen:

- Man muss dokumentieren, welche Attribute für welche Service-Version gültig sind. Für komplexe Datentypen, die von verschiedenen Services verwendet werden, kann das sehr kompliziert werden.
- Mit zunehmender Anzahl von Versionen werden die Datentypen größer und sind damit nicht mehr binärkompatibel zu vorherigen Versionen. Als Konsequenz muss sichergestellt sein, dass alle Bibliotheken mit der gleichen Version eines Datentyps kompiliert wurden. Wenn sich ein Datentyp für einen Service ändert und die neue Version des Service benötigt wird, muss man allen Code neu kompilieren, der die alte Version dieses Service bzw. Datentyps verwendet hat. Ansonsten bekommt man zur Laufzeit unerlaubte Speicherzugriffe, eine der problematischsten Fehlerarten in der IT, weil es dadurch zu völlig unberechenbarem Verhalten kommt (ein Absturz ist noch das Beste, was passieren kann, denn daran merkt man, dass es ein fatales Problem gibt).
- Falls ankommende Daten bei der Ankunft validiert (auf ein korrektes Format überprüft) werden, muss man sicherstellen, dass die zusätzlichen Attribute nicht zu einer Fehlermeldung über ein unkorrektes Format führen.

Um dies zu verdeutlichen, sei noch einmal die Situation von Abbildung 12-5 betrachtet. Mit dem Ansatz, immer den gleichen technischen Datentyp zu verwenden, ergibt sich die Situation wie in Abbildung 12-6 dargestellt.

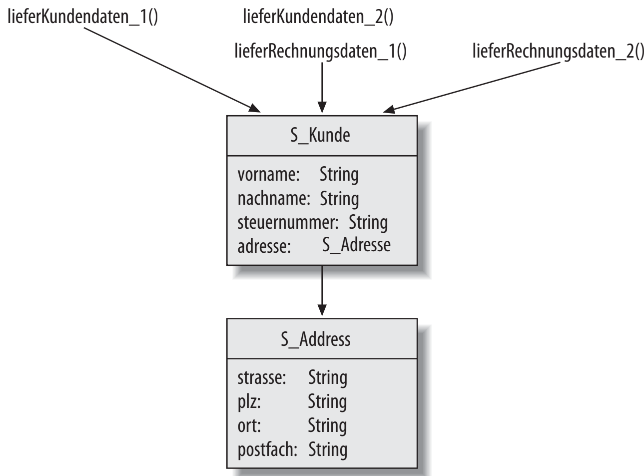


*Abbildung 12-6: Verschiedene Services mit verschiedenen Versionen, die gemeinsame Datentypen verwenden*

Für `lieferKundendaten_1()` werden dabei die beiden Attribute `steuernummer` und `postfach` ignoriert, und sowohl für `lieferKundendaten_2()` als auch für `lieferRechnungsdaten_1()` spielt das Attribut `steuernummer` keine Rolle. Zumindest dies sollte dokumentiert werden, was gar nicht so einfach sicherzustellen ist, denn wenn man die Dokumentation zu `lieferKundendaten_1()` schreibt, weiß man noch nichts über zukünftige Attribute, und wenn man `S_Adresse` verändert, ist es nicht offensichtlich, wo sich diese Änderung überall auswirkt.

Die erwähnte Gefahr der binären Inkompatibilität ergibt sich, wenn zunächst ein Nutzer eine Bibliothek verwendet, die `lieferRechnungsdaten_1()` aufruft und somit die alte Version des Datentyps und später zusätzlich eine Bibliothek mit `lieferRechnungsdaten_2()` kompiliert, die die neuen Datentypen verwendet. Wenn in diesem Fall nicht sichergestellt ist, dass *alles* neu kompiliert wird, wird es kritisch.

Diese Strategie ist daher nur zu empfehlen, wenn sich sicherstellen lässt, dass alle Beteiligten immer den gesamten Code neu kompilieren und nicht alte Bibliotheken mit neuen kombinieren.

### 12.3.3 Verwendung generischer Datentypen

Eine dritte Option besteht darin, nur generischen Code zu verwenden, sodass der tatsächliche Datentyp technisch gar keine Rolle spielt. Alle Zugriffe werden zur Laufzeit ausgewertet.

Um den Unterschied zwischen einem statischen und einem generischen API zu verdeutlichen sei hier zunächst zunächst ein Beispiel für statischen Code angegeben, der `lieferKundendaten_1()` verwendet:

```
S_Kunde_1 kunde;
S_Adresse_1 adresse;
String strasse;
input.setzeKundenID(id);
kunde = serviceAPI.lieferKundendaten_1(input);
adresse = kunde.getAdresse();
strasse = adresse.getStrasse();
```

Mit einer generischen Schnittstelle würde der gleiche Code wie folgt aussehen:

```
Data kunde, adresse;
String strasse;
input.setValue("KundenID",id);
kunde = serviceAPI.lieferKundendaten_1(input);
adresse = kunde.getValue("adresse");
strasse = adresse.getValueAsString("strasse");
```

Alternativ könnten die beiden letzten Zeilen auch zu einer kombiniert werden:

```
strasse = kunde.getValueAsString("adresse.strasse");
```

Dieser Ansatz ist definitiv eine Option, die in Betracht gezogen werden sollte. Es handelt sich allerdings um einen Ansatz, der derzeit relativ unüblich ist. Speziell im Kontext von Web-Services generieren alle üblichen Generatoren (als Default) typisierte APIs, was auf Dauer zu den oben angesprochenen Versionierungsproblemen führt. Der Vorteil von typisierten Schnittstellen ist natürlich, dass man Fehler schon zur Kompilierzeit findet. Mit dem generischen Ansatz stellt sich erst zur Laufzeit heraus, ob ein Zugriff auf ein Attribut zulässig ist. Aber für große Systeme kann sich das schnell auszahlen, denn die Versionierungsproblematik entfällt. Das Problem ist nur, dass man diesen Vorteil nicht sieht, solange erste Prototypen nur wenige Services anbieten. Später kann es dann zu spät sein, die Versionierungsstrategie umzustellen.

### 12.3.4 Zusammenfassung der Versionierung von Datentypen

Hoffentlich ist es mir gelungen klarzumachen, dass die Versionierung von Datentypen ein wichtiges und komplexes Thema ist, mit dem man sich bei großen verteilten Systemen befassen muss. Ich habe drei mögliche Ansätze vorgestellt und deren Vor- und Nachteile erläutert. Eine vierte Möglichkeit bestünde darin, ganz auf strukturierte Datentypen zu verzichten, was natürlich nicht wirklich funktioniert, wenn mit Hilfe von mehr oder weniger grobgranularen Services komplexe Datenstrukturen ausgetauscht werden sollen.

Wie ich bereits erwähnt habe, wird der Ansatz mit generischen APIs in der Praxis nicht oft genutzt. Man beachte, dass es hier nicht nur um das reine Protokoll, sondern auch um alle APIs geht, die auf das Protokoll abgebildet werden. Wenn man wie bei Web-Services ein Austauschformat wie XML hat, sind generische Schnittstellen eigentlich gar kein Problem. Aber dafür generische APIs auf allen Plattformen anzubieten ist keine kleine Herausforderung.

Der Ansatz mit den gemeinsamen Datentypen für verschiedene Versionen ist am gefährlichsten, weil es sehr hässliche Probleme gibt, wenn man nicht sicherstellt, dass beim Kompilieren bzw. Übersetzen des Source-Codes eines Prozesses immer die gleiche Version des Datentyps verwendet wird. Im Grunde kann man dann nur Source-Code ausliefern und diese Verantwortung den Nutzern übertragen.

Wir haben dieses Thema in Projekten immer wieder ausführlich diskutiert und sind doch in der Regel zur ersten Option mit technisch versionierten Datentypen zurückgekommen. Diese Politik führt aber leicht zu Beschwerden von Programmierern über ein offensichtlich völlig untaugliches Konzept, das noch nicht einmal Konsistenz über alle Datentypen eines Anbieters sicherstellt (und wir haben irgendwann einmal diskutiert, ob es nicht möglich ist, alle Datentypen unternehmensweit zu vereinheitlichen; siehe Seite 51). Dies ist aber im Grunde der Preis für die Anforderung nach möglichst unabhängigen verteilten Systemen mit unterschiedlichen Eigentümern. Ein Datentyp kann nicht einfach mal eben überall geändert werden.

Service-Nutzer sollten aus diesem Problem eine wichtige Konsequenz ziehen: Sie sollten sich unabhängig von den Versionierungsdetails der Anbieter machen. Das bedeutet, Nutzer sollten intern eigene Datentypen verwenden, die sie in einer dünnen Mapping-Schicht jeweils auf die aktuellen Datentypen der Anbieter abbilden. Diese Mapping-Schicht kann dabei mit der Aufgabe verknüpft werden, ein API auf ein Protokoll abzubilden, wie es in Abschnitt 5.3.3 auf Seite 71 diskutiert wird.

Aber Vorsicht, man sollte die Dinge nicht zu komplex machen, indem man von vornherein ein umfangreiches Konzept für komplexe Mapping-Strategien von Service-Nutzern entwickelt und verbindlich vorschreibt. Man sollte ganz einfach anfangen und sich erst mit dem Problem befassen, wenn es akut wird.

## 12.4 Konfigurationsmanagement-getriebene Versionierung von Services

Wie ich zu Beginn dieses Kapitels dargelegt habe, gibt es eine Anforderung zur Versionierung von Services, die zum Thema Konfigurationsmanagement führt. Es geht um die Frage, wie man verschiedene Versionen (oder Revisionen) eines Service verwaltet, die aber *nicht* gleichzeitig in ein und derselben Laufzeitumgebung verwendet werden. Während eine Version in Betrieb ist, befindet sich

eine neue in Test und Abnahme und eine dritte gerade in der Entwicklung. Hier werden typischerweise die klassischen Werkzeuge zur Versionskontrolle von Artefakten verwendet (CVS, ClearCase, Subversion, PVCS, RCS, SCCS usw.).

Bei diesem Thema ist zu beachten, dass man grundsätzlich in der Lage sein muss, die verschiedenen zusammengehörigen Artefakte bzw. deren zusammengehörende Versionen als Einheit zu definieren. Dazu verwendet man typischerweise ein oder mehrere Labels, die jeweils für eine Konfiguration vergeben werden. Wenn man also einen Service modelliert, daraus APIs generiert, gegen diese implementiert, diesen Code dann kompiliert und die resultierenden Bibliotheken ausliefert, muss immer wieder sichergestellt sein, dass diese ganzen Artefakte als zusammengehörig behandelt werden. Man beachte, dass diese Prozesse verteilt durchgeführt werden (Anbieter und Nutzer also getrennt gegen eine Schnittstelle implementieren).

Wenn alle Artefakte Dateien sind, ist das Ganze einfach. Man benötigt nur ein Versionsverwaltungssystem, das es erlaubt, entsprechende Labels zu vergeben und unterschiedliche Versionen der Artefakte zu verwalten (dazu sollte auch die Anzeige von Unterschieden und das Zusammenführen von Änderungen gehören).

Wenn Artefakte keine Dateien sind, sind entsprechende Mechanismen erforderlich. Wenn Services zum Beispiel in einem Repository abgelegt werden (siehe Kapitel 17), das auf Basis einer Datenbank implementiert ist, braucht man eine entsprechende Unterstützung zur Versionsverwaltung in dieser Datenbank oder spezielle organisatorische Regelungen (etwa die, dass für verschiedene Konfigurationen verschiedene Repositories/Datenbanken existieren).

Auch hier sollten Möglichkeiten vorgesehen sein, verschiedene Versionen zu vergleichen oder verschiedene Änderungen zusammenzuführen (englisch: merge). Dazu mögen dann spezielle Tools, Skripte oder Ähnliches notwendig sein, die man unter Umständen erst selbst entwickeln muss.

## 12.5 Versionierung in der Praxis

Die Häufigkeit von Änderungen ist nicht zu unterschätzen. In der Praxis können sehr viel häufiger neue Anforderungen und damit sehr viel mehr Versionen vorkommen, als man ursprünglich einmal angenommen hat. Ein Kunde von mir hatte in seinen ersten Richtlinien eine maximale Anzahl von drei gleichzeitig in Betrieb befindlichen Versionen vorgesehen; eine Zahl, die in der Praxis mehr als verdoppelt wurde.

Aus diesem Grund möchte ich nachfolgend noch zwei Aspekte diskutieren: Wer ist von neuen Versionen betroffen und wie werden ggf. doch Änderungen ohne neue Versionen eingeführt?

Abschließend möchte ich danach noch kurz auf eine Erfahrung mit virtuellen ESBs eingehen, die damit zu tun hat, dass sich in der Praxis fachliche

und Konfigurationsmanagement-getriebene Versionierung doch nicht so einfach trennen lassen.

### 12.5.1 Auswirkungen von Änderungen

Grundsätzlich sollte die Änderung eines Service so einfach wie möglich sein und so wenig Beteiligte wie möglich betreffen. Im Prinzip sollten Änderungen an einem Service ausschließlich den Anbieter und die Nutzer und sonst niemanden betreffen. Man beachte, dass zentrale Teams, die eine Architektur für verteilte Systeme aufsetzen, dazu neigen, zu viel Zentralismus einzuführen, was aber sehr leicht zu Flaschenhälsen führen kann.

Eine Infrastruktur (ein ESB) kann zum Beispiel anbieten, dass Service-Aufrufe automatisch beim Transport validiert werden. Das bedeutet allerdings, dass eine Änderung an der Schnittstelle nicht nur beim Anbieter und Nutzer, sondern auch in der Infrastruktur ausgeliefert werden muss. Dies kann man vorsehen, es sollte dann aber vollautomatisch und sofort erfolgen.

Anders herum formuliert sollte ein ESB deshalb in Bezug auf die unterstützte fachliche Funktionalität so generisch wie möglich sein. Ist eine Nachricht einmal unterwegs, sollte sie immer ankommen. Lediglich an den Endpunkten kann man (als Teil der Infrastruktur) Maßnahmen zur Validierung einbauen, die aber nicht zu Flaschenhälsen führen dürfen.

### 12.5.2 Verwendung von Call-Constraints

Bei einem meiner Kunden haben wir ein Konzept eingeführt, das wir »Call-Constraints« (wörtlich übersetzt »Aufrufeinschränkungen«) genannt haben. Jeder Service besaß einen Parameter, der dazu genutzt werden konnte, spezielle Situationen und Anforderungen anzuzeigen, die Auswirkungen auf das Verhalten eines Service haben konnten. Eingeführt wurde ein derartiger Parameter, weil es immer wieder passierte, dass ein relativ grobgranularer Service für bestimmte Nutzer zu viele Daten lieferte und deshalb zu lange dauerte (siehe Kapitel 13). Hinzu kommt, dass man zum Designzeitpunkt nicht unbedingt alle Randbedingungen kennt, unter denen der Service von einem Nutzer aufgerufen werden kann, und diese Randbedingungen für die Performance kritisch werden können (wenn man zum Beispiel Services hat, die Kundendaten mit allen Verträgen liefern, und dies dann auch Geschäftskunden mit Hunderten von Verträgen sein können).

Stellte man einen derartigen Fall in der Praxis fest, konnte man ohne Änderung der Signatur im Nachhinein noch ein gemeinsames Kennzeichen vereinbaren, das im Call-Constraints-Parameter dann vom Nutzer übergeben wurde, sodass der Anbieter für dieses Kennzeichen ein Spezialverhalten einbauen konnte. So war es möglich, selbst bei in Betrieb befindlichen Services Sonderbehandlungen einzubauen, ohne die Schnittstelle zu verändern.

Im Grunde ist das aber eine Änderung am Verhalten des Service, die man in diesem Fall allerdings sogar noch zur Laufzeit einbauen kann. Das dafür ausgetauschte spezielle Kennzeichen sorgt dabei dafür, dass andere Nutzer davon nicht betroffen sind. Nichtsdestotrotz sollte man Folgendes berücksichtigen:

- Es handelt sich nicht um einen generellen Mechanismus, mit dem ein Nutzer angeben kann, welche Daten er im Einzelnen benötigt und welche nicht. Es handelt sich um ein einfaches »Kennwort«, das man ausnahmsweise vereinbart, um eine notwendige Sonderbehandlung (übergangsweise) einzubauen.
- Innerhalb der Implementierung eines Service kann die Berücksichtigung eines derartigen Kennzeichens kompliziert werden, wenn es zu einer detaillierten Sonderbehandlung erst durchgereicht werden muss (etwa als Teil eines »Service-Kontext«).
- Es handelt sich um einen pragmatischen Ansatz, der nicht zum Prinzip erklärt werden sollte. Sonst landet man irgendwann bei rein generischen Services, bei denen das eigentliche Verhalten nur durch Aufrufparameter gesteuert wird und nicht einmal mehr am Service-Namen erkennbar ist. Call-Constraints sollten als temporärer Workaround betrachtet werden (wohlwissend, dass nichts beständiger ist als temporäre Workarounds).

Weitere Details zu Call-Constraints sind in Abschnitt 13.3.1 auf Seite 204 zu finden.

## 12.5.3 Virtuelle ESBs

In der Praxis hat sich gezeigt, dass die Grenzen zwischen fachlich und Konfigurationsmanagement-getriebener Versionierung ab einer gewissen Größenordnung von SOA leicht wieder verschwimmen kann. Das Problem ist, dass Entwicklungs- und Auslieferungszyklen von Software immer kürzer werden und bei großen verteilten Systemen nicht aufeinander abgestimmt sind.

Muss ein neuer oder geänderter verteilter Geschäftsprozess getestet werden, werden einige Systeme Änderungen einbringen, während andere Systeme so bleiben wie bisher. Im Prinzip ändert sich der Zustand zu testender und auszuliefernder Systeme im Wochen- oder Tagesrhythmus. Es lässt sich aber nicht für jeden Tag eine eigene Testumgebung bereitstellen (zumal manche Systeme für Integrationstests sogar die produktiven Systeme verwenden), weshalb man im Grunde zumindest in einer verteilten Testumgebung dann doch mehrere Versionen eines Service parallel betreiben muss. Während ein Tester die Software testet, die nächste Woche ausgeliefert wird, testet ein anderer Tester schon einmal die Auslieferung in drei Wochen.

Wir haben deshalb bei einem Kunden dafür das Konzept der »virtuellen ESBs« eingeführt. Dadurch können verschiedene Versionen eines Service gleichzeitig in der Infrastruktur betrieben werden. Beim Start eines Geschäftsprozesses kann dazu mit einem Label die Konfiguration angegeben werden, für die der Test

gelten soll (ohne Label gibt es Defaultsysteme wie z.B. das System für die nächste Auslieferung). Im Grunde haben wir das Labeln unterschiedlicher Versionen der Artefakte, das aus Sicht der Konfigurationsmanagement-getriebenen Versionierung notwendig ist, auf die Laufzeitumgebung ausgedehnt, weshalb wir nun doch mehrere Services-Versionen in der gleichen Laufzeitumgebung haben, obwohl dies fachlich gar nicht notwendig ist. Da diese aber nicht gleichzeitig in Betrieb gehen, wurde diese Versionierung nicht Teil der Service- und Datentypnamen.

## 12.6 Zusammenfassung

- SOA erfordert eine Strategie für sanfte Migrationen von neuen Service-Versionen.
- Der beste Ansatz zur Versionierung besteht darin, jede Änderung an einem in Betrieb befindlichen Service technisch als neuen Service zu betrachten.
- Falls (auf einzelnen Plattformen) typisierte APIs für Services angeboten werden, sollten auch Datentypversionen technisch als unterschiedliche Datentypen gehandhabt werden (dazu gibt es aber auch Alternativen).
- Um eine Explosion von Service- und Datentypversionen zu vermeiden, sollten Prozesse zur Abkündigung und Außerbetriebnahme von Services eingeführt werden. Dies ist einer der Gründe, weshalb Service-Aufrufe protokolliert werden sollten.
- Damit ein Service-Nutzer unabhängig von der Versionierungsstrategie eines Anbieters ist, sollte er (früher oder später) eine dünne Schicht einführen, die seine internen Datentypen von den externen Datentypen der Anbieter entkoppelt.
- Änderungen an Services sollten nur Anbieter und Nutzer betreffen (nichts und niemanden sonst).
- Um zur Laufzeit ausnahmsweise spezifische Änderungen einbauen zu können, können »Call-Constraints« eingeführt werden. Dies sollte aber nicht zu generischen Services führen.
- In virtuellen ESBs verwischen die Grenzen von fachlicher Versionierung und Versionierung aus Sicht des Konfigurationsmanagements.