
Leseprobe aus:**SOA - Expertenwissen**

Methoden, Konzepte und Praxis serviceorientierter Architekturen

Hrsg. Gernot Starke & Stefan Tilkov

dpunkt-Verlag 2007, 886 Seiten,

Mit Beiträgen von 50 internationalen SOA-Experten

ISBN-10: 3898644375

ISBN-13: 978-3898644372



Evolution von zentralen Objektmodellen zu einer service-orientierten Komponenten-Architektur

Dirk Tschierschke

Thilo Mezger

GEBIT Solutions GmbH

Abstract

Der Entwurf eines unternehmensweiten Datenmodells und dessen Abbildung in ein unternehmensweites Objektmodell galt lange Zeit als State-of-the-Art der objektorientierten Programmierung. Ein zentrales Objektmodell versprach auf der einen Seite einen gemeinsamen redundanz-freien Zugriff auf die Datenschicht für alle Anwendungen, andererseits aber führte gerade diese Zentralität zu starker Verzahnung und großen Abhängigkeiten.

Als "natürlicher" Ausweg aus diesem Dilemma bietet sich eine SOA-Architektur an, die die Geschäftsprozesse und aus diesen abgeleitet, sogenannte prozessorientierte Komponenten in den Vordergrund stellt.

Diese komponenten-orientierte SOA erlaubt weiterhin alle Vorteile von OO innerhalb einer Komponente, dabei wird an den Komponentengrenze aber bewusst mit OO-Paradigmen gebrochen. So werden bewusst Assoziationen zwischen Objekten gekappt oder Redundanzen zugelassen, um Komponenten zu trennen und lose zu koppeln.

Im Rahmen des Entwurfs und der Realisierung dieser unternehmensweiten SOA bedarf es neben diverser neuen technischen Ansätzen auch einiger organisatorischer Erneuerungen im Bereich der Software-Entwicklung sowie in der Kommunikation aller Projektbeteiligter. So gilt es u.a., Teams zu schaffen, die den Service-Gedanken verinnerlicht haben, um Verantwortung für Komponenten delegieren zu können.

Mit der Fokussierung auf Geschäftsprozesse statt auf Entitäten und der prozessorientierten Komponentenbildung entstand so in der IT eines weltweit operierenden Retailers eine service-orientierte Komponentenarchitektur.

1 Ausgangspunkt

In vielen Unternehmen findet man heute unternehmensweite Datenmodelle, in denen die vollständige Fachlichkeit des Unternehmens modelliert wurde. Sämtliche Anwen-

dungen greifen auf diese gemeinsame Datenbasis zu. Projekt- oder anwendungsspezifische Daten sind Ausschnitte des unternehmensweiten Datenmodells.

Die Anwendungen nutzen dabei meist eine zentrale Bibliothek, die den Zugriff auf die Geschäftsobjekte ermöglicht, die in Tabellen eines relationalen Datenbanksystems gespeichert werden. Konkret kann dies zum Beispiel über EJB Entity Beans oder einen beliebigen Access Layer geschehen. Häufig wird das vollständige unternehmensweite Datenbankmodell in einer einzigen Geschäftsobjektbibliothek abgebildet. Diese Geschäftsobjektbibliothek wird dann von mehreren Anwendungen genutzt.

Diese Modellierung führt jedoch zu entscheidenden Nachteilen.

1.1 Probleme

Eine zentrale Bibliothek mit Geschäftsobjekten stellt keine sinnvolle Kapselung der Persistenzschicht dar, da sich die Entwickler einer Anwendung entscheiden müssen, bei Datenbankänderungen (und damit einer neuen Version der Objektbibliothek) entweder diese neue Version einzusetzen oder das Risiko einzugehen, eine nicht aktuelle Version einzusetzen, wenn sie der Meinung sind, dass die Änderung für diese Anwendung nicht relevant ist.

Diese starke Verzahnung und Abhängigkeit und der damit verbundene Wartungsaufwand der Anwendungen bei Datenbankänderungen war der entscheidende Nachteil, weshalb nach neuen Lösungen gesucht werden musste.

1.2 Lösungsstrategie

Eine Abkehr von einem unternehmensweiten Datenbankmodell hin zu kleinen anwendungsspezifischen Modellen schien weder machbar noch sinnvoll. Also blieb nur das Aufteilen der Objektbibliothek in überschaubare Komponenten.

Um des Weiteren die Abstraktion zwischen den Anwendungen und der Persistenzschicht zu erhöhen, sollte auf diese Komponenten in einem zweiten Schritt nur noch über Services zugegriffen werden.

1.3 Evolutionäres Vorgehen

Es ist klar, dass ein solcher Umbau der Architektur nicht auf einen Schlag geschehen kann. Eine „Service-orientierte Architektur“ zu schaffen war keineswegs das direkte ursprüngliche Ziel.

Es soll an dieser Stelle erwähnt werden, dass der Begriff „SOA“ zum Zeitpunkt der Entwicklung dieser Lösungsstrategie noch keineswegs existierte. Es wurden lediglich Auswege aus jeweils aktuell existierenden Problemen der Architektur gesucht.

Die einzelnen Stufen der Evolution, die letztendlich dann doch von ganz allein zu einer SOA führten, sollen nun nachfolgend beschrieben werden.

2 Bildung von Komponenten

2.1 Komponententypen

Mit dem Thema der Komponentenbildung in der Software-Entwicklung beschäftigen sich bereits viele Veröffentlichungen.

Als Kriterium zur Bildung von Komponenten sollten die Geschäftsprozesse des Unternehmens dienen. Da es natürlich immer auch Geschäftsobjekte gibt, die sich keinem Geschäftsprozess zuordnen lassen – das gilt insbesondere für Referenzdaten (Stammdaten) – musste es auch ein zweites Kriterium geben.

Es schien also sinnvoll, zwischen zwei Typen von Komponenten zu unterscheiden:

- Prozess-Komponenten
- Referenzdaten-Komponenten

2.2 Prozess-Komponenten

Eine Prozess-Komponente enthält alle Geschäftsobjekte mit zugehöriger Geschäftslogik eines einzelnen Geschäftsprozesses eines Unternehmens. So kann es typischerweise Komponenten für Umsatzplanung, Bestellung, Lieferung etc. geben.

Geschäftsobjekte eines Geschäftsprozesses sind meist stark voneinander abhängig, wobei in der Regel zwischen Objekten verschiedener Prozesse eher eine losere Kopplung vorliegt. Aus diesem Grund ist dieses Kriterium sehr gut zur Bildung von Komponenten geeignet.

Es ist sehr schwierig eine konkrete Aussage über die Größe einer Komponente zu machen. Die mittlere Größe liegt etwa bei ca. 25 Komponenten, wobei es durchaus starke Abweichungen nach oben und unten geben kann.

Falls eine Komponente zu groß und damit zu unhandlich wird, kann sie durchaus in kleinere zerteilt werden, wenn sich Bereiche identifizieren lassen, die eher lockerer gekoppelt sind, beispielsweise könnte so *Planung* zerteilt werden in *Vorplanung* und *Quantitative Planung*.

2.3 Referenzdaten-Komponenten

Die Referenzdaten (Stammdaten), die sich keinen einzelnen Geschäftsprozessen zuordnen lassen, da sie allgemeingültig für das Unternehmen sind, werden in Referenzdaten-Komponenten abgebildet.

Meist lassen sich auch hier Kriterien finden, nach denen die Geschäftsobjekte weiter gruppiert werden können, um nicht eine große Komponente zu haben. Diese Kriterien sind meist sehr unternehmensspezifisch. Typischen Gruppen könnten aber etwa *Finanzdaten* sein, was vielleicht alle Geschäftsobjekt rund um Währungen, Währungssymbole etc. enthält oder *Warengruppen*.

Üblicherweise umfassen Referenzdaten-Komponenten etwa 10 bis 25 Geschäftsobjekte.

2.4 Konsequenzen aus der Komponentenbildung

Um die einzelnen Komponenten möglichst unabhängig zu entwerfen, wurden ganz bewusst Assoziationen zwischen zwei Geschäftsobjekten, die noch im alten unternehmensweiten Modell existierten, gekappt.

Wenn beispielsweise das Geschäftsobjekt *Bestellposition* aus der neuen Prozess-Komponente *Bestellung* eine Assoziation auf die Klasse *Währung* hatte, um anzugeben, in welcher Währung eine Bestellung vorgenommen wurde, so wurde diese Assoziation aus dem Modell entfernt, da das Geschäftsobjekt *Währung* in einer neuen Referenzdaten-Komponente enthalten ist.

Im Datenbank-Modell bleibt diese Relation jedoch weiterhin bestehen, um die referentielle Integrität zu gewährleisten.

In der Anwendungslogik hat man für das Geschäftsobjekt *Bestellposition* also nur einen Fremdschlüssel für *Währung*. Über diesen Schlüssel kann dann aber das Währungsobjekt aus einer zweiten Komponente nachgelesen werden.

Das Trennen von Assoziationen, über die man ja sonst sehr bequem über das gesamte Objektmodell navigieren konnte, scheint erst einmal etwas unverständlich. In der Praxis hat sich aber gezeigt, dass die Nachteile, Komponenten voneinander abhängig zu machen, weit größer sind als die Vorteile, über die Assoziationen navigieren zu können.

Zwischen zwei Geschäftsobjekten *einer* Komponente bleiben die Assoziationen selbstverständlich weiterhin bestehen.

3 Bildung von Teams

Ein ganz entscheidender Teil des Erfolgs bestand darin, die Verantwortung für die entstandenen Komponenten auf einzelne Entwickler bzw. kleine Teams zu verteilen. Jeder Entwickler hat so – je nach Größe des Unternehmens oder der jeweiligen Komponente – eine oder mehrere Komponenten, für die er ganz persönlich verantwortlich ist.

Dadurch besteht für den einzelnen Entwickler eine weitaus höhere Motivation, in seinen Komponenten für gute Qualität zu sorgen. Das muss nicht unbedingt bedeuten, dass nur dieser eine Entwickler allein Code in dieser Komponente erstellen darf. Seiner Verantwortung kann er auch dadurch nachkommen, dass er etwa das Design oder Code Audits durchführt.

Diese Delegation der Verantwortung schafft bereits einen Dienstleistungsgedanken, wo jeder Entwickler bestmöglichst Dienste („Services“) seiner Komponenten anbietet. Dies bietet den besten Nährboden für eine service-orientierte Architektur.

4 Identifizieren der Services

4.1 Entstehung der Services bei Alt-Anwendungen

Nachdem die Komponenten soweit gefunden und Geschäftsobjekte zugeordnet wurden, mussten nun Services identifiziert werden.

Dies können in einem ersten Evolutionsschritt erst einmal einfache Services sein, die lediglich Zugriff auf die Geschäftsobjekte bieten. Um eine größere Unabhängigkeit zu erreichen sollte der Zugriff allerdings nur auf eine Abstraktion der Geschäftsobjekte etwa in Form von *Value Objects* erlaubt werden. Dadurch kann in vielen Fällen bestehender Code, der seither direkt mit den Geschäftsobjekten gearbeitet hatte, deutlich schnell umgestellt werden auf die neuen Services.

In vielen Fällen konnten jedoch recht schnell viel Fachlogik in die neuen Komponenten übernommen werden, um so zu deutlich allgemeineren Schnittstellen zu kommen. Statt einer simplem Service-Methode „speichereNeuenAuftrag“, die sehr „persistenzlastig“ ist und dem Beispiel aus dem vorangehenden Absatz entspricht, hat man nun Service-Methode „erzeugeNeuenAuftrag“, der sämtliche Logik dieses Geschäftsprozesses kapselt.

Gerade bei der Umstellung von Systemen mit der alten Architektur werden durch die bestehenden Anwendungen die zu erzeugenden Service-Methoden schon sehr stark vorgegeben. Es ist aber selbstverständlich die Aufgabe des Architekten abzuwägen, wie speziell oder allgemein die Methoden zu entwerfen sind.

4.2 Entstehung der Services aus Anwendungsfällen

Es hat sich gezeigt, dass ein Service-orientiertes Vorgehen keine reine Aufgabenstellung der Entwickler ist. Vielmehr erreicht man eine deutlich höhere Qualität, wenn bereits bei der Ermittlung von neuen Anforderungen „service-orientiert“ gedacht wird.

Bei der Anforderungsanalyse für eine neue Anwendung sollten dabei die Use Cases bereits richtig „geschnitten“ werden und die Fachlichkeit eines einzelnen Ge-

schäftsvorfalles beschreiben. Dabei ist wichtig, diese Beschreibung unabhängig von der zu entwickelnden Anwendung zu schreiben, da man in der Regel ja auch anwendungsunabhängige und damit wiederverwendbare Software entwickeln möchte.

Um bereits bei der Analyse eine solch ideale Aufteilung der Use Cases für eine SOA zu erreichen, macht es Sinn, dass in einer so frühen Phase des Projektes bereits ein Software-Architekt diese Aufteilung mit begleitet. Das mag erst etwas ungewöhnlich erscheinen und danach aussehen, dass die fachlichen Beschreibungen sich bereits an der späteren Implementierung anlehnen sollen. Die Praxis hat aber gezeigt, dass beide Disziplinen ähnliche Kriterien für das Schneiden haben und die Erfahrungen aus dem Software-Development hinsichtlich Aufteilungen auch im Bereich der Anforderungsanalyse sehr wertvoll sein können.

Meist werden im Rahmen der Beschreibung von Use Cases auch Pre- und Postconditions ermittelt. Diese können häufig direkt verwendet werden, um den Vertrag für einen Service zu schreiben („Design by contract“).

Somit lassen sich Services zumindest schon grob identifizieren. Um allerdings ganz konkret die Signaturen von Services zu entwerfen, sind noch weit mehr Schritte zu unternehmen. Dabei gilt es unter anderem auf die Stabilität der Schnittstelle zu achten. Üblicherweise reicht es aber, wenn ein Architekt die Services identifiziert und grob entwirft. Die Details können ruhig in der Eigenverantwortung der Entwickler verbleiben.

Insgesamt hat sich dieses Use-Case-orientierte Vorgehen als sehr vorteilhaft erwiesen, da man dadurch nicht nur gut geschnittene Services erhält, die direkt einzelne Geschäftsprozesse abbilden, sondern auch noch synchron dazu die Anwendungsfälle in derselben Granularität.

5 Komponentensysteme

Nachdem nun Komponenten erstellt und Service-Methode identifiziert wurden, lag nun die jeweilige Fachlogik zu einem Geschäftsprozess in wiederverwendbaren Komponenten vor. Dieser scheinbar schon sehr gute Stand bildet aber noch nicht das Ende der Evolution, da auch hier noch entscheidende Nachteile vorliegen.

Die Fachlogik ist zwar wiederverwendbar für einzelne Anwendungen und Systeme, allerdings wurden die Fachkomponenten redundant überall dort eingesetzt, wo sie benötigt würden. Damit entsteht sehr leicht eine Situation, bei der nicht mehr wirklich überschaubar ist, wo welche Komponente in welcher Version verwendet wird. In den meisten Fällen gibt es aber zu einem Zeitpunkt nur einen gültigen Stand eines Geschäftsprozesses und damit kann es auch nur eine gültige Version einer Komponente geben.

Dies lässt sich über die Zentralität der Services lösen, bei der die Komponenten jeweils nur einmal zentral in einem Unternehmen zur Verfügung stehen. Wenn aus Gründen der Ausfallsicherheit die Services doch redundant vorliegen, so existieren sie aus Architektursicht doch nur einmal zentral. Sollten doch einmal mehrere Varianten eines Services gleichzeitig gültig sein, so lässt sich dies durch „Versionierung von Services“ erreichen.

Um die Services zentral zur Verfügung zu stellen, hat es sich als vorteilhaft erwiesen, mehrere ähnliche Komponenten zu einem Komponentensystem zusammenzufassen. So entsteht beispielsweise aus den Komponenten, die Services zu den verschiedensten Geschäftsprozessen aus dem Bereich Warenverteilung enthalten, tatsächlich ein Komponentensystem *Warenverteilung*.

Jedes Komponentensystem bekam eine zusätzliche Fassade in Form von grobgranularen Services, die verwendet werden, um über Systemgrenzen hinweg aufgerufen zu werden. Diese Fassade stellt eine weitere Abstraktionsschicht dar. Des Weiteren lässt sich so eine zusätzliche Sichtbarkeitschicht einführen, über die gesteuert wird, welche Services von Fremd-Komponentensystemen aufgerufen werden können.

6 Aktuelle Systemarchitektur

6.1 Überblick

Die Service-Landschaft besteht aus mehreren Komponentensystemen, die alle großen Fachbereiche des Unternehmens abbilden wie etwa Einkauf, Warenlieferung, Verkauf etc. Diese Systeme kommunizieren untereinander jeweils über grobgranulare Service-Methode der Komponentensystem-Fassaden.

Ein Komponentensystem setzt sich zusammen aus mehreren Komponenten (Prozesskomponenten oder Stammdatenkomponenten) zu einem bestimmten Fachbereich. Diese Komponenten kommunizieren untereinander über Services.

Die Komponenten wiederum enthalten die eigentlichen Services, wobei ein Service einen Geschäftsprozess abbildet.

Unverändert ist weiterhin das unternehmensweite Datenmodell. Hier wäre es evtl. in die Zukunft denkbar, auch hier Tabellenbereiche zu entkoppelt entsprechend der Assoziationen, die im Objektmodell zur Komponentenbildung gekappt wurde. Ein solcher Schritt ist aber derzeit weder geplant noch scheint er aktuell notwendig zu sein.

6.2 Technische Realisierung

Die Umsetzung fand in der Programmiersprache Java mit EJB als Komponententechnologie statt. Services wurden in Form von Stateless Session Beans implementiert und die Komponenten in Form von EJB-Jar-Files verpackt, wobei es jeweils ein Interface- und ein Implementierungs-Jar gab. Die Jar-Files der Komponenten wurden dann zu einem Enterprise Archive (EAR) zusammengebunden, welches damit ein Komponentensystem abbildet.

Auch die Fassaden eines Komponentensystems wurden als Session Beans realisiert, wobei hier ganz besonderer Wert auf die Modellierung der Schnittstellen gelegt wurde, um eine möglichst stabile API zu erhalten.

Bei den Services handelt es sich also um keine Web Services, sondern aus technischer Sicht eher um klassische EJB Programmierung. Dies mag erst einmal merkwürdig erscheinen, ist doch der Begriff SOA immer ganz eng mit Web Services verknüpft. Dies muss aber nicht immer so sein. Der Grund für die Verwendung von EJB ist historisch bedingt, da zu Beginn des Projektes die Web Services Implementierungen von Java noch in den Kinderschuhen steckten.

Unsere Erfahrung hat aber gezeigt, dass man mit den Prinzipien von SOA unabhängig von der zugrundeliegenden Technologie bessere Anwendungen entwerfen kann.

7 Fazit

Die Entwicklung einer service-orientierten Architektur ist nicht nur „auf der grünen Wiese“ möglich, sondern die Praxis hat gezeigt, dass sich sehr wohl eine komplette IT-Landschaft mit einem unternehmensweiten Datenmodell auf die neue Architektur umstellen lässt.

Dabei war ein evolutionäres Vorgehen sehr vorteilhaft, bei dem ganz pragmatisch Schritt für Schritt umgestellt werden konnte.

Neben all den neuen technischen Ansätzen waren auch organisatorische Erneuerungen notwendig, um etwa Komponentenverantwortlichkeiten zu delegieren und klare Zuständigkeiten zu definieren. Durch die hohe Eigenverantwortlichkeit eines Entwicklers für seine Komponente und die dadurch entstandene Motivation konnte ein sehr ausgeprägter Dienstleistungsgedanke entstehen. Dieser gelebte Dienstleistungsgedanke bildet die Basis für eine Service-Landschaft.