Integration von Webservices
in Applikationen

Integration von Webservices in Applikationen 1
1. Servicemodelle 1
  1.1. Utility-Service 2
  1.2. Business-Service 2
  1.3. Controller-Service 3
2. Modellierung Serviceorientierter Komponenten 3
  2.1. Probleme in vielen Projekten 4
  2.2. XWIF-Prozess für Komponenten Design 4
    2.2.1. Businesskomponenten zusammen tragen 4
    2.2.2. Identifizieren der Business-Logik für die Webservices nur beschränkt nutzbar sind 5
    2.2.3. Identifizieren von Wiederverwenbarkeit 6
    2.2.4. Identifizieren von Interfaceabhängigkeiten 6
    2.2.5. Bestimme Ebenen der Zusammenarbeit 6
    2.2.6. Erstellen von feineren Aufgabenbezogenen Komponenten 7
    2.2.7. Gruppieren der Methoden auf Grund ihres Typs 8
    2.2.8. Bestimmen der Kandidaten die für eine SOA relevant sind 8
    2.2.9. Betrachten der Kandidaten die nicht für SOA geeignet sind 8
    2.2.10. Zusammenführen von Klassenübergreifenden Aufgaben die zusammengehören 8
  2.3. XWIF-Prozess für Service-Interface Gestaltung 9
    2.3.1. Servicemodel wählen 9
    2.3.2. Bestimmen des Anwendungsbereiches der Geschäftslogik 10
    2.3.3. Bestimmen potentieller Klienten 10
    2.3.4. Bestimmen der benötigten Daten 10
    2.3.5. Prüfen der gegenseitigen Aufrufe 11
    2.3.6. Bestimmen der benötigten Komponenten 11
    2.3.7. Modellieren des Serviceinterfaces 12
    2.3.8. Visualisieren der Interaktionsszenarien 12
    2.3.9. Design der Nachrichtenstruktur 13
    2.3.10. Servicemodel verfeinern 13
3. Strategien zur Integration von SOA 14
  3.1. Definition Konsistenter Kriterien 14
  3.2. Parameter vs. Methodengesteuerte Schnittstellen 14
  3.3. Design gemischter Granularitäten 14
  3.4. Separate Standards für Interne und Externe Services 15
  3.5. Webservices Dritter 16
4. Modellierung von Service-Kompositionen 16
5. Erweiterungen der Servicefunktionalität 17
6. Integration von SOAP-Nachrichten 17

1. Servicemodelle

Jeder Service hat eine bestimmte Aufgabe zu erfüllen, die sich von jeder anderen unterscheiden sollte, damit wir keine redundanten Services zur Verfügung stellen.
Aber viele Services ähneln sich in der Art der Aufgabe die sie erfüllen sollen und diese bestimmt wiederum den Charakter des Services. Für die Beispiele in diesem Seminarthema sind die folgenden drei Servicetypen relevant und werden somit hier vorgestellt. Es gibt noch weitere Servicetypen, die aber in den späteren Kapiteln des Buches „Service-Oriented-Architecrure" von Thomas Erl besprochen werden.

1.1. Utility-Service

Ein Utility-Service kann als ein gewöhnlicher Dienstleistungsservice gesehen werden. Gewöhnlich in der Hinsicht, als dass er Aufgaben für Klienten bearbeitet, die nicht Businesslogicbezogen sind.
Ein Utilityservice ist so allgemein gefasst, dass er viele Klienten aus mehreren Bereichen haben kann, egal ob diese die Selben oder ähnlichen Ziele verfolgen.
Ein Utilityservice kann somit

- Daten verwalten
- Daten transformieren
- Datenberechnen

Ein Utility-Service hat folgende typische Eigenschaften.

Zugriffsvolumen	Sehr hohe Zugriffsraten. Gerade durch dritte können solche Services sehr stark in Anspruch genommen werden.
Schnittstelleneigenschaften	Normalerweise grobe Schnittstellen mit mehreren Parametern. Da der Service meist von außerhalb aufgerufen wird, sollte Kommunikationshäufigkeit beschränkt werden. Möglichst viel Information pro Transport.
Einsatzvorrausetzung	Auf Grund der hohen Nachfrage bei Utilityservices, sollten diese besser auf eigenen Servern angesiedelt sein. Somit können sie an einer Stelle angesiedelt werden, und trotzdem volle Arbeit leisten.

1.2. Business-Service

Ein Business-Service fügt mehrere Komponenten zu einem kompletten Geschäftsprozess zusammen, so dass über diesen Webservice auf eine komplette Geschäftslogik zugegriffen werden kann. Diese Services sind sehr Spezifisch auf ihre Aufgabe bezogen, und somit weniger allgemein wie die Utility Services.

Ein Business-Service hat folgende typische Eigenschaften.

Zugriffsvolumen	Mittelmass bis Hoch. Hängt davon ab wie oft der Geschäftsvorfall im Unternehmen benötigt wird
Schnittstelleneigenschaften	Detailliert bis grob. Die Schnittstellen müssen genau auf den Vorfall

	zugeschnitten sein. Somit hängt es vom Vorfall selber ab. Es gibt auch Redundante Schnittstellen. Feine für Interne und Grobe für Externe Aufrufe.
Einsatzvorrausetzung	Aus Sicherheits- und Performancegründen sollten die Business-Services immer möglichst na an den Anwendungskomponenten gelagert sein.

1.3. Controller-Service

Ein Controller-Service ist eine Art Fassade für eine Menge von Webservices.
Braucht ein Klient mehrere Services um seine Aufgabe zu erfüllen, muss er mehrere Aufrufe starten um seine Daten zusammeln. Des Weiteren muss er dann diese Daten auch noch selber zusammenführen. Das bedeutet zusätzliche Rechenzeit und den SOAP-Overhead für jede Anfrage.
Gibt es nun eine Zusammenstellung von Servicen die besonders oft benötigt wird, ist es sinnvoll einen Service, der diese Aufgaben für einen ausführt, dazwischen zu schalten. Diese Services nennt man Controller-Services

Zugriffsvolumen	Mittelmass bis Hoch. Hängt davon ab wie oft der Geschäftsvorfall im Unternehmen benötigt wird. Aber als Repräsentant für mehrere Geschäftsvorfälle, ist das Volumen hier höher als beim Business-Service
Schnittstelleneigenschaften	Grobe Schnittstellen. Controller-Services vereinen existierende Serviceschnittstellen in kleinere Businessservice umfassende Operationen.
Einsatzvorrausetzung	Je nach dem, wie groß die Menge der Businessservices ist, die der Controllerservice vertritt und wo sich deren Daten befinden, kann ein eigener eine Vorraussetzung für Controller-Services sein.

2. Modellierung Serviceorientierter Komponenten

Für das Einbinden eines Webservcice-Layer, der performant arbeitet und nicht durch einen Overhead an Operationen und Informationen gebremst wird, muss die Architektur dieser Software gut durchdacht sein.
Beim Modellieren Serviceorientierter Komponenten geht es darum, die Aufgaben die die Software erledigen soll, aufgrund gemeinsam benötigter Informationen oder sehr ähnlicher Aufgaben, zu gruppieren.
Für diese strukturierte Gruppierung gibt es zwei vom XWIF beschriebene Prozesse, die es einem erleichtern sollen, die Serviceorientierte Architektur zu bilden.
Der Ablauf dieser Prozesse, die in den folgenden Kapiteln beschrieben werden, erinnert durchaus an einen Übergang der Sequentiellen zur Objektorientierten Programmierung. Jedoch geht der Prozess für die SOA in meinen Augen an einigen Stellen sogar noch weiter. Objekte die für eine Software eine gänzliche Einheit wie zum Beispiel eine Rechnung darstellen, können somit, obwohl sie in der realen Welt eher als unteilbar erscheinen, noch mal unterteilt werden, um für Services, die nur einen kleinen Teil der Rechnung brauchen, auch nur exakt diesen Teil zur Verfügung stellen zu können.

Punkte über die man sich für die Prozesse Gedanken machen muss:

- Ausmaß des Services den die Software braucht?
- Welche Geschäftsabläufe werden benötigt?
- Wie viel Funktionalität sollte ein Service haben?

- Wo liegen die Technischen Grenzen des Systems?
- Wie kann ich die Komponenten optimieren?
- Wie sollen die Serviceschnittstellen aussehen?

2.1. Probleme in vielen Projekten

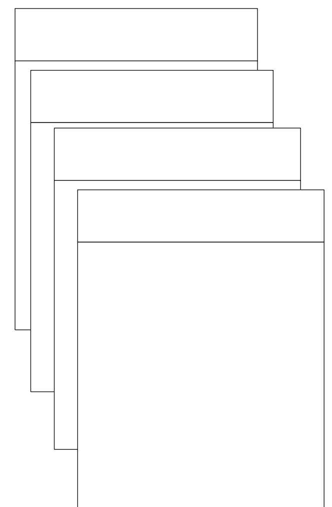
In Thomas Erls Augen, wird die SOA in vielen Projekten nicht weit genug berücksichtigt. Einige Projekte berücksichtigen die SOA sogar gar nicht.
SOA ist für viele eine Designtechnik, die die bisherigen Modellierungsstrategien ersetzen wird. Die Entscheidung in den Köpfen der Projektleiter scheint laut Thomas Erl eher eine entweder oder Debatte zu sein, obwohl es doch viel mehr ein fließender Übergang sein sollte.
Des Weiteren begehen die Projektleiter den Fehler, anzunehmen, es sei zu teuer die Mitarbeiter im bereich SOA und Webservices zu schulen. Jedoch wird bei diese Annahme übersehen, dass die Webservices auf jedes Unternehmen zukommen können, und die nachträglichen Anpassungen dann viel Teurer ausfallen würden, als die Sensibilisierung der Mitarbeiter auf die SOA gekostet hätte.

2.2. XWIF-Prozess für Komponenten Design

Der erste vom XWIF beschriebene Prozess beschäftigt sich im Wesentlichen mit den Eigenschaften der Komponenten und dessen Methoden. In den ersten fünf Schritten werden die Komponenten nicht verändert, sondern lediglich deren Methoden für eine eventuell kontextbezogene Trennung Markiert.
In den letzten fünf Schritten werden die Komponenten dann aktiv in kleinere Komponenten zerteilt, um somit flexibler Schnittstellen auf deren tatsächliche Arbeit begrenzen zu können.

2.2.1. Businesskomponenten zusammen tragen

Da der Prozess so ausgelegt ist, dass die Komponenten der Applikation schon existieren und in Richtung Serviceorientierung umgestaltet werden sollen, ist der erste Schritt das zusammen sammeln der Komponenten und Klassen der Applikation. Laut Thomas Erl ist es, wenn das Thema für einen neu ist, einfacher eine SOA aus bereits bestehenden Komponenten zu erstellen, anstelle von Anfang an zu versuchen eine Optimale Architektur aufzubauen.
In unserem Beispiel handelt es sich um die Klassen ProcessInvoice, ProcessOrder, ProcessPO und ProcessInventory.



In Unserem Beispiel werden wir mit der Klasse ProcessInvoice weiter machen.

2.2.2. Identifizieren der Business-Logik für die Webservices nur beschränkt nutzbar sind

Einige Funktionalitäten einer Applikation können entweder durch die Entwicklungsumgebung schon Webservice fähig sein, durch Add-Ons schnell Webservice fähig werden oder überhaupt nicht für Webservices geeignet sein.
Diese Funktionalitäten sollen im zweiten Schritt erkannt und markiert werden, damit in diese Komponenten nicht zuviel Zeit gesteckt wird.

In unserem Beispiel gehen wir davon aus, dass wir kein Transaktionsmanagement in unsere Services einbauen wollen, und haben somit Zwei Methoden, die für die Implementierung in Webservices nicht in Frage kommen.

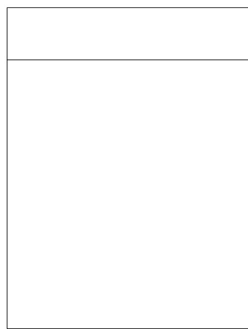
Die Methoden updateInvoice() und submitInvoice() sind schreibende Methoden, und benötigen somit ein Transaktionskonzept. Wir markieren diese Methoden Rot, da wir sie für Webservices nicht in betracht ziehen wollen.

Die Methode getInvoiceHistory() greift auf Daten zurück, für die wir eine sichere Nutzerverwaltung brauchen, da die Historischen Daten einer Rechnung nicht für jeden zugänglich sein sollen.
Somit benötigt diese Methode eine Zusätzliche Authentifizierung, die in den Webservice implementiert werden muss. Dafür markieren wir auch diese Methode.

2.2.3. Identifizieren von Wiederverwenbarkeit

Einige Methoden sind mit Sicherheit auch an anderen Stellen der Applikation von Bedeutung und könnten gut an diesen Stellen wieder verwendet werden. Jedoch ist es auch nicht immer sinnvoll alle anderen Methoden, die sich in unserer Klasse befinden, mit einzubinden, nur weil wir eine Methode aus der Klasse verwenden wollen. Wir markieren in diesem Schritt also alle Methoden von denen wir ausgehen, dass wir sie evtl. in anderen Bereichen der Applikation nutzen wieder verwenden wollen.

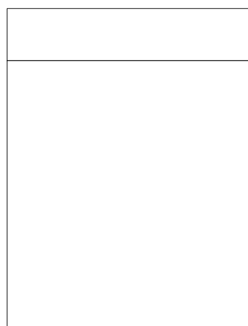
getInvoicePDF() ist zum Beispiel eine solche Funktion, da wir mit Sicherheit auch die Möglichkeit, andere Dokumente als PDF abspeichern zu können, einrichten wollen.



2.2.4. Identifizieren von Interfaceabhängigkeiten

In den meisten Fällen, greifen Methoden die extern aufgerufen werden auf interne Private Schnittstellen zu um Ihre Arbeit verrichten zu können.
Solche Methoden sollten in der Regel nicht getrennt werden.

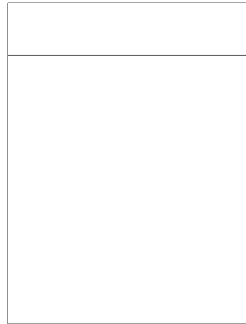
getInvoiceHeader() und getInvoiceDetails() sind Methoden die in unserem Fall auf die gleichen Methoden zurückgreifen um die Daten der Rechnung liefern zu können. Wir markieren also auch diese Methoden.



2.2.5. Bestimme Ebenen der Zusammenarbeit

Mit den Ebenen der Zusammenarbeit sind vor allem Interne und Externe Ebene gemeint. Ein Service, der nur in dem eigenen Unternehmen genutzt wird wäre ein Interner Service. Services die auch von Anderen Klienten, zum Beispiel über das Internet, genutzt

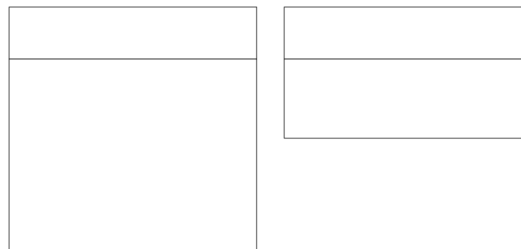
werden sind Externe Services. Interne und Externe Services sollten auch nach Möglichkeit getrennt werden, da Ihre Schnittstellen unterschiedlichen Standarten entsprechen.
Da wir mit der Methode getInvoiceX12() eine Methode haben, die mit großer Wahrscheinlichkeit von Außen aufgerufen wird und somit eher eine Methode eines externen Service bildet, sollte auch diese markiert werden. Die Anderen Methoden werden in diesem Beispiel primär als Methoden Interner Services betrachtet



2.2.6. Erstellen von feineren Aufgabenbezogenen Komponenten

Jetzt haben wir fast alle Methoden mit einer Eigenschaft markiert und werden nun aufgrund dieser Eigenschaften neue, kleinere Komponenten zusammenstellen, die keinerlei Overhead an Operationen haben, die für die geleistete Aufgabe nicht benötigt werden.
Für die eigene Applikation können diese auseinander genommen Komponenten oder Klassen über eine Fassade wieder zusammengesetzt werden. Für Webservices ist dann aber die Möglichkeit der kleinen Agilen Komponenten gegeben.

In unserem Fall trennen wir nun die Komponente in Zwei klassen. Eine Schreibende, die für unsere Webservices nicht relevant ist, und eine die Daten zur Invoice liefert.



2.2.7. Gruppieren der Methoden auf Grund ihres Typs

Jetzt kommen die in den ersten fünf Schritten verteilten Eigenschaften der Methoden zum Tragen.
Es gab in der Klasse GetInvoice nun noch drei Methoden, die auf Grund Ihrer Eigenschaften zu trennen sind. getPDF() benötigt als einzelne Methode eine PDF-Bibliothek, getX12() ist ein Externer Service und getHistory() brauch eine Authentifizierung.
Also stecken wir diese Methoden jeweils in eine einzelne Klasse.

2.2.8. Bestimmen der Kandidaten die für eine SOA relevant sind

Welche Methoden sind nun wirklich relevant für Webservices. Wir haben die Komponenten nun Aufgaben bezogen unterteilt und somit eine Struktur die uns hilft die Komponenten effizient für Webservices einzusetzen.
Aber aus der Aufteilung folgt nicht zwingend, dass auch alle Komponenten für einen Webservice genutzt werden müssen.
Aus diesem Grund werden in diesem Schritt die Komponenten bestimmt, die für Webservices in Frage kommen.
In unserem Beispiel wären das die folgenden Komponenten.

2.2.9. Betrachten der Kandidaten die nicht für SOA geeignet sind

Nun sind in der Regel noch Komponenten übrig, die wir nicht für Webservices als relevant eingestuft haben. Diese Komponenten kann man alle wieder zu einer zusammen führen. In unserem Beispiel ist es nur noch eine Komponente. Insofern ist das nicht nötig.

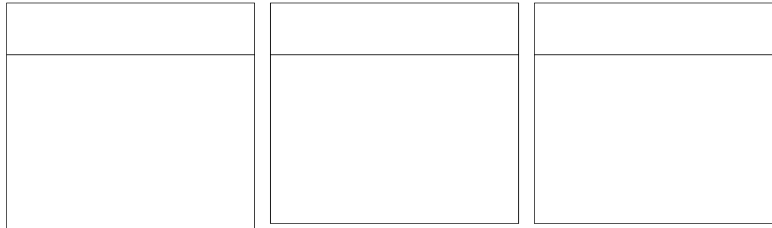
GetR

getInvoiceH

2.2.10. Zusammenführen von Klassenübergreifenden Aufgaben die zusammengehören

Die Klassen GetReports, GetPDF und GetEDI sind Komponenten, die neben den Methoden die die eigentlichen Daten liefern sollen auch noch zusätzliche Funktionalitäten brauchen. Diese Funktionalitäten stehen im Zusammenhang mit den Eigenschaften, die wir in den

letzten Punkten bestimmt haben. GetPDF brauch eine PDF-Bibliothek, GetReports eine Authentifizierungsmöglichkeit und GetEDI brauch so was wie eine EDI-Bibliothek.
Beim Bearbeiten der nächsten Klassen werden diese Eigenschaften womöglich wieder auftreten. Somit füllen sich die Komponenten mit den Methoden aus den anderen Klassen, da sie die Bibliotheken dann einfach wieder verwänden können.
Die Komponenten sehen nach dem Durchlaufen unserer Beispielklassen dann wie folgt aus.



2.3. *XWIF-Prozess für Service-Interface Gestaltung*

Der zweite vom XWIF beschriebene Prozess stellt einen Leitfaden zum strukturierten designen von Webservice-Interfaces dar.
Die ersten fünf Schritte stellen die für den Service benötigten Klassen zusammen und die letzten fünf Schritte beschäftigen sich dann mit dem konkreten Design der Interfaces.

In unserem Beispiel werden wir uns auf die Prozesse GetReports und GetInvoice konzentrieren und deren Service-Interfaces designen.

2.3.1. Servicemodel wählen

Im ersten Schritt wird für den zu erstellenden Service das Servicemodell bestimmt, dass die Aufgaben des Services am besten beschreibt. Das ist für die Implementierung des Services entscheidend, da sich die einzelnen Modelle in der Reaktion auf das Nutzungsvolumen oder in der Art der Veröffentlichung unterscheiden.

Servicemodelle
Utility-Service
Business-Service
Controller-Service
Proxy-Service
Wrapper-Service
Coordination-Service für atomare Transaktionen
Process-Services
Coordination-Services

GetR

Für Unsers Beispiele sind nur die ersten drei Modelle von Interesse. Die graue geschriebenen Modelle werden in Späteren Kapiteln des Buches „Service-Oriented Architecture“ besprochen.

getInvoiceH
getOrderHis
getInvetoryH

Ein Service kann grundsätzlich auch mehreren Modellen angehören. Für die Architektur und die Performance ist es aber wichtig, dass man sich über die Zugehörigkeiten möglichst früh bewusst wird.

„Services are best classified as utility services, once it has been confirmed that their overall purpose is to be shared and reused“ (Thomas Erl, „Service-Oriented Architecture” Seite 208 Abs. 2)

Der Service GetReports aus unserem Beispiel wird eine Schnittstelle zu unseren historischen Daten der Dokumente werden, und trägt somit den Charakter eines Utility-Services, da er anderen Klienten Daten zu Verfügung stellt.

Der Service GetInvoice wird uns alle benötigten Daten zu einer Rechnung zu Verfügung stellen und wird in diesem Fall eher als Business-Service eingeordnet.

2.3.2. Bestimmen des Anwendungsbereiches der Geschäftslogik

Der Anwendungsbereich eines Services sollte möglichst genau spezifiziert werden, damit die Aufgaben der einzelnen Services exakt getrennt sind.
Ein Service, der schnell erstellt wurde, erledigt womöglich die Aufgabe für die er bestimmt ist. Gelangt er aber mit seinen Aufgaben in den Bereich anderer Services, müssen wir entweder einen der Services um diese Funktionalität erleichtern, oder haben eine ungewollte Redundanz in unserem Servicemodell.

2.3.3. Bestimmen potentieller Klienten

Beim Erstellen des Webservices, sollte darauf geachtet werden, wer in Zukunft unseren Service nutzen wird. In erster Linie sind konkrete Aufgabenziele vorhanden, die der Service erfüllen soll.
Da wir aber eine Servicearchitektur modellieren, müssen wir damit rechnen, dass sich die Klientenmenge eigenständig erweitert. Auch über diese „Neukunden“ müssen wir uns vorher Gedanken gemacht haben, um die Schnittstellen und die Daten die wir anbieten möglichst optimal zu gestallten.

Folgende Service-Klienten sollten auf jeden Fall in Betracht gezogen werden.

- Teile der eigenen Software die in Zukunft teil eines Services werden
- Andere Applikationen die servicefähig werden
- Services die unseren Service einbinden (umschließen) wollen
- Externe Klienten denen der Zugriff auf unsere Services gewährt wird.

2.3.4. Bestimmen der benötigten Daten

Sobald der Umfang des Services bestimmt ist, sollten die Benötigten Daten zusammengetragen und übersichtlich dargestellt werden.

Wichtig ist hier, dass die Daten aus den Datenübermittlungen von den intern genutzten Daten getrennt werden.
Daten die zum Beispiel zur Validierung übergebener Daten benötigt werden, werden nur intern genutzt. Sie müssten gesondert von den Übergebenen Parametern aufgeleistet werden, um einen guten Überblick über die zu verarbeitenden Daten zu bekommen.

2.3.5. Prüfen der gegenseitigen Aufrufe

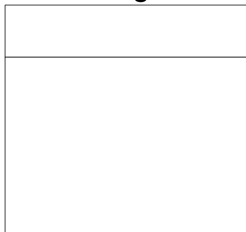
Mittlerweile sind die Funktionalitäten und benötigten Daten der Services bekannt, und es sollte geprüft werden, ob die Wege der Daten von der Datenquelle bis in unseren Service, wirklich die kürzesten sind.
Ruft ein Service einen anderen Service in einer Art auf wie wir es auch könnten, kann es wesentlich sinnvoller sein, direkt auf den anderen Service zuzugreifen. Noch Anschaulicher wird es beim direkten Zugriff auf die Datenbank. Haben wir die gleichen Möglichkeiten auf die Datenbank zuzugreifen wie der Service den wir dafür nutzen, dann ist es vielleicht sinnvoller es selber zu erledigen Vorraussetzungen hierfür ist, dass die Daten von dem Service der dazwischen wäre, nicht verändert werden, oder zum Beispiel eine Zugriffssicherung implementiert ist. Dann wäre der Zugriff nicht mehr Identisch und wir müssten doch den Service nutzen.

2.3.6. Bestimmen der benötigten Komponenten

Um die Serviceschnittstellen gestallten und implementieren zu können, müssen wir die benötigten Komponenten zusammen tragen.

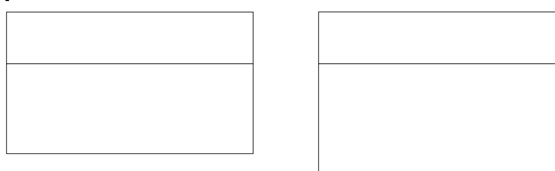
Welche Komponenten werden für die Arbeit die der Service leisten soll benötigt?

Für GetReports wird lediglich eine Klasse benötigt:



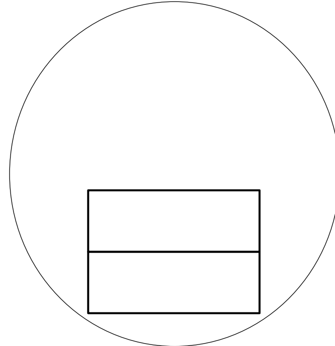
In diesem Fall haben wir eine Komponenten die exakt die Methoden für die Aufgaben unseres Services enthält.

Bei GetInvoice ist es ein wenig anders. Für die Umsetzung dieser Schnittstelle benötigen wir zwei Komponenten, die zusammen die Aufgaben für den Service leisten können
.

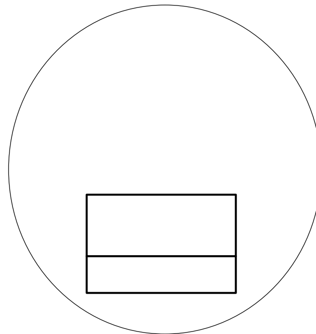


2.3.7. Modellieren des Serviceinterfaces

Jetzt haben wir die Komponenten zusammengetragen und wissen was unser Service leisten soll. Aber die Schnittstellen, über die der Klient auf unsere Dienste zugreifen kann müssen wir jetzt noch gestallten.



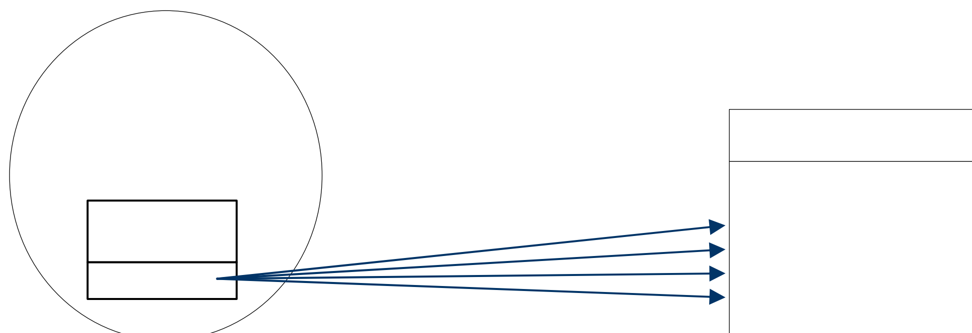
GetInvoce wird Zwei Schnittstellen haben, über die man die Rechnung abrufen kann, oder sich die Daten der Rechnung im EDI Format übermitteln lassen kann.



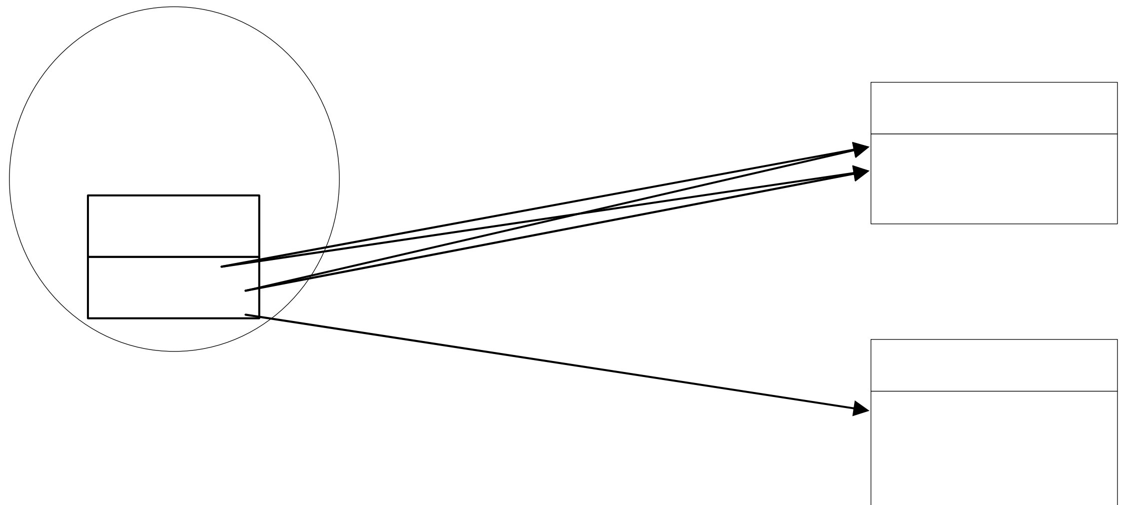
GetReports wird nur eine Schnittstelle haben, über die man auf die Historischen Daten der Dokumente zugreifen kann.

2.3.8. Visualisieren der Interaktionsszenarien

Um einen Überblick zu bekommen, wie die Komponenten Arbeiten und sich gegenseitig aufrufen, sollte man für jede Schnittstelle einmal ein Aufrufszenario zeichnen.
Diese Szenarien helfen einem nicht nur die eigenen Schnittstellen besser zu verstehen, sie können auch eine gute Basis für Testfälle in der Zukunft bilden.



Beim Aufruf der getHistory() Methode des Services GetReports ist zu beachten, dass nicht alle Methoden gleichzeitig aufgerufen werden, sondern der Aufruf per Parameter gesteuert werden sollte. Diese Form der Schnittstelle wird auch im Kapitel 3.2 noch besprochen



Bei der Schnittstelle getInvoiceEDI() ist sehr gut zu sehen, dass sie die Aufgabe nicht mit einer einzigen Komponente erledigen kann. Sie holt die Daten über die Komponente GetInvoice und lässt sie dann von der Komponente GetEDI() in das EDI Format umwandeln.

2.3.9. Design der Nachrichtenstruktur

In diesem Schritt soll die Nachrichtenstruktur der eingehenden und ausgehenden Nachrichten auf XML Basis erstellt werden.
Zum erstellen dieser Nachrichtenstrukturen gibt es Bibliotheken und Tools die dann die Gewünschten Daten zu XML Dokumenten konvertieren.
Bei der Suche nach solchen Tools bin ich auf das Tool MapForce von Altova gestoßen, welches visuell die Möglichkeit gibt, Komponenten und Datenbanken zu verknüpfen. Die Komponenten zum erstellen der XML-Nachrichten werden von dem Tool eigenständig erzeugt.
Das Tool hat auf mich einen sehr guten Eindruck gemacht, ist aber leider mit einem Preis von knapp 1000 € nicht ganz billig.

2.3.10. Servicemodel verfeinern

Am ende dieser Kette ist ein guter Punkt, um noch mal zu prüfen, ob für die einzelnen Services auch das korrekte Servicemodell gewählt wurde.
Mit den Methoden die in den Folgenden Kapiteln vorgestellt werden, sollten diese Serviceinterfaces nun noch feiner unter die Lupe genommen werden.

3. Strategien zur Integration von SOA

3.1. Definition Konsistenter Kriterien

Um das Modell möglichst konsistent zu halten und Chaos zu vermeiden, sollte sich immer an ein festgelegten Ablauf der Modellierung und an immer die selben Kriterien gehalten werden. Die Prozesse des XWIF sind hierfür gut geeignet und als Stütze gedacht. Es ist aber auch denkbar eigene Schritte einzufügen oder bereits vorhandene an zu passen. Jedoch sollten diese Änderungen aber für alle Webservices gelten.

3.2. Parameter vs. Methodengesteuerte Schnittstellen

Es gibt zwei Möglichkeiten wie man die Schnittstellen eines Services, bezogen auf die zu leistende Aufgabe, aufruft.
Methodengesteuerte Schnittstellen würden vom Namen her in der Art Nomen+Verb aufgerufen werden. Dies hätte den Vorteil, dass immer auf den ersten Blick zu sehen ist, um welche Aufgabe es sich handelt.
Parametergesteuerte Schnittstellen wären dagegen weniger aussagekräftig. Sie können für mehrere Aufgaben gleichzeitig stehen, wobei dann über einen zusätzlichen Parameter vom Aufrufenden Klienten gesteuert würde, welche konkrete Aufgabe zu erfüllen sei. Das hat den Nachteil, dass man nicht direkt aus der Bezeichnung der Schnittstelle erkennen kann, um welche Tätigkeit es sich genau handelt. Dagegen steht aber der große Vorteil der dynamischen Erweiterbarkeit. Funktionalitäten können hinzugefügt werden, ohne dass die Schnittstelle einer großen Änderung unterzogen werden muss. Den Klienten muss Lediglich diese Funktionalität im Zusammenhang mit dem dazugehörigen Parameter bekannt sein. Klienten die diese nicht kenne, nutzen die Schnittstelle wie gewohnt weiter.

3.3. Design gemischter Granularitäten

Jeder Webservice hat eine gewisse Granularität. Damit ist der Umfang der einzelnen Funktionen die ein Webservice liefert gemeint. Hat ein Webservice eine einzige Funktion, die einen ganzen Haufen an Daten liefert, so handelt es sich um eine recht grobe Schnittstelle. Wir erschlagen mit einem einzigen Aufruf einen riesigen Berg an Arbeit, haben aber auch immer einen großen Berg an Daten der transportiert werden muss.
Es gibt da aber auch die Funktionen, die nur kleine Teilaufgaben erledigen, oder auch eigenständige Aufgaben, die aber an sich keinen großen Umfang haben. Das sind dann Services mit einer kleinen Granularität. Diese einzelnen kleinen Schnittstellen stellen eine sehr Flexible Möglichkeit für den Klienten dar, die Aufgabenlösung so selber zusammenzustellen.
Bei einem Design Gemischter Granularitäten hat der Service nun sowohl die Groben als auch die feinen Schnittstellen. Der Klient kann sich somit aussuchen, ob er die gesamte Aufgabe vom Service bearbeitet haben will, oder ob er sich Stück für Stück über die Feinen Schnittstellen so an sein Ziel rantastet. Der Klient hätte somit bei letzterer Lösung auch die Möglichkeit auf Grund von bestimmten Ergebnissen, die Aufgaben zwischendurch abzubrechen.
Man könnte nun zu der Annahme kommen, dass durch das Bereitstellen der feinen sowie auch der groben Schnittstellen, eine gewisse Redundanz der Aufgaben entsteht. Bei schlechter Modellierung wäre das auch der Fall, aber

in den Folgenden Kapiteln lernen wir auch Modellierungskonzepte kennen, die die Redundanz durch interne Aufrufe vermeiden.
Welche Granularität eine Webserviceschnittstelle nun hat, häng offensichtlich davon ab, welche Aufgaben sie erledigen soll.

3.4. Separate Standards für Interne und Externe Services

In diesem Kapitel geht Thomas Erl in meinen Augen sehr ungenau darauf ein, was er mit dem Standard für Services meint. Ich gehe aber davon aus, dass es sich um den Aufbau der Schnittstellen in Bezug auf die Granularität, Namenskonventionene, Methoden- oder Parametersteuerung sowie um Skalierbarkeit und Fehlerbehandlungen handelt.

Interner Service

Interne Services sind Services die grundsätzlich von innerhalb des Unternehmens aufgerufen werden. Sie können natürlich auch indirekt, durch Aufrufen einer Externen Schnittstelle, von Externen Klienten ausgelöst werden. Die eigentliche Schnittstelle der Internen Services ist aber nur im Unternehmen selber bekannt und zugänglich.

Externer Service

Externe Services sind von Außen aufrufbar. Sie ermöglichen es Kunden von außen auf Daten in unserem Unternehmen zuzugreifen und zu verarbeiten.
Externe Services sind Bestandteile unseres Unternehmens, und können somit ohne weiteres auf unsere Internen Services zugreifen.

Natürlich wäre die optimalste Situation, dass sich Interne und Externe Services nicht Unterscheiden. Dann wäre die Portierung zwischen Internen und Externen Services wesentlich einfacher.
Aber in gewissen Teilen unterscheiden sich Interne und Externe Services dann doch.

Bei Internen Aufrufen können wir für unsere Unternehmen recht genau bestimmen wie hoch wohl der Zugriff auf den jeweiligen Service sein wird.
Bei Externen Services ist diese Aussage wesentlich schwieriger zu treffen.
Somit muss bei der Implementierung von Externen Services wesentlich mehr auf die Skalierbarkeit geachtet werden.

Externe Services sollten bei UDDI-Servern registriert werden und es sollten auf jeden Fall WSDL-Dokumente existieren.
Bei unseren Internen Services liegt es mehr in unseres Hand ob unseren Services bei einem UDDI-Server anmelden wollen. Wir wissen ja schließlich, welche Services wir im Hause haben und müssen somit nicht zwingend eine Dynamisch Suchmöglichkeit implementieren.

Bei Externen Services ist es wesentlich wichtiger eine Werteprüfung und ein Exceptionhandling zu realisieren.

Möchte man einen Externen Service intern nutzbar machen, kann man entweder die Inkonsistenz der Internen Schnittstellen aller Services in kauf

Nehmen, oder man fügt dem Externen Service einfach ein internes Interface hinzu.

3.5. Webservices Dritter

Nicht immer ist es nötig, einen Webservice selber zu implementieren. Es gibt bereits viele Lösungen im Internet, auf die man eigentlich einfach nur noch zugreifen müsste. Und das ist ja eigentlich auch der Sinn von Webservices. Eine Aufgabe, die an ein er Stelle, so wie wir sie brauchen, implementiert wurde, kann von uns ganz einfach über das Inter- oder Intranet genutzt werden.
Aber ein wichtiger Punkt ist hier „so wie wir sie brauchen“! Bevor man einen Webservice eines anderen Unternehmens nutzt, sollte man sich genau die Spezifikationen ansehen. Macht der Service wirklich exakt das was man für die Eigene Aufgabe braucht?
Webservices können von vielen Personen erstellt werden, und somit auch von Personen, die nicht sauber genug programmieren. Nimmt man nun den Dienst eines solchen Services in Anspruch, kann das zu vielen nicht beeinflussbaren Fehlern in der eigenen Applikation führen.
Wichtig ist des Weiteren auch die Verfügbarkeit. Wenn ich einen externen Webservice in meine eigene Applikation einbinde, dann muss ich auch sicher sein, dass der Dienst durchgehend verfügbar ist. Ansonsten laufe ich Gefahr, dass das Unternehmen still steht, ohne dass ich eingreifen kann, denn die Softwarekomponente die streikt liegt nicht in meinem Handlungsbereich.
Thomas Erl empfiehlt hier ein „Service Level Agreement“ mit den Anbietern abzuschließen.
Braucht man länger zum suchen eines solchen Services, als ihn selber zu implementieren, ist wohl nahe liegend, dass man ihn lieber selber implementiert. Das spart Zeit, und man lagert einen Handlungsraum weniger aus.

4. Modellierung von Service-Kompositionen

Servicekompositionen sind Zusammenstellungen von Services, so dass sie zusammen eine Bestimmte Aufgabe erfüllen können. Sie sind besonders Flexibel, weil sie aus vielen kleinen einzelnen Komponenten bestehen, die man dann Fallspezifisch aufruft und deren Informationen zusammen setzt.
Komplette Business-Logiken können somit aus einzelne Teilaufgaben zusammengestellt werden.
Durch diese Zersplitterung in die einzelnen Komponenten haben wir den Vorteil, dass wir mehr Möglichkeiten haben Komponenten wieder verwendbar zu machen. Je kleiner eine geleistete Teilaufgabe, desto größer wird die Menge der Potentiellen Klienten.
Da jedoch mit jedem Aufruf dieser einzelnen Services nun aber ein SOAP-Envelope dazu kommen, haben wir den Nachteil des offensichtlichen Daten-Overheads.
Des Weiteren hat der Klient nun die zusätzliche Aufgabe, selber die einzelnen kleinen Services in eine Nutzbare und für das Problem korrekte Aufrufreihenfolge zu bringen und die Daten nach Aufrufen aller Services zusammen zu setzen.
Hier bieten die Controllerservice einen guten Dienst. Sie übernehmen dies Aufgabe für den Klienten.

Ein Controllerservice ruft Intern auf der Serverseite alle benötigten Services in der korrekten Reihenfolge auf und stellt die Daten zusammen. Somit muss sich der Klient nicht mehr um dies Aufgabe kümmern, und der SOAP-Overhead, der durch die Einzelnen Aufrufe ggf. über das Internet entstehen würde ist in das Intranet des Unternehmens verlagert worden.
Controllerservice Kompensieren somit die Nachteile von Servicekompositionen und wir können unsere Business-Services trotzdem in kleinere Teilaufgaben zersplittern.

Aber die Nachteile der Service-Kompositionen sind nicht komplett entfernt. Sie sind lediglich in das Unternehmen verlagert worden.
Es sollte trotzdem darauf geachtet werden, dass die Service-Kompositionen nicht zu groß werden, da intern immer noch der Overhead der SOAP-Nachrichten auftritt. Bei zu großer Zersplitterung können die Vorteile auch in Nachteile umschwenken.

5. Erweiterungen der Servicefunktionalität

Darstellungshilfe

Um Webservices für den Klienten noch effizienter und angenehmer zu gestallten kann man, neben der angefragten Information, zusätzlich auch noch Information für die Verarbeitung der Daten liefern.
Ein Beispiel welches Thomas Erl liefert ist die Übertragung eines HTML-Blockes zur Darstellung der Daten. Auch die Übermittlung eines HTML-Formulars, um sicher zu stellen, dass der Klient immer die Korrekte Eingabemaske hat könnte eine Hilfe für Klienten sein.

Solche Daten müssen natürlich nicht jedes mal mit gesandt werden. Gibt man den Formularen eine Versionsnummer und fügt dem Service die Abfrage der derzeitigen Version hinzu, kann der Klient selber entscheiden, ob er den Overhead des Formulars eingehen will oder nicht.

Nutzungsverhalten

Um die Reaktionsfähigkeit des Services verbessern zu können kann man auch das Verhalten der Klienten studieren.
Wenn man die Aufrufreihenfolgen der Klienten aufzeichnet könnte man voraussagen, welche Funktion er wohl als nächstes aufrufen könnte. Die Daten kann man dann aus Performancegründen schon mal bereitstellen.

Wird an einer bestimmten Stelle häufig abgebrochen, kann das auch ein Anzeichen dafür sein, dass unser Service an der stelle unglücklich designed wurde. Sind uns solche Stellen bekannt, können wir sie auch gut ausbügeln.

6. Integration von SOAP-Nachrichten

Kompression

SOAP Nachrichten haben ja bekanntlich die Eigenschaft, dass sie sehr geschwätzig sind, weil sie auf XML basieren.
Diese Geschwätzigkeit ist der alten Binären Datenübertragunstechnik was die Performance angeht natürlich unterlegen.
Also muss man Wege finden wie man dieses Performanceproblem in Grenzen hält.

In erster Linie wäre es Sinnvoll den Overhead möglichste gering zu halten. Konkreter gesagt, so wenig SOAP Enveloaps wie möglich zu haben. Aber die Struktur der Nachrichten an sich sind durch die XML-Syntax auch schon ein wesentlich größeres Volumen als die Binären Daten früher.

Um dem entgegen zu gehen kann man alternativ, oder zusätzlich die Nachrichten vor der Übertragung komprimieren.
Immer wenn große Datenvolumina über tragen werden sollen, könnte man die Daten durch Kompression verringern und hätte somit die Performance ein wenig verbessert.
Es sei denn, man komprimiert ausnahmslos alle Nachrichten. Man muss natürlich auch darauf achten, dass die Kom- und Dekompression nicht länger dauern als die Übertragung der Plain Daten selber. Denn dann wäre keine Zeit gewonnen.

Sicherheit
Ein weiterer Nachteil der SOAP Nachrichten ist, dass Ihre Daten direkt aus dem Datenstream lesbar sind. Hier wird der Vorteil der Einfachheit wieder zu einem Problem.
Nachrichten können entweder gelesen oder sogar, von Routern zwischen den Partner, geändert werden.
SSL löst diese Problem weitest gehend. Jedoch aber nur auf der Transportsicherheit. Die Nachrichtensicherheit ist mit SSL immer noch nicht gegeben.
Thomas Erl gibt an, dass man sich zu diesem Thema auch in den Bereichen XML Key Management, XML Encryption, SAML, XACML und XML Digital Signature einarbeiten sollte.

Referenzen:

„Service-Oriented Architechture (A Field Guide to Integrationd XML and Webservices)"
von Thomas Erl (April 2004)

Links:

Altova MapForce
http://www.altova.com/products_mapforce.html