Pacta sunt servanda

Praktische Service-Evolution

Phillip Ghadir, Daniel Lübke

Bei der Serviceorientierung befreien wir uns von dem Ballast, erforderliche Funktionalität selbst bereitzustellen. Wir verwenden stattdessen lieber bestehende Dienste. Als Konsument fordern wir von benötigten Services sowohl funktionale Zusicherungen als auch die Einhaltung gewünschter Qualitätsmerkmale ein. Im Allgemeinen ist kein spezielles Setup notwendig: Der Anbieter registriert den Service. Der Konsument findet den Service über die Registratur. – Und ab geht's. Der Artikel zeigt, worauf es ankommt, wenn man mit der Serviceorientierung aus technischer Perspektive auch mittel- bis langfristig erfolgreich sein will.

Die oberste Direktive des Service-Entwurfs

Jeder gute Service bietet dem Konsumenten einen Mehrwert. Im Idealfall exponiert er Geschäftsfunktionen und/oder den Zugriff auf Geschäftsdaten. Der Zugriff erfolgt über Remote-Schnittstellen und ist typischerweise von der Technologie unabhängig. Ein guter Service wird durch seine Schnittstelle vollständig in Bezug auf Funktionalität und Qualitätsmerkmale spezifiziert. Die oberste Direktive lautet: *Der Konsument erfährt nichts über die Implementierung des Service!*

Daher gehören zur Spezifikation eines Service neben Aussagen zur Syntax und Semantik auch Aussagen zu den qualitativen Aspekten. Wollen wir als Serviceanbieter die Einhaltung von Qualitätsmerkmalen zusichern, sollten wir erwägen, von Konsumenten eine Registrierung (oder den Erwerb der Zusicherung) zu verlangen.

Kopplung erfolgt ausschließlich über Service-Discovery auf Basis einer Schnittstelle. Abhängigkeiten zu konkreten Serviceimplementierungen sind verboten.

Geld scheffeln ...

... kann nur, wer Konsumenten gewinnt, hält und zufriedenstellt.

In vielen Fällen ist – neben der Funktionalität des Service – entscheidend, wie verlässlich der Service ist, und wie hoch die Zugangshürden sind. Verlässlichkeit bedeutet, dass wir das spezifizierte Verhalten sowie die zugesicherten Qualitätsmerkmale (wie z. B. Transaktionsdurchsatz, Reaktionszeiten, Verfügbarkeit oder Antwortzeiten) eines Service nicht mehr verändern. Eine Erweiterung der Funktionalität oder der Qualitätsmerkmale ist dahingegen zulässig.

Die Zugangshürden werden maßgeblich durch die verwendete Infrastrukturtechnologie sowie die Pragmatik und die Dokumentation einer Serviceschnittstelle beeinflusst. Die Höhe der Zugangshürden beeinflusst damit die Zahl potenzieller Servicekonsumenten.

Verlässlichkeit

Wenn fachliche Änderungen eine Anpassung der Aufrufparameter zur Folge haben, darf dies nicht eine Anpassung aufseiten des Servicekonsumenten erfordern. Erzwungene Änderungen auf der Konsumentenseite sind gleichbedeutend mit einem ungeplanten Einstellen des Service.

Da bekommt der Erfahrungswert, dass sich 30 % aller Anforderungen jährlich ändern, eine völlig neue Bedeutung: Serviceänderungen, die eine Änderung der Konsumenten nach sich ziehen, sind vielfach unerwünscht. Je häufiger solche erzwungenen Änderungen erfolgen, desto aufwendiger und teurer ist der Service für die Konsumenten.

Drei Serviceschnittstellen hoch im Licht

Abbildung 1 verdeutlicht den Zusammenhang zwischen Servicekonsumenten und unserem Service. In der Regel nehmen wir vereinfacht an, dass Konsumenten und Anbieter über eine Schnittstelle voneinander entkoppelt werden. Tatsächlich gibt es allerdings – unabhängig von der verwendeten Infrastrukturtechnologie – drei Schnittstellen:

- die Schnittstelle, auf deren Basis der Konsument realisiert wurde,
- die Schnittstelle, die der Service gerade realisiert,
- die idealisierte Schnittstelle, von der wir alle annehmen, dass wir uns auf sie geeinigt haben.

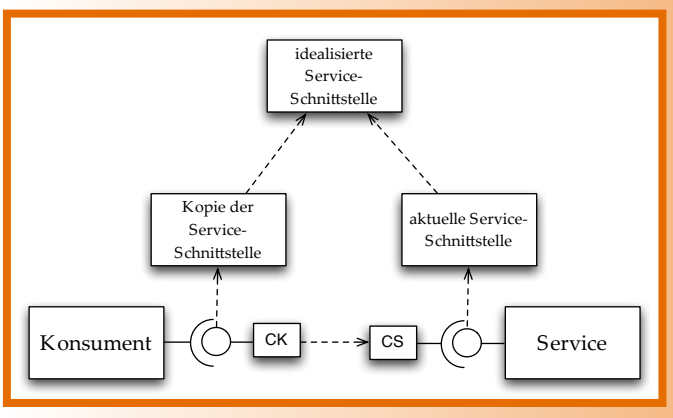

Abb. 1: Konstellation von Konsument und Service nach einigen Änderungen

Wir verstehen die idealisierte Serviceschnittstelle konzeptionell als Superklasse der beiden anderen Schnittstellen und fordern, dass jede Version der Serviceschnittstelle eine Spezialisierung der idealisierten Serviceschnittstelle ist. Insbesondere für die Serviceorientierung fordern wir die Einhaltung von Liskovs Substitutionsprinzip [SOLID]: Keine Schnittstelle darf mehr voraussetzen oder weniger zusichern als die idealisierte Schnittstelle.

Darüber hinaus sind unter anderen folgende Grundelemente erfolgreicher Serviceimplementierung zu beachten: Zeichencodierung (neudeutsch: Encoding), Hierarchisierung/Strukturierung der auszutauschenden Daten, Lesbarkeit der Daten, Unabhängigkeit von der Reihenfolge der Datenfelder, Eindeutigkeit der Referenzen, Eindeutigkeit des Informationstyps, Zugriff auf Basis von IDs anstatt über die Dokumentenstruktur usw.

Tipps für den Umgang mit Änderungen

Das Open-Closed-Principle (siehe [SOLID]) besagt, dass Software offen für Erweiterungen implementiert, aber geschlossen gegen Änderungen implementiert werden soll. Anders formuliert: Eine Erweiterung soll möglich sein, ohne bereits bestehende, getestete Funktionalität zu verändern.

Änderungen haben praktisch immer etwas vom Einstellen eines Dienstes – wie in „Geschlossen wegen Geschäftsaufgabe“: Gestern noch eine Eisdiele – morgen eröffnet ein Rasierapparat-Spezialgeschäft. Paypal – damals eine Bezahlplattform für Mobiltelefone – stellte seinen Dienst ein, um sich voll auf den Internet-basierten Bezahldienst zu konzentrieren. Nokia hörte auf, Gummistiefel zu produzieren und entwickelte Fernseher. Keine Änderung, sondern Einstellen eines Dienstes und Anbieten eines neuen.

Wenn wir in der Serviceorientierung von Änderungen sprechen, spiegelt dies häufig keine Änderung im Dienst an sich wider, sondern vielmehr eine Änderung in technischen Belangen oder im Bezug auf die Dienstqualität. Vielfach sind es also eher Erweiterungen an der Serviceschnittstelle oder an der Serviceimplementierung, wobei der Service eigentlich „unverändert“ oder besser weiter genutzt werden kann.

Änderungen der Implementierung tun nicht weh

Ein Beispiel für die Änderungen der Serviceimplementierung ohne dabei den Service selbst zu verändern, ist das Austauschen des Logistik-Dienstleisters im Hintergrund durch einen anderen, der den gleichen Logistik-Dienst anbietet. Für den Konsumenten ändert sich durch diese Änderung nichts. Ein Hoch auf ordentliche Kapselung!

Wenn sich die Signatur ändert

Schnittstellenänderungen können dazu führen, dass Konsumenten angepasst werden müssen, um den Service weiterhin nutzen zu können. Um dies zu vermeiden, müssen technische Schnittstellenänderungen sorgsam eingeführt werden. Neue Funktionen im Service sind tendenziell unproblematisch. Je nach gewählter Technologie kann bereits das Umsortieren von Datenfeldern (der Service-Request-Parameter) problematisch werden.

Enge Kopplung an z. B. in Java oder XSD strikt definierte Datentypen kann Probleme bereits in den Konnektoren auf

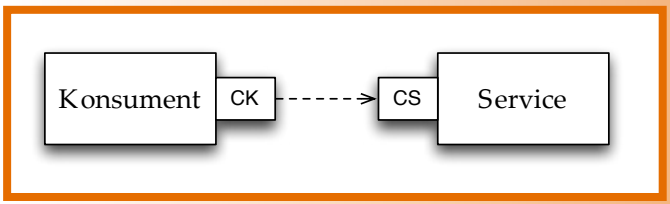

Abb. 2: CK – Konnektor des Konsumenten, CS – Konnektor des Service

Konsumenten- oder Serviceseite (CK und CS in Abb. 2) verursachen, noch bevor der Serviceanbieter eine Anfrage oder der Konsument eine Antwort zur Kenntnis nimmt.

Wir sollten auf strikte Bindung an Strukturen verzichten und mittels definierter Bezeichner/IDs die benötigten Daten selbst über flexible Mechanismen wie z. B. XPath ermitteln. Daher empfehlen wir die Verwendung von hierarchischen Datenstrukturen (objektorientierte bzw. hierarchisch strukturierte Daten). Abbildung 3 stellt auf der rechten Seite dar, wie dies gemeint ist: Anstatt alle Detailfelder gleichberechtigt in ein Dokument zu schreiben, sollten zusammengehörige Daten gruppiert werden. Alle Beteiligten müssen sich dann auf eine Detailebene nur mit den Informationen befassen, die in dem jeweiligen Kontext notwendig sind.

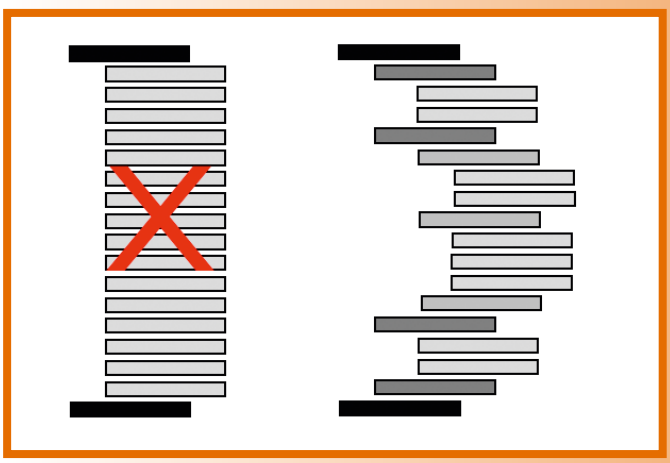

Abb. 3: Strukturierte Daten (rechts) sind unstrukturierten (links) vorzuziehen

Wenn sich jährlich circa 30 % der Anforderungen ändern ...

Darüber hinaus gilt für die Datenstrukturdefinition an den Schnittstellen: Von Anfang an muss Platz für Ad-hoc-Erweiterungen vorgesehen werden: Man sollte stets für jeden Knoten mit unbekannten Zusatzinformationen oder geänderter Feldreihenfolgen rechnen. Hierfür bieten sich zum Beispiel Key-Value-Listen (nicht automatisiert validierbar) oder in XML z. B. Elemente vom Typ XML-Any an.

Um eindeutig feststellen zu können, ob zum Beispiel optionale Daten aus der Key-Value-Liste verwendet werden sollen oder nicht, müssen sowohl Anbieter als auch Konsument erkennen können, zu welcher konkreten Version der vereinbarten Schnittstelle ein Datum gehört. So kann zum Beispiel für ein Release 1.2.6 ein Wert aus /miscellaneous/customer_age gelesen werden, während der Wert in Release 2.0 definiert unter /customer/age abgelegt wird. Analog

könnten auch unterschiedlich belegte Werte geeignet interpretiert werden.

Hingegen bietet XML-Any den Vorteil, dass man separat definierte Schemata für zusätzliche Informationen im Nachgang referenzieren kann.

Möglichkeiten, Schnittstellen technisch anzupassen

Ein Weg, eine Schnittstelle zu erweitern und dabei sicherzustellen, dass kein bestehender Konsument angepasst werden muss, ist das Hinzufügen einer neuen, erweiterten Schnittstelle bei gleichzeitigem Erhalt der alten Schnittstelle.

Eine neue Schnittstelle könnte zum Beispiel unter neuem Namen (entweder im gleichen oder im separaten Namensraum) bereitgestellt werden. Geänderte Funktionen können einfach unter neuem Namen hinzugefügt werden, ohne die ursprünglichen zu ersetzen. (Dank [Refactoring] in der Implementierung sind wir nicht gezwungen, Quelltext zu duplizieren.)

Anti-Corruption-Layer und Adaptation

Je nach Paradigma gibt es sowohl synchrone Request/Response-Zugänge als auch asynchrone Zugänge. Manchmal sind Anpassungen im Transport-Protokoll notwendig. An diesen Stellen können wir z. B. auf Middleware wie ESBs zurückgreifen.

Bei der Änderung des Zugangswegs kann die Middleware Datenformate, Transportprotokolle und andere Parameter anpassen. Gleichermaßen sollten wir spätestens bei den ersten Änderungen mittels Refactoring eine Schicht einziehen, die der Entkopplung von interner Struktur (dem internen Domänenmodell des Service) und der öffentlich deklarierten Datenformate (den Datenstrukturen unserer Schnittstellen) dient. Im Domain-Driven Design [DDD] ist so ein Ansatz als Anti-Corruption-Layer bekannt. Andere nennen eine solche Schicht Adaptation- oder Mediation-Layer. Das Grundprinzip bleibt: Änderungen an Schnittstellen und Implementierungen sollen nicht auf die bestehenden Kontrakte nach außen durchschlagen.

Services folgen dem Markt

In Organisationen ist das Deployment von Services oftmals gut geregelt. Es gibt Reviews und Quality Gates, die die Serviceschnittstelle und die -implementierung durchlaufen müssen. Wesentlich wichtiger und oftmals nicht so detailliert festgelegt sind jedoch die Fragen, wie eine neue Serviceversion eingeführt wird und wie eine alte abgekündigt wird.

Änderungen in den Services sind unkritisch, solange der Konsument davon nicht betroffen ist. Wird z. B. eine neue Operation zu einer Schnittstelle hinzugefügt und alle anderen Operationen verändern sich nicht, dann ist der Konsument nicht gezwungen, etwas zu ändern. Dadurch, dass Konsument und Provider durch ein unabhängiges Zwischenformat wie XML oder JSON voneinander entkoppelt sind, können Änderungen transparent und abwärts kompatibel gemacht werden, die in CORBA nicht möglich gewesen wären. So können optionale Eingabeparameter deklariert werden, die vom alten Client einfach niemals generiert werden.

Bleibt die Frage, was bei einer Serviceänderung versioniert werden soll? Die gesamte Serviceschnittstelle, der Service oder nur eine einzelne Serviceoperation?

Bei SOAP-basierten Webservices werden oftmals die Interfaces versioniert, indem z. B. die Namespaces eines WSDL-Porttyps und der entsprechenden Schemata geändert werden.

Gerade wenn Konsumenten nicht alle Operationen, sondern nur einen Teil verwenden, kann die dedizierte Versionierung auf Operationsebene die Kopplung zwischen den Systemen reduzieren (siehe auch Interface-Segregation-Principle in [SOLID]).

Neben den klassischen Serviceimplementierungen, die ganz normal über Versionskontrollsysteme verwaltet werden, gibt es zunehmend auch aggregierte Services.

Automatisierte Geschäftsprozesse in BPEL

In BPEL entwickelte Prozesse werden nicht nur im Versionskontrollsystem, sondern auch serverseitig in einer BPEL-Laufzeitumgebung versioniert. So können auch Prozessinstanzen von unterschiedlichen Prozessversionen gleichzeitig aktiv sein, sodass selbst nach einem Deployment einer neuen Version noch alte Prozessinstanzen aktiv sind, die noch auf alte Serviceschnittstellen zugreifen. Dies kann dann problematisch werden, wenn die Geschäftsprozesse wirklich langlaufend sind, also z. B. mehrere Monate dauern. Die alte Serviceversion muss dann noch zur Verfügung stehen, damit der Prozess erfolgreich abgeschlossen werden kann.

Serviceversionierung richtig gemacht

Eine Serviceimplementierung realisiert eine oder mehrere Serviceschnittstellen. Der Konsument benutzt einen Service, der lediglich über eine Schnittstelle und eine bekannte Adresse identifiziert wird. Der Konsument kann auf unterschiedlichste Weise realisiert sein: Von klassischen Backend-Systemen über Portale können auch Serviceorchestrierungen, wie z. B. (BPEL-) Prozesse, Services aufrufen.

Gerade bei den Prozessen wird die Unterscheidung zwischen (Prozess-)Definition und (Prozess-)Instanz wichtig. Denn im Gegensatz zu klassischen Anwendungen laufen Prozessinstanzen noch so lange mit dem alten Modell weiter, bis sie beendet sind. So kann es sein, dass es noch eine Instanz einer alten Version gibt, die selbst nach langer Zeit noch aktiv ist und eine alte Serviceschnittstelle anspricht, obwohl das neue Software-Release bereits einige Zeit zurückliegt.

Einige Workflowsysteme erlauben das Migrieren von Prozessinstanzen auf neue Prozessdefinitionen, jedoch ist dies nicht in allen Systemen und nicht für alle Änderungen automatisiert möglich. Bei Änderungen der Serviceschnittstelle gibt es zwei unterschiedliche Arten der Änderungen:

- abwärts kompatibel: Bestehende Konsumenten können ohne Probleme auf Services der neuen Version zugreifen. Hierunter fallen z. B. optionale Eingabeparameter.
- nicht abwärts kompatibel: Bestehende Konsumenten können die neue Version nicht mehr aufrufen. Hierunter fallen z. B. neue Pflichteingabeparameter, neue Ausgabeparameter und XML-Namespace-Änderungen.

Abwärts kompatible Änderungen werden oftmals mit Minorversionen angegeben (z. B. 2.1, 2.2, …). Wichtig ist hierbei, dass diese Minorversionen nicht über den Namespace angezeigt werden dürfen, weil dies sonst doch die Abwärtskompatibilität zerstört. Die Version kann in der Dokumentation angegeben werden. Möchte man zur Laufzeit sehen, welche Version ein System schickt, so muss diese Information als separates Feld in der eigentlichen Nachricht geführt

werden. Allerdings sollte man dieses nicht zur Schemavalidierung verwenden, da die Minorversion abwärts kompatibel ist. Das Ansprechen eines solchen Feldes bei der Deserialisierung oder bei der Validierung ist ein klares Zeichen für schlechtes Design.

Realisierungsstrategien

Nachdem die unterschiedlichen Versionen explizit gekennzeichnet sind, stellt sich zwangsläufig die Frage, wie eine neue Version eingeführt und die alte abgelöst werden soll. Nur ganz selten ist es möglich, die neue Version zu deployen und dabei die alte komplett zu ersetzen. Vielmehr werden in einer Transitionsphase mehrere Serviceversionen angeboten werden müssen. Dies erlaubt es den Servicekonsumenten, ihre Software in ihrem eigenen Releasezyklus auf die neue Serviceversion umzustellen.

Die Dauer dieser Transitionsphase wird durch zwei Faktoren bestimmt:

- die längsten Releasezyklen der Servicekonsumenten und
- die längste Instanzdauer aller Servicekonsumenten.

Im Idealfall ist die Transitionsphase so lang, dass alle alten Instanzen der Servicekonsumenten beendet sind und jeder Servicekonsument einen Software-Release durchführen konnte.

Doch wie bietet man technisch möglichst sauber zwei Serviceversionen parallel an? Dafür gibt es – wie so oft – verschiedene Architekturalternativen. Es seien hier exemplarisch drei aufgeführt:

- Zweite Service-Fassade vor der Geschäftslogik: Wenn das Design des Service sauber aufgebaut ist, sollte eine Service-Fassade die eingehenden XML- oder JSON-Daten parsen und auf die eigentlichen Domänenobjekte abbilden. Danach wird die Geschäftslogikschicht aufgerufen. Die Logik ist somit serviceunabhängig und es kann eine zweite Fassade für die neue Serviceversion unabhängig von der ersten entwickelt werden.
- Dedizierter Versionsadapter: Der alte Serviceaufruf wird von einem neuen Service entgegengenommen, der den Aufruf in einen Aufruf für die neue Version umwandelt und den neuen Service aufruft. Das Ergebnis des Aufrufs wird ebenfalls auf die alte Version transformiert und an den Servicekonsumenten zurückgeliefert.
- Message-Transformation auf einem ESB: Die Transformationen der alten und neuen Nachrichten können, sofern vorhanden, auch in der Infrastruktur gelöst werden, indem z. B. auf einem ESB die entsprechenden Regeln definiert werden.

Fazit

Serviceorientierung ermöglicht uns, eine Reihe von Bootstrapping-Problemen „wegzudefinieren“, die auf der Inanspruchnahme von Services beruht. Für die Serviceorientierung ist Verlässlichkeit ein genauso wesentliches Merkmal wie die Anpassbarkeit an sich ändernde Situationen. Die Einführung einer Adapter-Schicht ist dabei ein geeigneter Weg. Um aber Serviceschnittstellen weiterentwickeln zu können, sollten dazu die bestehenden Kontrakte stets aufrecht erhalten werden.

Wichtig für alle Beteiligten ist, dass man anhand der Anfragen, Antworten und der darin enthaltenen Daten erkennen kann, wer (welche Implementierung) diese Daten – auf Basis welcher Spezifikation – erzeugt hat.

Literatur und Links

[BPEL] T. van Lessen, D. Lübke, J. Nitzsche, Geschäftsprozesse automatisieren mit BPEL, dpunkt.verlag, 2011

[DDD] E. Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, Addison-Wesley, 2004

[Refactoring] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

[SOLID] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall International, 2008

[WSDL] Web Services Description Language, `http://www.w3.org/TR/wsdl20/`

[XSD] XML Schema Definition Language, `http://www.w3.org/TR/xmlschema-0/` und `http://www.w3.org/TR/xmlschema-1/`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Dr.-Ing. Daniel Lübke hat Wirtschaftsinformatik an der TU Clausthal studiert und an der Leibniz Universität Hannover am Fachgebiet Software Engineering promoviert. Zurzeit arbeitet er bei der innoQ Schweiz GmbH und berät dort Kunden in SOA- und MDA-Projekten. Zudem ist er Maintainer des BPELUnit-Projekts, welches Entwickler beim Testen von BPEL-Prozessen unterstützt.
E-Mail: daniel.luebke@innoq.com