Eine Normalform für Services

Bernhard Humm, Oliver Juwig[1]

sd&m Research
Carl-Wery-Straße 42
81739 München
{Bernhard.Humm, Oliver.Juwig}@sdm.de

Abstract: Ein guter Service-Entwurf ist entscheidend für den Erfolg einer serviceorientierten Architektur (SOA). Aber was bedeutet „gut" in diesem Kontext? In diesem Papier werden konkrete Kriterien für den Service-Entwurf angegeben, welche die lose Kopplung von Komponenten fördern. Diese Kriterien wurden aus umfangreicher industrieller Projektpraxis destilliert. Eines dieser Kriterien, die Normalform für Services, wird formal definiert und anhand von Beispielen erläutert.

1 Einleitung

Das Schlagwort der *serviceorientierten Architektur (SOA)* (zum Beispiel [RHS05]) beherrscht gegenwärtig die Diskussion um die Gestaltung unternehmensweiter IT-Anwendungslandschaften. Unternehmen erwarten sich von einer SOA eine höhere Flexibilität ihrer Geschäftsprozesse und die Reduktion von IT-Kosten. Die Erreichung dieser Ziele ist möglich, die Einführung einer SOA erfordert jedoch neben guter Planung auch eine klare Vorstellung der fachlichen Zielarchitektur.

In seinem Artikel „Wann liefert eine serviceorientierte Architektur echten Nutzen?" [Ric05] arbeitet Richter heraus, dass zur Erreichung dieser Ziele der Aufbau einer guten fachlichen Architektur für eine IT-Anwendungslandschaft entscheidend ist.

Aber was bedeutet „gut" in diesem Kontext? Wie ist die fachliche Architektur einer IT-Anwendungslandschaft zu gestalten? In der SOA-Literatur (zum Beispiel [Woo04, KBS04, Erl04]) werden dazu allgemeine Hinweise gegeben, vor allem zum zentralen Konzept der *losen Kopplung*. Leider stehen jedoch sowohl die Forschung als auch die Industrie noch am Anfang, wenn es um konkrete, nachprüfbare Kriterien für den Entwurf von Services geht.

Um die häufig verwendete Analogie zwischen Software-Architektur und der Architektur von Gebäuden anzuwenden: es fehlt in der Software-Architektur nach Prinzipien der Statik, mit dessen Hilfe Architekten im Voraus die Tragfähigkeit einer Lösung berechnen können.

[1] Wir danken herzlich Herrn Prof. Armin B. Cremers, Universität Bonn, für wertvolle Beiträge zu diesem Papier.

In diesem Papier stellen wir konkrete Kriterien für den Entwurf von Services vor, die die lose Kopplung von Komponenten fördern. Diese Kriterien haben wir destilliert aus der industriellen Erfahrung von über zehn SOA-Vorhaben in verschiedenen Branchen mit einem Gesamtvolumen von über 100 Bearbeiterjahren.

Das Papier ist wie folgt aufgebaut. In Abschnitt 2 werden die Kriterien vorgestellt. In den Abschnitten 3-6 wird eines der Kriterien, die Normalform von Services, formalisiert. Abschnitt 7 zieht ein Fazit und gibt einen Ausblick.

2 Komponenten und Services

2.1 Komponenten

Die primäre Strukturierung einer IT-Anwendungslandschaft erfolgt nach fachlichen *Komponenten*. Für Komponenten existiert keine allgemein anerkannte Definition. Siedersleben definiert in [Sie04] eine Komponente als wesentliche Einheit des Entwurfs, der Implementierung und Planung. Sie exportiert und importiert Schnittstellen und verbirgt ihre Implementierungsdetails. Sie kann hierarchisch aufgebaut sein.

Diese Definition – wie viele andere Definitionen auch (zum Beispiel [Szy02]) – ist anwendbar für fachliche und technische Komponenten, sowohl im Großen als auch im Kleinen. Im Kontext einer SOA betrachten wir fachliche Komponenten im Großen, also ganze Anwendungen oder Domänen.

2.2 Services

Die Schnittstellen von Komponenten im Kontext einer SOA nennt man *Services*. Ein Service ist eine definierte fachliche Funktionalität, die als Bestandteil eines größeren Verarbeitungsablaufs verwendet werden kann. Als solcher stellt ein Service einen Teil der abstrakten Sicht auf die exportierende Komponente dar und verbirgt alle Implementierungsdetails. Die Definition eines Service hat den Charakter einer vertraglichen Übereinkunft zwischen Service-Anbieter und Service-Nutzer (*Service Level Agreement*). Ein Service besteht aus *Service-Operationen*.

2.3 Kriterien zum Entwurf von Services

Services sollten lose gekoppelt werden – das ist eines der Grundprinzipien der SOA. Diese Forderung nach einer losen Kopplung der einzelnen Komponenten einer Anwendungslandschaft lassen sich nach unseren praktischen Projekterfahrungen in den nachfolgenden Kriterien für die Gestaltung der Operationen von Services präzisieren.

1. *Grobgranular*: Wenige Aufrufe, die viel bewirken, sind besser als viele Aufrufe, die erst zusammen den gewünschten Effekt erzielen. Daher sind wenige mächtige – *grobgranulare* – Serviceoperationen besser als viele kleine.

2. *Vollständig und redundanzfrei (normal):* Schnittstellen sind vollständig, wenn ihre Operationen die gesamte Funktionalität einer Komponente abdecken: Abfragen (lesende Operationen) und Kommandos (schreibende Operationen). Redundanzfrei bedeutet, dass weder in Abfragen noch in Kommandos dieselbe Arbeit an verschiedenen Stellen gemacht wird. Sind Schnittstellen vollständig und redundanzfrei, so bezeichnen wir sie als *normal*.

3. *Idempotent:* Idempotente Serviceoperationen zeichnen sich dadurch aus, dass ein mehrmaliger Aufruf mit denselben Parametern denselben Effekt hat wie der einmalige. Die Idee kommt aus der Batch-Verarbeitung: Man baut Batches nach Möglichkeit so, dass ein versehentlich angestoßener zweiter Lauf kein Unheil anrichtet. Idempotenz wirkt sich positiv auf die Stabilität von Anwendungen aus.

4. *Kontextfrei (Transaktionen):* Serviceoperationen werden stets als Transaktionen [GR93], das heißt, ganz oder gar nicht ausgeführt. Wir fordern jedoch, dass Serviceoperationen *nicht* in anwendungsübergreifende Transaktionen integriert werden. Anwendungsübergreifende Transaktionen koppeln Anwendungen eng aneinander – das widerspricht dem Prinzip der losen Kopplung. Kontextfreiheit in Bezug auf Transaktionen muss erkauft werden durch die Implementierung von kompensierenden Operationen, die im Fehler- oder Stornierungsfall aufgerufen werden können.

5. *Kontextfrei (Sessions):* Die gerufene Anwendung kennt keine Benutzer und keine Sessions. Sie reagiert auf Operationsaufrufe, woher auch immer sie kommen. Der Aufrufer stellt – falls erforderlich – die Verbindung zwischen seinen Aufrufen her, nicht die gerufene Anwendung. Der Aufrufer oder die Systemumgebung stellen die Prüfung von Berechtigungen sicher. Auch die Kontextfreiheit in Bezug auf Sessions hat den Zweck, Rufer und Gerufenen zu entkoppeln.

6. *Technikneutral:* Die Schnittstelle verrät nichts über die eingesetzte Implementierungstechnik. Technische Schlüssel werden beispielsweise niemals nach außen gegeben.

2.4 Beispiel

Wir verdeutlichen die Kriterien anhand eines Praxisbeispiels. Die Versicherungs-Anwendungs-Architektur [VAA] ist eine Referenzarchitektur des Gesamtverbands der Deutschen Versicherungswirtschaft. Dort werden fachliche Komponenten und ihre Serviceoperationen (dort bezeichnet als *Anwendungsfälle*) spezifiziert. Für die Komponente `Partner` ist eine Serviceoperation `Partner ändern` definiert, aber interessanterweise weder eine Operation `Partner anlegen` noch eine Operation `Partner löschen`. Die Operation `Partner ändern` beinhaltet laut der Spezifikation in [VAA] die folgende Funktionalität:

1. Prüfung, ob der Partner bereits besteht. Falls nicht: Neuanlage

2. Änderung der Eigenschaften des Partners. Dies kann auch das logische Löschen des Partners beinhalten
3. Validierung der Änderungen
4. Dublettenprüfung: bei Identifikation von Dubletten, Anstoßen der Operation `Dubletten zusammenlegen`

Dieser Service-Entwurf überrascht auf den ersten Blick, da doch augenscheinlich mehrere, unabhängige Operationen zusammengefasst werden. Bei genauer Prüfung stellt sich jedoch heraus, dass alle obigen Kriterien erfüllt werden:

1. *Grobgranular*: Die Operation umfasst nicht nur alle möglichen Attributänderungen eines Partners, sondern auch das Anlegen und logische Löschen.
2. *Vollständig und redundanzfrei (normal)*: Zwei separate Operationen für das Anlegen und das Ändern von Partnern hätten Redundanzen, zum Beispiel in der Validierungslogik. Durch das Zusammenlegen der Funktionalität wird diese Redundanz vermieden. Die Schnittstelle ist auch vollständig, da sowohl das Anlegen, das Ändern wie auch das logische Löschen möglich sind – das physische Löschen von Partnern ist fachlich nicht vorgesehen. Daher ist die Schnittstelle in Normalform.
3. *Idempotent*: Die Dublettenprüfung stellt die Idempotenz der Operation auch bei einer Neuanlage sicher. Beim ersten Aufruf der Operation wird der Partner neu angelegt. Bei jedem weiteren Aufruf mit identischen Parametern identifiziert die Dublettenprüfung dasselbe Partnerobjekt. Die Funktionalität für das Ändern und logische Löschen ist von Natur aus idempotent.
4. *Kontextfrei (Transaktionen)*: Die Operation `Partner ändern` kann selbst als kompensierende Operation für einen fehlgeschlagenen Aufruf von `Partner ändern` ausgeführt werden. Bei einer Attributänderung müssen die ursprünglichen Attributwerte als Parameter mitgegeben werden. Bei einer Neuanlage muss der Partner logisch gelöscht werden und umgekehrt. Mit Hilfe dieser kompensierenden Aktionen können anwendungsübergreifende Transaktionen vermieden werden.
5. *Kontextfrei (Sessions)*: Die Operation kennt keine Benutzer und keine Sessions.
6. *Technikneutral*: Die Spezifikation der Operation verrät nichts über die eingesetzte Implementierungstechnik.

So erfüllt `Partner ändern` gleichzeitig alle Kriterien für einen guten Service-Entwurf. In der Projektpraxis ist dies jedoch selten der Fall – nicht zuletzt deshalb, da die Kriterien häufig untereinander oder mit anderen Anforderungen in Konflikt stehen. So muss bisweilen die Grobgranularität durch eine verschlechterte Performance erkauft werden – hier sind Kompromisse zu finden.

In den folgenden Abschnitten formalisieren wir das Kriterium der Normalform für Services. Wir beginnen mit der Definition einiger Grundbegriffe.

3 Definition von Grundbegriffen

3.1 Komponenten, Services und ihre Operationen

Definition: Ein *Komponententyp* oder kurz eine *Komponente C* definieren wir durch die Menge der Services *{Service$_1$, ..., Service$_n$}*, die ihre Aufrufschnittstelle bilden. Im Kontext dieses Papiers ist nur die Aufrufschnittstelle von Komponenten relevant.

Beispiel: *C* = `PartnerVerwaltung`, *Service$_1$* = `PartnerPflege`

Definition: Ein *Service* fasst eine Menge von *Operationen {o$_1$, ... o$_n$}* zusammen. Die Signatur einer Operation *o* ist definiert durch Eingabeparameter und ihre Typen und optional den Typ eines Rückgabewerts: *ResultType m(Type$_1$ param$_1$, ... Type$_n$ param$_m$)*.
Bei einem *Operationsaufruf o(param$_1$, ..., param$_n$)* → *result* werden die aktuellen Parameter an die formalen Parameter gebunden, *m* ausgeführt und eventuell *result* zurückgegeben. Der Einfachheit halber subsumieren wir Ausnahmen unter Rückgabewerten.

Beispiel: Operation `void legePartnerAn(Number id, String name);`
Aufruf `legePartnerAn(4711, "Müller")`

3.2 Komponenten-Instanzen und ihre Zustände

Definition: Eine Komponente *C* kann beliebig viele *Komponenten-Instanzen c* haben. Zu jedem Zeitpunkt *t* hat *c* einen definierten Zustand s_t. Hier geht es um den externen, also nach außen sichtbaren fachlichen Zustand von *c*. Interne beziehungsweise technische Zustände werden hier nicht betrachtet, da sie sich sichtbar nur über ihren externen fachlichen Zustand repräsentieren.
Ein *Zustand* s_t ist eine Menge von Aussagen *{i$_1$, i$_{2, ...}$}*, die zum Zeitpunkt *t* für *c* gültig sind.

Beispiel: *i$_1$ = ∃ Partner #4711; i$_2$ = Partner #4711 hat Name „Müller“;*
i3 = ¬∃ Partner #4712

Wie das Beispiel verdeutlicht, kann der Zustand einer Komponenten-Instanz unendlich viele Aussagen beinhalten. Das Zustandskonzept ist nicht beschränkt auf Komponenten, die Objekte wie zum Beispiel Partner beinhalten. Auch der Zustand von Komponenten-Instanzen, die rechnen, wie zum Beispiel ein Währungsrechner oder ein Fourier-Transformator, können damit beschrieben werden.

Definition: Für jede Komponente *C* sei eine Menge *S = {s$_1$, s$_2$, ...}* aller *erlaubter Zustände* definiert.

Beispiel: Angelegte Partnerobjekte müssen eine gültige Postleitzahl haben.

3.3 Abfragen und Kommandos

Wir unterscheiden zwei Arten von Operationen: *Abfragen q* und *Kommandos cmd*.

Definition: *Abfragen* sind lesende Operationen: *ResultType q(Type$_1$ param$_1$, ... Type$_n$ param$_n$).* Ein Aufruf von *q* verändert den Zustand einer Komponenten-Instanz *c* nie: $s_t = s_{t+1}$ bei Aufrufzeitpunkt *t*.
Der Aufruf von *q* mit seinen Eingabeparametern *param$_1$, ... param$_n$* und dem Rückgabewert *result* kann *interpretiert* werden als eine Teilmenge des Komponentenzustands s_t. Wir führen dazu die *Interpretation I* ein und schreiben $I(result) \subseteq s_t$.

Beispiel: `findePartner(4711)` → `result` mit `result.name = "Müller"`. Dann ist die Interpretation $I(result) = \{i_1, i_2\}$ mit
$i_1 = \exists$ *Partner #4711;* i_2 = *Partner #4711 hat Name „Müller“.*

Definition: *Kommandos* sind schreibende Operationen:
void cmd(Type1 param$_1$, ... Type$_n$ param$_n$). Ein Aufruf von *cmd(param$_1$, ... param$_n$)* zum Zeitpunkt *t* führt die Komponenteninstanz *c* vom Zustand s_t in den neuen Zustand s_{t+1} über. Die *Zustandsänderung* Δs des Aufrufs von *cmd(param$_1$, ... param$_n$)* auf *c* zum Zeitpunkt *t* ist eine Menge von Aussagen $\Delta s = s_{t+1} \setminus s_t$

Beispiel: `void legePartnerAn(4711, "Müller")`. $\Delta s = \{i_1, i_2\}$ mit
$i_1 = \exists$ *Partner #4711;* i_2 = *Partner #4711 hat Name „Müller“.*

Mischformen zwischen Abfragen und Kommandos, das heißt Abfragen mit Seitenenffekten, betrachten wir nicht – zum einen aus Gründen der Einfachheit, zum anderen da es guter Stil ist, auch in der Programmierung Abfragen und Kommandos strikt zu trennen.

Eine Komponenteninstanz mit Zustand, Abfragen und Kommandos entspricht dem Konzept der *Datenräume (data spaces)* aus [CH78a-c]. In den folgenden Abschnitten formalisieren wir das zweite Kriterium zum Entwurf von Services aus Abschnitt 2.3.

4 Normalform für Services

4.1 Vollständige Abfragemengen

Definition: Die Menge der Abfragen $Q = \{q_1, ... q_n\}$ einer Komponente *C* ist *vollständig*, wenn zu jedem Zeitpunkt *t* der gesamte Zustand einer Komponenteninstanz *c* erfragt werden kann: $\cup_{I(result)} = s_t$ mit $q_i(param_{i1}, ... param_{im}) \rightarrow result$ für alle Abfragen q_i $(1 \leq i \leq n)$ und alle möglichen Parameterbelegungen $param_{ij}$ $(1 \leq j \leq m)$.

Beispiel: In einer einfachen Partnerverwaltung (Anlegen, Ändern und Löschen von Partnern) ist die Abfrage `Partner findePartner(Number id)` vollständig: durch Aufzählen aller möglichen IDs können alle Partnerdaten gelesen werden.

4.2 Vollständige Kommandomengen

Definition: Die Menge der Kommandos *CMD* einer Komponente *C* ist *vollständig*, wenn alle erlaubten Zustände $s \in S$ aller Komponenteninstanzen *c* stets erreicht werden

können: für je zwei Zustände s_{start}, $s_{end} \in S$ gibt es eine Folge von Kommandos $cmd_1 \circ ... \circ cmd_n$ $(cmd_i \in CMD, 1 \leq i \leq n)$ mit Parameterbelegungen $param_{ij}$, so dass c schrittweise von s_{start} in s_{end} überführt wird. Dabei bezeichnet $\circ$ die Hintereinanderausführung von Kommandos. Mehrfachanwendungen und Weglassen einzelner Kommandos aus *CMD* sind möglich.

Beispiel: In einer einfachen Partnerverwaltung ist die Menge der folgenden Kommandos vollständig:

```
void legePartnerAn(Number id, String name);
void aenderePartner(Number id, String name);
void loeschePartner(Number id)
```

4.3 Orthogonale Abfragemengen

Definition: Die Abfragen $q_1, ..., q_n$ einer Komponente C sind *orthogonal*, wenn sie jeweils unterschiedliche Aussagen liefern:
$I(result_i) \cap I(result_j) = \varnothing \; (1 \leq i, j \leq n; \; i \neq j)$ mit $q_i(param_{i1}, ... \; param_{im}) \rightarrow result_i$; $param_{ij}$ beliebig; Zustand s_t beliebig.

Beispiel: Die beiden folgenden Abfragen sind orthogonal:

```
String gibName(Number partnerID);
Date gibGeburtsdatum(Number partnerID)
```

4.4 Orthogonale Kommandomengen

Definition: Die Kommandos $CMD = \{cmd_1, ..., cmd_n\}$ einer Komponente C sind *orthogonal*, wenn sie jeweils unterschiedliche Zustandsübergänge bewirken: $\Delta_i \cap \Delta_j = \varnothing \; (1 \leq i, j \leq n; \; i \neq j)$ mit Δ_i ist die Zustandsänderung bei einem Aufruf von cmd_i auf einer Komponenten-Instanz c bei beliebigen Parameterbelegungen $param_{i1}, ... \; param_{im}$ und beliebigem erlaubten Zustand s.

Beispiel: Die folgenden beiden Kommandos sind *nicht* orthogonal:

```
void legePartnerAn(Number id, String name)
void aenderePartner(Number id, String name)
```

Begründung: Bei den Aufrufen `legePartnerAn(4711, "Müller")` und `aenderePartner(4711, "Müller")` enthält die Zustandsänderung beider Operationsaufrufe die Aussage *Partner #4711 hat Name "Müller"* (Annahme: beide Operationsaufrufe bewirken wirklich etwas, das heißt sie brechen nicht mit einer Ausnahme ab und vor dem Aufruf von `aenderePartner` war der Name ungleich `"Müller"`).

4.5 Normalform

Definition: Operationen einer Komponente C sind sind *in Normalform* oder kurz sind *normal,* wenn sowohl die Teilmenge der Abfragen Q als auch die Teilmenge der

Kommandos *CMD* vollständig und orthogonal sind.
Eine normale Teilmenge der Operationen einer Komponente *C* nennen wir eine *Basis.*

Als Beispiele dienen die obigen Beispiele für vollständige und orthogonale Abfragen- und Kommandomengen.

Wir empfehlen nicht uneingeschränkt, dass alle Services einer SOA in Normalform vorliegen müssen. Unsere Architekturempfehlungen geben wir in Abschnitt 6. Zuvor benötigen wir jedoch noch weitere Definitionen, um sinnvolle Ableitungen durchführen zu können.

5 Ableitungen

In der Praxis sind Operationen häufig nicht orthogonal: sie umfassen die Funktionalität anderer Operationen, zum Beispiel weil sie diese in der Implementierung aufrufen. Um diesen Sachverhalt zu formalisieren, benötigen wir Definitionen für abgeleitete Abfragen und Kommandos.

5.1 Abgeleitete Abfragen

Definition: Eine Abfrage *q2* ist *abgeleitet* von einer Abfrage *q1 (q2 ≥ q1)*, wenn *q2* mindestens alle Aussagen von *q1* liefert: $I(result_2) \supseteq I(result_1)$ mit $q_i(param_{i1}, ..., param_{in}) \rightarrow result_i$ *(i = 1, 2)* für alle möglichen Parameterbelegungen $param_{i1}, ..., param_{in}$ und in allen erlaubten Zuständen $s \in S$.

Beispiel:
`Partner findePartner(Number id) ≥ String gibName(Number id)`
unter der Annahme, dass mit `partner.name` der Partnername erfragt werden kann.

5.2 Abgeleitete Kommandos

Definition: Ein Kommando cmd_2 ist *abgeleitet* von einem Kommando cmd_1 $(cmd_2 \geq cmd_1)$, wenn cmd_2 mindestens alle Zustandsänderungen von cmd_1 durchführt:
$\Delta_2 \supseteq \Delta_1$ mit Δ_i ist die Zustandsänderung der Ausführung von $cmd_i(param_{i1}, ..., param_{in})$ *(i = 1, 2)* zum Zeitpunkt *t* bei jedem erlaubten Zustand *s* und jeder Parameterbelegung $param_{i1}, ..., param_{in}$.

Beispiel:
```
void legePartnerAn(Number id, String name)
  ≥ void legePartnerAn(Number id)
void legePartnerAn(Number id, String name)
  ≥ void aenderePartner(Number id, String name)
```

5.3 Theorem orthogonal ⇒ ableitungsfrei

Theorem: Orthogonale Abfragen sind niemals voneinander abgeleitet. Das gleiche gilt für orthogonale Kommandos:

1. Abfragen:
   Sei Q = {q_1, …, q_n} die orthogonale Abfragemenge einer Komponente C. Dann gilt: $\neg\exists\, q_i, q_j\ (1 \le i, j \le n, i \ne j)$: $q_i \ge q_j$.
2. Kommandos:
   Sei $CMD = \{cmd_1, ..., cmd_n\}$ die orthogonale Kommandomenge einer Komponente *C* Dann gilt: $\neg\exists\, cmd_i, cmd_j\ (1 \le i, j \le n, i \ne j)$: $q_i \ge q_j$.

Beweis:

1. Abfragen: Annahme $\exists\, q_i, q_j\ (1 \le i, j \le n, i \ne j)$: $q_i \ge q_j$. Dann $I(result_i) \supseteq I(result_j)$ (gemäß Definition Ableitung) ⇒ Widerspruch zu Definition orthogonal.
2. Kommandos: Annahme $\exists\, cmd_i, cmd_j\ (1 \le i, j \le n, i \ne j)$: $cmd_i \ge cmd_j$. Dann $\Delta_i \supseteq \Delta_j$ (gemäß Definition Ableitung) ⇒ Widerspruch zu Definition orthogonal.

5.4 Minimale Vollständigkeit

Definition: Eine Menge von Abfragen *Q* und Kommandos *CMD* einer Komponente *C* heißt *minimal vollständig*, wenn man keine einzige Operation weglassen kann, ohne die Vollständigkeit zu verlieren.

5.5 Minimalitätstheorem

Theorem: Jede Basis einer Komponente ist minimal vollständig:
Sei *Q* eine normale Abfragemenge und *CMD* eine normale Kommandomenge einer Komponente *C* (Basis von *C*). Dann ist jede echte Teilmenge $Q' \subset Q$, $CMD' \subset CMD$ nicht mehr vollständig.

Beweis:

1. Abfragen:
   Sei *Q* vollständig und orthogonal und $Q' \subset Q$. Annahme: *Q'* sei vollständig. Sei $q_i \in Q \setminus Q'$ (eine vermeintlich überflüssige Abfrage). Dann kann $I(result_i)$ nicht mehr erfragt werden (da *Q* orthogonal) ⇒ *Q'* nicht vollständig ⇒ Widerspruch.
2. Kommandos:
   Sei *CMD* vollständig und orthogonal und $CMD' \subset CMD$. Annahme: *CMD'* sei vollständig. Sei $cmd_i \in CMD \setminus CMD'$ (ein vermeintlich überflüssiges Kommando). Dann kann Δ_i nicht mehr erreicht werden (da *CMD* orthogonal) ⇒ *CMD'* nicht vollständig ⇒ Widerspruch.

6 Architekturempfehlung und Beispiel

Auf der Basis der Definition der Normalform und ihren Ableitungen lauten nun unsere Empfehlungen für den Entwurf von Services in einer SOA wie folgt:

1. Trenne klar Abfragen von Kommandos
2. Spezifiziere für jede Komponente eine Basis von Serviceoperationen, also Abfragen und Kommandos in Normalform
3. Bei Bedarf spezifiziere abgeleitete Operationen, die die Benutzerfreundlichkeit, Performance und / oder andere nicht-funktionale Eigenschaften verbessern. Diese Operationen müssen nicht zwingend die Trennung zwischen Abfragen und Kommandos durchhalten. Trenne klar zwischen Basis und abgeleiteten Operationen, das heißt weise abgeleitete Operationen separat aus.

Die Spezifikation einer Basis von Serviceoperationen für eine Komponente führt zu Redundanzfreiheit bei ausreichend mächtiger Funktionalität – das ergibt sich direkt aus der Definition der Normalform und dem Minimalitätstheorem. Damit sind die Operationen einer Basis ideal wieder verwendbar. Leider sind Serviceoperationen realer Komponenten in der industriellen Praxis selten in Normalform. Hier hilft die klare Definition zu einem besseren Entwurf.

Auf der anderen Seite sind abgeleitete Operationen in der Projektpraxis essentiell. Oft ist es notwendig, Operationsaufrufe aufgrund von Infrastrukturkosten zusammen zu legen oder häufig nacheinander ausgeführte Operationen in einer Serviceoperation zu aggregieren. Die strikte Trennung zwischen Basis und Ableitungen hilft jedoch, in der Implementierung Redundanzen zu vermeiden oder zumindest zu kontrollieren.

Siehe dazu das folgende Codebeispiel:

```
Service PartnerPflege {
  // Basis: Kommandos
  void legePartnerAn(Number id)
  void aenderePartner(Number id, String name)
  // Basis: Abfragen
  Partner findePartner(Number id)
  // abgeleitete Operationen
  Partner legePartnerAn(number id, String name)
}
```

Interessant an diesem simplen Beispiel ist die Operation `Partner legePartnerAn(number id, String name)`. Es ist die übliche Operation zum Anlegen neuer Objekte: Attributbelegungen (hier am Beispiel des Namens) als Eingabe und erzeugtes Objekt als Ausgabe. Bei genauem Hinsehen stellt man jedoch fest, dass die Operation eine Mischung aus Kommando und Abfrage ist und von allen drei Basisoperationen abgeleitet ist. Und in der Tat besteht hier die Gefahr von unerwünschter Redundanz, da mit der Belegung von Attributen in der Regel Plausibilisierungen, zum Beispiel auf die Gültigkeit von Adressen, verbunden sind.

Daher ist es ratsam, bei der Implementierung dieser Operation sich auf die Implementierung von `aenderePartner`, von der sie abgeleitet ist, abzustützen.

7 Fazit und Ausblick

Nur mit einer guten fachlichen Architektur kann eine SOA den gewünschten Nutzen der Flexibilität von Prozessen und Reduktion von IT-Kosten erzielen. Aber was bedeutet „gut“ in diesem Kontext? Lose Kopplung ist eines der zentralen Prinzipien, aber dieses Prinzip allein ist noch wenig konkret anwendbar und überprüfbar. In diesem Papier haben wir daher konkrete Kriterien für einen guten Service-Entwurf angegeben.

Das Kriterium der Normalform haben wir formalisiert, was eine Prüfung dieses Kriteriums im konkreten Projektkontext erlaubt. Die theoretischen Grundlagen für diese Formalisierung wurden schon in den 1970er Jahren mit dem Konzept der Datenräume gelegt [CH78a-d]. Neu in diesem Papier ist die Erweiterung der Konzepte der Vollständigkeit und Orthogonalität von Anfragen auf Kommandos und die Anwendung dieser Konzepte als Entwurfsempfehlungen für eine SOA.

Der Schnittstellen-Entwurf von Serviceoperationen nach den Regeln der strikten Trennung von Basis und Ableitung hat sich in unserer Projektpraxis seit Jahren bewährt. Zuerst intuitiv ausgeführt und als gutes Design bezeichnet, wurde er angeregt durch die Diskussion über die SOA von uns formalisiert. In Reinform wird die in diesem Papier formulierte Normalform jedoch in keinem Projekt umgesetzt. Dies ist vergleichbar mit der Normalform im Datenbankentwurf, mit der eben auch in der Praxis aus wohlüberlegten Gründen gebrochen wird. Als Entwurfsideal steht die Normalform jedoch immer am Anfang aller Überlegungen.

Die anderen genannten Prinzipien lassen sich mehr oder weniger auf Basis der vorgestellten Begriffsdefinitionen formalisieren:

1. *Grobgranular*: Die Granularität einer Abfrage lässt sich durch die Menge der Aussagen bestimmen, welche aus deren Aufrufen interpretiert werden können. Bei Kommandos ist es die Menge der Aussagen der Zustandsänderung. Abgeleitete Operationen sind in ihrer Granularität gröber als Operationen, von denen abgeleitet wurde.

2. *Idempotent:* Ein Kommando ist idempotent, wenn bei einem zweiten Aufruf (und damit jedem weiteren Folgeaufruf) mit identischen Parametern die Menge der Zustandsänderungen leer ist.

3. *Kontextfrei (Transaktionen):* Das Konzept von Transaktionen erweitert das Konzept von Datenräumen insofern, dass Maschinen angenommen werden, auf denen Operationen ausgeführt werden. Diese Maschinen erlauben Parallelität und können Fehlern unterliegen. Das Kriterium lautet, keine serviceübergreifenden Transaktionen zu verwenden. Die Formalisierung dieses Kriteriums ist nicht trivial.

4. *Kontextfrei (Sessions):* Eine Benutzer-Session ist im Wesentlichen eine Historie von Operationsaufrufen. Ansätze für eine Formalisierung bieten Historien-Konzepte für Datenräume [CH91].

5. *Technikneutral:* Bei diesem Kriterium geht es um die Trennung der Fachlichkeit von der implementierenden Technik (siehe auch [Sie04]). Ansätze für eine Formalisierung bieten Konzepte zur Implementierung von Datenräumen [CH78d].

Klar ist: Auch wenn diese Kriterien formalisiert sind, sind in jedem konkreten Projekt weiterhin viele Entwurfsentscheidungen vom Software-Architekten zu treffen. So hilft beispielsweise die Formalisierung der Granularität einer Operation nur bedingt, die *adäquate* Granularität zu finden. Dies wird auch dadurch deutlich, dass die Kriterien oft sowohl untereinander, als auch zu anderen Systemanforderungen im Konflikt stehen.

Um in der Analogie zur Architektur von Gebäuden zu bleiben: die Regeln der Statik ersetzen den Architekten nicht, aber sie helfen ihm, tragfähige Lösungen zu entwickeln. So ist auch dieser Beitrag zu sehen als ein Beitrag zur Statik der Software-Architektur. Viele weitere müssen noch folgen.

Literaturverzeichnis

[CH78a] Cremers, A. B.; Hibbard, N: Orthogonality of Information Structures. Acta Informatica 9, S. 243 – 261. Springer-Verlag, 1978.

[CH78b] Cremers, A. B.; Hibbard, N: Functional Behaviour in Data Spaces. Acta Informatica 9, S. 293 - 307. Springer-Verlag, 1978.

[CH78c] Cremers, A. B.; Hibbard, N: Data Spaces with Indirect Adressing. Mathematical Systems Theory 12, S. 151 - 173. Springer-Verlag, 1978.

[CH78d] Cremers, A. B.; Hibbard, N: Formal Modeling of Virtual Machines, IEEE Transactions on Software Engineering 4, S. 426-436, 1978

[CH91] Cremers, A. B.; Hibbard, N: Axioms for Concurrent Processes, Lecture Notes in Computer Science 555, S. 54-68 G. Springer Verlag, 1991.

[Erl04] Thomas Erl: Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services. Prentice Hall PTR. April 2004

[GR 93] Jim Gray, Andreas Reuter: Transaction Processing : Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems Morgan Kaufmann. 1st edition 1993

[KBS04] Dirk Krafzig, Karl Banke, Dirk Slama: Enterprise SOA : Service-Oriented Architecture Best Practices. The Coad Series. Prentice Hall PTR. November 2004

[RHS05] Jan-Peter Richter, Harald Haller, Peter Schrey: Aktuelles Schlagwort Serviceorientierte Architektur. Informatik-Spektrum. Springer-Verlag, 2005.

[Ric05] Jan-Peter Richter: Wann liefert eine serviceorientierte Architektur echten Nutzen? Proceedings Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen, S. 231-242.

[Sie04] Johannes Siedersleben: Moderne Software-Architektur – umsichtig planen, robust bauen mit Quasar. dpunkt Verlag. 2004.

[Szy02] C. Szyperski: Component Software. Addison Wesley, 2002.

[VAA] Gesamtverband der Deutschen Versicherungswirtschaft e.V.: Versicherungs-Anwendungs-Architektur (final edition). http://www.gdv-online.de/vaa/

[Woo04] Dan Woods: Enterprise Services Architecture. Galileo Press. Bonn, 2004.