

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. F. Leymann
Prof. Dr. B. Mitschang

Betreuer: Dipl. Ing. R. Junghuber

begonnen am: 01. September 2001

beendet am: 28. Februar 2002

CR-Klassifikation: H.4.1, H.3.5, H.5.3

Diplomarbeit Nr. 1960

Automatisierte Nutzung von UDDI für IBM MQ Series Workflow

Andreas Fried

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Für meine Eltern

Inhaltsverzeichnis

1	Einführung	1
1.1	Einbettung und Aufbau der Arbeit	1
1.2	Web Services	1
1.3	Simple Object Access Protocol - SOAP	4
1.4	Web Services Description Language - WSDL	6
1.5	Universal Description Discovery and Integration - UDDI	8
1.6	Zusammenspiel von SOAP, WSDL und UDDI	11
1.7	IBM MQ Series Workflow	13
2	Web-Service-Erweiterungen für MQWF	17
2.1	Workflow-Prozesse als Web Services	17
2.2	Einbinden von Web Services in Workflow-Prozesse	19
3	Veröffentlichung von Workflow-Prozessen	22
3.1	Überblick: Veröffentlichungsprozess	22
3.2	Einzelschritte des Veröffentlichungsprozesses	23
3.3	Realisierungsalternativen für den Gesamtprozess	29
4	Einbinden von Web Services	36
4.1	Überblick: Einbinden eines Web Service	36
4.2	Auffinden von Web Services in UDDI	37
4.3	Binden von Web Services	47
4.4	Statisches Einbinden	49
4.5	Dynamisches Einbinden	56
5	Lösung für das Veröffentlichen	71
5.1	Anforderungen	71
5.2	Zugriffskomponente für UDDI	72
5.2.1	Spezifikation	72
5.2.2	Entwurf	73
5.3	Zugriff auf Parameter in Dateien	81
5.3.1	Spezifikation	83
5.3.2	Entwurf	86
5.4	Steuerung des Gesamtvorganges	87
5.4.1	Spezifikation	87
5.4.2	Entwurf	88
6	Lösung für das Einbinden	92
6.1	Anforderungen	92
6.2	Zugriff auf UDDI	93
6.2.1	Spezifikation	93
6.2.2	Entwurf	94
6.3	Einbindungsprozess	101
6.3.1	Spezifikation	101
6.3.2	Entwurf	102

6.4	Praktische Erfahrungen beim dynamischen Einbinden	105
7	Zusammenfassung und Ausblick	107
A	Code-Beispiele	109
A.1	WSDL	109
A.2	UDDI-Strukturen	111
A.3	Generierung von WSDL aus einer FDL-Prozessdefinition	114
	Literatur	117

Abbildungsverzeichnis

1.1	Web-Services-Modell	3
1.2	UDDI Kerndatenstrukturen	9
1.3	Interoperabilitäts-Stapel	11
1.4	WSDL-Beschreibung in UDDI	12
1.5	SOAP, WSDL und UDDI im Web-Services-Modell	13
1.6	Aufbau von IBM MQ Series Workflow	15
2.1	SOAP-Schnittstelle zu MQWF	17
2.2	Das Software-Werkzeug fdl2wsdl	18
2.3	Aufruf eines Web Service als Aktivitäts-Implementierung	19
2.4	UPES-Aufruf, abgeleitet aus WSDL-Dokument	20
2.5	UPES-Aufruf über Java-Proxy	20
2.6	Das Softwarewerkzeug wsdl2fdl	21
3.1	Überblick über den Veröffentlichungsprozess	23
3.2	Kapselung eines Workflow-Prozesses	24
3.3	UDDI Zugriff über SOAP-Nachrichten	26
3.4	Zugriff auf UDDI über das UDDI4J-API	27
3.5	Zugriff auf UDDI über WSTK	28
3.6	Publizieren einer WSDL-Beschreibung in UDDI	30
3.7	Veröffentlichung, gesteuert durch WFMS	31
3.8	Veröffentlichung, gesteuert durch Anwendungsprogramm	32
3.9	Skript-gesteuerter Veröffentlichungsprozess	33
4.1	Auffinden und Binden eines Web Service für MQWF	37
4.2	Auffinden eines Web Service durch Suchen und Auswählen	38
4.3	Suchen von Web Services mit bestimmter Schnittstelle	41
4.4	Suchen von Web Services durch Auswahl von Unternehmen	42
4.5	Suchen und Filtern von Web Services durch Zugriff auf UDDI	43
4.6	Suche nach Web Services mit Service-Kategorie	44
4.7	Auswählen eines BindingTemplate-Objektes	45
4.8	Schritte zum Binden eines Web Service an MQWF	50
4.9	Auffinden über UDDI4J-API	52
4.10	Statischen Einbinden, gesteuert über Skript	53
4.11	Statisches Einbinden, unterstützt durch Java-Anwendung	54
4.12	Statischer und dynamischer Teil des Einbindens	57
4.13	Programmaktivität zum Aufruf eines Web Service, eingebunden ins Workflow-Modell	58
4.14	Prozess-Definition für den Start des dynamischen Teils	59
4.15	Zugriffsmöglichkeiten auf Daten beim Auffinden	62
4.16	Aufruf zur Generierung der Proxy-Klasse aus MQWF	63
4.17	Dynamische Generierung einer Programmdefinition	64
4.18	Aufruf eines Web Service als Prozessaktivität	66
4.19	Dynamisches Einbinden als Sub-Prozess	67
4.20	Dynamisches Einbinden als Java-Anwendung	68
4.21	Statische Schritte bei Aufruf über generischen Proxy	68
4.22	Dynamische Schritte, eingebunden in den Workflow-Prozess	69

5.1	Klassen für die Steuerung des UDDI-Zugriffs	74
5.2	Datenquellen für den Veröffentlichungsprozess	75
5.3	Klassendiagramm: Veröffentlichung der WSDL-Schnittstelle	76
5.4	Veröffentlichung einer WSDL-Schnittstelle	77
5.5	Klassendiagramm: Veröffentlichung der WSDL-Implementierung	79
5.6	Veröffentlichung einer WSDL-Implementierung	80
5.7	Klassendiagramm: Klasse zum Lesen der Konfigurationsdatei	86
5.8	Zweistufiger Veröffentlichungsprozess	89
5.9	Erster Teil des Veröffentlichungsprozesses	90
5.10	Zweiter Teil des Veröffentlichungsprozesses	91
6.1	Kernklassen für den Auffindvorgang	95
6.2	Sequenzdiagramm: Auffinden eines Web Service	95
6.3	Klassendiagramm: Auffindvorgang	96
6.4	Sequenzdiagramm: Suche nach geeigneten Web Services in UDDI	100
6.5	Einbindungsprozess: Zugriff auf UDDI	103
6.6	Einbindungsprozess: Generierung der Proxy-Klassen und der Programmdefinition	104
6.7	Klassendiagramm: Zugriff auf dynamische Suchparameter	104

1 Einführung

1.1 Einbettung und Aufbau der Arbeit

Das Thema dieser Arbeit ist eingebettet in den Themenkomplex: Anbindung eines Workflow Management Systems (WFMS) an das World Wide Web (WWW). In jüngerer Zeit wurden neue Formen zur Anwendungsintegration auf Basis der Internet-Technologie entwickelt. Dazu gehören das Simple Object Access Protocol (SOAP) und die Web-Service-Technologie. Universal Discovery Description and Integration (UDDI) ist ein Baustein dieser Web Service Technologie. Am Beispiel von IBM MQ Series Workflow sollte erörtert werden, wie die zentrale Auskunftsstelle UDDI für die Integration von Workflow- und Web-Service-Technologie genutzt werden kann.

Im ersten Abschnitt werden die Technologien, die die Grundlage für diese Diplomarbeit bilden, kurz vorgestellt und dabei einige wichtige Begriffe eingeführt.

Die Abschnitte 2 bis 4 bilden den ersten Hauptteil dieser Arbeit. Darin werden die Interaktionsmöglichkeiten zwischen dem Workflow-System IBM MQ Series Workflow (MQWF) und dem Verzeichnis für Web Services Universal Description, Discovery and Integration (UDDI) beleuchtet und mögliche Realisierungsansätze für eine automatisierte Interaktion vorgestellt.

Der zweite Hauptteil umfasst die Abschnitte 5 und 6. Darin wird die Implementierung zweier Interaktionsprozesse zwischen MQWF und UDDI beschrieben. Dabei wird sowohl eine Lösung für das Veröffentlichen einer Workflow-Prozesses als Web Service in UDDI vorgestellt als auch für das Einbinden von in UDDI veröffentlichten Web Services in MQWF. Grundlage für die Ausführungen sind die Programme, die im Rahmen dieser Diplomarbeit erstellt wurden.

In Abschnitt 7 werden die Ergebnisse dieser Arbeit zusammengefasst.

Diese Diplomarbeit entstand im Rahmen einer Kooperation zwischen der Universität Stuttgart und der IBM Deutschland Entwicklung GmbH in Böblingen. Sämtliche Programme, die im Rahmen dieser Arbeit entwickelt wurden, sind auf Rechnern der IBM gespeichert.

1.2 Web Services

Definition

Der Begriff Web Service wird in [Kre01] definiert als Schnittstelle zu einer Software, auf deren Operationen mit Hilfe von standardisierten XML-Nachrichten über ein Kommunikationsnetzwerk zugegriffen werden kann. Zu dieser Schnittstelle gehört eine Beschreibung aller für die Nutzung notwendigen Details, wie Nachrichtenformate, Protokolle und genaue Lage des Web Service im Netzwerk. Die Beschreibung von Web Services erfolgt in einer standardisierten, auf XML basierenden Notation.

Somit bilden Web Services die Grundlage zur Realisierung eines auf höherer Ebene angesiedelten Kommunikationsprotokolls zwischen Programmen, in dessen Vordergrund die Anwendungssemantik steht und weniger die Netzwerktechnologie. Voraussetzung dafür ist die Nutzung von standardisierten und auf breiter Ebene eingesetzten Technologien wie z.B. XML und HTTP.

Die Eigenschaft, dass die Implementierungsdetails eines Web Service durch die Schnittstelle verborgen werden, ist vor allem im Umfeld der betrieblichen Datenverarbeitung interessant. So kann die Web-Services-Technologie zur Integration von vorhandener Software zu einer neuen, komplexeren Anwendung über Programmiersprachen und Plattformgrenzen hinweg eingesetzt werden. Dazu kommt der Aspekt der Öffnung von unternehmenseigenen Anwendungen für den Zugriff über das World Wide Web.

Im weniger technologischen Sinne beschreibt ein Web Service in den meisten Fällen die Software-Lösung irgendeiner unternehmens- bzw. geschäftsbezogenen Problemstellung z.B. Bestellungen, Auskünfte, etc., mit dem Ziel, die Geschäftswelt noch weiter zu automatisieren. Hinter einem Web Service steht ein breites Spektrum an Implementierungsmöglichkeiten, vom einfachen Funktionsaufruf bis hin zu komplexen Anwendungen.

Web-Services-Modell

Die Grundlage des Web-Services-Modells bilden drei Rollen, Service Provider, Service Requestor und Service Registry, sowie deren Interaktionen, find, publish und bind. Rollen und Interaktionen arbeiten auf den Objekten Web Service, d.h. die Implementierung des Web Service, und dessen Beschreibung.

Der Service Provider stellt die Implementierung eines Web Services in einem Netzwerk zur Verfügung. Gleichzeitig erstellt er eine Beschreibung dieses Services in standardisierter Form. Diese Beschreibung veröffentlicht er in einem für die potentiellen Nutzer zugänglichen Verzeichnis, der Service Registry. Ein Service Requestor, also der Nutzer des Services findet in dem Verzeichnis die Beschreibung des gewünschten Services. Die Beschreibung enthält alle notwendigen Informationen, um dem Service Requestor den Aufruf bzw. die Nutzung des Web Service zu ermöglichen. Hervorzuheben ist, dass das Aufsuchen und Binden nicht nur statisch erfolgen muss sondern auch dynamisch zur Laufzeit möglich ist. In Abbildung 1.1 auf der nächsten Seite wird das Zusammenspiel der Rollen und deren Interaktion dargestellt.

Beschreibung der Rollen im Web-Services-Modell

Service Provider bezeichnet den Eigentümer des Web Service aus Unternehmenssicht. Aus technischer Sicht stellt der Service Provider die Plattform dar, auf der die Implementierung zur Verfügung gestellt wird.

Service Requestor bezeichnet ein Unternehmen, das eine bestimmte Problemstellung mit Hilfe eines Web Service lösen möchte. Aus technischer Sicht wird ein Programm als Service Requestor bezeichnet, mit dem ein Web Service gesucht oder auf einen Web Service zugegriffen wird.

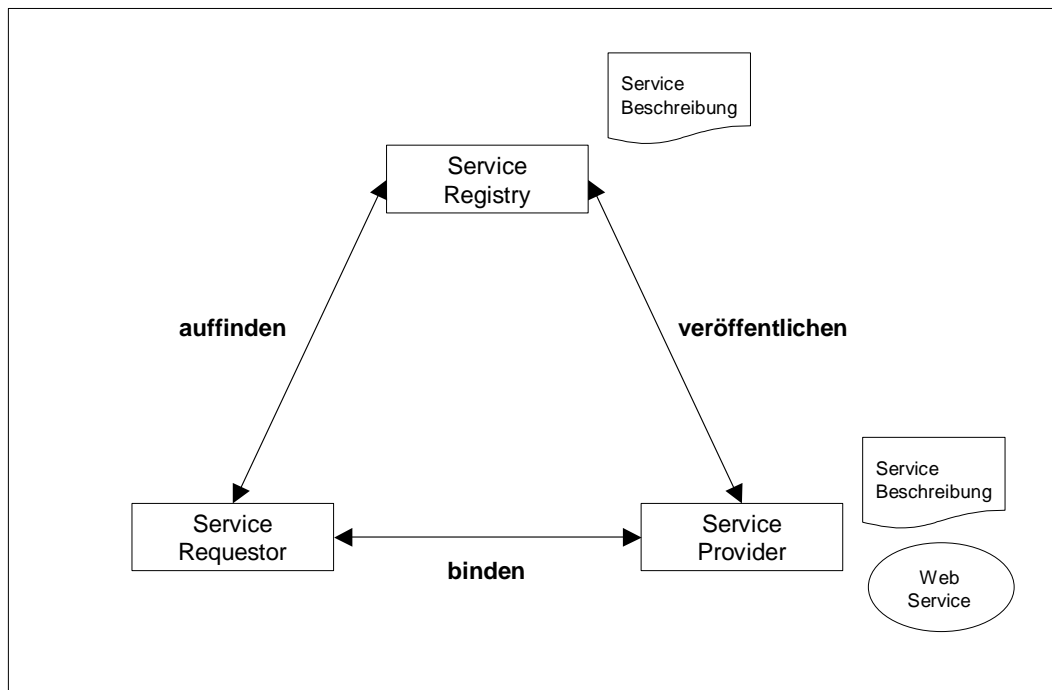


Abbildung 1.1: Web-Services-Modell

Service Registry bezeichnet ein Verzeichnis, in dem Service Provider die Beschreibung ihrer Web Services veröffentlichen und in dem Service Requestors diese Informationen suchen und abrufen können.

Beschreibung der Operationen

Publish bezeichnet das Veröffentlichen der Beschreibung eines Web Service, um diesen von außen zugänglich zu machen. Format und Ort der Veröffentlichung sollte so gewählt werden, dass ein potentieller Nutzer die Beschreibung suchen und auffinden kann.

Find bezeichnet das Suchen bzw. Auffinden einer Web Service Beschreibung in einem geeigneten Verzeichnis. Die Suche kann in verschiedenen Phasen des Lebenszyklus einer Anwendung vorkommen: zur Entwicklungszeit, um auf die Schnittstelleninformationen zuzugreifen oder zur Laufzeit für die Informationen zum Binden, z.B. Netzwerkadresse oder Protokoll.

Bind bezeichnet den Start der Interaktion des Service Requestor mit dem Web Service zur Laufzeit. Hierzu werden die Detailinformationen der Web Service Beschreibung benutzt.

Lebensphasen eines Web Service

Build bezeichnet die Implementierung eines Web Services und das Erstellen der Beschreibung.

Deploy Um einen Web Service zur Verfügung zu stellen, muss dieser in seine Ablaufumgebung eingebracht werden, z.B. Registrierung beim Application Server. Im weiteren Sinne gehört auch das Veröffentlichen der Web Service Beschreibung zum Deployment.

Run In dieser Phase steht der Web Service anderen Anwendungen zum Aufruf, insbesondere für den Netzwerkzugriff, zur Verfügung.

1.3 Simple Object Access Protocol - SOAP

Im Folgenden werden die Teile des Simple Object Access Protocol (SOAP), die im Kontext dieser Arbeit relevant sind, kurz vorgestellt. Ausführliche Informationen können in der SOAP-Spezifikation [BE⁺00] nachgelesen werden. Beim Beginn dieser Arbeit standen stabile Implementierungen für SOAP in der Version 1.1 zur Verfügung, obwohl bereits Version 1.2 veröffentlicht wurde. Aus diesem Grund beziehen sich die weiteren Ausführungen zu SOAP auf die Version 1.1 der Spezifikation. Die Weiterentwicklung von SOAP wird von einer Arbeitsgruppe des W3C Konsortiums koordiniert.

SOAP ist ein auf XML basierendes Protokoll zum Austausch von Nachrichten in einer vernetzten, verteilten Umgebung. Die Grundlage für SOAP ist das folgende unidirektionale Kommunikationsmodell: ein Sender schickt eine Nachricht an einen Empfänger. Daraus können verschiedene synchrone oder asynchrone Kommunikationsmuster, z.B. Frage-Antwort, entfernter Prozeduraufruf oder Nachrichtenversand abgeleitet werden.

In der Spezifikation wird SOAP nicht an ein bestimmtes Transportprotokoll gebunden, obwohl es im Hinblick auf das Hyper Text Transfer Protocol (HTTP) entworfen wurde. Somit kann SOAP zum Datenaustausch zwischen Programmen in allen gängigen Netzwerkumgebungen eingesetzt werden.

Die SOAP 1.1 Spezifikation konzentriert sich auf folgende Punkte:

- Rahmen für den Nachrichtenaustausch,
- Regeln für die Darstellung von Daten in XML,
- Realisierung eines entfernten Prozeduraufrufs (RPC) auf Basis von HTTP.

Der Rahmen für das Versenden von Nachrichten besteht aus wenigen XML-Elementen und Attributen. So besteht eine SOAP-Nachricht aus einem Envelope-Element, das ein optionales Header- und ein Body-Element enthält. Das Header-Element ist bestimmt für das Versenden von Metainformationen an die SOAP-Umgebung, z.B. für die Middleware. Im Body-Element befindet sich die eigentliche Nachricht. Ergänzt wird dieser Rahmen durch eine Fehlermeldung (SOAP-Fault), mit der dem Sender mögliche Fehlersituationen aufgezeigt werden können.

Der Großteil der SOAP-Spezifikation befasst sich mit der Darstellung der zu übertragenden Daten. Prinzipiell können mit SOAP Daten in jeder Repräsentation verschickt werden. Ein Ziel der SOAP-Spezifikation ist jedoch, einen Standard für die Datenrepräsentation einzuführen mit dem plattformübergreifende Integration und Kommunikation möglich ist. Aus diesem Grund werden Regeln für die Darstellung von Daten und Datentypen, insbesondere strukturierter Datentypen, aufgestellt. Als Basis dient das in der XML Schema Spezifikation eingeführte Typsystem.

Die Autoren von SOAP betonen ausdrücklich, dass die Nachrichten nicht auf das dargestellte Typsystem und die Darstellungsregeln beschränkt ist. Vielmehr sei SOAP im Hinblick auf Einfachheit und Erweiterbarkeit konzipiert [BE⁺00].

Außerdem wird in der SOAP-Spezifikation beschrieben, wie ein entfernter Prozeduraufruf (Remote Procedure Call, RPC) auf Basis des HTTP-Protokolls realisiert wird. Die Struktur von HTTP ist für den RPC sehr gut geeignet. Dabei wird der in SOAP dargestellte Prozeduraufruf mit HTTP POST an die Zielumgebung geschickt und die Antwortnachricht zusammen mit der HTTP-Statusmeldung über dieselbe Verbindung zurückgeschickt. Im folgenden Beispiel werden der RPC über HTTP sowie der Aufbau einer SOAP-Nachricht dargestellt:

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host: www.plasticmoney.de
Content-Type: text/xml; charset="utf-8
Content-Lenght: nnnn
SOAPAction: "urn:creditcardchecker-service"

<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  <soap:Body>
    <checkCreditCard
      xmlns="http://www.plasticmoney.de/CreditCardChecker">
      <creditCardNumber>1234-456-789-0</creditCardNumber>
    </checkCreditCard>
  </soap:Body>
</soap:Envelope>
```

Listing 1.1: SOAP-RPC Aufruf

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Lenght: nnnn

<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
```

```
<soapv:Body>
  <checkCreditCardResponse
    xmlns="http://www.plasticmoney.de/CreditCardChecker">
    <checkResult>1</checkResult>
  </checkCreditCardResponse>
</soap:Body>
</soap:Envelope>
```

Listing 1.2: SOAP-RPC Antwort

1.4 Web Services Description Language - WSDL

Die Web Services Description Language (WSDL) wurde gemeinsam von den Firmen Ariba, IBM und Microsoft entwickelt, um eine standardisierte Beschreibung der Kommunikationsbeziehungen zwischen Web Services zu ermöglichen. Zur Zeit dieser Arbeit liegt WSDL in der Version 1.1 vor. Zunächst werden die Modellannahmen für WSDL beleuchtet. Im Anschluss daran werden die Sprachelemente und die Struktur eines WSDL-Dokumentes erläutert und in einem abschließenden Beispiel zusammengefasst.

Ergänzend zu dem in Abschnitt 1.2 dargestellten Ausführungen werden Web Services (die WSDL Spezifikation benutzt den Begriff Network Service) in [CW⁺01] definiert als Menge von Kommunikations-Endpunkten, die in der Lage sind, Nachrichten auszutauschen. Ein Kommunikations- oder Netzwerk-Endpunkt ist an eine Netzwerk-Adresse, ein konkretes Protokoll und an bestimmte Nachrichtenformate gebunden.

WSDL führt eine Trennung zwischen abstrakten, wiederverwendbaren und konkreten Teilen einer Service-Definition ein. So behandelt WSDL die Nachrichten und Transportprotokolle als abstrakte Informationen. Diese werden in abstrakten Endpunkt-Typen kombiniert. Durch Zuordnung einer Netzwerkadresse wird aus dem Endpunkt-Typ ein konkreter Endpunkt. Alle Endpunkte zusammen ergeben einen konkreten Web Service.

Um alle Aspekte eines Web Service zu definieren benutzt WSDL folgende Elemente:

Types Container für Datentypdefinitionen, die ein beliebiges Typsystem benutzen. WSDL benutzt als eingebautes Typsystem das Typsystem der XML Schema Definition, siehe auch [DGM⁺01].

Message abstrakte, typisierte Definition der Daten, die im Rahmen des Web-Services-Modell ausgetauscht werden.

Operation abstrakte Definition der Aktionen eines bestimmten Web Service. Diese Aktionen werden innerhalb eines Endpunkt-Typs definiert.

PortType Menge der abstrakten Operationen, die von einem oder mehreren Web Services unterstützt werden.

Binding Zuordnung eines konkreten Protokolls und Datenformates für einen ausgewählten Endpunkt-Typ (PortType).

Port einzelner konkreter Endpunkt, definiert durch die Kombination einer Netzwerkadresse und eines Binding-Elementes. Ein Port-Element wird innerhalb eines Service-Elementes definiert.

Service Menge zusammengehörender Endpunkte. Dies ist die komplette Beschreibung der konkreten Operationen, die durch einen Web Service zur Verfügung gestellt werden.

Durch das Binding-Element wird in der WSDL Spezifikation ein allgemeiner Mechanismus für das Binden von Nachrichten an Protokolle definiert, mit dem beliebige Transportprotokolle und Nachrichtenformate zugeordnet werden können. Bereits in der WSDL-Spezifikation werden drei konkrete Protokolle definiert, die über diesen Erweiterungsmechanismus eingebunden werden können:

- SOAP 1.1
- HTTP GET/POST
- MIME

Diese eben dargestellte Offenheit von WSDL für zusätzliche Protokolle gilt analog für die anderen WSDL-Elemente, z.B. Typen und Nachrichten. Der angedeutete Erweiterungsmechanismus erlaubt u.a. beliebige Typsysteme einzuführen und darauf die entsprechenden Nachrichtenformate zu definieren ohne dass WSDL als Beschreibungssprache geändert werden muss.

Bislang wurde nur wenig über die Dokumentationsmöglichkeiten von WSDL ausgesagt. Dabei erlaubt WSDL, jedem Element eine ausführliche, an Menschen adressierte Beschreibung in Form eines Documentation-Elementes hinzuzufügen. Der Inhalt dieses Documentation-Elementes kann vom Benutzer im Rahmen der XML-Konventionen frei definiert werden. Somit können in WSDL sowohl technische als auch anwendungsbezogene Aspekte eines Web Service beschrieben werden.

WSDL erlaubt durch einen Importmechanismus, die Beschreibung eines Web Service auf mehrere Dokumente aufzuteilen. Dies ist aus Gründen der Wiederverwendung vorteilhaft, insbesondere bei einer Aufteilung in abstrakter und konkreter Endpunkt-Beschreibung oder bei der Definition von Datentypen, deren Schema von mehreren WSDL- und anderen Dokumenten referenziert bzw. importiert werden können. Tatsächlich ist in der praktischen Anwendung die Aufteilung der WSDL Beschreibung in Service-Interface, das sind die abstrakten Teile, und Service-Definition, das ist der konkrete Teil, häufig zu beobachten.

In Anhang A.1 auf Seite 109 ist die Beschreibung des Web Service zur Überprüfung der Kreditkartennummer aus Abschnitt 1.2 als WSDL-Dokument dargestellt, das aufgeteilt ist in Service-Schnittstelle und Service-Definition.

1.5 Universal Description Discovery and Integration - UDDI

In diesem Abschnitt werden die Konzepte von UDDI erläutert, insbesondere vor dem Hintergrund der Ideen und Absichten der Erfinder von UDDI. Eine Diskussion der Nutzungsmöglichkeiten von UDDI für MQ Series Workflow erfolgt in den Kapiteln 3 und 4.

UDDI, Universal Description, Discovery and Integration, ist ein Ansatz für den standardisierten, gemeinsamen Zugriff auf die Informationen, die ein Unternehmen zu Web Services veröffentlicht hat, vergleichbar mit einer zentralen Informationsstelle. Wie in Abschnitt 1.2 über Web Services ausgeführt, übernimmt UDDI im Web-Services-Modell die Rolle der Service Registry.

Der Begriff UDDI steht einerseits für ein Konzept, zum anderen auch für die konkrete Implementierung eines solchen Verzeichnisses, das den UDDI-Spezifikationen entspricht. Die Spezifikationen, die zum Zeitpunkt dieser Arbeit in der Version 2.0 vorliegen, wurden von einem Industriekonsortium der führenden Software-Hersteller erarbeitet und als gemeinsamer Quasi-Standard verabschiedet.

Eine der Grundideen für UDDI ist, Informationen über Unternehmen und deren Web Services in einem zentralen Verzeichnis abzulegen, damit andere Unternehmen, die diese Web Services nutzen wollen, diese auf einfache Art und Weise auffinden und abrufen können. Dazu ist es notwendig, dass alle technischen Informationen über den Web Service, wie Netzwerkadresse und Protokolle, erreichbar sein müssen. Ohne ein derartiges Verzeichnis müssten diese Informationen von Hand oder mit Hilfe von Suchmaschinen im World Wide Web gesucht werden. Dabei muss angenommen werden, dass keine einheitliche Struktur für diese Informationen besteht und somit ein automatisches Auffinden und Auswerten praktisch unmöglich ist.

Für UDDI Version 2.0 existieren insgesamt vier Teilspezifikationen: Datenstrukturen, Programmierschnittstelle, Operations- und Replikations-Spezifikation. In den beiden erstgenannten Dokumenten werden die Datenstrukturen zur Speicherung und für die Zugriffsoperationen festgeschrieben. Sie stellen also die zentralen Teile der Spezifikation für die Nutzer dar. Die beiden letztgenannten Teile richten sich in erster Linie an die Betreiber von UDDI-Verzeichnissen. Diese werden im weiteren Verlauf der Arbeit nicht weiter betrachtet.

Zunächst soll das Informationsmodell von UDDI näher beleuchtet werden. Den Kern von UDDI bilden vier Datenstrukturen: BusinessEntity, BusinessService, Binding-Template und TModel. In diesen vier Datenstrukturen werden alle nötigen Informationen zu einem Unternehmen und dessen veröffentlichten Web Services gespeichert. Abbildung 1.2 auf der nächsten Seite illustriert die Beziehungen der Kern-Datentypen.

Alle Datenstrukturen von UDDI werden in XML spezifiziert. Im folgenden werden die vier Kern-Datenstrukturen inhaltlich näher erläutert. Eine vollständige und detaillierte Beschreibung aller Datenstrukturen findet man in der UDDI Data Structure Specification [ERvR01].

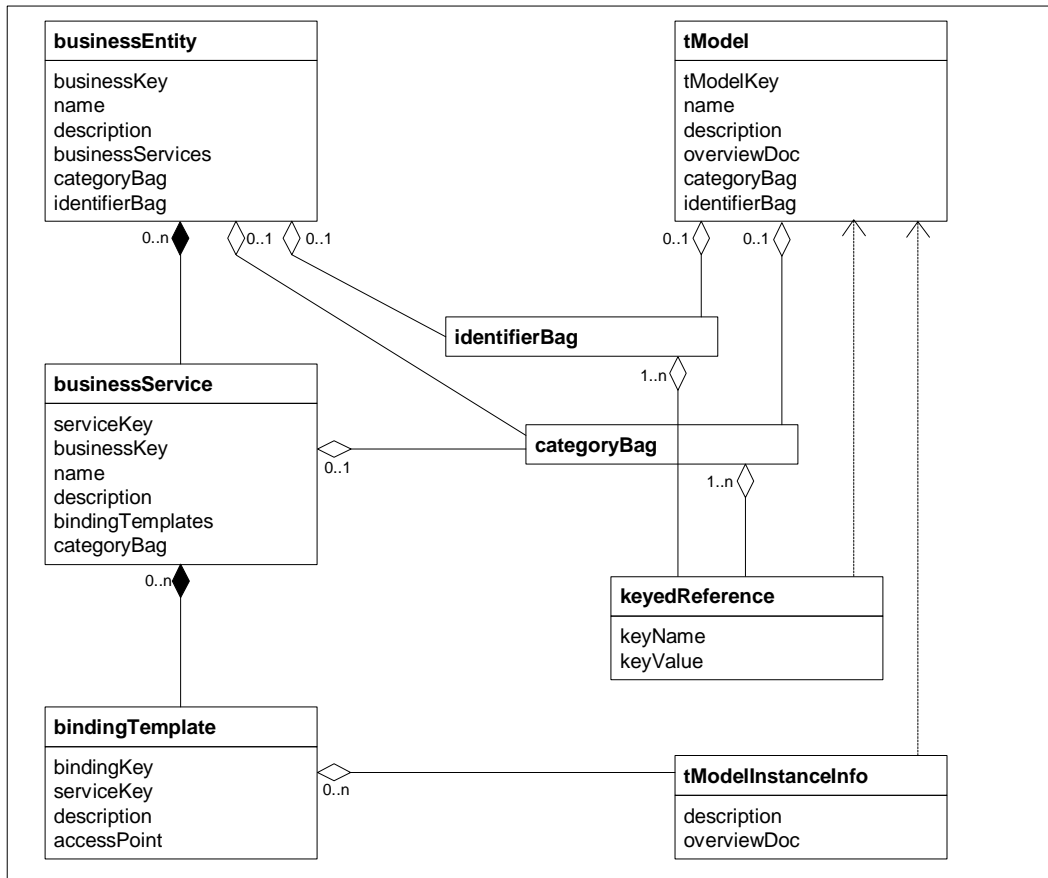


Abbildung 1.2: UDDI Kerndatenstrukturen

Ein `BusinessEntity`-Objekt bildet das hierarchisch oberste Element eines UDDI-Eintrages. Es enthält eine Beschreibung des Unternehmens wie Adresse, Geschäftsfeld, Ansprechpartner, etc., also eine kompaktes Bündel derjenigen Informationen, die ein Interessent benötigt, um eine Geschäftsbeziehung mit dem Unternehmen anzubahnen. Weiter enthält ein `BusinessEntity`-Objekt eine Liste mit veröffentlichten `BusinessService`-Objekten. Komplettiert wird ein `BusinessEntity`-Objekt durch die Möglichkeit, verschiedene Kategorien zuzuordnen und Bezeichner zu vergeben, um ein gezielteres Auffinden zu ermöglichen.

Ein `BusinessService`-Objekt beinhaltet eine abstrakte Beschreibung dessen, was ein veröffentlichter Web Service an Funktionalität zur Verfügung stellt. Die Informationen sind in erster Linie an Menschen adressiert und sollen die Aspekte eines Web Service beschreiben, die zur Auswahl einer bestimmten Funktionalität notwendig sind. Auch für `BusinessService`-Objekte kann eine Kategorisierung vorgenommen werden. Nicht zuletzt enthält ein `BusinessService` eine Liste mit Detailinformationen zum Aufruf von Web Services in einer oder mehreren `BindingDetail`-Strukturen.

Ein `BindingDetail`-Objekt beinhaltet die technischen Informationen für einen Web Service. Neben einer verbalen, d.h. an Menschen adressierten Beschreibung, wird die

Netzwerkadresse in Form eines URL und die Zugriffsmethode, z.B. HTTP, festgelegt. Daneben besteht noch die Möglichkeit für ein BindingDetail Objekt die Konformität mit einer bestimmten Spezifikation festzulegen. Dazu wird eine Referenz auf eine TModel-Datenstruktur, die weiter unten beschrieben wird, gespeichert.

Die bislang vorgestellten Datenstrukturen stehen in einer „Ist-enthalten-Beziehung“ Das äußerste Element ist dabei das BusinessEntity-Objekt. Darin enthalten sind die BusinessService- und darin wiederum die BindingDetail-Objekte. Letztere können nicht unabhängig in UDDI existieren. Unabhängig sind dagegen neben den BusinessEntity- auch die TModel-Strukturen.

Eine TModel-Struktur enthält Metadaten über eine Spezifikation, d.h. UDDI gibt einen Rahmen vor, wie solche Beschreibungen zugänglich gemacht werden können. Ein TModel enthält einen eindeutigen Schlüssel, einen Namen, eine Beschreibung und einen URL, der auf die eigentliche Spezifikation zeigt. Die beiden bedeutendsten Spezifikationsarten im Zusammenhang mit UDDI sind zum einen Schnittstellendefinitionen für Web Services, zum anderen Klassifizierungen bzw. Taxonomien.

Man kann TModel-Strukturen auf zwei Arten referenzieren. Die einfache Referenz wird im Rahmen einer Web Service Spezifikation eingesetzt, um diese i.S. eines Fingerabdrucks zu identifizieren. Man kann eine Referenz auch mit einem Schlüsselwert versehen, um einerseits die Spezifikation zu identifizieren, zum anderen auf einen bestimmten Abschnitt oder Wert zu verweisen. Letztere Technik wird insbesondere für Klassifizierungen und Taxonomien im Zusammenhang mit der Kategorisierung angewendet.

Die UDDI Programmer's API Specification 2.0 [MER01] beschreibt den Zugriff auf die UDDI Informationen in Form von XML Nachrichten. Als Protokoll für die Übermittlung der Nachrichten dient das Simple Object Access Protocol (SOAP). Diese Programmierschnittstelle ist zur Einbindung von UDDI in eigene Anwendungen vorgesehen und bietet einen elementaren, vollständigen Satz an Zugriffsoperationen für alle UDDI-Daten. Auf Basis dieser Operationen können auch komplexere Operationen, z.B. komfortable Aufsuchoperationen definiert werden.

Die Programmierschnittstelle gliedert sich in die beiden Bereiche Veröffentlichen und Aufsuchen. Im ersten Teil sind die Nachrichtenformate zum Speichern und modifizieren zusammengefasst, im zweiten die Nachrichten zum Suchen und für den Zugriff auf Detailinformationen. Für das Veröffentlichen sind zusätzlich zu den Nutzdaten Autorisierungsinformationen erforderlich und eine SSL-Verbindung vorgeschrieben. Für das Auffinden von UDDI Einträgen wurden zwei verschiedene Anfragetypen zu Grunde gelegt: zum einen existieren breit angelegte Operationen zum Auffinden, die als Resultat eine möglichst lange Trefferliste mit den notwendigsten Informationen von geringem Umfang enthält, zum anderen gezielte Anfragen, mit denen detaillierte Informationen abgerufen werden können. Mit den beiden Anfragetypen lassen sich drei verschiedene Anfragemuster realisieren: Suchen und überblicksartige Anzeige von Informationen, Spezialisierung einer allgemeineren Anfrage, um Details abzufragen und Direktzugriff mit Schlüsselwerten.

Daneben gibt es eine Benutzungsschnittstelle, mit der interaktiv über einen Web-Browser auf die UDDI Informationen zugegriffen werden kann. Diese Schnittstelle

ist vor allem für die Nutzer interessant, die mehr über ein Unternehmen und deren Web Services erfahren wollen und sich weniger für die technische Umsetzung interessieren. Auch diese Schnittstelle erlaubt die wesentlichen Funktionen zum Auffinden, Veröffentlichen und Modifizieren von UDDI-Einträgen. Darüber hinaus ist es jedem Betreiber eines UDDI-Verzeichnisses frei gestellt, weitere Schnittstellen zur Verfügung anzubieten.

Die aktuelle Form von UDDI ist durch die Tatsache geprägt, dass einige, durchaus zentrale Fragen im Web-Services-Modell noch nicht abschließend geklärt, keinesfalls standardisiert sind. Dies gilt insbesondere für die „Quality of Services“, also die Frage, wie sicher und zuverlässig Nachrichten zwischen Web Services ausgetauscht werden können. Aus diesem Grund ist UDDI in der Version 2.0 ein Metamodell mit technischer Prägung, bietet aber dennoch einen soliden Grundstock an Funktionalität, um einen sofortigen Einsatz zu ermöglichen.

1.6 Zusammenspiel von SOAP, WSDL und UDDI

Die in den vorhergehenden Abschnitten vorgestellten Technologien SOAP, WSDL und UDDI behandeln allesamt orthogonale Problembereiche im Web-Services-Modell. Dennoch sind die Konzepte im Hinblick auf das Zusammenwirken aller Komponenten entwickelt worden. Häufig ist die Rede von einem Stapel [M⁺00], der in Abbildung 1.3 dargestellt ist. In diesem Abschnitt soll der Zusammenhang der drei Technologien illustriert werden.

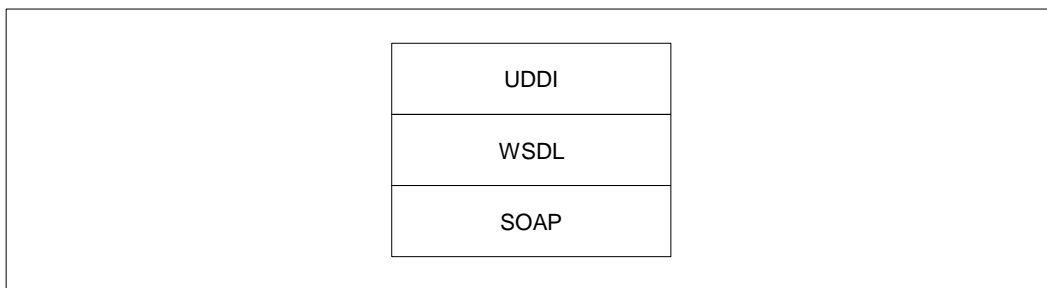


Abbildung 1.3: Interoperabilitäts-Stapel

Auf der untersten Ebene des Stapels ist das Kommunikationsprotokoll SOAP angesiedelt. Aufgrund seiner Flexibilität hat SOAP im Web-Services-Umfeld große Bedeutung erlangt. Für viele Schnittstellen zu Web Services existiert eine SOAP-Implementierung. Diese Tatsache wird unter anderem daran deutlich, dass in der WSDL-Spezifikation bereits eine Erweiterung für SOAP definiert wurde. Im Hinblick auf UDDI spielt SOAP ebenfalls eine wichtige Rolle. So ist die Programmierschnittstelle von UDDI für das Veröffentlichen und das Auffinden von Web-Service-Beschreibungen spezifiziert in Form von SOAP-Nachrichten.

Komplizierter ist das Zusammenspiel von WSDL und UDDI. Zwar ist WSDL ein Mittel zur Beschreibung von Web Services und UDDI ein Platz, an dem Informationen zu Unternehmen und deren Web Services gespeichert werden können, dennoch

überlagern sich an manchen Stellen zwei unterschiedliche Ansätze zur Beschreibung von Web Services. Dennoch bietet UDDI aufgrund seines weitergehenden Informationsmodells die Möglichkeit, Informationen über Web Services in Form eines WSDL Dokumentes zu veröffentlichen. Dabei wird von der Möglichkeit von WSDL Gebrauch gemacht, einen Web Service getrennt nach Schnittstelle und Implementierung, d.h. die konkrete Netzwerkadresse, darzustellen. Abbildung 1.4 verdeutlicht die Vorgehensweise zur Veröffentlichung von WSDL-Beschreibungen in UDDI, so wie sie in [CER01] empfohlen wird.

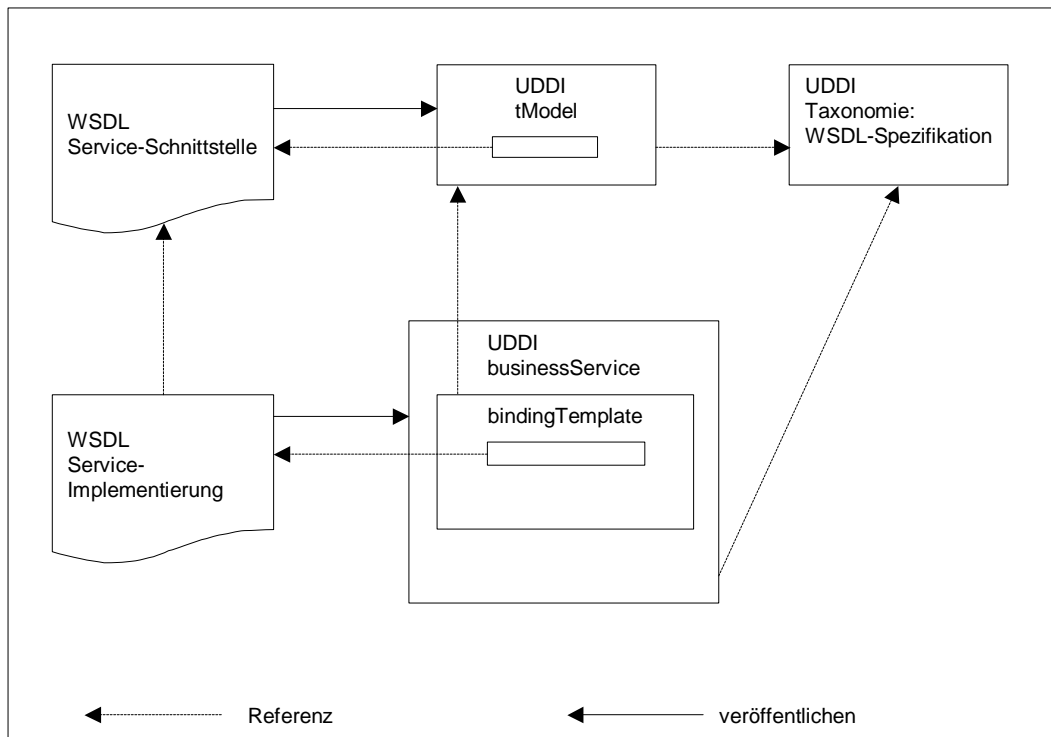


Abbildung 1.4: WSDL-Beschreibung in UDDI

Ein Web Service wird in zwei getrennten WSDL-Dokumenten beschrieben: der Service-Schnittstelle mit Types-, Messages-, PortType- und Binding-Elementen und der Service-Implementierung, bestehend aus einem Service-Element mit einem oder mehreren Port-Elementen. Beide Dokumente sind durch die WSDL-Import-Anweisung verknüpft. In UDDI werden Schnittstellendefinitionen als TModel gespeichert. Dafür wird nicht das WSDL-Dokument selbst in UDDI eingefügt, sondern eine Referenz (URL) auf den tatsächlichen Speicherort. Anschließend wird das TModel-Objekt mit Hilfe einer von UDDI zur Verfügung gestellten Taxonomie für Spezifikationstypen als WSDL-Spezifikation gekennzeichnet. Die Informationen der Service-Implementierung werden in einem UDDI-BusinessService-Objekt mit einem oder mehreren BindingTemplate-Objekten gespeichert. Wie in Abschnitt 1.5 beschrieben, wird eine Referenz auf die Service-Implementierung innerhalb der TModelInstanceInfo-Struktur eingetragen. Das BusinessService-Objekt kann nun mit zusätzlichen Informationen, z.B. Kategorien ergänzt werden.

2. Dimension: Organisatorische Informationen beschreiben die bei der Ausführung beteiligten Personen, Arbeitsgruppen und Rollen. Dabei wird festgelegt, welche Schritte von welchen Personen oder Rollen ausgeführt werden sollen.
3. Dimension: EDV-Infrastruktur. Hier werden den einzelnen Schritten Anwendungsprogramme und Systeme zugeordnet.

Diese drei Dimensionen eines Geschäftsprozesses werden in einem Workflow-Modell zusammengefasst. Dieses Modell kann mit Hilfe des Modellierungswerkzeuges grafisch dargestellt werden oder in Textform mit Hilfe der Flow Definition Language (FDL). Die in der FDL eingesetzten Konstrukte sind z.B.:

PROCESS beschreibt einen Workflow-Prozess. Ein Workflow-Prozess umfasst Aktivitäten, Kontroll- und Datenflüsse. Prozesse besitzen Ein- und Ausgabedaten.

ACTIVITY bezeichnet eine Aktivität. Eine Aktivität ist vergleichbar mit einem Einzelschritt im Prozess und kann entweder durch ein Programm (**PROGRAM_ACTIVITY**) oder durch einen (Sub-)Prozess (**PROCESS_ACTIVITY**) implementiert werden. Auch Aktivitäten besitzen Ein- und Ausgabedaten.

PROGRAM beschreibt eine Programmdefinition mit Ein- und Ausgabedaten. Dabei werden das ausführbare Programm, die Parameter und die Plattform festgelegt.

STRUCTURE definiert eine Datenstruktur, die aus einfachen oder komplexen Datentypen aufgebaut sein kann. Als komplexe Datentypen stehen Arrays und Records zur Verfügung.

CONTROL Zwischen Aktivitäten wird ein Kontrollfluss mit bestimmten Übergangsbedingungen festgelegt, die erfüllt sein müssen, bevor die nächste Aktivität ausgeführt werden kann.

DATA legt den Datenfluss zwischen den Aktivitäten fest. Dabei kann auch definiert werden, wie Datenwerte zwischen Datenstrukturen weiter zu gegeben sind.

Eine ausführliche Beschreibung des Prozessmodells bietet [LR00].

MQWF umfasst mehrere Komponenten mit spezifischen Aufgaben und ist in drei Schichten aufgebaut, vgl. Abbildung 1.6 auf der nächsten Seite. Schicht 1 umfasst die Workflow-Clients, Programmierschnittstellen (APIs) und die Program Execution Agents, die für den Aufruf von Anwendungsprogrammen verantwortlich sind. Schicht 2 wird aus den unterschiedlichen Servern für die Administration, Prozesssteuerung und Ausführung von Back-End-Applikationen gebildet. Schicht 3 besteht aus dem Datenbanksystem. Zur Kommunikation zwischen den Komponenten können unterschiedliche Kommunikationssysteme eingesetzt werden z.B. das nachrichtenorientierte IBM MQ Series.

Für die weiteren Ausführungen dieser Arbeit sind folgende Komponenten wichtig:

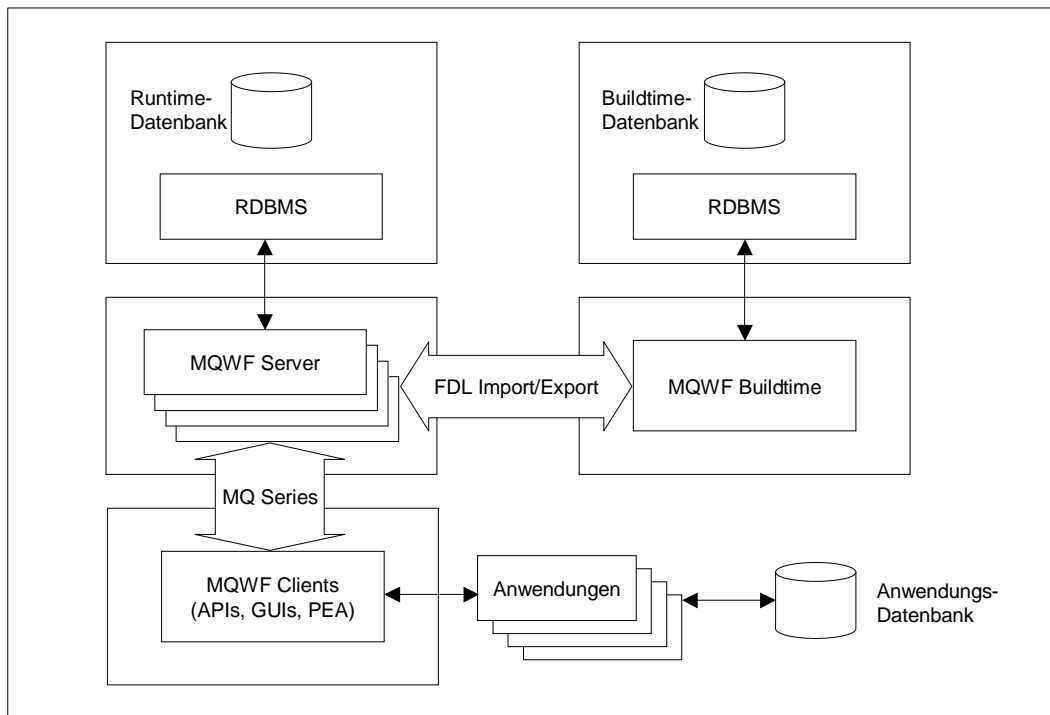


Abbildung 1.6: Aufbau von IBM MQ Series Workflow

(Workflow) Execution Server ist verantwortlich für Instanziierung, Ausführung und Beendigung von Prozessen. Zu den Aufgaben gehört weiter, das Verschicken von Work-Items, Management von Work-Lists sowie das Logging des aktuellen Prozessstatus.

User Defined Program Execution Server (UPES) ist eine Komponente mit der benutzerdefinierte Anwendungen an MQWF angebunden werden können. Aktiviert wird die Anwendung durch eine MQWF-Nachricht, die über MQ Series verschickt wird. Die Interpretation dieser Nachricht liegt dann voll in der Verantwortung der Anwendung. UPES werden z.B. zur Integration von Anwendungen eingesetzt, die über die vorhandenen Mechanismen nicht angesprochen werden können.

Workflow-API bezeichnet die Programmierschnittstelle zu MQWF. Über diese Programmierschnittstelle kann z.B. aus Anwendungen heraus auf Datenstrukturen zugegriffen oder Prozesse gestartet werden. Diese Programmierschnittstelle steht unter anderem auch für die Java-Plattform zur Verfügung.

Buildtime ist die Modellierungskomponente von MQWF, mit der Workflow-Modelle erstellt werden können. Die Daten der Buildtime-Komponente werden getrennt von denen in der Ausführungsumgebung gehalten und müssen vor der Ausführung explizit importiert werden. Zum Austausch von Workflow-Modellen zwischen Modellierungs- und Ausführungsumgebung wird die Flow Definition Language benutzt.

Eine ausführliche Beschreibung der Architektur und der Komponenten befindet sich in [IBM01a].

2 Web-Service-Erweiterungen für IBM MQ Series Workflow

Das Workflow Management System (WFMS) IBM MQ Series Workflow (MQWF) besitzt bereits verschiedene Möglichkeiten zur Nutzung der Web-Services-Technologie. Diese Möglichkeiten sind als Erweiterung zum eigentlichen WFMS unter dem Namen Web Service Process Management Toolkit (WSPMTK) erhältlich. Dieses WSPMTK gibt sowohl den Rahmen für die Nutzung der Web-Service-Technologie vor als auch für die weiteren Ausführungen in dieser Arbeit. In den folgenden Abschnitten werden diese Web-Service-Erweiterungen näher beschrieben.

2.1 Workflow-Prozesse als Web Services

Ein Teil der Web-Service-Erweiterungen schafft die Voraussetzungen dafür, dass Workflow-Prozesse als Web Services zur Verfügung gestellt und über einen SOAP-RPC auf Basis des HTTP-Protokolls aufgerufen werden können. Dafür werden zwei Komponenten zur Verfügung gestellt: eine generische Schnittstelle zu MQWF und ein Software-Werkzeug zur Erstellung einer Web Service Beschreibung.

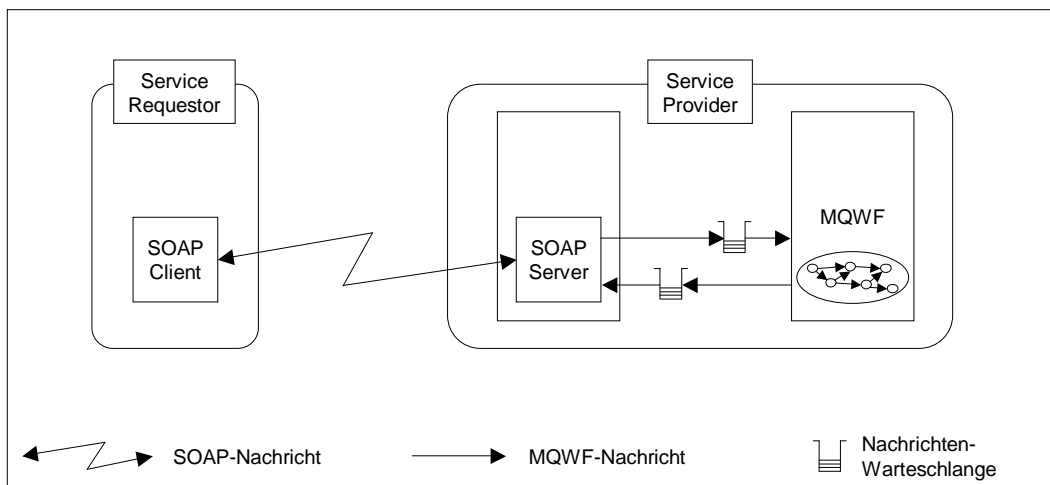


Abbildung 2.1: SOAP-Schnittstelle zu MQWF

Die Aufgabe der generischen Schnittstelle besteht darin, SOAP-Nachrichten zu empfangen, diese in eine MQWF-Nachricht zu transformieren und dann zur Verarbeitung an MQWF weiterzuleiten. Die Rückgabewerte werden entsprechend zurücktransformiert und als SOAP-Nachricht an den SOAP-Client zurückgeschickt. Diese Schnittstelle ist in Abbildung 2.1 als SOAP-Server dargestellt. Die Details zu dieser generischen SOAP-Schnittstelle werden in der Dokumentation zum WSPMTK beschrieben.

Mit Hilfe des Software-Werkzeuges `fd12wsdl` können die Klassen für die eben beschriebene Schnittstelle erzeugt werden. Außerdem wird durch `fd12wsdl` die in Ab-

schnitt 1.4 dargestellte Web-Service-Beschreibung in Form eines WSDL-Dokumentes generiert. Abbildung 2.2 zeigt die Ein- und Ausgaben von `fdl2wsdl`.

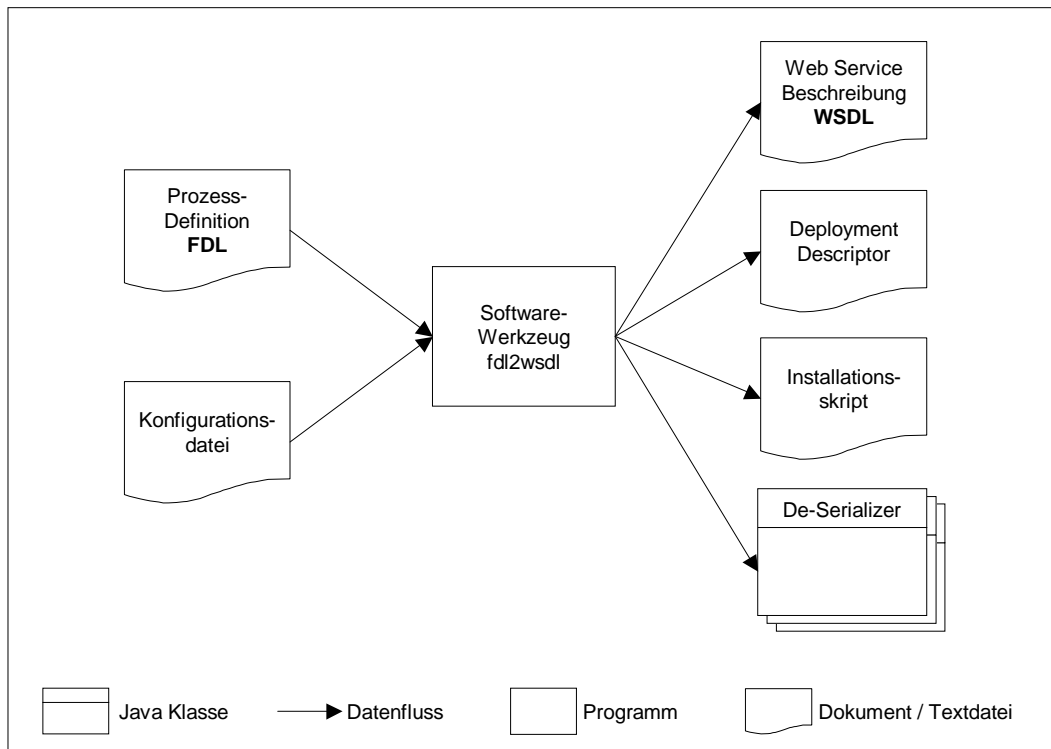


Abbildung 2.2: Das Software-Werkzeug `fdl2wsdl`

Die Eingaben für das Software-Werkzeug bestehen aus der Prozessdefinition in FDL und einer Konfigurationsdatei, die alle notwendigen Parameter zur Erzeugung der Schnittstelle und der Web-Service-Beschreibung enthält, die nicht Teil der FDL-Beschreibung sind. Diese Parameter werden in der Dokumentation zum WSPMTK erläutert.

Aus der Prozessbeschreibung und der Konfigurationsdatei werden folgende Ausgaben erzeugt:

De-/Serializer Java-Klassen zur Typumwandlung zwischen SOAP- und MQWF-Datenformaten.

Deployment-Descriptor XML-Datei, die benutzt wird, um eine SOAP-Schnittstelle beim Application Server zu registrieren. Für jede Prozessdefinition innerhalb der FDL-Datei wird ein Deployment-Descriptor generiert.

Installationskript in Form einer MS-DOS-Stapelverarbeitungsdatei, mit der die De-/Serializer-Klassen übersetzt, und die Schnittstellen beim Application Server registriert werden können.

Web-Service-Beschreibung in Form eines WSDL-Dokumentes. Für jeden in der FDL-Datei enthaltenen Prozess wird ein Web Service im WSDL-Dokument erzeugt.

Proxy-Klassen die für den externen Aufruf des Workflow-Prozesses über die Web-Service-Schnittstelle benutzt werden können. Für jedes Service-Element in der WSDL-Beschreibung wird eine Proxy-Klasse erzeugt.

Für jeden Prozess in der FDL-Definition wird durch `fdl2wsdl` ein WSDL-Service-Element mit einem SOAP-Binding erzeugt. Der Nachrichtenaustausch zwischen SOAP-Client und SOAP-Server erfolgt über entfernten Prozeduraufruf, so wie in der SOAP 1.1 Spezifikation [BE⁺00] beschrieben (SOAP-RPC über HTTP).

Die Eingabe- bzw. Ausgabestrukturen für die Prozesse werden als WSDL-Message-Elemente modelliert. Dabei wird für jede Datenstruktur der FDL-Definition ein XML-Schema-Datentyp erzeugt. In Anhang A.3 auf Seite 114 ist ein Beispiel einer FDL-Struktur und der daraus generierten WSDL-Beschreibung dargestellt.

2.2 Einbinden von Web Services in Workflow-Prozesse

Durch die Web-Service-Erweiterungen zu wird eine Möglichkeit zur Verfügung gestellt, wie Web Services als Implementierung von Aktivitäten benutzt werden können. Zum Aufruf der Web Services wird ein User Defined Program Execution Server (UPES) benutzt. Der UPES empfängt MQWF-Nachrichten und ruft die entsprechenden Web Services z.B. über SOAP-Nachrichten auf. Abbildung 2.3 veranschaulicht dieses Prinzip.

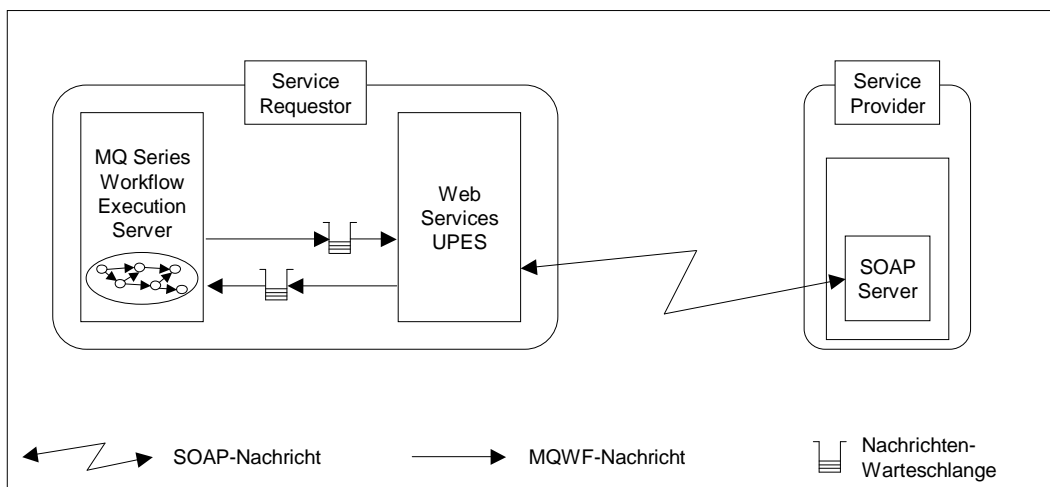


Abbildung 2.3: Aufruf eines Web Service als Aktivitäts-Implementierung

Der im WSPMTK enthaltene UPES kann auf zwei Arten benutzt werden. Eine Möglichkeit ist, wie in Abbildung 2.4 auf der nächsten Seite dargestellt, dem UPES eine WSDL-Beschreibung zu übergeben, aus der dieser die notwendigen Informationen bezieht, um den Web Service aufzurufen. Die zweite Möglichkeit besteht in der Übergabe einer Java-Klasse, die den Aufruf des Web Service kapselt (Proxy), vgl. Abbildung 2.5 auf der nächsten Seite. Die Details zum Betreiben des UPES sind in der Dokumentation zum WSPMTK beschrieben.

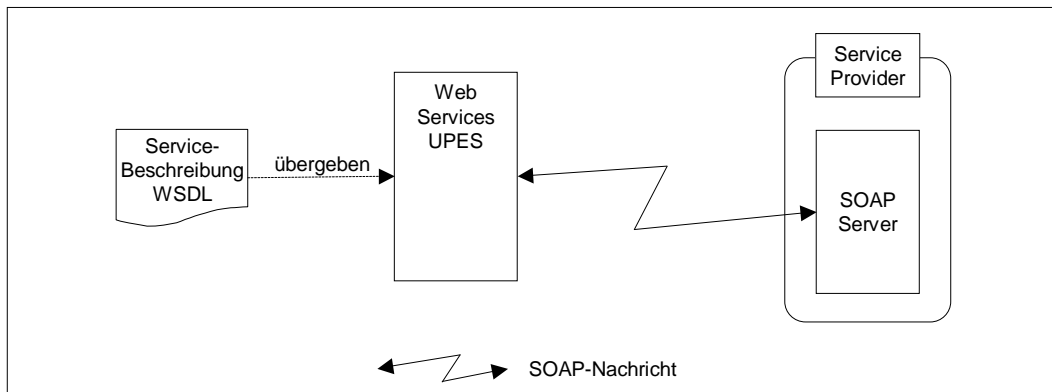


Abbildung 2.4: UPES-Aufruf, abgeleitet aus WSDL-Dokument

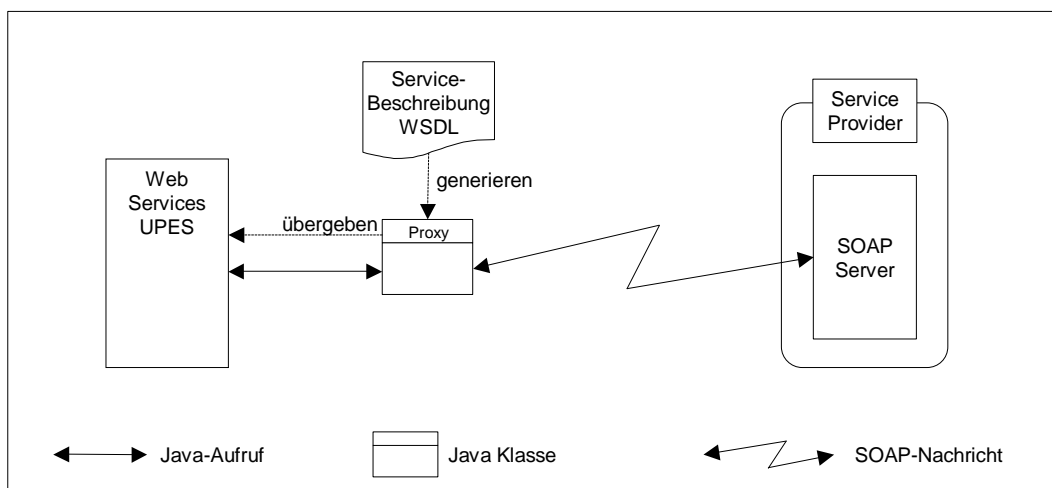


Abbildung 2.5: UPES-Aufruf über Java-Proxy

Bevor von einem Workflow auf einen Web Service über diese generische Web-Service-Schnittstelle zugegriffen werden kann, müssen die notwendigen Definitionen für die Aktivitäts-Implementierung und den UPES in die Runtime-Umgebung von MQWF importiert werden. Zu diesem Zweck kann mit Hilfe eines Software-Werkzeuges mit der Bezeichnung `wsdl2fdl` aus der WSDL-Beschreibung ein FDL-Dokument erzeugt werden, das alle notwendigen FDL-Konstrukte enthält. Dieses Programm befindet sich noch im Entwicklungsstadium und ist noch nicht Teil des aktuellen WSPMTK. Alle Parameter für MQWF, die nicht aus der WSDL-Beschreibung abgeleitet werden können, werden `wsdl2fdl` in Form einer textbasierten Konfigurationsdatei übergeben, vgl. Abbildung 2.6 auf der nächsten Seite.

Die Eingaben zu `wsdl2fdl` bestehen aus einem WSDL-Dokument und einer Konfigurationsdatei. Aus den Eingaben wird eine Datei mit folgenden FDL-Konstrukten generiert:

STRUCTURE Datenstrukturen, abgeleitet aus dem WSDL-Types-Element, so-

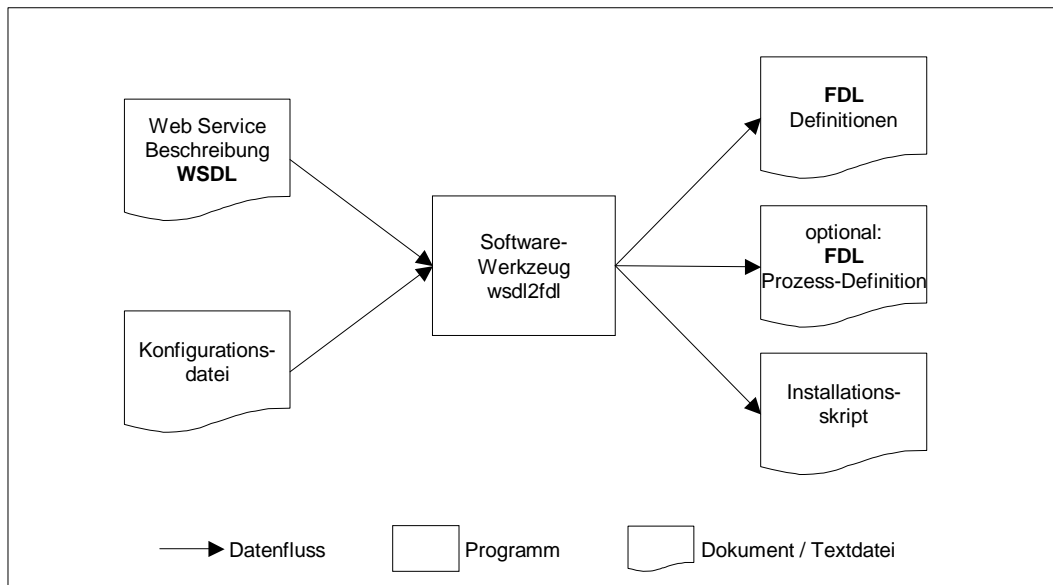


Abbildung 2.6: Das Softwarewerkzeug wsdl2fdl

fern diese als XML-Schema beschrieben sind. Eine Einschränkung besteht bezüglich Arrays mit offenen Arraygrenzen und verschachtelter Strukturen.

PROGRAM Implementierung einer Programmaktivität, für jedes WSDL-Operation-Element eines WSDL-PortType.

SERVER Definition des UPES, abgeleitet aus den Parametern der Konfigurations-datei.

PROCESS Optional kann auch eine unvollständige Prozessdefinition erstellt werden, die zwar Programmaktivitäten enthält, nicht jedoch Daten- und Kontrollflüsse.

Zusätzlich wird eine Stapelverarbeitungsdatei generiert, die für den Import der erzeugten FDL-Definitionen in die Runtime-Umgebung von MQWF benutzt werden kann. Dabei ist zu beachten, dass die Prozessdefinition ohne nachträgliche Modifikation nicht importiert werden kann.

3 Veröffentlichen von Workflow-Prozessen als Web Service in UDDI

In Abschnitt 1.2 wurde das Web-Services-Modell in seinen Grundzügen erläutert. Dabei wurde deutlich gemacht, dass eine Trennung besteht zwischen dem Erstellen und dem Veröffentlichen eines Web Service. Betrachtet man den Weg vom i.A. vorhandenen Workflow-Prozess zu UDDI, sind beide Vorgänge tangiert, insbesondere unter Gesichtspunkten der Automatisierung.

In den folgenden Abschnitten wird der Gesamtprozess zur Veröffentlichung analysiert und die notwendigen Teilschritte näher beleuchtet. Diese werden im Hinblick auf bestehende Systeme und Implementierungen erläutert und auf neu zu schaffende Voraussetzungen überprüft. Auf dieser Grundlage werden alternative Ansätze zur Automation dieses Gesamtprozesses vorgestellt und bewertet.

3.1 Überblick: Veröffentlichungsprozess

Für das Veröffentlichen eines Workflow-Prozesses als Web Service in UDDI sind folgende Schritte notwendig:

- Ermittlung der Prozessdefinition in FDL
- Erstellung und Installation der notwendigen Web-Service-Programme
- Generierung der Web-Service-Beschreibung in WSDL
- Registrierung der Web-Service-Beschreibung in UDDI

Diese Schritte werden in Abbildung 3.1 auf der nächsten Seite als Pfeile dargestellt.

Ausgangspunkt für die Veröffentlichung ist die Workflow-Definition. Diese kann entweder aus der Runtime- oder aus der Buildtime-Datenbank von MQWF exportiert und in Textform gespeichert werden. Nach dem Export liegt die Prozessbeschreibung in Flow Definition Language (FDL) vor (vgl. Abschnitt 1.7).

Die FDL Beschreibung muss folgende Definitionen enthalten:

- Workflow-Prozess mit Name, Ein- und Ausgabedaten
- Datenstrukturen der Input- und Output-Container.

Aus der Workflow-Definition kann mit dem Software-Werkzeug `fd12wsdl` die Web-Service-Schnittstelle zu MQWF erzeugt werden. Die Schnittstelle besteht aus einen generischen Übersetzer, mit dem SOAP- in MQWF-Nachrichten transformiert werden können und den Java-Klassen, die diese Übersetzung zwischen den Prozess- Datenstrukturen und den SOAP-Datenstrukturen im Einzelfall implementieren. Zur

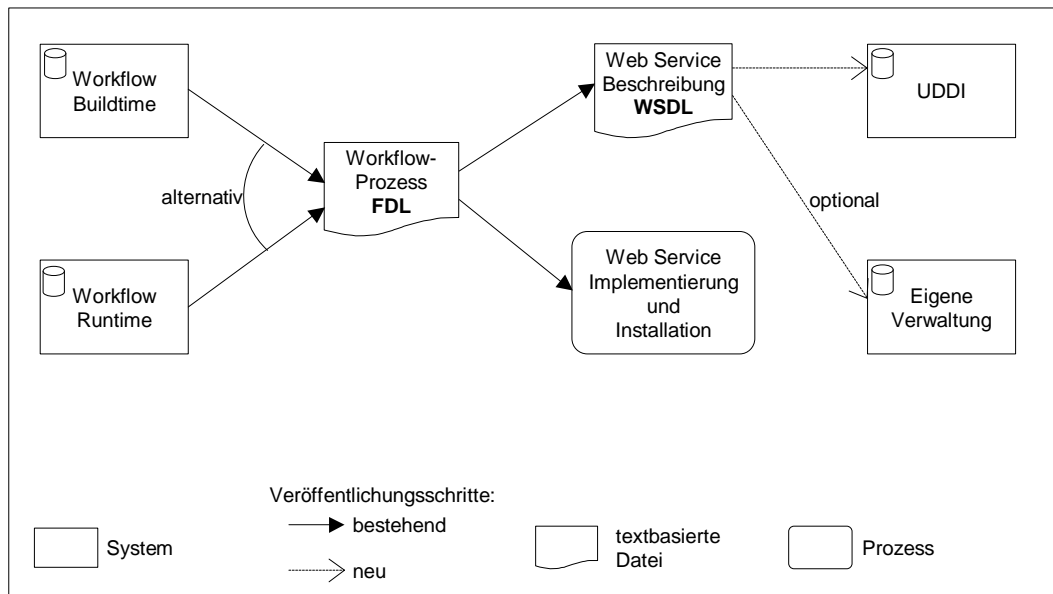


Abbildung 3.1: Überblick über den Veröffentlichungsprozess

Registrierung dieser Schnittstelle als SOAP-Service wird außerdem eine Beschreibungsdatei (Deployment-Descriptor) und eine Stapelverarbeitungsdatei zur automatischen Installation generiert.

Weiter wird mit `wsdl2fdl` eine Web-Service-Beschreibung in WSDL erzeugt. Damit die Beschreibung extern zur Verfügung steht, muss sie über das Internet, z.B. über einen Web Server, zugänglich gemacht werden. Dies kann durch Eintrag in eine Datenbank oder durch einfaches Kopieren in ein entsprechendes Verzeichnis ermöglicht werden. Um den Web Service selbst verfügbar zu machen, wird dieser mit Hilfe der Skriptdatei beim Application Server angemeldet.

Im nächsten Schritt wird der Web Service in UDDI registriert. Dazu müssen die UDDI-Einträge von Hand über das Web-Browser-Interface erstellt oder mit Hilfe eines weiteren Software-Werkzeuges generiert und über die Programmierschnittstelle in UDDI registriert werden. Voraussetzung dafür ist, dass der Benutzer bereits ein Konto bei einem UDDI-Betreiber angemeldet hat. Im weiteren Verlauf dieser Arbeit wird nur noch die UDDI-Programmierschnittstelle betrachtet.

In manchen Situationen ist eine lokale Speicherung von Informationen über veröffentlichte Web Services sinnvoll. Zu diesem Zweck könnten z.B. Namen, Schlüssel oder Versionen in einer Datenbank abgelegt werden. Dieser Schritt ist optional und wird im weiteren Verlauf dieser Arbeit ausgeklammert.

3.2 Einzelschritte des Veröffentlichungsprozesses

Nach der eher grundsätzlichen Betrachtung des Veröffentlichungsprozesses im vorhergegangenen Abschnitt wird jetzt jeder Einzelschritt im Detail betrachtet.

Voraussetzungen bei der Prozessdefinition

Vor allem bei bestehenden Prozessdefinitionen ist vor dem Veröffentlichen sicherzustellen, dass nur solche Daten nach außen hin sichtbar gemacht werden, die nicht geheim bleiben sollen. Dies gilt vor allem für die Prozess-Output-Container. Workflows, die unverändert beibehalten werden und deren Ausgabedaten trotzdem verborgen bleiben sollen, müssen somit in einem neuen Prozess gekapselt werden. Die neue Prozessdefinition enthält neben den neuen Datenstrukturen und den entsprechenden Zuordnungen nur eine Prozessaktivität, die den vorhandenen Prozess aufruft.

Dieses Prinzip wird in Abbildung 3.2 verdeutlicht. Der gekapselte Prozess P2 enthält geheime Daten. Durch die Kapselung von P2 in P1 und den entsprechenden Datenzuordnungen bleiben die geheimen Daten verborgen. Ein Nebeneffekt dieser Vorgehensweise ist eine Verkleinerung der Schnittstelle und des zu übertragenden Datenvolumens.

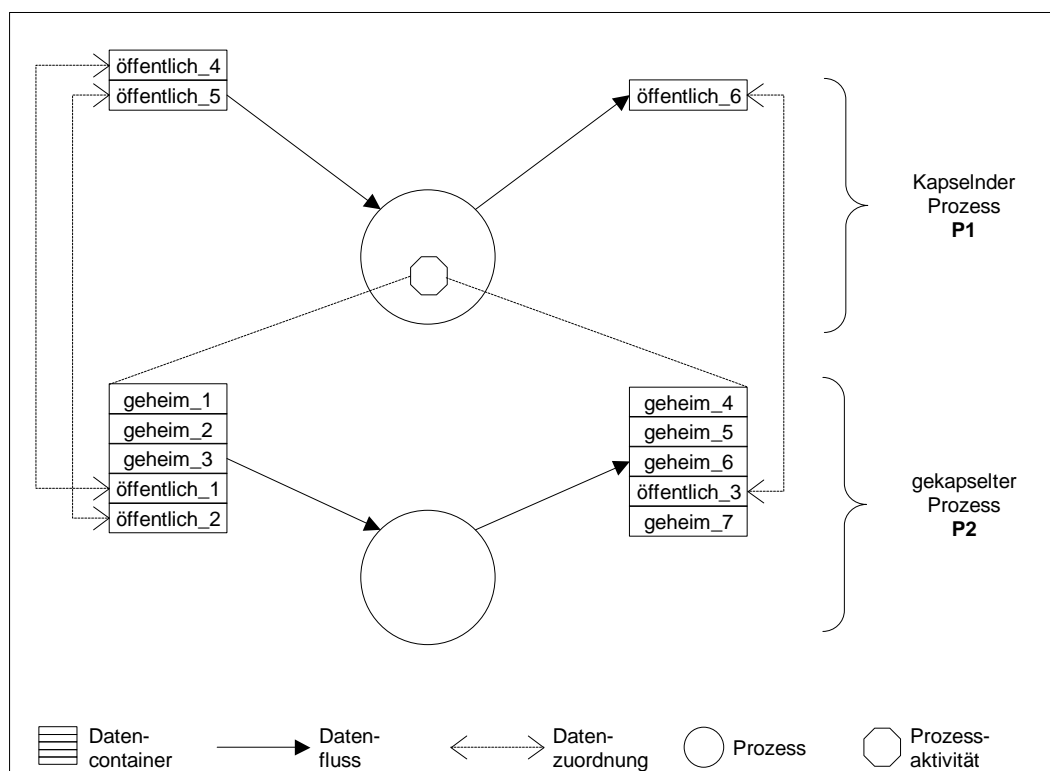


Abbildung 3.2: Kapselung eines Workflow-Prozesses

Um einen Workflow-Prozess als Web Service zu veröffentlichen sind auch Informationen notwendig, die nicht Teil der Prozess-Definition sind. Neben dem Namen für den Web Service muss festgelegt werden, welche Methoden zum Starten eines Workflow-Prozesses unterstützt werden sollen. Die folgenden Methoden werden durch MQWF zur Verfügung gestellt:

call synchroner Aufruf eines Prozesses

spawn asynchroner Aufruf eines Prozesses

terminate Abbruch der Prozessausführung

restart Neustart eines Prozesses

inquire Abfrage von Statusinformationen über einen Prozess.

Ohne Angabe einer oder mehrerer Methoden wird durch `fd12wsdl` nur der call-Aufruf veröffentlicht.

Bei der Namensvergabe ist darauf zu achten, dass der Namensraum so gewählt wird, dass Namenskonflikte bei den beteiligten Web Services und deren Methoden bzw. Datenobjekten ausgeschlossen sind.

Eine ausführliche Beschreibung aller Steuerungsparameter befindet sich in der Dokumentation des WSPMTK.

Generierung und Installation der Web-Service-Schnittstelle

Bis die Web-Service-Schnittstelle zu einem Workflow-Prozess etabliert ist, muss ein zweistufiger Prozess durchlaufen werden. Im ersten Schritt werden, wie in Abschnitt 2.1 beschrieben, die notwendigen Java-Klassen zur Typumwandlung, die WSDL-Beschreibung und ein Installationskript erzeugt. Im zweiten Schritt werden die Klassen übersetzt und die Schnittstelle anschließend in der SOAP-Umgebung registriert. Für diesen zweiten Schritt kann das generierte Installationskript benutzt werden.

Web Service Beschreibung öffentlich zugänglich machen

Das WSDL-Dokument steht nach der Erzeugung in dem in der Konfigurationsdatei zu `fd12wsdl` festgelegten Verzeichnis. Damit ein potentieller Nutzer des neu etablierten Web Service von außen auf die notwendigen Schnittstelleninformationen zugreifen kann, muss eine dafür eine Zugriffsmöglichkeit geschaffen werden. Das Abrufen des WSDL-Dokumentes erfolgt im Idealfall mit Hilfe eines Web Browsers, der das Dokument über den URL von einem Web Server anfordert.

Eine simple Möglichkeit, die Web-Service-Beschreibung bei einem Web Server zu registrieren ist, die Datei in ein bestimmtes Verzeichnis des Web Servers zu kopieren. Daneben stehen noch weitere Möglichkeiten zur Verfügung, die je nach Infrastruktur des Unternehmens beliebig komplex ausfallen können, z.B. Content-Management-Systeme.

Veröffentlichung des WSDL-Dokumentes in UDDI

Bevor ein Web Service in UDDI veröffentlicht werden kann, müssen folgende Voraussetzungen erfüllt sein:

- Der Benutzer bzw. das Unternehmen muss beim UDDI-Betreiber registriert sein.
- Für das Unternehmen muss ein BusinessEntity-Objekt in UDDI existieren, bei dem die Web Services eingetragen werden können.

Beide Voraussetzungen werden im Verlauf dieser Arbeit als gegeben betrachtet.

Wie in Abschnitt 1.5 ausgeführt verfügt UDDI über eine Schnittstelle in Form von SOAP-Nachrichten. Die Nachrichten zum Registrieren müssen dabei aus den Informationen der WSDL-Beschreibung erzeugt werden. Der Versand der Nachrichten kann dann auf verschiedene Arten erfolgen. Im Rahmen dieser Diplomarbeit werden drei Möglichkeiten vorgestellt: direkter Zugriff über SOAP, indirekter Zugriff über eine Java Programmierschnittstelle für UDDI (UDDI4J - UDDI for Java) und indirekter Zugriff über eine Java Programmierschnittstelle mit UDDI-Erweiterung (WSTK - IBM Web Services Toolkit).

Im Fall von direktem Zugriff auf UDDI über SOAP, dargestellt in Abbildung 3.3 muss der Programmierer des Anwendungsprogramms für die korrekte Erzeugung der SOAP-Nachrichten sorgen, ebenso für die Fehlerbehandlung für SOAP und UDDI. Die notwendigen Informationen zu den Web Services müssen aus der WSDL-Beschreibung abgeleitet werden.

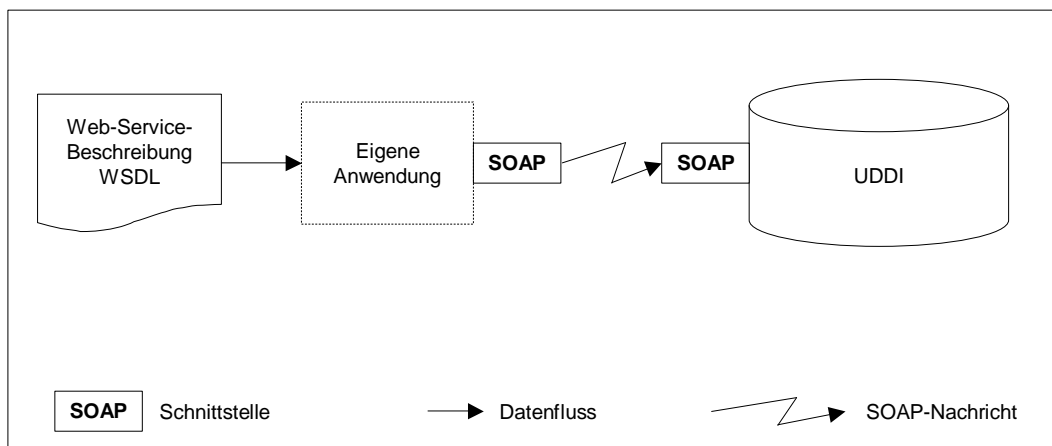


Abbildung 3.3: UDDI Zugriff über SOAP-Nachrichten

Abbildung 3.4 auf der nächsten Seite zeigt den Zugriff über UDDI4J. Auch in diesem Fall müssen die notwendigen Informationen aus dem WSDL-Dokument extrahiert werden, für den Aufbau der SOAP-Nachrichten und zur Fehlerbehandlung stehen jedoch Java-Klassen zur Verfügung.

Das Auslesen der Daten aus der WSDL-Beschreibung entfällt bei der Nutzung des WSTK. Wie in Abbildung 3.5 auf Seite 28 dargestellt, wird das WSDL-Dokument direkt durch Java-Klassen im WSTK analysiert und die entsprechenden SOAP-Nachrichten generiert. Somit ist das Anwendungsprogramm hauptsächlich für den Kontrollfluss verantwortlich.

Wie in Abschnitt 1.5 erläutert sollte die WSDL-Beschreibung für UDDI getrennt nach Schnittstelle und Implementierung vorliegen. Als Folge davon sind für die Veröffentlichung eines Web Services zwei Zugriffe auf UDDI erforderlich: einer für die Registrierung der Schnittstelle als TModel, der andere für den Implementierungsteil, der in den UDDI-Strukturen BusinessService und BindingTemplate gespeichert wird. Für die Registrierung des Implementierungsteils ist der Schlüssel des

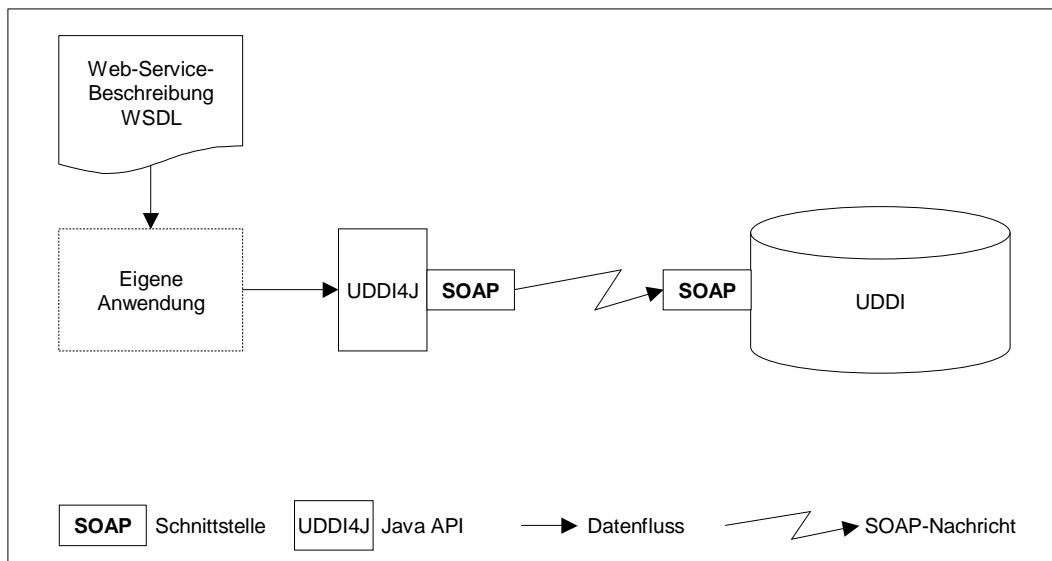


Abbildung 3.4: Zugriff auf UDDI über das UDDI4J-API

TModel-Objektes erforderlich, damit eine korrekte Zuordnung von Schnittstelle und Implementierung gewährleistet ist. Der Schlüssel (TModelKey) kann aus der Antwortnachricht der Interface-Registrierung extrahiert werden.

Um eine getrennte Registrierung zu ermöglichen, müssen die beiden Teilbeschreibungen zunächst aus dem WSDL-Dokument erzeugt werden. Als Resultat dieser Arbeit wurde das Software-Werkzeug `fd12wsdl` so abgeändert, dass die WSDL-Beschreibung bereits nach Schnittstelle und Implementierung getrennt erstellt werden kann.

Ein wesentlicher Punkt, der für die Nützlichkeit von UDDI als zentrale Auskunftsstelle für Web Services sehr bedeutend ist, besteht darin, die technischen Beschreibungen mit anwendungsbezogenen Informationen zu kombinieren, um ein gezieltes und möglichst effizientes Auffinden zu ermöglichen. UDDI bietet dafür die Möglichkeit an, BusinessService-, BusinessEntity- und TModel-Strukturen, in Kategorien einzuteilen.

Eine Kategorie bezieht sich auf eine Taxonomie, die als TModel in UDDI gespeichert ist. Für die Zuordnung eines Objektes zu einer Kategorie wird der TModel-Schlüssel, ein Schlüssel zur Identifikation eines Taxonomie-Eintrags und der Klartext dieses Taxonomie-Eintrags benötigt. Diese drei Informationen werden in einem KeyedReference-Objekt gespeichert. Um einem Objekt eine Kategorie zuzuordnen, muss ein KeyedReference-Objekt in dessen CategoryBag eingetragen werden. Soll ein Objekt mehreren Kategorien zugeordnet werden, muss für jede Kategorie ein entsprechendes KeyedReference-Objekt registriert werden. Das folgende Beispiel zeigt die Kategorisierung eines BusinessService-Objektes.

Listing 3.1 zeigt ein TModel-Objekt, das die Spezifikation einer fiktiven Taxonomie für Finanzdienstleistungen repräsentiert. Die eigentliche Spezifikation kann über die

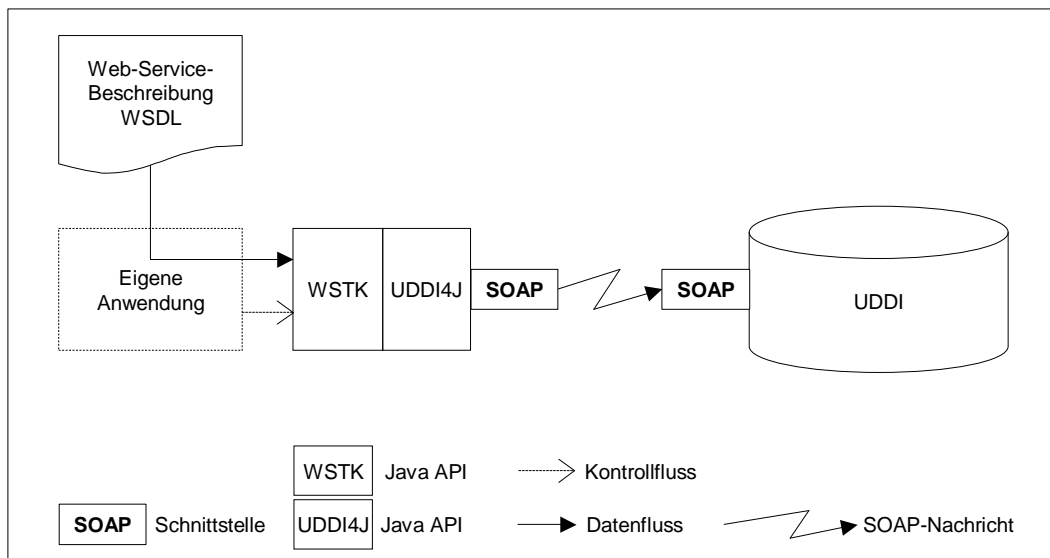


Abbildung 3.5: Zugriff auf UDDI über WSTK

fiktive URL `www.banking.de/tax/finanz.html` abgerufen werden. Weiter ist angedeutet, dass ein TModel ebenfalls kategorisiert werden kann.

```
<tModel tModelKey="123" ...>
  <name lang="de">Taxonomie fuer Finanzdienstleistungen</name>
  <overviewDoc>
    ...
    <overviewURL>www.banking.de/tax/finanz.html</overviewURL>
  </overviewDoc>
  <categoryBag>...</categoryBag>
</tModel>
```

Listing 3.1: Spezifikation einer Taxonomie

Listing 3.2 zeigt ein BusinessService-Objekt, das der Kategorie Kreditkartenüberprüfung der o.a. Taxonomie zugeordnet wurde.

```
<businessService>
  ...
  <bindingTemplates>...</bindingTemplates>
  <categoryBag>
    <keyedReference tModelKey="123"
      keyValue="9834"
      keyName="Kreditkartenueberpruefung"/>
  </categoryBag>
</businessService>
```

Listing 3.2: Kategorisierung eines Business-Service-Objektes

Die UDDI-Spezifikation gibt einige Kategorien vor, für die bereits TModel-Objekte in UDDI registriert sind:

NAICS eine US-amerikanische Taxonomie für Industrieunternehmen

UNSPSC eine US-amerikanische Taxonomie für Leistungen von Industrieunternehmen

ISO 3166 Zuordnung von bestimmten Ländercodes, z.B. DE für Deutschland

GENERAL KEYWORDS Schlagwörter

Daneben schreibt UDDI auch noch eine Taxonomie für Spezifikationstypen vor, anhand derer TModel-Objekte kategorisiert werden können. Darin gibt es z.B. eine Kategorie für Spezifikationen in WSDL. Darüber hinaus können beliebig viele andere Taxonomien als TModel registriert und für die Kategorisierung benutzt werden.

Beim Registrieren eines Web Service in UDDI müssen solche Kategorisierungsinformationen extra hinzugefügt werden, denn diese können nicht aus der WSDL-Beschreibung abgeleitet werden. Die Schnittstelle, die als TModel gespeichert wird sollte neben den anwendungsbezogenen Kategorien auch die eingebaute Taxonomie „WSDL Document“ enthalten.

Ähnlich wie die Kategorien müssen auch die nicht technischen Informationen über die Web Services zu den UDDI-Strukturen hinzugefügt werden. Solche Dokumentationsinformationen existieren in der Regel bereits in der FDL-Beschreibung der Workflow-Prozesse in den Konstrukten DESCRIPTION bzw. DOCUMENTATION. Diese werden allerdings in der aktuellen Version von `fd12wsdl` nicht in die WSDL-Beschreibung übernommen und müssen deshalb in einem späteren Schritt, entweder nachträglich in die WSDL-Beschreibung oder direkt in die UDDI-Strukturen eingefügt werden.

Für jeden schreibenden Zugriff auf UDDI sind Autorisierungsdaten in Form eines Benutzernamens und einem Kennwort erforderlich. Zusammen mit den Netzwerkadressen der Schnittstellen müssen diese Autorisierungsdaten dem Anwendungsprogramm zugänglich gemacht werden.

Abbildung 3.6 auf der nächsten Seite fasst den Veröffentlichungsvorgang, ausgehend von einem nach Schnittstelle und Implementierung getrennten WSDL-Dokument zusammen.

3.3 Realisierungsalternativen für den Gesamtprozess

In den folgenden Ausführungen werden drei alternativen Ansätze für die Automatisierung des Gesamtprozesses betrachtet:

- Kapselung des Gesamtprozesses in einem Anwendungsprogramm,

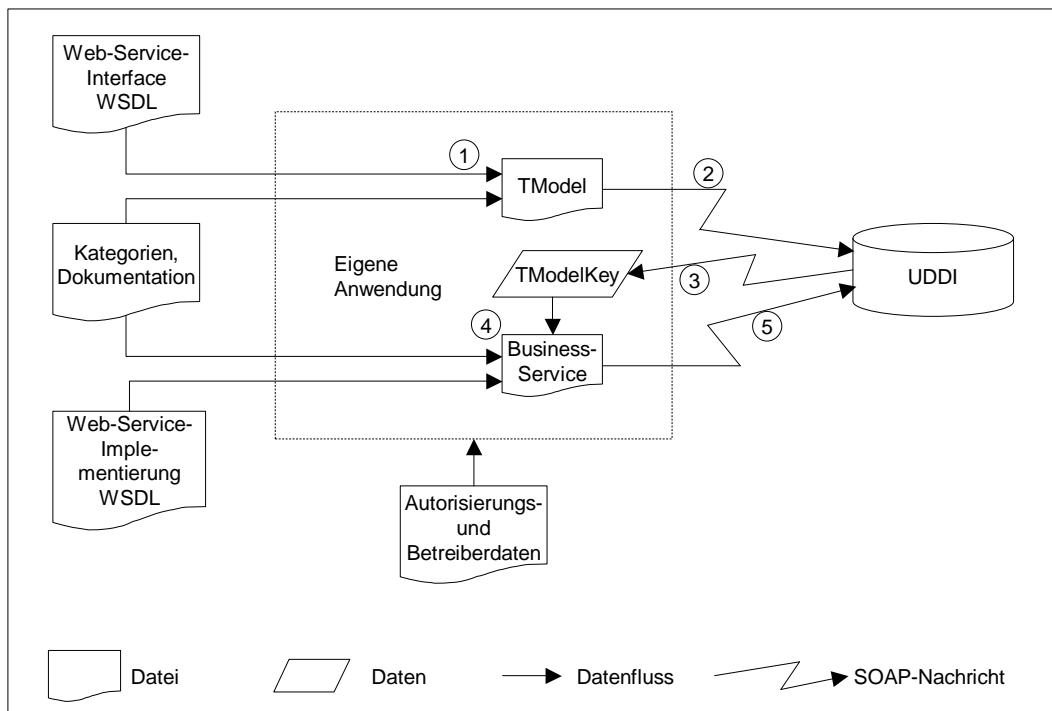


Abbildung 3.6: Publizieren einer WSDL-Beschreibung in UDDI

- Realisierung als Workflow-Prozess,
- Realisierung in Form einer Skriptsprache.

Dabei liegt der Schwerpunkt der Betrachtungen auf konzeptioneller Ebene, wenn auch in Einzelfällen auf Detailprobleme Bezug genommen wird, sofern diese einen bedeutenden Einfluss auf die Wahl der Realisierungsalternative hat.

Als Ergebnis des vorhergehenden Abschnittes wird deutlich, dass bei der Realisierung sowohl vorhandene Software-Werkzeuge als auch neu zu realisierende Programme integriert werden müssen. Die zu integrierenden Werkzeuge sind zum Großteil als Java-Klassen implementiert, die über Stapelverarbeitungsdateien ausgeführt werden. Ein Teil der notwendigen Klassen und Skriptdateien wird im Laufe des Veröffentlichungsvorganges generiert.

Ein wichtiger Aspekt ist auch, dass die benötigten Informationen in sehr unterschiedlicher Form vorliegen, z.B. die Workflow-Definitionen in einer FDL-Datei, die SOAP-Informationen in einer Konfigurationsdatei. Darüber hinaus werden in einzelnen Quellen sowohl prozessbezogene, z.B. Namensräume, als auch administrative Daten, z.B. Arbeitsverzeichnis, gehalten. Schließlich müssen neue, UDDI-bezogene Datenquellen integriert werden.

Zusammenfassend kann man feststellen, dass bei der Wahl der Realisierungsalternative drei Probleme maßgeblich sind: Integration vorhandener Software-Werkzeuge,

heterogene Datenquellen in Dateiform und das Einbinden von zur Laufzeit generierten Klassen und Skripte.

Die erste zu betrachtende Alternative ist die Prozessautomation mit Hilfe eines WFMS. Die Idee dabei ist, die Einzelschritte als Programmaktivitäten zu modellieren, die durch das WFMS koordiniert werden. Die erforderlichen Parameter werden aus den entsprechenden Daten-Containern ausgelesen und an die Software-Werkzeuge übergeben. Nach der Ausführung werden die Ergebnisse in die Ausgabe-Container zurückgeschrieben. Abbildung 3.7 illustriert den Veröffentlichungsprozess als Workflow.

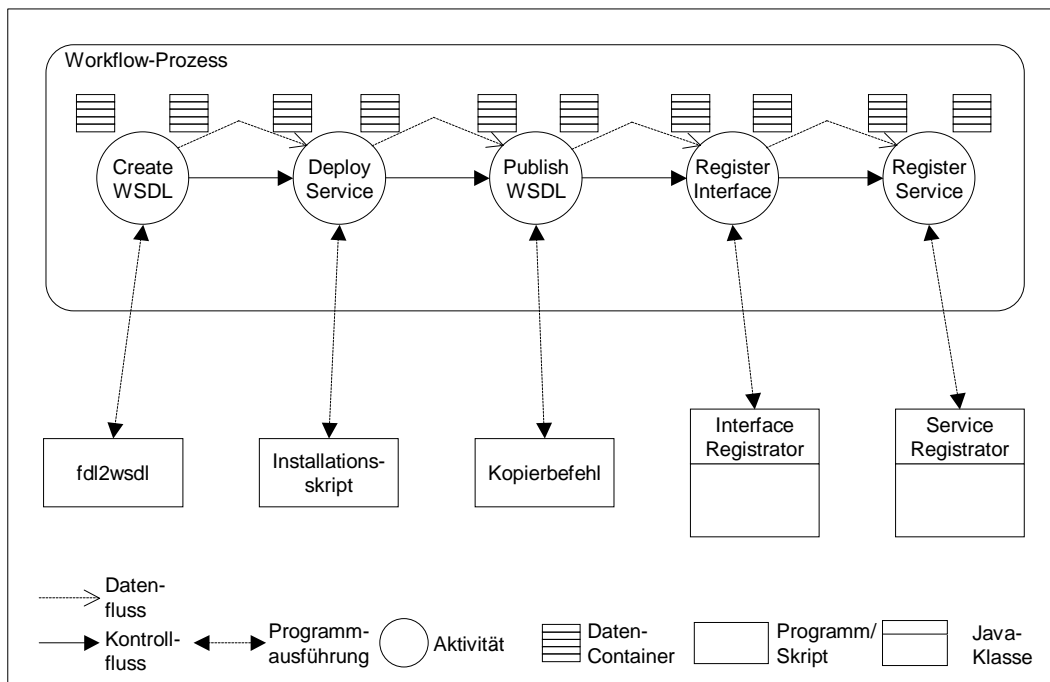


Abbildung 3.7: Veröffentlichung, gesteuert durch WFMS

Die Integration der unterschiedlichen Software-Werkzeuge ist bei diesem Ansatz auf einfache Art und Weise zu bewerkstelligen. Das gilt auch für die Kontrolllogik für den Gesamtprozess, bei dem die Einzelschritte sequentiell ausgeführt werden. Als Problem erweist sich dagegen die Integration der generierten Stapelverarbeitungsdatei. Der Name dieser Stapelverarbeitungsdatei ist abhängig von der Workflow-Definition und von der Konfigurationsdatei zu `fd12wsdl`. Zwar kann beim Veröffentlichungsprozess eine Programmaktivität für das Skript generiert werden, jedoch muss entweder die Programm-Definition bei jedem Durchlauf geändert und neu ins WFMS importiert werden oder das Werkzeug so geändert werden, dass die generierte Stapelverarbeitung immer den gleichen Namen erhält.

Ein weiteres, gravierendes Problem ist das zurückschreiben von Werten in die Ausgabe-Container. Das Übergeben von Container-Werten als Programmparameter ist auf einfache Weise möglich, nicht jedoch das Zurückschreiben von Resultaten. Rückgabewerte müssen von den Anwendungsprogrammen selbst in die Container geschrie-

ben werden, z.B. durch Nutzung des Java-API für MQ Series Workflow. Da die Software-Werkzeuge nicht für den Einsatz als Programme in MQWF konzipiert wurden, müssen diese entweder geändert oder Hilfsprogramme entwickelt werden, die die Ausgabedateien der Software-Werkzeuge analysieren und die erforderlichen Daten in die Ausgabe-Container schreiben.

Ebenso problematisch ist die Angabe von Parametern, die immer nur für einen bestimmten zu veröffentlichenden Workflow-Prozess gelten, z.B. der Dateiname für die Kategorie-Informationen. Eine Möglichkeit ist, alle veränderlichen Parameter beim Prozess-Start von Hand einzugeben. Um sich die Eingabe der Parameter von Hand zu ersparen, können diese auch direkt in der Prozessdefinition als Vorgabewerte gespeichert werden. Die Vorgaben können dann beim Prozessstart von Hand überschrieben werden, falls abweichende Parameter benötigt werden. Bei jeder dauerhaften Änderung der Vorgabewerte muss die Prozessdefinition geändert werden.

Eine wichtige Grundannahme für das Ausführen von Prozessen unter der Regie von WFMS ist eine hohe Wiederholungsrate solcher Vorgänge. Ob diese im Fall der Veröffentlichung von Prozessen als Web Service in UDDI erreicht wird, kann dann angezweifelt werden, wenn nur bestimmte Web Services außerhalb eines Unternehmens veröffentlicht werden sollen, z.B. externe Bestellungen, etc. Werden viele Prozesse veröffentlicht, z.B. im Rahmen einer internen Web Service Architektur mit privater UDDI-Nutzung, kann durchaus eine hohe Wiederholungsrate erreicht werden.

Die zweite Alternative zur Realisierung des Veröffentlichungsprozesses ist, ein Anwendungsprogramm zu entwickeln, das den Gesamtvorgang steuert. Als Programmiersprache bzw. Plattform bietet sich Java an, denn die zu integrierenden Software-Werkzeuge sind ebenfalls in Java realisiert. Letztere können also auch direkt aus einem Programm gestartet werden, ohne dass auf die Stapelverarbeitungsdateien zurückgegriffen werden muss. Dieser zweite Ansatz ist Abbildung 3.8 dargestellt.

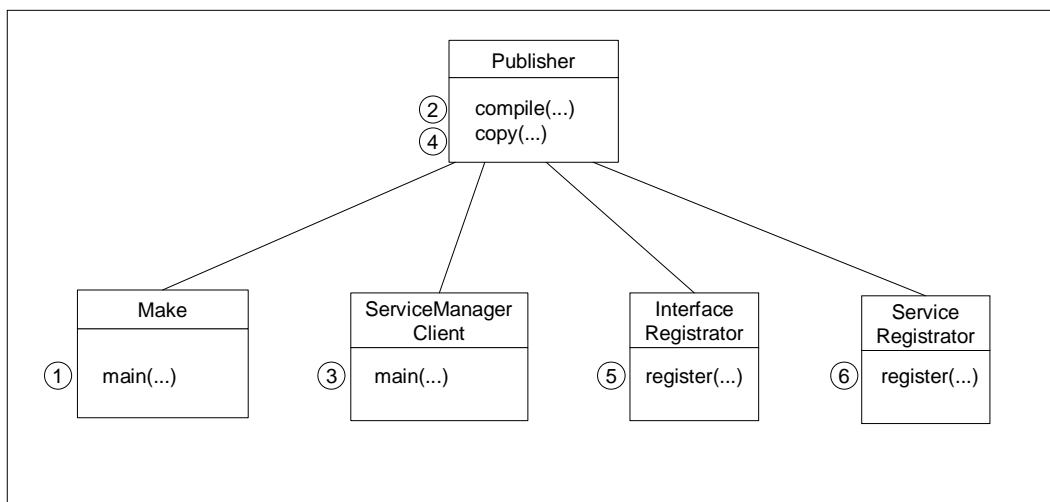


Abbildung 3.8: Veröffentlichung, gesteuert durch Anwendungsprogramm

Ähnliche Probleme wie beim ersten Ansatz bereitet der Zugriff auf die Rückgabewerte der zu integrierenden Werkzeuge. Auch hier müssen entweder die Werkzeuge verändert werden oder Klassen zur Analyse der Ausgabedateien entwickelt werden, um bestimmte, aus verschiedenen Quellen generierte Informationen für die weitere Verarbeitung zur Verfügung zu haben. Dabei entfällt gegenüber dem ersten Ansatz der Zwischenschritt zur Speicherung der Daten in einem Container innerhalb von MQWF.

Der Zugriff auf Systemprogramme, z.B. Copy, ist nicht so einfach, wie beim ersten Ansatz. Systemprogramme bzw. -Befehle müssen entweder in Java nachgebildet werden oder über einen Systemaufruf realisiert werden.

Schwerwiegender ist jedoch, dass die zweite Alternative ähnlich unflexibel gegenüber Änderungen im Veröffentlichungsprozess ist. So müssen beim ersten Ansatz das Prozessmodell und evtl. der Zugriff auf die Rückgabewerte neu gestaltet, beim zweiten muss das Anwendungsprogramm überarbeitet werden.

Die dritte Alternative für die Automatisierung des Veröffentlichungsprozesses ist, eine Skriptsprache zu verwenden, vgl. Abbildung 3.9. Dabei werden die vorhandenen Software-Werkzeuge unverändert aufgerufen. Die Problematik der Rückgabewerte ist ähnlich wie bei den vorhergehenden Ansätzen, jedoch erlauben die meisten Skriptsprachen die Verwendung von Platzhaltern, so dass ein Zugriff auf Rückgabewerte in bestimmten Fällen nicht mehr notwendig ist.

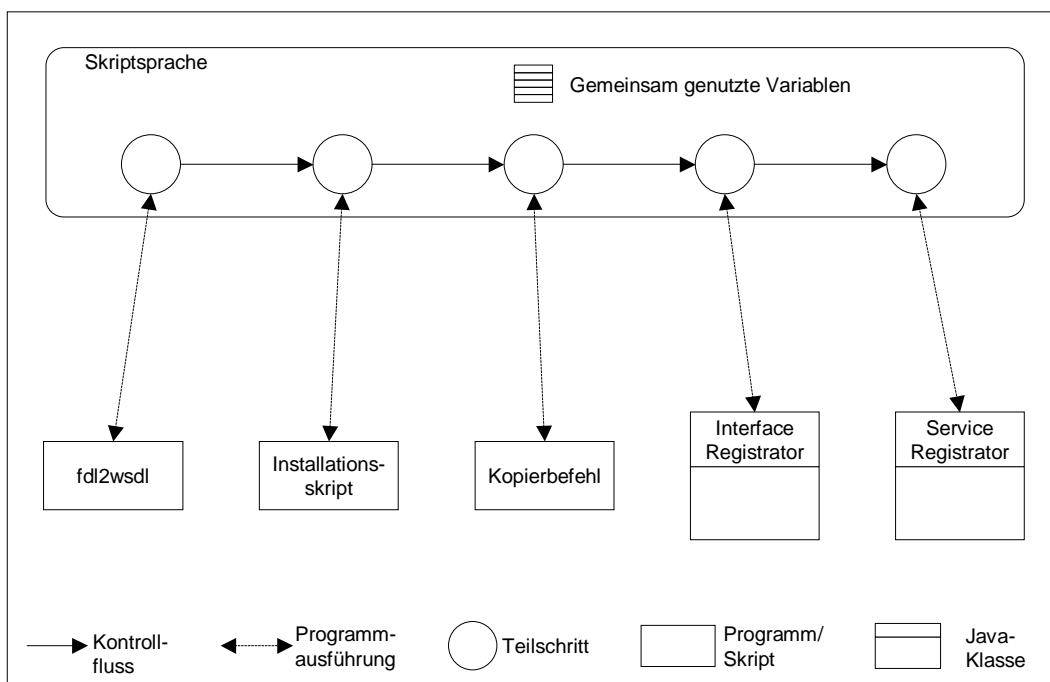


Abbildung 3.9: Skript-gesteuerter Veröffentlichungsprozess

Für die Verwendung einer Skriptsprache spricht auch die Flexibilität bei Veränderungen im Prozess. So können neuere Versionen von aufgerufenen Programmen mit

minimalem Änderungsaufwand in den Prozess integriert werden. Insgesamt ist der Programmieraufwand geringer als in den beiden anderen Alternativen. Zwar muss auch hier der Datenzugriff programmiert werden, wenn keine Platzhalter verwendet werden können, jedoch muss der Prozessablauf selbst nicht in einer Programmiersprache nachvollzogen werden. Auch können in einer Skriptsprache die Systembefehle, z.B. Copy, auf einfache Weise genutzt werden, während diese bei der Benutzung von Java entweder über den System-Exit aufgerufen oder aber explizit programmiert werden müssen.

Eine interessante Eigenschaft mancher Skriptsprachen ist, dass ein Skript in der gleichen Sprache rekursiv aufgerufen werden kann. Dabei hat das gerufene Skript Zugriff auf die gleichen Variablen wie das rufende. Diese Eigenschaft kann im Hinblick auf die Steuerung des Prozesses dann von Vorteil sein, wenn wie bei `fd12wsdl` Skripte generiert werden. Wird als Ausgabe die gleiche Skriptsprache verwendet wie zur Prozesssteuerung, können z.B. Rückgabewerte, die für spätere Verarbeitungsschritte relevant sind, in globalen Variablen gespeichert werden.

Die dargestellten Eigenschaften der Realisierungsalternativen werden in der folgenden Tabelle zusammengefasst:

Realisierungs- alternative	Workflow-Prozess	Java-Anwendung	Skript
Kontrolllogik	Prozessmodell	Java-Klasse	Skriptsprache
Datenfluss	Direkte Parameter- übergabe, Rückgabewerte über Workflow-API, Analyse von Zwischenresultaten notwendig	Direkte Parameter- übergabe, Zugriff auf Zwischenresultate	Direkte Parameter- übergabe, Verwendung von Platzhaltern, Analyse von Zwischenresultaten notwendig
Änderungs- aufwand	Prozessmodell ändern und importieren	Programm ändern und neu übersetzen	Skript editieren
Menschliche Interaktion	Evtl. Eingabe von Parametern	Keine	Keine
Steuerungs- parameter	Vorgabewerte, Externe Datei, Manuelle Eingabe	Externe Datei	Externe Datei, Direkt im Skript
Integration heterogener Programme	Alle Programme, Keine Rückgabe- werte	Java-Klassen	Alle Programme, Keine Rückgabe- werte
GUI-Anbindung	Indirekt möglich	Direkt möglich	Indirekt möglich
Generierung von Teilschritten	FDL erzeugen und importieren	Java-Klassen erzeugen und übersetzen	Skript generieren und aufrufen

Im Regelfall wird ein Veröffentlichungsvorgang für einen Workflow-Prozess einmal durchlaufen. Aus diesem Grund sollte eine leicht zu modifizierende Alternative gewählt

werden, die Änderungen ohne größeren Programmieraufwand zulässt. Unter Gesichtspunkten der Flexibilität und des Änderungsaufwandes stellt die Skriptlösung eine interessante Alternative dar. Soll eine grafische Benutzungsschnittstelle integriert werden, bietet die Java-Lösung Vorteile.

4 Einbinden von in UDDI publizierten Web Services in Workflow-Prozesse

In diesem Abschnitt wird das Einbinden von in UDDI veröffentlichten Web Services auf Basis der in Abschnitt 2 dargestellten Web-Service-Erweiterungen zu MQWF erörtert.

4.1 Überblick: Einbinden eines Web Service

Ausgangspunkt für das Einbinden von Web Services in einen Workflow-Prozess ist die Überlegung des Prozess-Modellierers, ob eine Lösung für eine bestimmte Aufgabe im Geschäftsprozess neu implementiert werden muss, oder ob eine solche Lösung als Web Service im Intra- bzw. Internet zur Verfügung steht. Fällt die Entscheidung für die Web-Service-Alternative, sind eine Reihe von Einzelschritten erforderlich, bis letztlich der Web Service aus einem Workflow-Prozess heraus aufgerufen werden kann:

- Auffinden von Web Services, die eine bestimmte Aufgabe erfüllen,
- Auswahl des Web Service, der tatsächlich eingebunden wird,
- Entscheidung, wie der UPES-Aufruf für den Web Service erfolgen soll, entweder über ein WSDL-Dokument oder eine Proxy-Klasse,
- Evtl. Generierung einer Proxy-Klasse,
- Einbinden der erforderlichen Definitionen für die Datenstrukturen und Programme in die Prozess-Definition,
- Aufruf des Web Service aus einem Workflow-Prozess heraus.

Die eben dargestellten Schritte müssen allerdings nicht zwangsweise sequentiell und nicht zusammenhängend durchlaufen werden. Wie später noch zu sehen sein wird, ist der Ausführungszeitpunkt und die Ausführungsreihenfolge abhängig von der Entscheidung, ob ein Web Service statisch, also zum Modellierungszeitpunkt, oder dynamisch, d.h. zum Ausführungszeitpunkt des Workflow-Prozesses, eingebunden werden soll.

In Abbildung 4.1 auf der nächsten Seite sind diese Schritte dargestellt. Das Auffinden umfasst das Suchen und die Auswahl eines Web Services. Das Binden steht als Sammelbegriff für die Generierung der FDL-Konstrukte, den Import in die Runtime-Datenbank von MQWF und den eigentlichen Aufruf des Web Service.

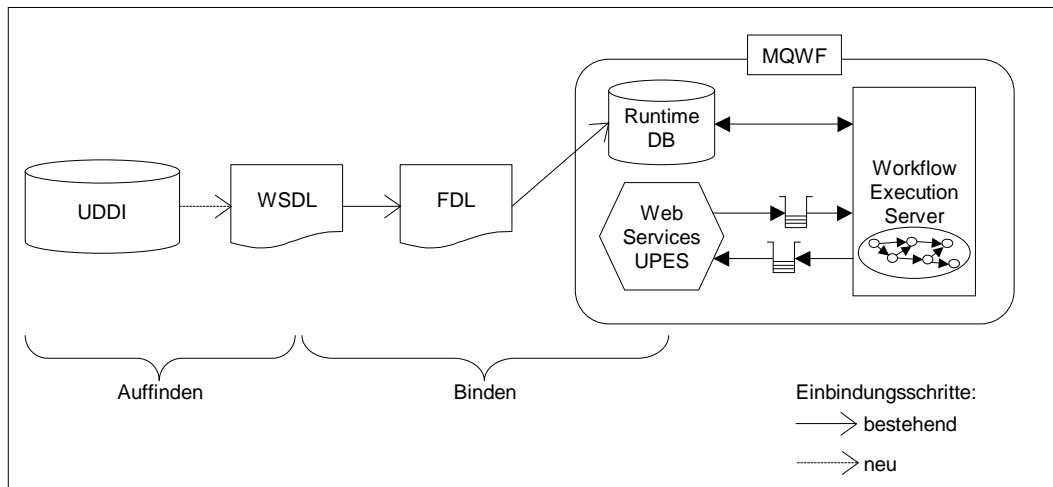


Abbildung 4.1: Auffinden und Binden eines Web Service für MQWF

4.2 Auffinden von Web Services in UDDI

Das Auffinden eines Web Service in UDDI hat zum Ziel, einen Web Service in UDDI zu finden, mit dem eine bestimmte Aufgabe gelöst werden kann und der somit als Implementierung einer Programm-Aktivität genutzt werden kann. Dieser Vorgang kann unterteilt werden in zwei Teilaufgaben: das Suchen und die Auswahl, vgl. Abbildung 4.2 auf der nächsten Seite.

Das Ergebnis des Suchens ist eine Menge an Web Services, die als Resultat einer oder mehrerer Suchanfragen an UDDI hervorgeht. Aus dieser Treffermenge, muss schließlich der Web Service ausgewählt werden, der dann als Aktivitäts-Implementierung tatsächlich benutzt wird. Die Größe der Treffermenge hängt von folgenden Faktoren ab:

- unterstützte Anfragemöglichkeiten der UDDI-Programmierschnittstelle
- Anfrageformulierung
- Nutzung von Taxonomien bei den registrierten Web Services

Die aufgelisteten Faktoren beziehen sich auf die Nutzung von UDDI über die Programmierschnittstelle. Da das Modellieren eines Workflow-Prozesses ein Vorgang ist, bei dem menschliche Interaktion im Vordergrund steht, kann das Auffinden auch über andere Schnittstellen erfolgen, z.B. über Web-Browser.

Anfragemöglichkeiten in UDDI

UDDI unterstützt zwei verschiedenen Anfragetypen mit unterschiedlicher Zielrichtung: eine Suchanfrage `find`, die als Resultat eine Liste mit qualifizierten Objekten ausgibt, bei der nicht alle Detailinformationen enthalten sind, und eine Detailanfrage `get`, mit der Detailinformationen zu einem bestimmten Objekt abgerufen werden

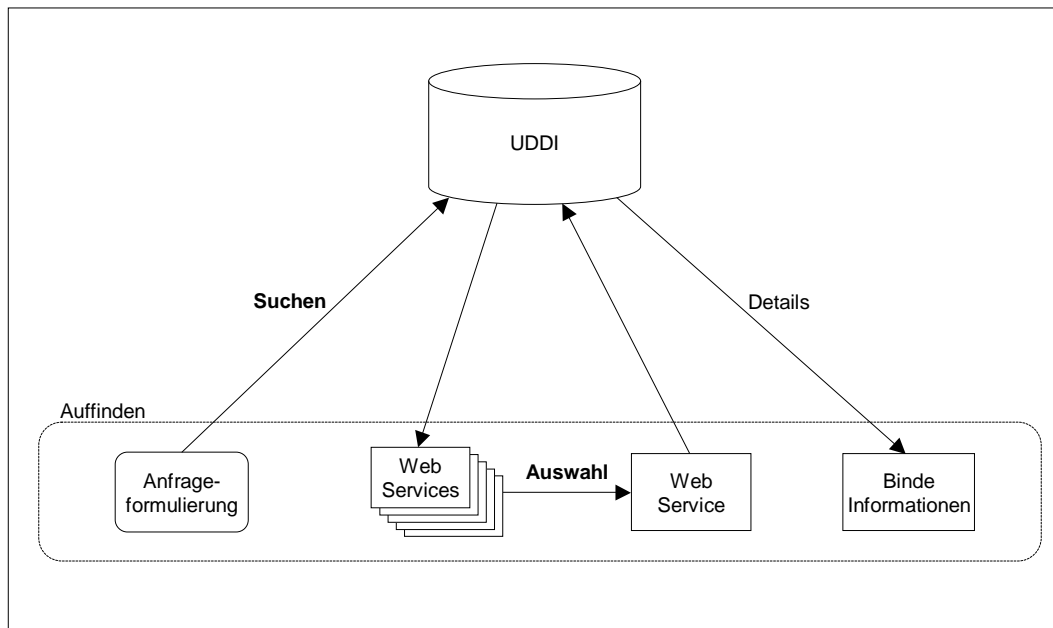


Abbildung 4.2: Auffinden eines Web Service durch Suchen und Auswählen

können. Der erste Typ wird immer dann eingesetzt, wenn Web Services gesucht, der zweite, wenn alle Detailinformationen zu einem ausgewählten Web Service abgerufen werden müssen.

Die Programmierschnittstelle Version 2.0 von UDDI stellt zum Suchen von registrierten Web Services die Abfragemethode `find_service` zur Verfügung, deren Syntax in Listing 4.1 ausführlich dargestellt ist. Dabei sind optionale Elemente und Attribute durch eckige Klammern `[]` gekennzeichnet. Das Ergebnis ist ein `<serviceList>`-Element, das eine Liste von Kurzbeschreibungen zu den qualifizierten Web Services enthält.

```
<find_service businessKey="xyz" [maxRows="n"] ...>
  [<findQualifiers/>]
  [<name/>[<name/>] [<name/>] [<name/>] [<name/>]]
  [<categoryBag/>]
  [<tModelBag/>]
</find_service>
```

Listing 4.1: Suchanfrage für Web Services

Listing 4.1 macht die Anfragemöglichkeit deutlich. Eine mögliche Ausprägung der Suchanfrage ist die Namenssuche, bei der bis zu fünf Namen gleichzeitig angegeben werden können. Dabei können die Namen zur Unterstützung von Suchmustern ein oder mehrere Platzhalter-Zeichen „%“ enthalten. Wird kein Name angegeben werden alle Web Services selektiert. Dabei kann die Anzahl der Rückgabewerte durch das

Setzen des optionalen Attributes `maxRows` begrenzt werden. Wird das Attribut nicht gesetzt, wird eine vom UDDI-Betreiber festgelegte Anzahl von Objekten zurückgegeben. Kommen dennoch mehr Objekte in Frage, wird ein Hinweis in die Ergebnisliste geschrieben, dass noch mehr Objekte qualifiziert wurden. Dennoch existiert keine Möglichkeit, die weiteren Teilergebnisse abzurufen.

Eine weitere Ausprägung der Suchanfrage ist die Selektion nach bestimmten Kategorien. Diese kann mit der Namenssuche kombiniert werden. Kategorien werden in Form eines `<categoryBag>`-Elementes zur Abfrage hinzugefügt, vgl. 3.2. Diese Kategorien werden dabei in der Grundeinstellung per logischem UND verknüpft.

Zur Unterstützung der Suche nach Web Services, die eine bestimmte Schnittstelle implementieren, können mehrere durch logisches UND verknüpfte Schlüsselwerte von TModel-Strukturen in Form eines `<tModelBag>`-Elementes angegeben werden.

Die Angabe bestimmter `findQualifier` ermöglichen z.B. die Sortierreihenfolge der Rückgabeliste oder eine Änderung der logischen Verknüpfungen.

Eine deutliche Schwäche für das Suchen von Web Services besteht darin, dass die `find_service`-Suchanfrage immer nur für ein BusinessEntity-Objekt gültig ist. Eine Suchanfrage nach Web Service ohne Angabe eines bestimmten Unternehmens ist nicht möglich bzw. muss durch ein eigenes Programm nachgebildet werden.

Der Einstieg zur Suche von Web Services beginnt also immer mit einer `find_business`-Anfrage, dargestellt in Listing 4.2.

```
<find_business [maxRows="n"] ...>
  [<findQualifiers/>]
  [<name/> [<name/>] [<name/>] [<name/>] [<name/>]]
  [<discoveryURLs/>]
  [<identifierBag/>]
  [<categoryBag/>]
  [<tModelBag/>]
</find_business>
```

Listing 4.2: Suchanfrage für BusinessEntity-Objekte

Die Ausführungen zur Namenssuche bei der `find_service`-Anfrage gelten analog auch für `find_business`. Zusätzlich kann die Suche verfeinert werden, in dem z.B. Identifizierer oder URLs angegeben werden können. Diese Möglichkeiten werden ausführlich in [MER01] beschrieben.

Die Angabe von TModel-Schlüsselwerten in der `<tModelBag>` bezieht sich auch hier auf Web Services, die diese Schnittstelle implementieren. Laut Spezifikation sollen nur Informationen über BusinessEntity-Objekte ausgegeben werden, die einen Web Service mit der spezifizierten Schnittstelle besitzen, d.h. in den die Kurzinformation sollen nur diese spezifischen Web Services aufgeführt werden.

Weiter kann die `<categoryBag>` bei der `find_service`-Anfrage auf zwei Arten benutzt werden. Einerseits können Kategorien für die Selektion von BusinessEntity-Objekten benutzt werden, andererseits kann durch das Setzen eines bestimmten

Eintrages im `<findQualifiers>`-Element die Anwendung des Selektionskriteriums auf die im BusinessEntity-Objekt enthaltenen BusinessService-Objekte umgeleitet werden.

Die Suche nach Schnittstellen ist dagegen grundsätzlich unabhängig von Business-Objekten. Listing 4.3 zeigt die Details der Suchanfrage.

```
<find_tModel [maxRows="n"] ...>
  [<findQualifiers/>]
  [<name/>]
  [<identifierBag/>]
  [<categoryBag/>]
</find_business>
```

Listing 4.3: Suchanfrage für Web-Service-Schnittstellen

Im Gegensatz zu den beiden anderen Anfragen kann für die Anfrage nach einem TModel nur ein einzelner Name angegeben werden. Jedoch ist auch hier die Nutzung des Platzhalters „%“ möglich. Die übrigen Parameter verhalten sich wie oben beschrieben. Für den Fall, dass nur Schnittstellen gesucht werden, die in WSDL formuliert sind, kann die in UDDI zur Verfügung gestellte Kategorie „WSDL Document“ benutzt werden.

Für alle Suchanfragen gilt, dass auch dann ein Listen-Objekt zurückgegeben wird, wenn kein entsprechendes Daten-Objekt gefunden werden kann. In diesem Fall ist das Suchergebnis die leere Liste.

Für den Prozess-Modellierer, der nach einem Web Service zur Lösung einer Aufgabe sucht, haben zwei Ansätze zur Formulierung der Suche große Bedeutung:

- Suche nach Web-Service-Schnittstellen mit anschließender Suche nach Web Services,
- direkte Suche nach Web-Services.

Im ersten Fall wird entweder über den Web-Browser oder durch Formulierung der `find_tModel`-Anfrage die UDDI-Repräsentation der Web-Service-Schnittstelle gesucht und ausgewählt. Aus dem Resultat kann dann der Schlüssel (`tModelKey`) ermittelt und in der `find_business`-Anfrage als Eintrag in der `<tModelBag>` benutzt werden. Die Ausgabe der zweiten Anfrage ist dann eine Liste mit verkürzten BusinessEntity-Informationen, bei der nur solche Unternehmen und Web Services betrachtet werden, die diese Schnittstelle implementieren. Die Suche kann außerdem durch die Angabe weiterer Parameter eingeschränkt werden. Diese Anfrage ist in Abbildung 4.3 auf der nächsten Seite dargestellt.

Diese Vorgehensweise ist besonders dann interessant, wenn UDDI für die Publikation von standardisierten Schnittstellen benutzt wird, z.B. für Standard-Schnittstellen für

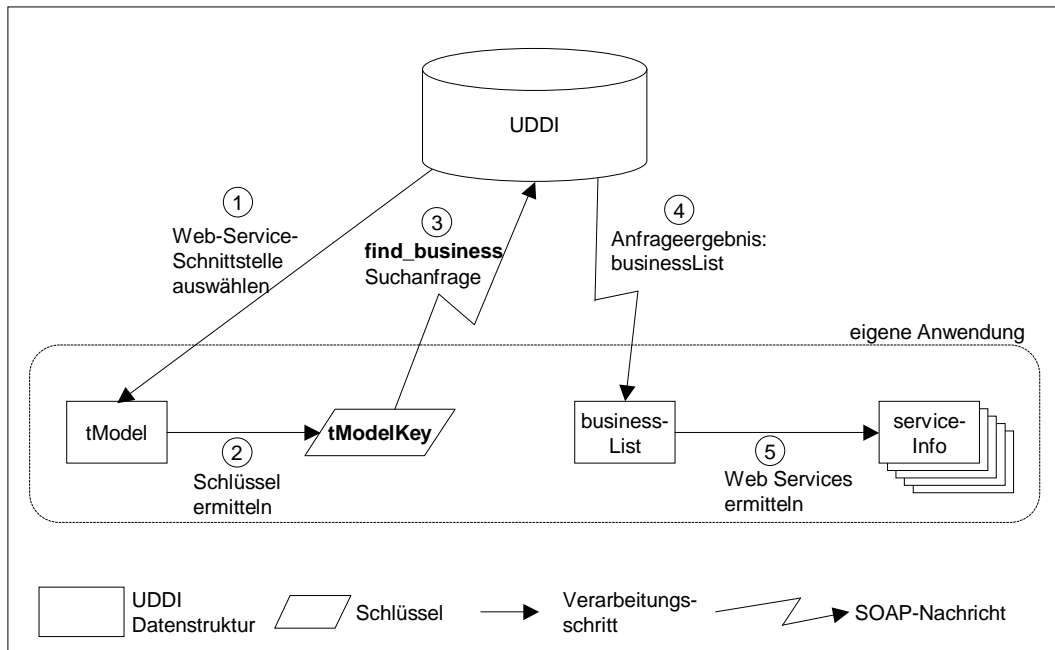


Abbildung 4.3: Suchen von Web Services mit bestimmter Schnittstelle

Bestellungen, Aktienorders, Auskünfte etc. Ein weiterer Aspekt für den ersten Ansatz ist die Empfehlung, in WSDL beschriebene Web Services getrennt nach Schnittstelle und Implementierung zu veröffentlichen, vgl. [CER01]

Im zweiten Fall werden Web Services direkt über eine `find_business`-Anfrage gesucht. Als Auswahlkriterium für die `BusinessEntity`-Objekte können, wie in Listing 4.2 auf Seite 39 dargestellt, Namen mit Platzhaltern, URLs, Identifizierer oder Kategorien angegeben werden. Als Resultat dieser Anfrage liegt dann eine Liste vor die alle qualifizierten `BusinessEntity`-Objekte mit allen zugehörigen Web Services enthält. Aus diesen Web Services müssen in einem zweiten Schritt diejenigen herausgefiltert werden, die für die gewünschte Lösung in Frage kommen (vgl. Abbildung 4.4 auf der nächsten Seite). Das Filtern kann z.B. über eine `find_service`-Anfrage realisiert werden, für die neben den `BusinessKeys` aus der Ergebnisliste eine eigene `<categoryBag>` angegeben werden kann. Eine solche `find_service`-Anfrage muss für jedes `BusinessEntity`-Objekt in der ersten Ergebnisliste formuliert werden und die Ergebnisse dieser Anfragen müssen zu einem Gesamtergebnis vereinigt werden. Diese Vorgehensweise ist in Abbildung 4.5 auf Seite 43 dargestellt.

Eine weitere Variante der direkten Suche ist, die `<categoryBag>` als Auswahlkriterium für `BusinessEntity`-Objekte zu benutzen und nicht für `BusinessServices`. Diese Möglichkeit wurde in Version 2.0 von UDDI eröffnet. Durch diese Art der Anfrageformulierung werden nur diejenigen Unternehmen und nur die Web Services betrachtet, die den Filterkriterien bzw. der Service-Kategorie entsprechen, vgl. Abbildung 4.6 auf Seite 44.

Auswahl

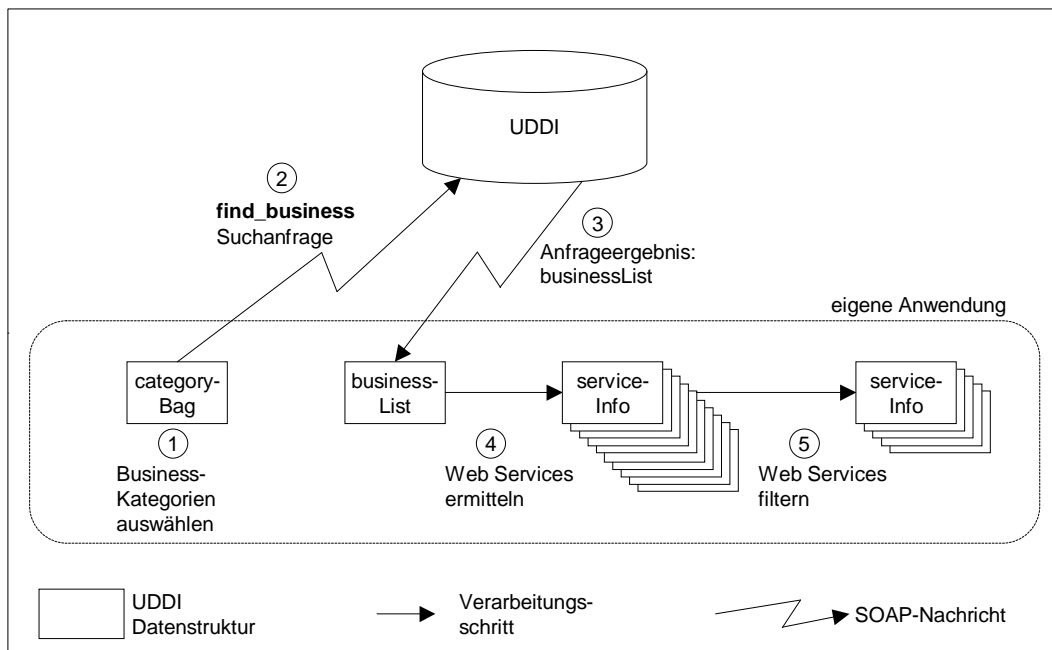


Abbildung 4.4: Suchen von Web Services durch Auswahl von Unternehmen

Das Ziel der Auswahl ist, in einer Menge von Web Services, die alle die gewünschte Funktion implementieren, denjenigen Web Service zu ermitteln, der später aus dem Workflow-Prozess heraus aufgerufen werden soll, vgl. Abbildung 4.2 auf Seite 38. Diese Menge liegt als Zwischenergebnis eines zurückliegenden Suchschrittes in Form eines in der UDDI Spezifikation dargestellten `<serviceInfos>`-Elementes vor.

Der ausgewählte Web Service dient dann als Einstiegspunkt zu weiteren Informationen, die für die weitere Verarbeitung notwendig sind, z.B. die WSDL-Beschreibung. Diese Informationen werden in einem von mehreren möglichen BindingTemplate-Objekten gehalten, die allerdings noch nicht im Zwischenresultat der Suchanfrage enthalten sind. Zur Ermittlung der BindingTemplate-Objekte eines BusinessService-Objektes ist noch mindestens ein weiterer UDDI-Zugriff erforderlich:

```
<find_binding serviceKey="..." [maxRows="n"] ...>
  [<findQualifiers/>]
  [<tModelBag/>]
</find_binding>
```

Listing 4.4: Anfrage zur Ermittlung von Binding-Informationen zu einem Web Service

Das Ergebnis dieser Anfrage ist eine Liste mit BindingTemplate-Objekten, aus der wiederum ein BindingTemplate ausgewählt werden muss, um den Web Service zu binden. Der Auswahlvorgang ist in Abbildung 4.7 auf Seite 45 dargestellt.

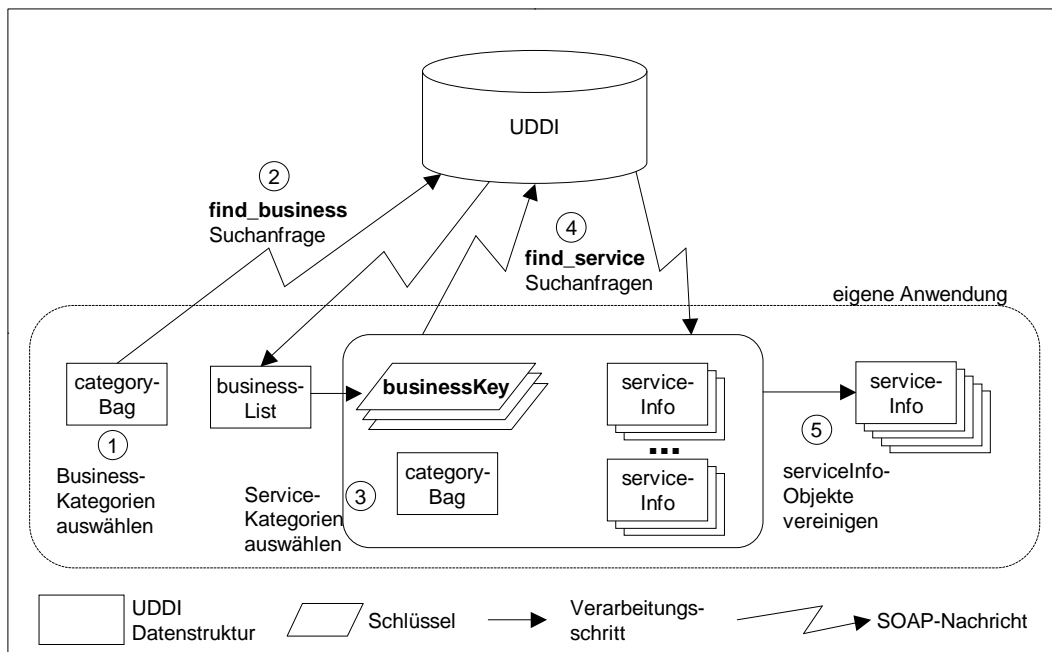


Abbildung 4.5: Suchen und Filtern von Web Services durch Zugriff auf UDDI

Als Auswahlkriterien für Web Services kommen sowohl wirtschaftliche als auch technische Eigenschaften in Frage:

- Preis, der dem Anbieter eines Web Services für die Inanspruchnahme bezahlt werden muss,
- „Quality of Services“, also Kriterien wie Verfügbarkeit, Sicherheit, Zuverlässigkeit und Antwortzeit beim Aufruf des Web Service,
- Präferenzregelung für gleichwertige Web Services,
- Negativliste zum Ausschluss bestimmter Web Services,
- bestimmte Transportprotokolle,
- Repräsentation in WSDL zur automatisierten Weiterverarbeitung.

Leider wird in UDDI nur zwei dieser Kriterien direkt unterstützt: das Transportprotokoll und die WSDL-Repräsentation. Wenn eine Web-Service-Beschreibung, wie in Abschnitt 1.6 dargestellt, getrennt nach Schnittstelle (TModel) und Implementierung (BindingTemplate) gespeichert wird, kann man beim Suchen der Schnittstelle die Kategorie „WSDL Document“ als Selektionskriterium angeben.

Das Transportprotokoll, kann über das Attribut `URLType` des `<accessPoint>`-Elementes abgefragt werden, nicht jedoch die Nachrichtenformate. So kann zwar als Attributwert HTTP als Transportprotokoll angegeben werden, nicht jedoch SOAP als Nachrichten- und Transportprotokoll. Höhere Protokolle und Nachrichtenformate

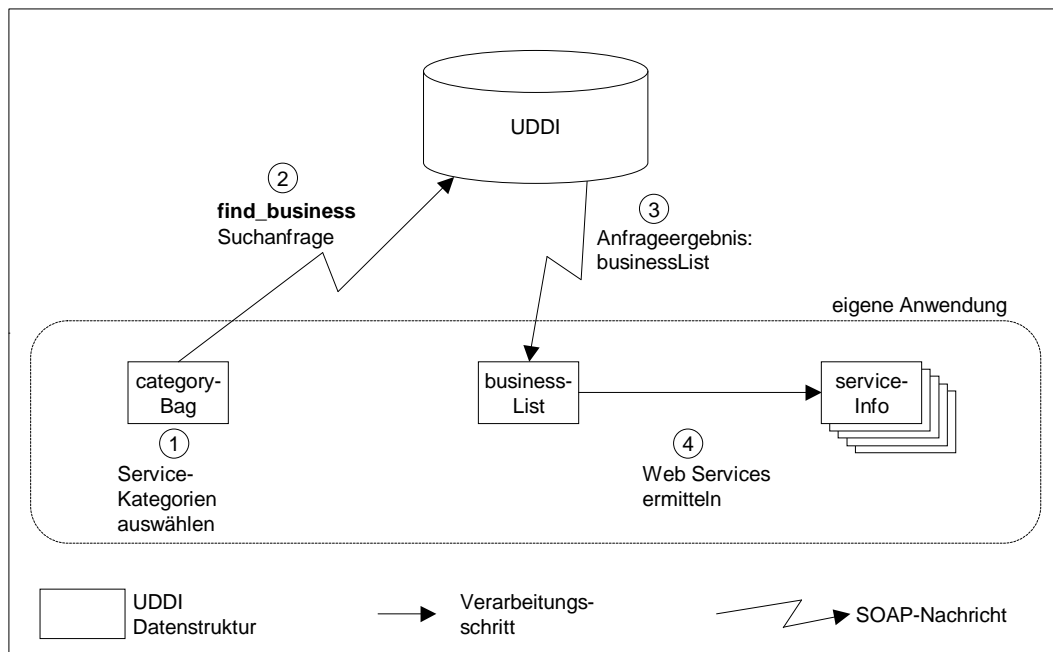


Abbildung 4.6: Suche nach Web Services mit Service-Kategorie

können indirekt aus der Web-Service-Beschreibung abgeleitet werden. Liegt diese im WSDL-Format vor, können daraus diejenigen Web-Service-Implementierungen benutzt werden, die das gewünschte Nachrichten- bzw. Transportprotokoll unterstützen.

Eine Darstellung von Preisen für das Ausführen von Web Services in UDDI ist in den Spezifikationen nicht vorgesehen. Preisdarstellungen sind z.B. denkbar für solche Web Services, die als Resultat Informationen liefern, z.B. aktuelle Aktienkurse. Dagegen spielt der Preis für Web Services mit denen Waren bestellt werden können, eine untergeordnete Rolle, da hier der Preis der Waren und nicht der Dienstleistung „Bestellung über das Internet“ entscheidend ist. In UDDI könnten Preise in Form einer Taxonomie realisiert werden oder innerhalb eines `<description>`-Elementes bei einem BusinessService- oder BindingTemplate-Objekt. Problematisch dabei ist, dass die Anfragen an UDDI auf reinen Textvergleichen basiert und die Vergleichsoperatoren \leq , \leq , \geq , \geq nicht verwendet werden können.

Auch die „Quality of Services“ bleiben in den UDDI-Datenstrukturen unberücksichtigt, so dass eine direkte Abfrage nicht möglich ist. Dieses durchaus sehr wichtige Kriterium für die Web-Service-Nutzung könnte ebenso wie die Preisinformationen in Form einer Taxonomie oder zusätzlichen Elementen inner halb einer `<description>` in UDDI integriert werden. Die Anwendung von Vergleichsoperatoren, z.B. um eine Verfügbarkeit von $\geq 95\%$ abzufragen, ist auch hier aus den oben dargestellten Gründen nicht möglich.

Da auch der Negationsoperator von UDDI für Abfragen nicht zur Verfügung gestellt wird, können auch keine Anfragen formuliert werden, bei denen bestimmte Web Services ausgeschlossen werden sollen. Ein solcher Ausschluss ist für viele Anwen-

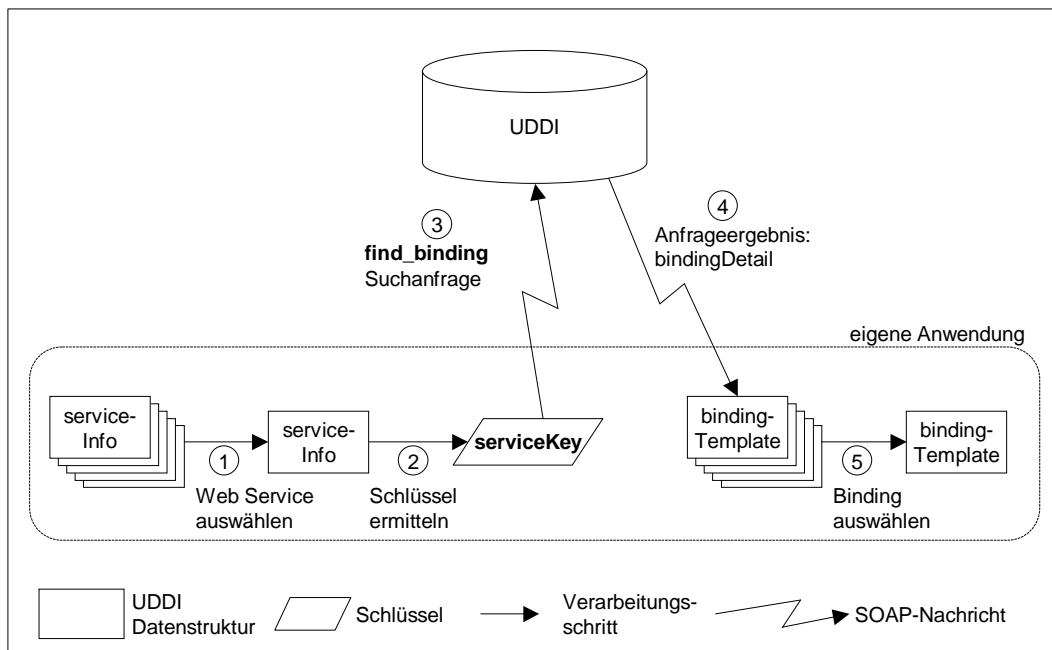


Abbildung 4.7: Auswählen eines BindingTemplate-Objektes

dingssituationen durchaus wünschenswert. Werden dennoch bestimmte Ausschlüsse gewünscht, muss ein entsprechendes Verfahren auf Seiten der Anwendung etabliert werden. Dies könnte z.B. über den Vergleich von UDDI-Resultaten und vorab ermittelten Schlüsselwerten realisiert werden.

Sollen dagegen bestimmte Präferenzen ausgedrückt werden, z.B. wenn Unternehmen A den gewünschten Web Service anbietet, ist dieser zu benutzen, wenn nicht, dann den von Unternehmen B usw., kann man bei der Namenssuche die gewünschte Reihenfolge vorgeben. Die Resultate werden dann in der gleichen Ordnung zurückgegeben. Die Namenssuche ist jedoch durch die Spezifikation auf fünf Einträge begrenzt. Sollen komplexere oder umfangreichere Präferenzen ausgedrückt werden, sind auch diese auf Seiten der Anwendung zu realisieren.

Die Ausführungen zeigen, dass das Problem der Auswahl eines bestimmten Web Service mit vorhandenen Mitteln, nur schwer zu automatisieren ist, weil einige wichtige Auswahlkriterien von UDDI nicht direkt unterstützt werden. Neue Lösungsansätze können entweder in UDDI integriert oder müssen individuell realisiert werden. Im ersten Fall verschiebt sich die Auswahl in Richtung UDDI-Suche und hat zur Folge, dass der Umfang der qualifizierten Web Services geringer wird. Beim zweiten Lösungsansatz verschiebt sich die Auswahl in Richtung proprietärer Verfahren mit der Folge, dass nur noch minimale UDDI-Informationen, z.B. Schlüssel oder die Adresse der WSDL-Beschreibung, für die Auswahl benötigt werden. Solche kompakten Informationen sind z.B. im BindingTemplate-Objekt enthalten.

Im konkreten Fall der Web-Service-Nutzung für MQWF kann folgende Strategie für das Auffinden Web Services in UDDI vorgeschlagen werden:

- Auswahl (manuell oder automatisch) einer Schnittstelle durch Spezifikation einer `find_tModel`-Anfrage mit bestimmten Kategorien aus Standard-Taxonomien und der Kategorie „WSDL Document“.
- Formulierung einer `find_business`-Anfrage, bei der als Suchkriterium der Schlüssel der Schnittstelle übergeben wird. Diese Suche kann eingeschränkt werden durch die Angabe von Namen und durch die Angabe von Unternehmenskategorien aus Standard-Taxonomien.
- Auswahl des geeigneten Web Service aufgrund von Informationen in den BindingTemplate-Strukturen. Sollen Präferenzen oder Ausschlüsse ausgedrückt werden sind diese auf Basis eines Schlüsselvergleiches in einer eigenen Anwendung zu realisieren.

Durch diese Strategie wird erreicht, dass nur Web Services mit der gleichen Schnittstelle gefunden werden und dass für diese Web Services eine Beschreibung in WSDL vorliegt. Damit ist einerseits der Umfang der Resultate minimiert, andererseits sind die beiden wichtigsten Kriterien für eine weitgehende Automatisierung erfüllt:

- Eine klar definierte Schnittstelle ist Voraussetzung für die Modellierung des Workflow-Prozesses, auch wenn der Aufruf bei verschiedenen Prozessen an unterschiedlichen Netzwerkadressen erfolgt.
- Die WSDL-Beschreibung ermöglicht die Nutzung der Software-Werkzeuge im WSPMTK.

Im Hinblick auf die Automatisierung des Auffindvorganges sind die Anfragemöglichkeiten von UDDI problematisch und weitaus weniger mächtig und komfortabel wie die Möglichkeiten, die z.B. SQL bietet. Die dargestellten Nachteile fallen weniger ins Gewicht, wenn folgende Punkte bei der Nutzung von UDDI berücksichtigt werden:

- Registrierung eines Web Service getrennt nach Schnittstelle und Implementierung.
- Nutzung einer begrenzten Anzahl an standardisierten, aussagekräftigen Taxonomien für BusinessService- und TModel-Strukturen.

4.3 Binden von Web Services

In diesem Abschnitt wird beleuchtet, welche Voraussetzungen erfüllt werden müssen und welche Aktionen notwendig sind, um Web Services in Workflow-Prozesse einbinden zu können. Dazu wird das Prozessmodell, der Aufrufmechanismus und die notwendigen Schritte betrachtet.

Voraussetzungen

Für jede Aktivität im Prozessmodell, die durch ein Computerprogramm implementiert wird, benötigt das WFMS eine Definition dieses Programms. In dieser Definition werden unter anderem Ein- und Ausgabedatenstrukturen, Programmparameter, Programmdatei, Betriebssystem und ausführender Rechner festgelegt, also alle Informationen, um das Programm zu lokalisieren, zu starten und die richtigen Daten zu verarbeiten. Soll ein Web Service als Aktivitätsimplementierung benutzt werden, müssen diese Informationen aus zwei Quellen zusammengetragen werden: aus der Web-Service-Beschreibung und aus der Beschreibung des Systems, aus dem heraus der Web Service aufgerufen wird. Aus diesen Quellen können dann die notwendigen Definitionen abgeleitet werden.

Da der Nutzer eines Web Service üblicherweise keinen Einfluss auf die Schnittstelle des Web Service hat, muss der Prozess an den Web Service angepasst werden. Dazu gehört, dass Datenstrukturen definiert werden, die Ein- und Ausgaben des Web Service aufnehmen können. Voraussetzung dafür ist jedoch, dass für alle auszutauschenden Daten die entsprechenden Datentypen auf der Seite von MQWF zur Verfügung gestellt werden können. Die Nutzung ist demnach auf solche Web Services beschränkt, für die eine Typanpassung ins MQWF-Typsystem vorgenommen werden kann.

Die Workflow-Datentypen (**STRUCTURE**) werden aus den Elementen `<message>` und `<types>` der WSDL-Beschreibung abgeleitet, so dass für alle Nachrichten, die zwischen MQWF und dem Web Service ausgetauscht werden, ein entsprechender MQWF-Datentyp zur Verfügung steht. Dabei müssen die Einschränkungen beachtet werden, die bei der Verwendung des Web Service UPES vorgegeben werden, vgl. 2.2 und Dokumentation zum WSPMTK. Diese Einschränkungen betreffen unter anderem die Namensvergabe: so müssen für die MQWF-Datenstrukturen die gleichen Namen benutzt werden wie für die korrespondierenden Nachrichten-Elemente des Web Service.

Weiter muss für jeden gewünschten Web-Service-Aufruf, in WSDL repräsentiert durch einen `<port>` in einem `<service>`-Element, sowohl ein **PROGRAM** als auch eine **PROGRAM_ACTIVITY** modelliert werden. Die Datenstrukturen für die Ein- und Ausgabe müssen dabei mit dem Nachrichtenformaten für den Web Service korrespondieren. Bei der Programm-Definition muss neben spezifischen Parametern auch die Komponente angegeben werden, die für die eigentliche Ausführung benutzt wird. Wie in 2.2 dargestellt, werden die Web Services über einen „User Defined Program Execution Server“ (UPES) aufgerufen. Auch für den UPES muss eine Definition erstellt werden. Die Parameter für die UPES-Definition können allerdings nicht aus der WSDL-Definition abgeleitet werden.

Schließlich müssen die neuen Prozess-Elemente in das vorhandene Prozessmodell eingebunden werden. Insbesondere muss definiert werden, wie die neuen Datenstrukturen bei Prozessausführung mit Daten gefüllt werden und wie das Resultat des Web Service in die vorhandenen Datenstrukturen zurückgeschrieben werden. Dazu muss für jeden Aufruf eine **DATA MAP** definiert werden, die die genaue Übergabe zwischen Datenfeldern innerhalb eines Workflow-Prozesses beschreibt. Die genaue Modellierung von Prozessen und Datenflüssen in MQWF wird in [IBM01b] beschrieben.

Auswahl der UPES-Konfiguration

Wie in Abschnitt 2.2 dargestellt wurde, kann der Web-Service-UPES auf zwei Arten aufgerufen werden: mit einer Java-Proxy-Klasse oder mit einem WSDL-Dokument. Welcher Aufruf benutzt wird hängt von folgenden Kriterien ab:

- Verfügbarkeit eines Software-Werkzeuges zur Generierung der Proxy-Klasse für den Web-Service-Aufruf,
- Datenstrukturen des Web Service,
- Ausführungszeit,
- administrativer Aufwand.

Ohne ein Software-Werkzeug, mit dem aus einer WSDL-Beschreibung die notwendigen Java-Klassen zum Aufruf eines Web Service generiert werden kann, müssen diese für jeden einzubindenden Web Service „von Hand“ erstellt werden. Dies hat wiederum zur Folge, dass das Einbinden von Web Services nicht automatisierbar ist. Im WSPMTK ist leider kein solcher Proxy-Generator enthalten. Dennoch sind derartige Generatoren, mit denen Java-Proxy-Klassen erzeugt werden können, erhältlich, z.B. im Web Services Toolkit (WSTK) von IBM.

In der aktuellen Version des WSPMTK besitzt der Web Service UPES noch einige Einschränkungen bezüglich der Datenstrukturen. So können bei der Nutzung von Proxy-Klassen keine Arrays als Parameter übergeben werden. Ansonsten können komplexe Typen sowohl als Parameter als auch als Rückgabewerte benutzt werden. Bei der Nutzung des WSDL-Aufrufes werden nur flache Datenstrukturen, also keine Arrays und keine komplexen Typen unterstützt. Folgende XML-Schema-Datentypen sind bei der zweiten Aufrufvariante zugelassen: string, boolean, float, double, long, int, short, byte, siehe auch Dokumentation zum WSPMTK.

Ein wichtiges Kriterium ist die Ausführungszeit beider Alternativen. Im Rahmen dieser Arbeit konnte dieses Kriterium nicht gemessen werden, jedoch erscheint plausibel, dass die Aufrufvariante über Java-Proxy-Klassen eine kürzere Laufzeit aufweisen wird, weil dabei die Analyse eines XML-Dokumentes entfällt, insbesondere wenn dieses über eine entfernte URL geladen werden muss.

In Fällen, in denen die drei aufgeführten Kriterien nicht greifen, kann der administrative Aufwand ausschlaggebend für die Wahl der UPES-Konfiguration sein. Bei der WSDL Variante entfällt der Aufwand für das Generieren und Installieren der Proxy-Klassen.

Schritte

Nachdem ein Web Service gefunden und eine Aufrufalternative für den UPES gewählt worden ist, kann der Web Service gebunden, d.h. für den Aufruf vorbereitet und aufgerufen werden. Ausgehend von einer UDDI-BindingTemplate-Struktur sind folgende Schritte notwendig:

Für die Aufrufalternative WSDL:

- Ermittlung der URL für das WSDL-Dokument, das den Web Service beschreibt aus dem BindingTemplate-Objekt,
- Erstellen der Workflow- Definitionen aus dem WSDL-Dokument,
- Modellierung der Datenflüsse zwischen den neuen und vorhandenen Prozess-Konstrukten,
- Importieren der geänderten Prozessdefinition in das WFMS

Für die Aufrufalternative Java-Proxy-Klasse:

- Ermittlung der URL für das WSDL-Dokument, das den Web Service beschreibt aus dem BindingTemplate-Objekt,
- Erstellen der Klassen für den Java-Proxy aus der WSDL-Beschreibung,
- Installieren der Java-Proxy-Klassen in die dafür vorgesehene Umgebung,
- Erstellen der Workflow- Definitionen aus dem WSDL-Dokument,
- Modellierung der Datenflüsse zwischen den neuen und vorhandenen Prozess-Konstrukten,
- Importieren der geänderten Prozessdefinition in das WFMS.

Beide Alternativen sind in Abbildung 4.8 auf der nächsten Seite dargestellt.

Nach der Diskussion der Vorgänge zum Auffinden und Binden von Web Services in MQWF werden nun die zwei Ansätze zum statischen und dynamischen Einbinden untersucht. Dabei werden jeweils die Vorteile und Probleme dargestellt und Realisierungsalternativen aufgezeigt.

4.4 Statisches Einbinden

Wie im vorhergehenden Abschnitt beschrieben finden das Auffinden und das Binden eines Web Service beim statischen Ansatz zum Modellierungs-Zeitpunkt eines Workflow-Prozesses statt. Wenn das Prozessmodell in die Runtime-Umgebung von MQWF importiert wird, sind alle erforderlichen Schritte zum Einbinden beendet und der Web-Service-Aufruf ein fester Bestandteil der Prozessdefinition.

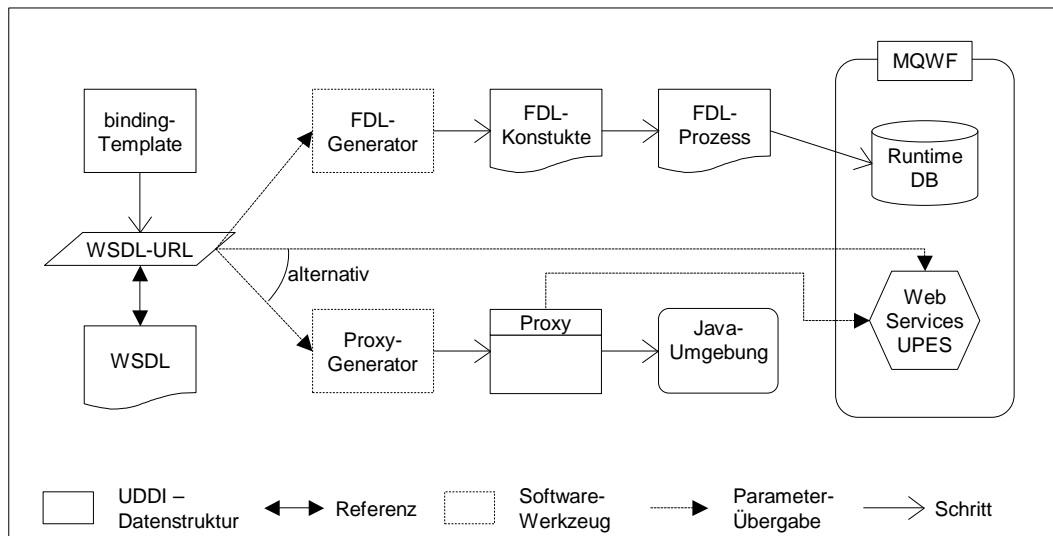


Abbildung 4.8: Schritte zum Binden eines Web Service an MQWF

Vorteile

Der statische Ansatz bietet Vorteile in unterschiedlichen Bereichen, die sowohl die Modellierungs- als auch die Ausführungsphase betreffen:

- Das Auffinden und Binden eines Web Service muss nicht innerhalb eines Workflow-Prozesses initiiert und die entsprechenden Schritte müssen somit nicht ins Prozessmodell aufgenommen werden.
- Dadurch dass die notwendigen Schritte zum Einbinden schon vor der Prozessausführung durchlaufen werden, kann beim Aufruf des Web Service mit einer insgesamt kürzeren Ausführungszeit gerechnet werden als wenn die Schritte erst sequentiell durchlaufen werden müssen. Dieser Laufzeitvorteil kann jedoch in manchen Fällen durch geeignete Prozessmodellierung minimiert werden.
- Der Vorgang zum Einbinden ist beim statischen Ansatz weniger zeitkritisch, da alle erforderlichen Schritte bereits während der Modellierungsphase durchlaufen werden.
- Mögliche Fehlersituationen bei den Einzelschritten, z.B. wenn beim Suchen kein geeigneter Web Service gefunden werden kann, können beim statischen Ansatz „offline“ behandelt werden. Im anderen Fall muss die Fehlerbehandlung im Workflow-Prozess modelliert werden.
- Der statische Ansatz ermöglicht menschliche Interaktion bei allen Einzelschritten. So können z.B. Korrekturen bei den generierten FDL-Konstrukten vorgenommen werden.
- Auch wenn keine Software-Werkzeuge zum Auffinden von Web Services oder zur Generierung von FDL, etc. vorhanden sind, können dennoch Web Services in MQWF eingebunden werden.

Probleme

Die größten Probleme beim statischen Ansatz resultieren aus Änderungen am Web Service. Solche Änderungen können alle Bereiche des Web Service betreffen: Schnittstelle, Nachrichten- und Transportprotokolle oder Netzwerkadressen. In jedem Fall muss nach einer solchen Änderung auch das Prozessmodell geändert werden, mit einer Ausnahme: bei einer geänderten Netzwerkadresse wird nur beim Aufruf über Proxy-Klassen eine Neu-Generierung dieser Klassen erforderlich.

Zur Entdeckung von geänderten Web Services sind zwei Ansätze denkbar: entweder werden die eingebundenen Web Services außerhalb der Workflow-Umgebung regelmäßig überprüft, z.B. in einer besonderen Administrationsumgebung, oder geänderte Web Services werden aufgrund einer Fehlersituation bei der Prozessausführung entdeckt. Im zweiten Fall entstehen Beeinträchtigungen im Produktionsablauf, vor allem wenn der Web Service häufig benutzt wird.

Realisierungsalternativen

Bevor mit der Diskussion von Alternativen zur Realisierung des gesamten Einbindvorganges begonnen wird, soll zunächst ein Überblick darüber gegeben werden, welche Teile des Prozesses automatisierbar sind und welche Software-Werkzeuge dafür zur Verfügung stehen.

Zu den nicht automatisierbaren Schritten gehört sicherlich die Modellierung des Workflow-Prozesses, in den der Web Service eingebunden werden soll und die Anbindung der `PROGRAM_ACTIVITY` für den Aufruf in den Kontroll- und Datenfluss des Prozesses. Letztere scheitert an der Zuordnung von Datenfeldern in den Prozess-Datenstrukturen zu den Web-Service-Datenstrukturen.

Nicht automatisierbar ist auch die Auswahl bzw. die Festlegung der Kategorien, die bei der Suche nach geeigneten Web Services in UDDI benutzt werden sollen.

Die übrigen Schritte sind grundsätzlich automatisierbar, wenn auch im Rahmen der aktuellen Web-Service-Erweiterungen zu MQWF dafür keine Software-Werkzeuge enthalten sind. Folglich müssen für die Schritte

- Web-Service-Suche,
- Web-Service-Auswahl,
- Generierung von Proxy-Klassen,
- Generierung von FDL-Konstrukten.

Software-Werkzeuge gesucht oder neu entwickelt werden.

Für die Generierung der FDL-Konstrukte existiert, wie in Abschnitt 2.2 ein Software-Werkzeug: `wsd12fdl`. Daneben kann zur Generierung der Java-Proxy-Klassen aus der WSDL-Beschreibung das Software-Werkzeug `proxygen` eingesetzt werden, das von IBM im Rahmen des Web Services Toolkit (WSTK) angeboten wird. Ein neuer Proxy-Generator kann außerdem durch Modifikation des Software-Werkzeuges

`fdl2wsdl` entwickelt werden. Komplet neu entwickelt werden müssen dagegen die Werkzeuge zum Suchen und Auswählen von Web Services.

Die Werkzeuge `wsdl2fdl` und `proxygen` sind beide in Java programmiert und werden mit Hilfe einer Stapelverarbeitungsdatei gestartet. Für die neu zu entwickelnden Werkzeuge stehen die in Abschnitt 3.3 dargestellten Schnittstellen für die Interaktion mit UDDI in Form von Java-Bibliotheken zur Verfügung. Also könnten die Werkzeuge zum Auffinden ebenfalls in Java realisiert werden. Ob die Bibliothek des WSTK, UDDI4J oder SOAP für den Zugriff auf UDDI benutzt wird, kann beantwortet werden, wenn man die Formen des Zugriffs und der Resultate betrachtet. Für die Anfragen müssen nicht wie beim Veröffentlichen Dokumente transformiert werden, sondern sie können direkt formuliert werden. Die Resultate bestehen aus UDDI-Objekten. Da das WSTK eine abstrakte Sicht auf Web-Service-Verzeichnisse bietet und diese intern in UDDI-Zugriffe umwandelt, UDDI4J jedoch direkten Zugriff auf UDDI bietet, ist letztere vorzuziehen. Beim Zugriff über SOAP müssen alle Aufrufe durch eigene Algorithmen erzeugt und alle Resultate analysiert werden. Aus Gründen der Wiederverwendung und unter Fehlergesichtspunkten scheidet der direkte SOAP-Zugriff aus. Der Zugriff ist in Abbildung 4.9 dargestellt.

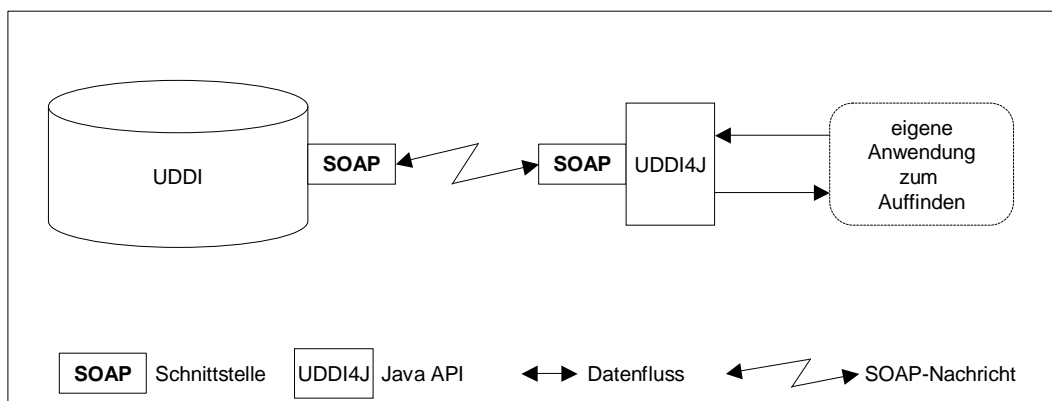


Abbildung 4.9: Auffinden über UDDI4J-API

Als Realisierungsalternativen für die Automatisierung des Gesamtvorganges werden eine die Lösung mit Hilfe einer Skriptsprache und eine Lösung als Java-Applikation betrachtet.

Bei ersten Lösung wird eine Skriptsprache verwendet, um die Einzelschritte zu steuern. Dabei werden die zu benutzenden Software-Werkzeuge unverändert aufgerufen. Wird als Skriptsprache ANT der APACHE-Group verwendet, können die Java-Klassen der Werkzeuge auch direkt gestartet werden, ohne den Umweg über die Stapelverarbeitungsdateien zu gehen. Dabei können die neu zu entwickelnden Anwendungen zum Auffinden so geschrieben werden, dass sie Rückgabewerte, z.B. die URL auf ein WSDL-Dokument, direkt an die Skript-Variablen zurückgeben. Dieser Ansatz wird in Abbildung 4.10 auf der nächsten Seite dargestellt.

Problematisch bei der Skriptlösung ist der Zugriff auf die Resultate, die von solchen Software-Werkzeugen erzeugt werden, die nicht im Hinblick auf die Verwendung mit

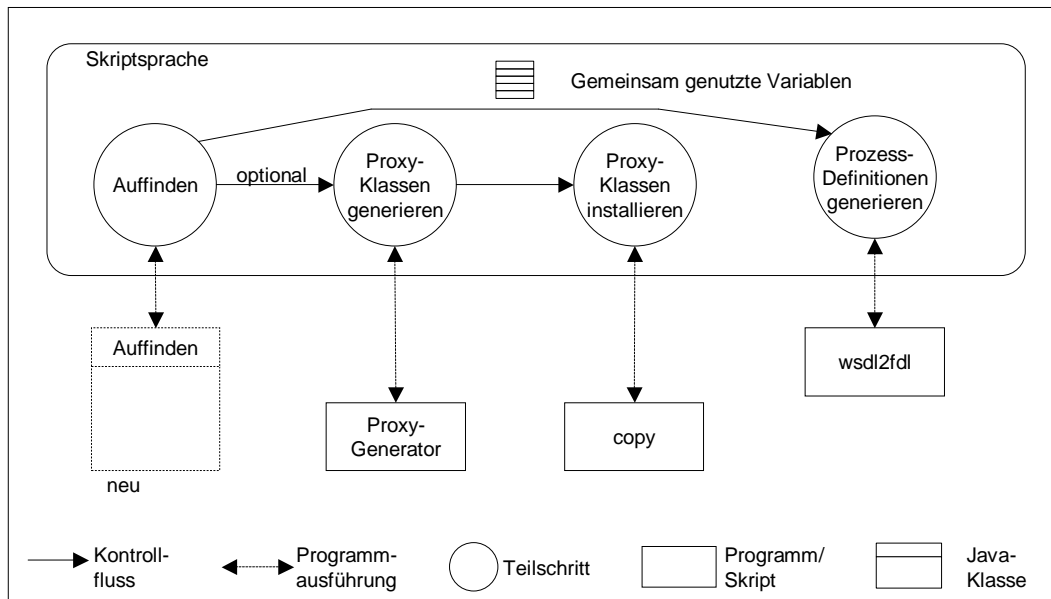


Abbildung 4.10: Statischen Einbinden, gesteuert über Skript

einer Skriptsprache entwickelt wurden, z.B. **proxygen**. So ist es nur durch Analyse des WSDL-Dokumentes möglich, zu ermitteln, wie die Klassennamen der Proxy-Klassen lauten. Zwar bieten viele Skriptsprachen die Möglichkeit für bestimmte Befehle, z.B. **copy** Platzhalter zu verwenden, dennoch besteht dabei immer das Risiko von unerwünschten Nebeneffekten. Wenn die Skriptlösung zum Einsatz kommt, sollten neu zu entwickelnde Werkzeuge so gestaltet werden, dass ein Zugriff auf Resultate aus der Skriptumgebung möglich ist.

Wie Abbildung 4.10 zu sehen ist, endet der automatisierte Vorgang, nachdem evtl. Proxy-Klassen generiert und die notwendigen Prozessdefinitionen erzeugt wurden. Der nächste Schritt im Gesamtprozess ist das nicht automatisierbare Einbinden der generierten FDL-Konstrukte in den Workflow-Prozess. Zu diesem Zweck wird ein Werkzeug zur Prozessmodellierung benutzt, z.B. MQWF Buildtime. Für das anschließende Importieren des neuen Prozessmodells in die Laufzeitumgebung von MQWF steht ein Kommandozeilen-Befehl zur Verfügung. Dieser kann entweder manuell durch den Prozess-Modellierer oder über ein Skript gestartet werden.

Die zweite Realisierungsalternative besteht darin, den Gesamtprozess als Java-Anwendung zu programmieren. Voraussetzung hierfür ist, dass die neu zu entwickelnden Anwendungen zum Auffinden auch in Form von Java-Klassen realisiert werden. Bei diesem Ansatz werden die vorhandenen Werkzeuge nicht über die Stapelverarbeitungsdateien gestartet, sondern direkt über die entsprechenden Methoden, vgl. Abbildung 4.11.

Auch bei diesem Ansatz ist der Zugriff auf Zwischenergebnisse der eingebundenen Software-Werkzeuge problematisch. Auch hier müssen Teilergebnisse analysiert werden, um die Parameter für weitere Aufrufe zu ermitteln. Dagegen ist die Benutzung von Platzhaltern bei der Java-Alternative nur dann möglich, wenn ein entsprechen-

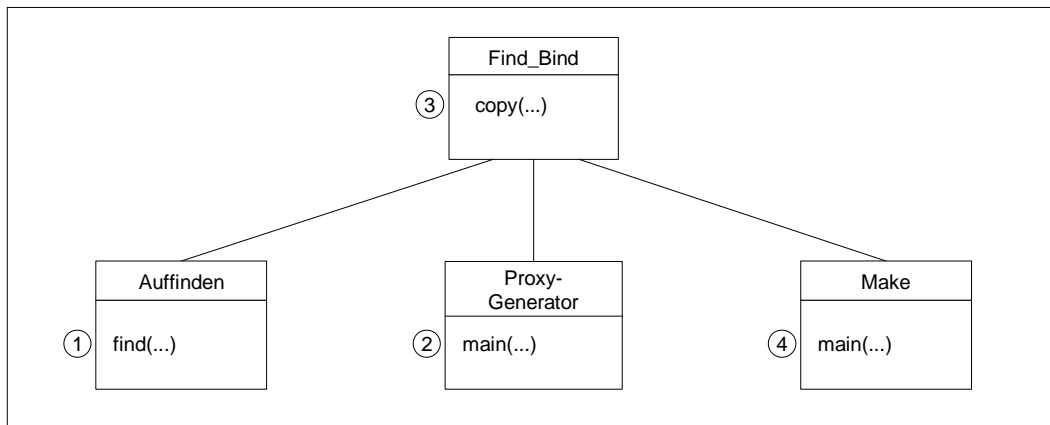


Abbildung 4.11: Statisches Einbinden, unterstützt durch Java-Anwendung

der Algorithmus implementiert wird. Ebenso muss der `copy`-Befehl entweder mit Java-Mitteln nachgebaut oder über einen System-Aufruf benutzt werden.

Auch beim Java-Ansatz gilt, dass neu zu entwickelnde Werkzeuge auf die Verwendung mit der übergeordneten Java-Anwendung ausgelegt sein sollten, um einen einfachen Zugriff auf Zwischenergebnisse zu ermöglichen. Im Fall von Java-Werkzeugen sollten dafür die entsprechenden Methoden, z.B. `getWSDL_URL()`, zur Verfügung gestellt werden.

Bewertung

Die Bewertung der eben dargestellten Realisierungsalternativen erfolgt nach folgenden Gesichtspunkten:

- Implementierungs- und Änderungsaufwand,
- Zugriff auf Zwischenergebnisse,
- Integration heterogener Werkzeuge,
- Anbindung an Benutzeroberfläche.

Bei der Skript-Lösung muss im Gegensatz zur Java-Lösung kein Programm im eigentlichen Sinn erstellt werden, das vor der Ausführung übersetzt werden muss. Dadurch ist die Skript-Alternative sehr flexibel, vor allem im Hinblick auf Änderungen im Prozessablauf. Diese Tatsache kann auch in Fällen ausgenutzt werden, wenn für bestimmte Aufgaben Teilskripte zur Laufzeit generiert werden. So könnte ein Skript zum Kopieren der generierten Proxy-Klassen durch den Proxy-Generator erzeugt werden und könnte dann aus dem Haupt-Skript heraus aufgerufen werden. Zwar können auch Java-Klassen zur Laufzeit erstellt werden, jedoch müssen diese vor dem Aufruf übersetzt werden.

Der Zugriff auf die Zwischenergebnisse ist bei beiden Alternativen ähnlich problematisch, da von den existierenden Werkzeugen die Rückgabe von Teilergebnissen

nicht vorgesehen ist. Werden dennoch solche Resultate benötigt, z.B. Klassen- und Package-Namen von generierten Objekten, müssen diese Informationen entweder aus den Eingabe- oder aus den Ausgabedateien ermittelt werden. Daraus ergeben sich Vorteile für die Skript-Lösung, bei der, je nach Skriptsprache Platzhalter und Suchmuster verwendet werden können. Im Fall von neu zu entwickelnden Software-Werkzeugen können beide Lösungen unterstützt werden.

Unter Gesichtspunkten der Integration heterogener Werkzeuge schneidet die Skript-Lösung besser ab, vor allem wenn ein neues Werkzeug in den vorhandenen Prozess integriert werden soll. Im Skript können Anwendungen, Java-Programme und Systemprogramme problemlos integriert werden. Bei der Java-Lösung sind die Aufrufe von fremden Anwendungen und Systemprogrammen immer mit einem höheren Programmieraufwand verbunden.

Ein Vorteil der Java-Lösung ist, dass die Anwendung weiter ausgebaut werden kann, z.B. durch die Anbindung an ein Fenstersystem. Dadurch könnte das Automatisierungswerkzeug mehr in Richtung eines Modellierungswerkzeuges erweitert werden, mit dem z.B. auch menschliche Interaktion mit UDDI ermöglicht werden kann, oder mit dem die Zwischenergebnisse des Auffindvorganges zur Auswahl angezeigt werden können.

Durch die Automation des statischen Einbindens wird ein Hilfsmittel zur Verfügung gestellt, mit dem Web Services, die eine gewünschte, vorgegebene Funktion erfüllen, in UDDI gesucht und alle Vorbereitungen zum Einbinden in die Prozessdefinition getroffen werden können. Die Entscheidung, welcher Web Service eingebunden wird, trifft jedoch nach wie vor der Mensch, der den Vorgang startet. Alle Entscheidungsparameter müssen dem Automatisierungs-Prozess beim Start übergeben werden. Am Ende des automatisierten Vorgangs müssen die erzeugten FDL-Konstrukte in den Workflow-Prozess eingebunden werden.

Der automatisierte Vorgang ersetzt vor allem die manuelle Suche in UDDI, mit dem (gravierenden) Nachteil, dass für aufgabenbezogene Informationen ausschließlich die Kategorien genutzt werden können. Da bei der Auswahl eines Web Service wichtige Entscheidungskriterien in UDDI fehlen, z.B. „Quality of Services“, reduziert sich die Auswahl auf die technischen Aspekte und auf die Möglichkeiten, die in eigenen Algorithmen realisiert werden. Die Vorbereitung der Suche und der Auswahl, z.B. die Festlegung der Kategorien und der Selektionskriterien kann durch die Automatisierung nicht ersetzt oder erleichtert werden.

4.5 Dynamisches Einbinden

Aus Sicht der Anwendung bietet das dynamische Einbinden von Web Services in Workflow-Prozesse einige Vorteile gegenüber dem statischen Ansatz. So kann bei jeder Ausführung des Prozesses neu entschieden werden, ob der Web Service beim aktuell billigsten Anbieter oder bei dem mit der schnellsten Antwortzeit gewählt wird. Die Reihe der Beispiele lässt sich beliebig erweitern.

Aus technischer Sicht bedeutet der dynamische Ansatz, dass das Auffinden und das Binden eines Web Services zur Ausführungszeit des Workflow-Prozesses stattfindet mit der Folge, dass diese Schritte aus dem laufenden Prozess heraus gestartet werden müssen. Die notwendigen Parameter für die Suche, zur Auswahl und zum Binden des Web Service werden dabei ebenfalls zur Ausführungszeit festgelegt. Zudem spielt im dynamischen Fall auch die Laufzeit der Einbindungs-Schritte eine bedeutende Rolle, da sich diese auf die Ausführungsdauer des Workflow-Prozesses auswirkt.

Vorüberlegungen

Wie in Abschnitt 4.4 über das statische Einbinden dargestellt wird, können manche Teile des Einbindevorganges nicht automatisiert werden. Dazu gehören die Einbettung des Aufrufes in das Prozessmodell und die Formulierung der Kategorien für die Suche nach geeigneten Web Services in UDDI. Das dynamische Einbinden wird dennoch ermöglicht, wenn der Vorgang in zwei Teile aufgeteilt wird. Im ersten Teil werden die Schritte abgearbeitet, die nicht automatisierbar sind oder für die menschliche Interaktion notwendig ist. Diese Schritte müssen zum Zeitpunkt der Modellierung durchlaufen werden und werden daher als statischer Teil betrachtet. Im dynamischen Teil, zur Ausführungszeit des Prozesses, werden dagegen die Schritte ausgeführt, die entweder automatisierbar sind oder die auf Resultate von Workflow-Aktivitäten aufbauen. Abbildung 4.12 auf der nächsten Seite illustriert die Aufteilung des Einbindungs-Vorganges in einen statischen und einen dynamischen Teil.

Statischer Teil

In den Ausführungen zum statischen Einbinden wird deutlich, dass die Zuordnung von Prozess-Datenstrukturen zu Datenstrukturen für den Web-Service nicht automatisiert werden kann, weil dazu Wissen über die Anwendung notwendig ist. Diese Zuordnung muss also vom Prozess-Modellierer vorgenommen werden. Diese Zuordnung von Datenstrukturen ist bereits auf Basis der Schnittstelle zum Web Service möglich. Die konkrete Netzwerkadresse des Web Service ist dazu nicht erforderlich. Dies bedeutet, dass die Schnittstelle zum Web Service statisch ins Workflow-Modell eingebunden, die Netzwerkadresse jedoch dynamisch ermittelt werden kann. Für das dynamische Einbinden kommen also nur unterschiedliche Web Services in Frage, die alle die gleiche Schnittstelle implementieren. Das automatische Einbinden von Web Services mit unterschiedlichen Schnittstellen scheitert jedoch an der Zuordnung der Datenstrukturen.

In Abschnitt 1.6 wird dargestellt, wie eine WSDL-Beschreibung in UDDI, getrennt nach Schnittstelle und Implementierung, gespeichert werden kann. Diese getrennte

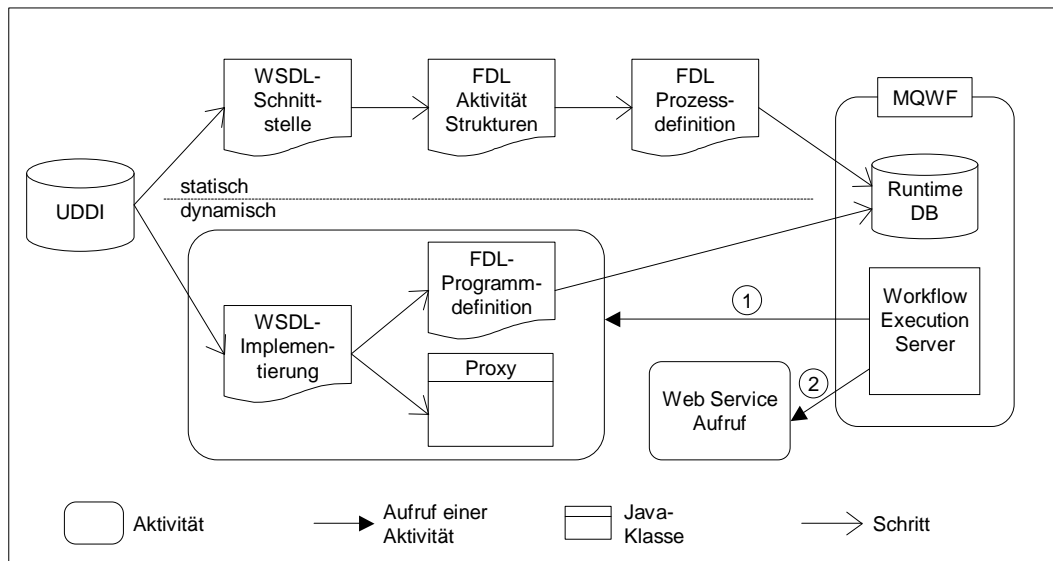


Abbildung 4.12: Statischer und dynamischer Teil des Einbindens

Speicherung kann für das dynamische Einbinden genutzt werden: beim statischen Teil wird nur die WSDL-Schnittstelle in UDDI ermittelt und beim dynamischen Teil nur die Implementierung. Die Schnittstelle enthält alle Informationen, die notwendig sind, um die statischen Prozess-Definitionen zu erzeugen.

Der statische Anteil umfasst folgende Schritte:

- Ermitteln der Schnittstelle in UDDI,
- Generieren der Datenstrukturen und einer Programmaktivität für den Web-Service-Aufruf,
- Einbindung der generierten Definitionen ins Prozessmodell,
- Import des Prozessmodells in die Runtime-Datenbank von MQWF.

Für die Ermittlung der Schnittstelle ist ein Zugriff auf UDDI notwendig. Dieser kann mit Hilfe von Suchanfragen oder über eine grafische Benutzungsschnittstelle durchgeführt werden. Wird eine geeignete TModel-Struktur gefunden, kann daraus die WSDL-Beschreibung des Web Service ermittelt werden. Diese WSDL-Beschreibung umfasst nur die WSDL-Schnittstelle, also die Elemente `<types>`, `<message>`, `<port-Type>` und `<binding>`.

Aus diesen Informationen kann eine FDL-Beschreibung für die notwendigen Workflow-Konstrukte `STRUCTURE` und `PROGRAM_ACTIVITY` generiert werden. Im anschließenden Modellierungsschritt werden diese in die Prozessdefinition eingebunden, in dem der Kontroll- und Datenfluss zwischen generierten und vorhandenen FDL-Konstrukten festgelegt wird. Der für den konkreten Web-Service-Aufruf notwendige UPES kann ebenfalls statisch definiert werden.

Der so modellierte Prozess wird schließlich in die Runtime-Datenbank von MQWF importiert und steht dann für die Prozessausführung zur Verfügung. Dennoch kann der Web Service noch nicht verwendet werden, denn das Bindeglied zwischen Programmaktivität und UPES fehlt noch im Workflow-Modell. Diese fehlende Programmdefinition wird erst im dynamischen Teil erzeugt.

Am Ende des statischen Teils gelten folgende Voraussetzungen:

- es existiert eine Prozessdefinition in der MQWF Runtime-Datenbank, die neben einer Programmaktivität zum Aufruf eines Web Service auch die erforderlichen Datenflüsse zwischen den Datenstrukturen dieser Programm-Aktivität und anderen Datenstrukturen dieser Prozessdefinition enthält,
- es existiert eine Definition des Web Service UPES in der MQWF Runtime-Datenbank,
- der UDDI-Schlüssel für die Schnittstelle ist bekannt,
- die Auswahl Aufrufalternative für den UPES ist abgeschlossen.

Weitere statische Vorgaben für das dynamische Einbinden betreffen die Kategorien für eine evtl. Einschränkung der Suche und die Art und Anzahl der Parameter für die Auswahl eines Web Service. Die Form in der diese Informationen gespeichert sind, hängt von der Alternative ab. Abbildung 4.13 illustriert die Prozessdefinition nach Abschluss des statischen Teils.

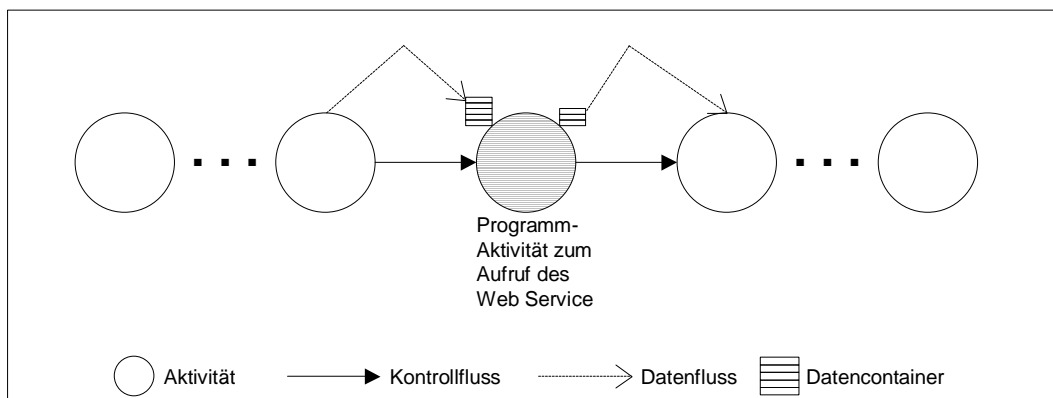


Abbildung 4.13: Programmaktivität zum Aufruf eines Web Service, eingebunden ins Workflow-Modell

Dynamischer Teil

Die vorhandenen Web-Service-Erweiterungen zu MQWF, die in Abschnitt 2 dargestellt werden, sind in der aktuellen Version auf das statische Einbinden ausgelegt. Dies betrifft vor allem den Aufrufmechanismus für Web Services in Form eines UPES. Zunächst sollen Probleme und Lösungsstrategien für das dynamische Einbinden untersucht werden, die bei der Nutzung des vorhandenen UPES auftreten. Anschließend wird eine alternative Lösung zum Aufruf von Web Services skizziert.

Nutzung des vorhandenen Web Service UPES

Die Nutzung des vorhandenen Web Service UPES macht die in Abbildung 4.12 auf Seite 57 skizzierten Schritte erforderlich, die einzeln oder gesammelt aus dem laufenden Workflow-Prozess gestartet werden müssen. Zu diesen Schritten gehören:

- Auffinden einer Web-Service-Implementierung,
- Ermitteln der WSDL-Beschreibung des Web Service,
- evtl. Generierung von Java-Proxy-Klassen für den Aufruf,
- Generierung einer Programm-Definition in FDL,
- Import der Programmdefinition in die MQWF Runtime-Datenbank.

Für die Einzelschritte stehen die in den vorangegangenen Abschnitten dargestellten Software-Werkzeuge zur Verfügung, d.h. für das Aufsuchen in UDDI muss eine Anwendung neu entwickelt werden, ein Proxy-Generator existiert in Form eines Java-Programms, `proxygen`, das über ein Stapelverarbeitungsprogramm gestartet werden kann und für die Generierung der FDL-Programm-Definition kann das Software-Werkzeug `wsdl2fdl` benutzt werden. Die vorhandenen Programme sind weder für den dynamischen Einsatz noch für den Austausch von Zwischenergebnissen in anderer Form als über Dateien konzipiert.

Die Konsequenz aus dem Einsatz dieser heterogenen Software-Werkzeuge ist, dass ein übergeordneter Kontroll- und Datenfluss etabliert werden muss, der wiederum aus dem laufenden Workflow-Prozess heraus gestartet wird. Dazu muss das Prozessmodell um Datenstrukturen und Aktivitäten für die dynamischen Schritte ergänzt werden, vgl. Abbildung 4.14. Diese Schritte müssen vor dem eigentlichen Web-Service-Aufruf abgeschlossen sein.

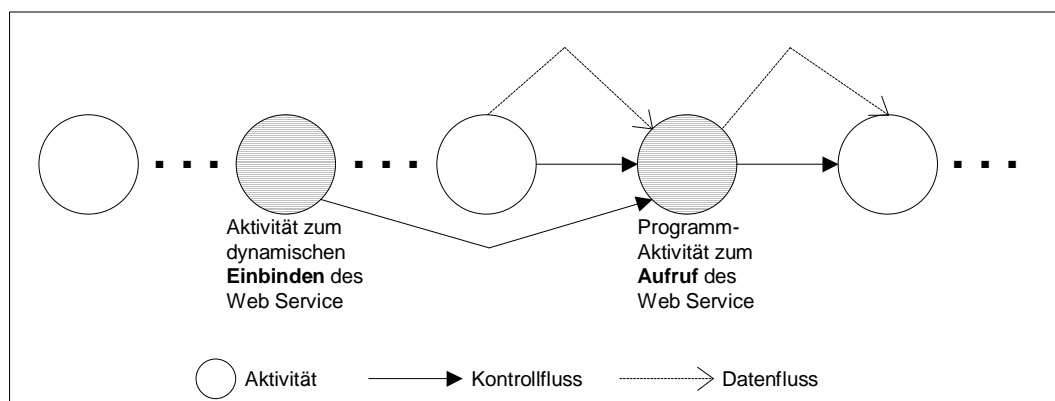


Abbildung 4.14: Prozess-Definition für den Start des dynamischen Teils

Zum Start des dynamischen Teils muss mindestens eine Programmaktivität mit den entsprechenden Datenflüssen ins Workflow-Modell aufgenommen werden. Die Implementierung des Einbindungs-Vorganges kann auf vielfältige Weise erreicht werden.

Denkbar ist die Implementierung als Workflow-Prozess, als Java-Applikation oder als Anwendung in einer Skriptsprache.

Bevor die Probleme und Fragen der Gestaltung des Gesamtprozesses betrachtet werden, werden zunächst die Einzelschritte näher beleuchtet.

Die notwendigen Information für den dynamischen Teil des Einbindens werden durch mehrere Suchanfragen an UDDI ermittelt. Da die Schnittstelle bereits statisch ermittelt wurde und der Schlüssel für die UDDI-Struktur bekannt ist, kann folgende Anfragevariante eingesetzt werden: zunächst wird eine `find_business`-Anfrage formuliert, die in der `<tModelBag>` den Schlüssel (`tModelKey`) der Schnittstelle enthält. Zur weiteren Einschränkung der Suchergebnisse können Kategorien oder Namen angegeben werden, vgl. Abschnitt 4.2. Durch die Angabe des Schnittstellen-Schlüssels, werden bei der Ermittlung des Suchresultates nur solche `BusinessEntity`- und `BusinessService`-Objekte berücksichtigt, für die ein `BindingTemplate` in UDDI gespeichert ist, das auf diese Schnittstelle verweist. Anschließend werden für die `BusinessServices` im Resultat der ersten Anfrage eine `find_binding`-Anfrage formuliert. Diese Anfragen enthalten einen `serviceKey` und eine `<tModelBag>` und liefern als Ergebnis eine Reihe von `BindingTemplates` mit einem Verweis auf die entsprechende Schnittstelle. Diese `BindingTemplates` enthalten schließlich die WSDL-Beschreibung der Web Services.

Die Möglichkeiten zur Auswahl des einzubindenden Web Service wurden bereits in Abschnitt 4.2 dargestellt. Ebenso, dass wichtige Auswahlkriterien für Web Services, die besonders beim dynamischen Ansatz von Interesse sind, z.B. „Quality of Services“, nicht in UDDI abgefragt werden können. Sobald jedoch Taxonomien für die entsprechenden Auswahlkriterien etabliert sind, können diese durch Angabe von Kategorien zur Verfeinerung der Suche benutzt werden. Zur Anwendung der dargestellten Verfahren für die Formulierung von Präferenzen oder Ausschlüssen, sind entweder Namen bei der `find_business`-Suchanfrage anzugeben oder die entsprechenden Algorithmen zu implementieren.

Um einen Reibungslosen Ablauf des Auffindens zu gewährleisten ist eine sorgfältige Planung des Datenflusses notwendig. Bei der Modellierung müssen vor allem die Datenquellen und die Lebensdauer berücksichtigt werden.

Die Daten, die für die Formulierung der Suchanfragen benötigt werden, stammen aus unterschiedlichen Quellen. Einerseits wird auf Daten zugegriffen, die beim statischen Teil des Einbindens festgelegt werden, andererseits auf solche, die während der Ausführung des Workflow-Prozesses erzeugt werden. Außerdem kommen noch weitere Datenquellen in Betracht, z.B. wenn spezielle Algorithmen zur Auswahl benutzt werden. Letztere werden bei den weiteren Ausführungen nicht betrachtet. Statisch festgelegt werden der Schlüssel für die Schnittstelle und evtl. Kategorien für die Einschränkung der `find_business`-Anfrage. Die dynamisch erzeugten Daten stehen in Form von Workflow-Containern zur Verfügung.

Die Daten aus den angeführten Datenquellen können nach ihrer Lebensdauer unterschieden werden:

- instanzbezogene Daten,

- prozessbezogene Daten,
- globale Daten.

Dabei sind instanzbezogene Daten immer nur für eine Prozess-Instanz gültig. Sie repräsentieren die dynamische Komponente des Einbindens. Dagegen sind die prozessbezogenen Daten für alle Instanzen eines Prozesses gültig. Diese umfassen die Daten, die durch den statischen Teil vorgegeben sind. Globale Daten sind gültig für mehrere Prozess-Klassen. Dazu gehören z.B. die Konfigurationsdaten des UDDI-Betreibers.

Die instanzbezogenen Daten werden in Form von Workflow-Datenstrukturen modelliert, da sie bei der Ausführung von Workflow-Aktivitäten erzeugt werden und nur für eine Prozess-Instanz gültig sind. Auf diese Daten kann auf zwei Arten zugegriffen werden: entweder sie werden der Anwendung als Aufrufparameter übergeben oder die Anwendung greift über die Schnittstelle zu MQWF auf die Containerinhalte zu.

Die prozessbezogenen Daten können sowohl innerhalb als auch außerhalb eines Workflow-Prozesses gehalten werden. Der Zugriff erfolgt ausschließlich lesend und ist deshalb unkritisch, auch wenn viele Prozesse gleichzeitig ausgeführt werden. Innerhalb eines Prozesses können solche Daten als Default-Werte modelliert werden.

Die globalen Daten gelten für viele unterschiedliche Prozesse. Auch hier erfolgt ausschließlich lesender Zugriff. Zur Vermeidung von Redundanz sollten solche Daten außerhalb der Workflow-Prozesse, z.B. in einer Konfigurationsdatei gespeichert werden.

In Abbildung 4.15 auf der nächsten Seite werden die Zugriffsmöglichkeiten dargestellt.

Ebenso wichtig wie die Planung der Eingabedaten für das Auffinden ist die Modellierung der Ausgabe, also in welcher Form und auf welchem Weg das Suchresultat an die nachfolgenden Schritte weitergegeben wird. Die Rückgabe ist davon abhängig, wie der Gesamtprozess zum dynamischen Einbinden realisiert ist. Da die Ausgabe instanzbezogen ist empfiehlt sich eine Rückgabe an den Workflow-Prozess. Dazu kann die Schnittstelle zu MQWF herangezogen werden. Erfolgt die Weiterverarbeitung in einer Skriptsprache oder im Rahmen einer Anwendung sollte das Resultat an die entsprechende Umgebung zurückgegeben werden. Ungeeignet im Hinblick auf Aspekte des Mehrbenutzerbetriebes ist die Speicherung von Zwischenresultaten in Dateien außerhalb der rufenden Umgebung.

Im Gegensatz zum statischen Ansatz spielt die Ausführungszeit für das dynamische Auffinden eine wichtige Rolle. Die Zeiten für die Anfrageformulierung, die UDDI-Zugriffe und den Auswahlalgorithmus müssen bei jedem Web-Service-Aufruf erneut aufgebracht werden. Wenn die Annahmen gelten, dass die Anzahl der Web-Service-Aufrufe sehr viel größer ist, als die Anzahl der angebotenen Schnittstellen und die 80:20-Regel [HR99] auch bei den Web-Service-Aufrufen angewendet werden kann, dann kann durch die Anwendung von Caching-Verfahren ein Großteil dieser zusätzlichen Ausführungszeit eingespart werden.

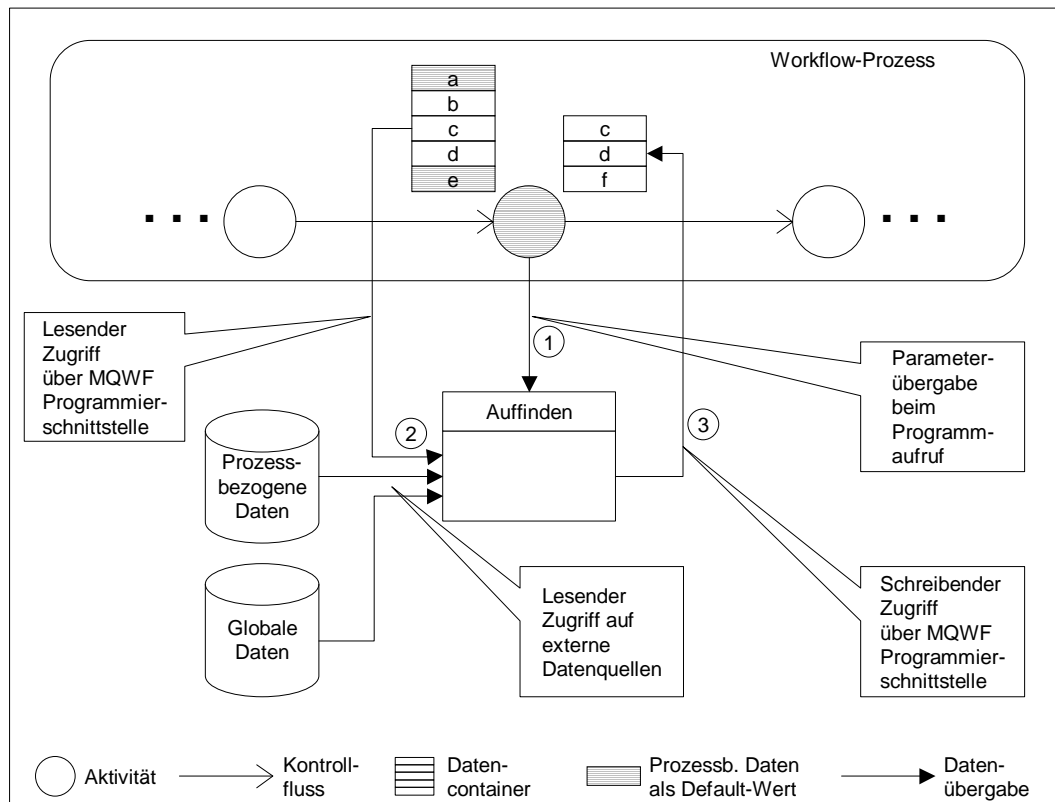


Abbildung 4.15: Zugriffsmöglichkeiten auf Daten beim Auffinden

Ein Cache zur Vermeidung von UDDI-Zugriffen kann dadurch realisiert werden, dass eine prozessbezogene Tabelle zur Verfügung gestellt wird, in der bei jeder Suchanfrage die Kombination der Auswahlkriterien und das Suchergebnis gespeichert werden. Bevor ein UDDI-Zugriff durchgeführt wird, wird die Tabelle überprüft, ob für die Kombination von Suchkriterien bereits ein Resultat ermittelt wurde. Ist dies der Fall kann mit dem gespeicherten Resultat weitergearbeitet werden. Schlägt der Aufruf fehl, z.B. weil der Web Service nicht mehr angeboten wird, muss die Suche wiederholt werden. Bei jedem durchlaufenen Auffindvorgang mit UDDI-Zugriff wird ein neuer Eintrag im Cache erzeugt. Dabei werden evtl. veraltete Versionen überschrieben.

In Abschnitt 4.3 werden die Möglichkeiten zum Binden dargestellt. Dabei wird deutlich, dass die Auswahl der Variante für den UPES-Aufruf Auswirkungen auf die notwendigen Schritte zum Einbinden hat. So müssen bei der Nutzung der Proxy-Variante die notwendigen Proxy-Klassen erzeugt werden. Im dynamischen Fall werden diese Klassen zur Ausführungszeit des Workflow-Prozesses generiert, nachdem in einem früheren Schritt der einzubindende Web Service ausgewählt wurde.

Die Nutzung des Software-Werkzeuges `proxygen` für das dynamische Einbinden ist aus folgenden Gründen problematisch:

- Zugriff auf instanzbezogene, prozessbezogene und globale Daten

- Rückgabewerte

Daneben gelten die Ausführungen zur Laufzeitoptimierung durch Caching beim UDDI-Zugriff analog.

Für die Ausführung von `proxygen` sind als Parameter die URL des WSDL-Dokumentes und das Ausgabeverzeichnis anzugeben. Die URL ist instanzbezogen und muss aus dem Resultat des Auffindschrittes ermittelt werden. Das Ausgabeverzeichnis ist entweder prozessbezogen oder global festgelegt. Da diese Daten aus verschiedenen Datenquellen ermittelt werden müssen, ist noch ein Zwischenschritt erforderlich, bei dem die Parameter ermittelt und für den Aufruf von `proxygen` kombiniert werden. Dies kann z.B. durch eine Java-Klasse realisiert werden, die die Parameter ermittelt und in eine Workflow-Datenstruktur ausgibt oder diese direkt an `proxygen` zur Ausführung weiterleitet. Wird als Ausgabeverzeichnis ein Verzeichnis gewählt, das im CLASSPATH des UPES registriert ist, kann der Schritt zum Kopieren der Klassen entfallen.

Nicht entfallen kann jedoch die Analyse der Ausgaben von `proxygen`. Der qualifizierte Klassenname der Proxy-Klasse sind als Parameter für den Aufruf des UPES erforderlich. Die Namen werden bei der Proxy-Generierung aus Werten in der WSDL-Beschreibung abgeleitet. Damit der Name ermittelt werden kann, muss in einem weiteren Zwischenschritt das WSDL-Dokument analysiert und an die rufende Umgebung, z.B. MQWF zurückgegeben werden. Als Optimierung kann diese Analyse mit der Ermittlung der Eingabeparameter zusammengefasst werden, vgl. Abbildung 4.16

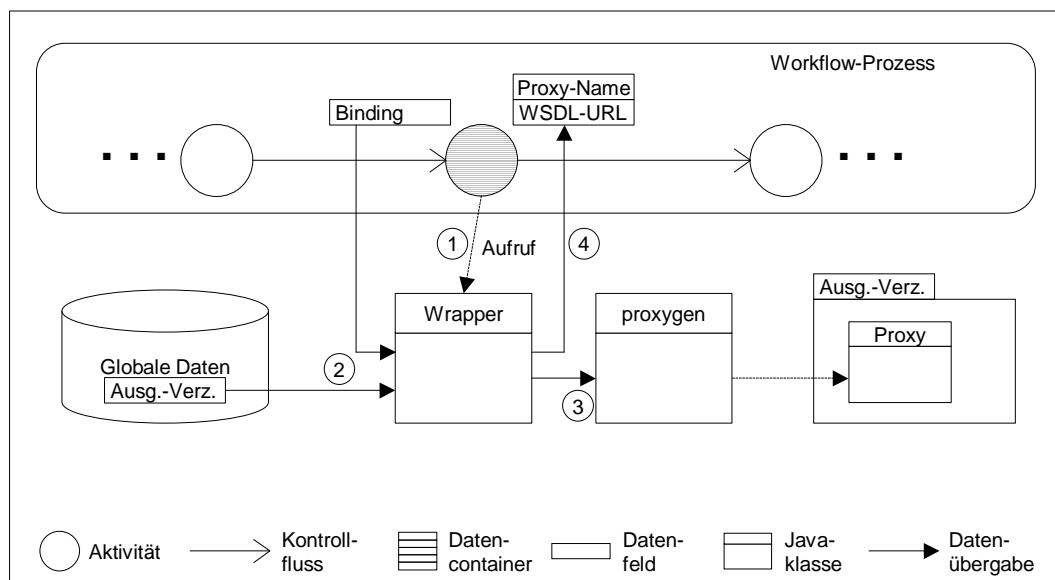


Abbildung 4.16: Aufruf zur Generierung der Proxy-Klasse aus MQWF

Durch einen geeigneten Caching-Mechanismus kann auch bei der Generierung der Proxy-Klassen Zeit eingespart werden. Ähnlich, wie beim Cache für die UDDI-Suche, könnte eine Tabelle mit WSDL-URLs und Proxy-Klassennamen zur Verfügung

gestellt werden. Ist ein Eintrag vorhanden, müssen die Proxy-Klassen nicht neu generiert werden.

Nachdem die Proxy-Klassen für den Aufruf generiert worden sind bzw. wenn die Aufrufvariante ohne Proxy-Klasse gewählt wird, muss aus der WSDL-Beschreibung eine Programmdefinition erzeugt und in die Runtime-Datenbank von MQWF importiert werden. Dazu kann das in Abschnitt 2.2 vorgestellte Software-Werkzeug `wsd12fdl` benutzt werden. Allerdings besitzt `wsd12fdl` im Hinblick auf die Verwendung für das dynamische Einbinden gravierende Nachteile:

- kein lesender und schreibender Zugriff auf Workflow-Datenstrukturen,
- Steuerung der Generierung über externe Konfigurationsdatei,
- Erzeugung von FDL-Konstrukten kann nicht auf PROGRAM eingeschränkt werden.

Beim Software-Werkzeug `wsd12fdl` bereitet der Zugriff auf Eingabeparameter bzw. die Rückgabe von Resultaten ähnliche Probleme wie bei `proxygen`. Weitaus gravierender ist, dass alle Einstellungen für den Übersetzungsvorgang in Form einer Datei an `wsd12fdl` übergeben werden. Diese Datei enthält neben prozessbezogenen bzw. globalen Daten auch instanzbezogene Informationen. Da `wsd12fdl` nur diese Möglichkeit der Parameterübergabe zur Verfügung stellt, muss diese für jede Prozess-Instanz modifiziert oder neu erzeugt werden, vgl. Abbildung 4.17.

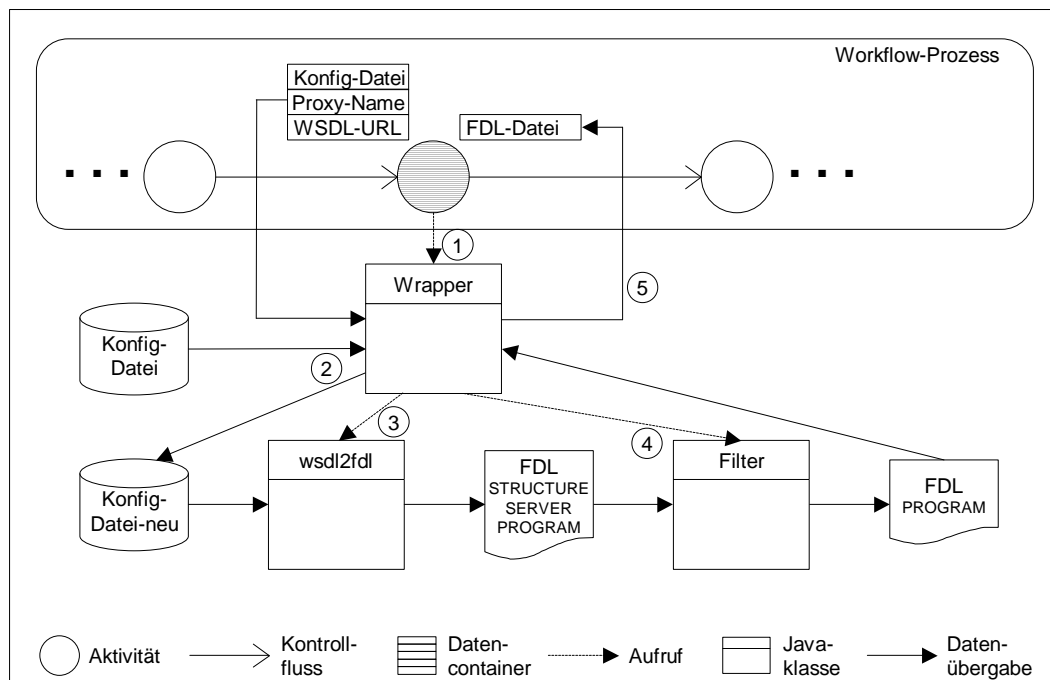


Abbildung 4.17: Dynamische Generierung einer Programmdefinition

Ein weiterer Nachteil besteht darin, dass durch `wsdl2fdl` für jede WSDL-Beschreibung folgende FDL-Konstrukte erzeugt werden: `STRUCTURE`, `SERVER`, `PROGRAM` und optional `PROCESS`. Die Erzeugung kann nicht auf `PROGRAM` eingeschränkt werden, obwohl nur dieses für den Web-Service-Aufruf benötigt wird. Folglich muss in einem nachgeschalteten Schritt dieses `PROGRAM` isoliert werden, damit beim späteren Import in die Runtime-Datenbank nur die fehlende Programmdefinition übernommen wird.

Zur Laufzeitoptimierung kann auch hier ein Caching-Mechanismus eingesetzt werden, im Idealfall in Kombination mit den Caching zur Proxy-Generierung. Dazu muss in die Tabelle statt der Proxy-Klasse die erzeugte Programmdefinition aufgenommen und bei Bedarf zurückgegeben werden. Auf diese Weise kann der gesamte Generierungsprozess eingespart werden.

Bei den bisherigen Betrachtungen wurde ein gravierendes Problem noch nicht betrachtet. Beim statischen Ansatz wurden alle FDL-Definitionen generiert, modifiziert und in die Runtime-Datenbank von MQWF übernommen, d.h. jedes FDL-Konstrukt existiert genau einmal. Betrachtet man den dynamischen Ansatz, fällt auf, dass für jede Prozess-Instanz, bzw. für eine Gruppe von Prozess-Instanzen ein `PROGRAM` erzeugt und in die Runtime-Datenbank importiert wird. Da der Programmname in der `PROGRAM_ACTIVITY` statisch festgelegt ist, wird das `PROGRAM` bei jedem Importvorgang mit der neuen Version überschrieben, was selbstverständlich unerwünschte Effekte auf laufende Prozesse nach sich ziehen kann. Dies kann verhindert werden, wenn der Import und der Web-Service-Aufruf, d.h. die Zugriffe auf das `PROGRAM` in derselben Transaktion ausgeführt werden. Dies ist nicht in allen Situationen möglich, z.B. wenn ein Web Service zur Kursabfrage eingebunden wird und anschließend Hunderte von Kursabfragen aufgerufen werden.

Eine Alternative zum Transaktionsschutz besteht darin, mehrere Programmaktivitäten zum Aufruf verschiedener Web Services in einem gemeinsamen Sub-Prozess zu kapseln. Statisch wird dann statt einer `PROGRAM_ACTIVITY` eine verb `PROCESS_ACTIVITY` modelliert und der gerufene Sub-Prozess wird dynamisch um eine Programmaktivität und die korrespondierende Programmdefinition ergänzt. Das Prinzip ist in Abbildung 4.18 auf der nächsten Seite dargestellt.

Zur Auswahl der `PROGRAM_ACTIVITY` wird beim Einbinden für jede `PROGRAM_ACTIVITY` im Sub-Prozess eine Transitionsbedingung erzeugt. Dieser Wert der Transitionsbedingung wird im rufenden Prozess gespeichert. Auf diese Weise kann sichergestellt werden, dass die richtige Programmaktivität gestartet wird. Dabei muss nicht für jede Prozess-Instanz eine neue Programmaktivität mit korrespondierender Programmdefinition generiert werden, sondern nur für unterschiedliche Web Services, vgl. Ausführungen über das Caching.

Die einzelnen Einbindungsschritte müssen nicht zwangsweise einzeln aus dem Workflow-Prozess heraus gestartet werden. Neben den bislang dargestellten Möglichkeiten sind weitere Realisierungsalternativen denkbar. Um den Anwendungsbezug des Prozessmodells nicht aufzuweichen, sollte die Anzahl an Aktivitäten, die die Administration des Systems betreffen, möglichst gering gehalten werden. Zu diesem Zweck sind grundsätzlich zwei Alternativen denkbar:

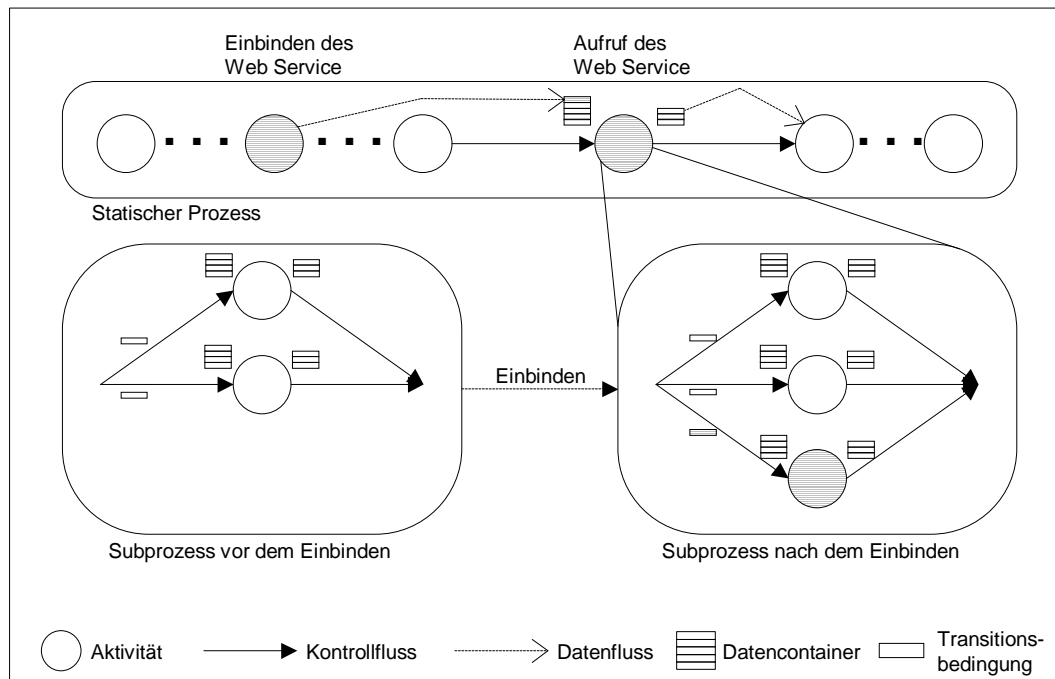


Abbildung 4.18: Aufruf eines Web Service als Prozessaktivität

- Kapselung der administrativen Schritte in einem Sub-Prozess.
- Zusammenfassen der administrativen Schritte in einem Anwendungsprogramm.

Bei beiden Ansätzen können alle notwendigen Schritte auf eine Aktivität im Hauptprozess zusammengefasst werden. Für die Instanz- und prozessbezogenen Daten werden bei beiden Alternativen Workflow-Datenstrukturen benutzt.

Bei der ersten Alternative werden die Einzelschritte durch MQWF gesteuert. Dabei besteht die Möglichkeit, mögliche Fehlersituationen durch alternative Pfade im Prozessmodell zu behandeln. Der Datenaustausch zwischen den Einzelschritten erfolgt über die MQWF-Programmierschnittstelle und Workflow-Datenstrukturen. Aus diesem Grund müssen beim Einsatz von geänderten Softwarewerkzeugen Änderungen am Prozessmodell und beim Datenzugriff vorgenommen werden. In Abbildung 4.19 auf der nächsten Seite ist angedeutet, wie die dynamischen Schritte als Sub-Prozess, also mit Hilfe von MQWF ausgeführt werden.

Bei der zweiten Alternative wird die Kontrolle des Einbindungs-Vorganges an ein Anwendungsprogramm übertragen, vgl. Abbildung 4.20 auf Seite 68. Zugriffe auf die MQWF-Datenstrukturen sind dann nur zu Beginn und am Ende des Prozesses erforderlich. Jedoch muss der Datenfluss zwischen den Werkzeugen auch bei dieser Alternative implementiert werden. Ein Vorteil dieses Ansatzes besteht darin, dass bei Änderungen an den eingesetzten Werkzeugen das Prozessmodell nicht geändert werden muss. Dagegen sind Änderungen am Anwendungsprogramm erforderlich, die jedoch bei geeigneter Implementierung lokal begrenzt werden können.

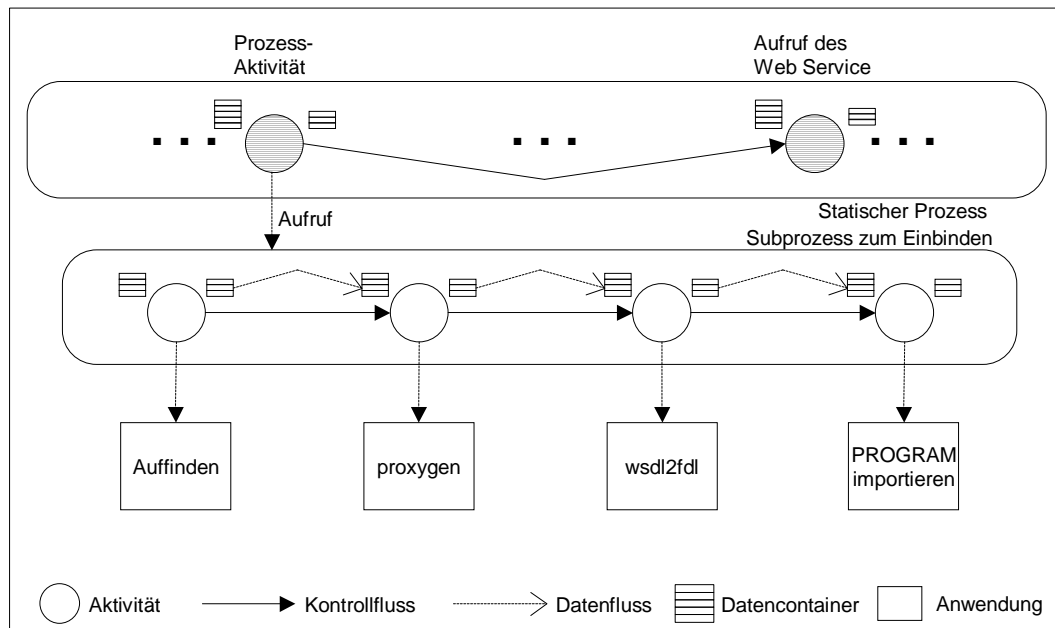


Abbildung 4.19: Dynamisches Einbinden als Sub-Prozess

Eine Alternative zum Anwendungsprogramm aus einem Guss ist, die Einzelschritte mit Hilfe einer Skriptsprache zu verbinden. Dieser Ansatz ist sehr flexibel gegenüber geringen Änderungen und bietet sich besonders dann an, wenn heterogene Software-Werkzeuge integriert werden müssen.

Nutzung alternativer Aufrufmechanismen

Die in Abschnitt 2.2 auf Seite 19 dargestellten Web-Service-Erweiterungen haben im Hinblick auf das dynamische Einbinden gravierende Nachteile:

- für jeden Web Service ist eine neue Programmdefinition erforderlich, auch wenn die Schnittstelle für diese Web Services gleich ist,
- für jeden Web Service müssen Proxy-Klassen erzeugt werden,
- für die Generierung der Proxy-Klassen wird kein Werkzeug zur Verfügung gestellt.

Unter der Voraussetzung, dass für das dynamische Binden nur solche Web Services in Frage kommen, für die eine WSDL-Beschreibung verfügbar ist und auf die über SOAP und HTTP zugegriffen werden kann, ist eine weniger komplizierte Lösung für das dynamische Einbinden denkbar. Bei genauer Betrachtung der eben genannten Voraussetzungen fällt auf, dass sich Web Services mit der gleichen Schnittstelle nur in einem einzigen Punkt unterscheiden: der Netzwerkadresse für den Zugriff.

Wenn nun eine Proxy-Klasse generiert wird, deren Netzwerkadresse für jeden Zugriff eingestellt werden kann, dann kann das gesamte Prozessmodell statisch definiert werden. Die Schritte zum Erzeugen und Installieren von Proxy-Klassen sowie

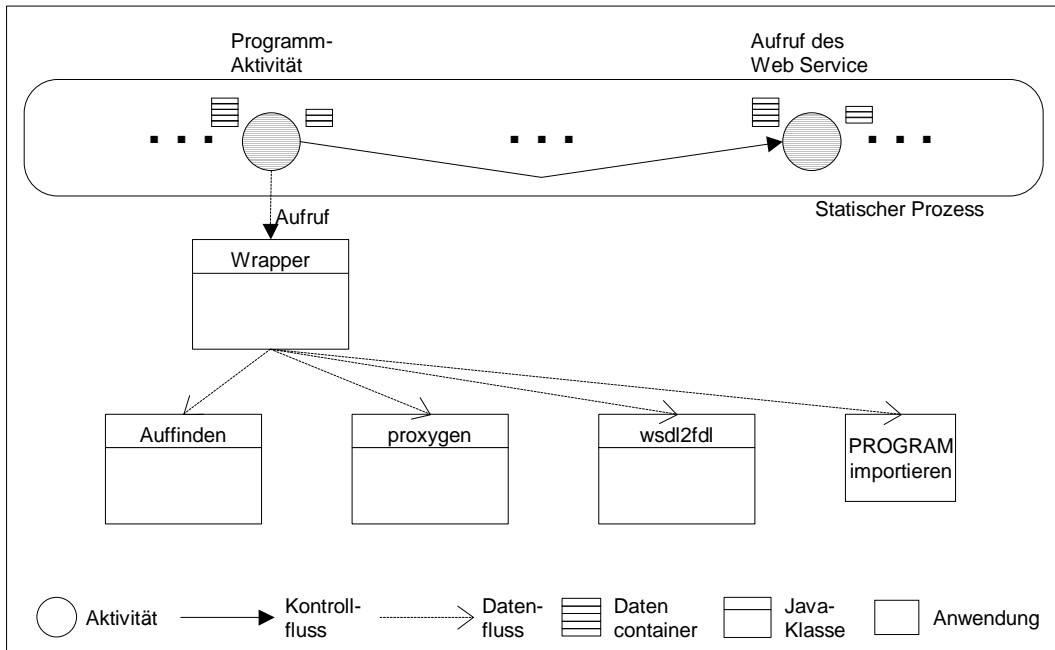


Abbildung 4.20: Dynamisches Einbinden als Java-Anwendung

zur Generierung und Einbindung von Programmdefinition können somit entfallen. Somit kann das Prozessmodell weitgehend frei von Aktivitäten für administrative Schritte gehalten werden. Darüber hinaus sind keine Vorkehrungen zu treffen, um die Ausführungszeit durch Caching-Maßnahmen zu verringern. Auch der Datenaustausch zwischen MQWF und den Software-Werkzeugen braucht nicht gesondert implementiert zu werden. Ebenso sind keine besonderen Maßnahmen erforderlich, um den konkurrierenden Zugriff verschiedener Prozess-Instanzen auf die Werkzeuge zu regeln. Die statischen Schritte sind in Abbildung 4.21 dargestellt.

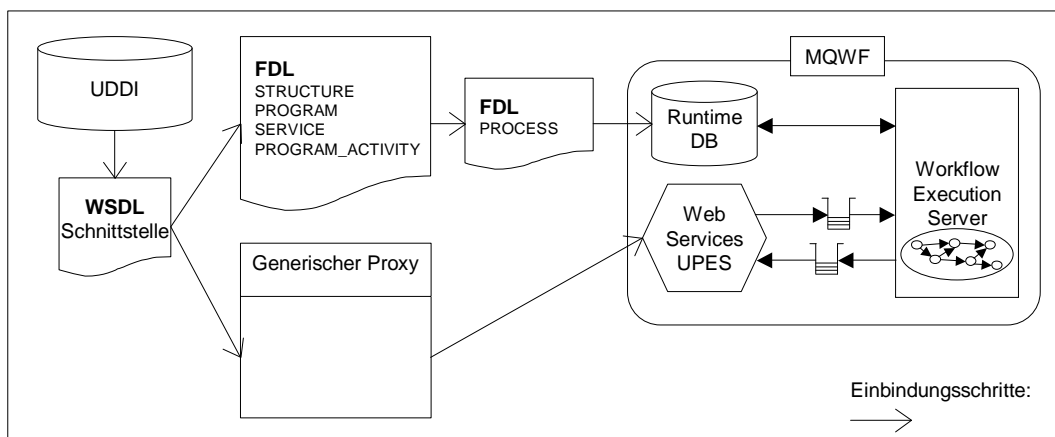


Abbildung 4.21: Statische Schritte bei Aufruf über generischen Proxy

Der Nachteil dieser Lösung ist, dass ein Proxy-Generator entwickelt werden muss,

der aus einer WSDL-Schnittstelle eine Proxy-Klasse generiert, die an beliebige Netzwerkadressen gebunden werden kann. Dann können zum Modellierungszeitpunkt alle Vorkehrungen für den Web-Service-Aufruf getroffen werden. Das Auffinden erfolgt nach wie vor dynamisch, hat aber nicht eine WSDL-Beschreibung als Resultat, sondern die Netzwerkadresse, die direkt aus den UDDI-Daten und nicht über den „Umweg“ einer WSDL-Beschreibung ermittelt werden kann. Abbildung 4.22 illustriert, wie der dynamische Teil in den Workflow-Prozess eingebunden wird.

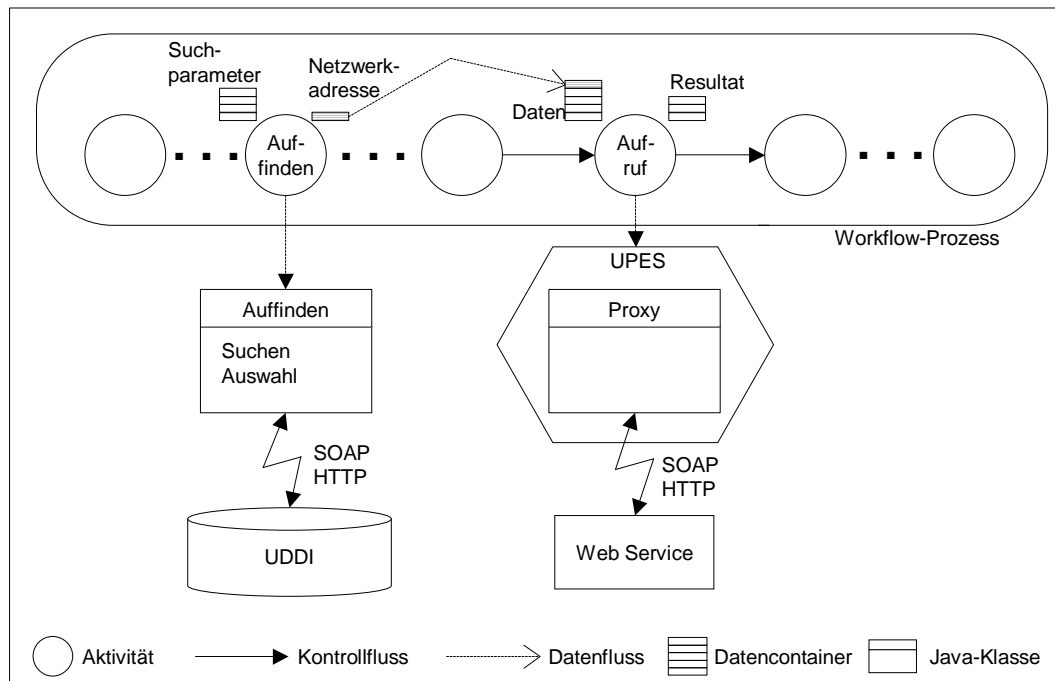


Abbildung 4.22: Dynamische Schritte, eingebunden in den Workflow-Prozess

Zusammenfassung

Das Dynamische Einbinden von Web Services in IBM MQ Series Workflow wird durch UDDI und die Web-Service-Erweiterungen grundsätzlich ermöglicht. Allerdings können nur solche Web Services dynamisch eingebunden werden, die die gleichen Methoden, Daten und Nachrichtenformate zur Kommunikation benutzen, also die gleiche Schnittstelle implementieren. Das dynamische Einbinden von Web Services mit unterschiedlicher Schnittstelle scheitert daran, dass der semantisch richtige Datenfluss im Prozessmodell nur statisch festgelegt werden kann.

UDDI unterstützt das dynamische Einbinden durch verschiedene Anfragemöglichkeiten über die Programmierschnittstelle. Die automatisierte Suche wird dabei durch das Kategorie-Konzept, das auf vorgegebenen Kategorien aufbaut, ermöglicht. Die gezielte Suche nach Web Services mit der gleichen Schnittstelle wird durch entsprechende UDDI-Anfragen direkt unterstützt. Für Informationen, die im Hinblick auf das dynamische Einbinden von besonderem Interesse sind, z.B. „Quality of Services“, bietet UDDI leider keine direkte Unterstützung. Der einzig mögliche Ansatz dafür besteht darin, entsprechende Taxonomien zu entwickeln und in UDDI zu registrieren.

Der Einbindungsprozess wird durch zwei Dinge erschwert: die Modellierung des Web-Service-Aufrufes und die Unterstützung durch Werkzeuge. Durch das Festschreiben der Übergabeparameter und der Proxy-Klassen für den Web-Service-Aufruf im Prozessmodell, wird bei jedem Einbinden eine Modifikation des Prozessmodells erforderlich. Dies hat negative Auswirkungen auf die Laufzeit des Prozesses und den konkurrierenden zugriff mehrerer Prozess-Instanzen auf das Prozessmodell. Die vorhandenen Werkzeuge sind für den statischen Einsatz konzipiert und können nur mit Hilfe von neu zu entwickelnden Hilfsprogrammen in den Prozess eingebunden werden. Für die UDDI-Anfragen und die Auswahl des Web Service müssen die Anwendungen noch entwickelt werden.

Eine dynamische Ausrichtung des Aufrufmechanismus für Web Services mit der gleichen Schnittstelle könnte den Einbindungsvorgang erheblich vereinfachen. Wenn der Aufrufmechanismus so gestaltet wird, dass zusätzlich zu den Nutzdaten die Netzwerk-Adresse des Web Service als Parameter übergeben werden kann, könnten alle Schritte zum Erzeugen von Aufrufkomponenten zur Laufzeit entfallen.

Für eine sinnvolle Nutzung von UDDI zum dynamischen Einbinden sind zwei Kriterien entscheidend. Einerseits müssen Informationen, wie „Quality of Services“, in UDDI integriert werden, andererseits ist der Aufrufmechanismus so zu gestalten, dass dieser statisch erzeugt und dynamisch mit der Netzwerkadresse parametrisiert werden kann.

5 Lösung für das Veröffentlichen

5.1 Anforderungen

Für das Veröffentlichen eines Workflow-Prozesses als Web Service in einem UDDI-Verzeichnis soll eine Lösung entwickelt werden, mit der alle notwendigen Schritte zusammenhängend ausgeführt werden können, vgl. Abschnitt 3.2. Die Einzelschritte sind:

- Generierung einer WSDL-Beschreibung des Web Service, getrennt nach WSDL-Schnittstelle und WSDL-Implementierung,
- Generierung der notwendigen Typumwandlungsklassen für die SOAP-Schnittstelle,
- Registrierung der SOAP-Schnittstelle beim SOAP Server,
- Speicherung der WSDL Beschreibung in UDDI, getrennt nach Schnittstelle und Implementierung.

Dabei sollen die Web-Service-Erweiterungen zu IBM MQ Series Workflow (MQWF) genutzt und das Software-Werkzeug `fdl2wsdl` integriert werden. Die Zugriffskomponente für UDDI muss neu entwickelt werden und über die Möglichkeit verfügen, Kategorisierungsinformationen für die UDDI-Objekte aus einer externen Datenquelle zu laden und einzufügen.

Die Lösung soll vorwiegend vom technischen Personal verwendet werden, um aus einer vorhandene Prozessdefinition, unabhängig vom laufenden Betrieb des Workflow Systems, eine Web-Service-Schnittstelle für den Prozess zu etablieren und diese in UDDI zu registrieren. Die Lösung soll von der Kommandozeile aus gestartet werden können. Alle notwendigen Informationen und Parameter für den Vorgang werden aus Dateien gelesen, d.h. eine grafische Benutzungsschnittstelle ist nicht erforderlich. Die Lösung ist auf Basis der Java 2 Plattform unter Verwendung des Java Development Kit 1.3 von Sun (JDK 1.3) zu entwickeln.

Die Eingabe für den Gesamtvorgang besteht aus einer Prozessdefinition in Flow Definition Language (FDL) und einer oder mehrerer Dateien mit Konfigurations- bzw. Kategorisierungsinformationen. Folgende Ausgaben werden erzeugt:

- Typumwandlungsklassen zwischen MQWF- und Web-Service-Datenstrukturen in Form von Java-Klassen,
- Registrierungs-Eintrag für den Web Service beim SOAP Server,
- je eine Datei für die Web-Service-Beschreibung in WSDL, getrennt nach WSDL-Schnittstelle und WSDL-Implementierung,
- in UDDI registriertes TModel-Objekt für die WSDL-Schnittstelle,

- in UDDI registriertes BusinessService-Objekt für die WSDL-Implementierung.

Weiter sollen folgende Nebenbedingungen gelten:

- Die Voraussetzungen der Web-Service-Erweiterungen sind zu beachten, d.h. die dafür notwendigen Java-Pakete und Programme sind auf der Zielplattform installiert und einsatzbereit.
- Beim UDDI-Betreiber ist ein Benutzerkonto eingerichtet und ein BusinessEntity-Objekt registriert. Die vom UDDI-Betreiber festgelegte Höchstzahl an gespeicherten Web Services ist nicht erreicht.

Aus den eben dargestellten Anforderungen können drei Schwerpunkte abgeleitet werden: Entwicklung einer Zugriffskomponente für UDDI, Integration von globalen und prozessbezogenen Konfigurationsinformationen vorhandener und neu zu entwickelnder Werkzeuge sowie Kontrolle der Einzelschritte. Die Spezifikationen und Entwürfe dieser drei Schwerpunkte werden in den folgenden Abschnitten dargestellt.

5.2 Zugriffskomponente für UDDI

5.2.1 Spezifikation

Die Zugriffskomponente für UDDI muss folgende Aufgaben erfüllen:

- Speicherung einer Web-Service-Schnittstelle in UDDI als TModel-Objekt.
- Speicherung einer Web-Service-Implementierung in UDDI als BusinessService-Objekt.
- Zuordnung von Kategorisierungsinformationen zu den UDDI-Strukturen.

Bei der Speicherung einer Web-Service Schnittstelle in UDDI sind folgende Eingaben erforderlich: URL auf eine WSDL-Schnittstelle und Kategorisierungsinformationen. Der URL muss dabei für die UDDI-Benutzer zugänglich sein, d.h. die WSDL-Schnittstelle kann von einem potentiellen Service-Requestor über HTTP abgerufen werden. Die Kategorisierungsinformationen bestehen aus null oder mehreren Tupeln aus TModelKey, Key und KeyDescription. Diese Tupel beinhalten die Informationen, die notwendig sind, um in UDDI ein KeyedReference-Objekt für die Speicherung einer Kategorie zu erzeugen. Neben diesen Web-Service-bezogenen Informationen sind noch folgende Daten wichtig: URL auf die Programmierschnittstelle des UDDI-Betreibers, Benutzer-ID und Passwort für das Benutzerkonto.

Die Ausgabe dieses Schrittes ist ein beim UDDI-Betreiber gespeichertes TModel-Objekt für die WSDL-Schnittstelle, das mit mehreren Kategorien klassifiziert ist, mindestens jedoch mit der Kategorie „WSDL Document“. Der Schlüssel zu diesem TModel-Objekt wird zu Kontrollzwecken lokal gespeichert.

Bei der Registrierung der Web-Service-Implementierung werden als Eingabedaten der URL auf das WSDL-Dokument für die Implementierung und Kategorisierungsinformationen in Form der oben beschriebenen Tupel benötigt. Daneben ist der Schlüssel auf das TModel-Objekt für die WSDL-Schnittstelle erforderlich. Weiter muss der Schlüssel zu einem BusinessEntity-Objekt, bei dem die Web-Service-Implementierung gespeichert wird, angegeben werden. Auch bei diesem Vorgang werden die allgemeinen Konfigurationsinformationen benötigt.

Die Ausgabe dieses Schrittes ist ein BusinessService-Objekt mit dem entsprechenden BindingTemplate-Objekt für den Web Service. Das BusinessService-Objekt enthält mehrere Kategorien in Form von KeyedReference-Objekten, mindestens jedoch die Kategorie „WSDL Document“. Das BusinessService-Objekt wird bei dem durch den Schlüssel angegebenen BusinessEntity-Objekt gespeichert. Das BindingTemplate-Objekt verfügt über eine InstanceDetails-Struktur, die auf das TModel-Objekt für die WSDL-Schnittstelle verweist.

Daneben müssen die in Abschnitt 5.1 Anforderungen berücksichtigt werden. Des Weiteren sollen die Empfehlungen für das Speichern von WSDL-Dokumenten in UDDI, die in [CER01] dargestellt sind, beachtet werden.

5.2.2 Entwurf

Für die beiden Veröffentlichungsvorgänge werden zwei Java-Klassen erstellt, die unabhängig voneinander eingesetzt werden können. Für das Veröffentlichen von WSDL-Schnittstellen wird die Klasse `InterfaceRegistrar` verwendet, für WSDL-Implementierungen die Klasse `ServiceRegistrar`. Beide Klassen besitzen jeweils eine `main()`-Methode zum Start der Verarbeitung. Für wiederkehrende Aufgaben in beiden Vorgängen, z.B. Lesen von Konfigurationsdaten oder Zuordnung von Kategorisierungsinformationen, werden Hilfsklassen zur Verfügung gestellt, vgl. Abbildung 5.1 auf der nächsten Seite. Für den UDDI-Zugriff stehen darüber hinaus die Klassen der Bibliotheken UDDI4J und WSTK zur Verfügung.

Die notwendigen Eingabedaten stammen aus unterschiedlichen Quellen: Parameter, die beim Start übergeben werden, und Konfigurationsdateien in verschiedenen Formaten. Diese Vielfalt an Datenquellen ist unter Gesichtspunkten der Integration, Redundanz und menschlicher Interaktion notwendig. Da der Veröffentlichungsprozess ohne menschliche Interaktion ablaufen soll, müssen die erforderlichen Daten vorab in Dateien zur Verfügung gestellt werden. Die Parameterübergabe ermöglicht sowohl die separate Ausführung der Veröffentlichung von Hand als auch die Einbindung in einen übergeordneten Prozess. Aus Redundanzgründen werden Einzeldaten möglichst nur an einer einzigen Stelle definiert. Dabei sollen Daten mit unterschiedlicher Lebensdauer nicht vermischt werden, mit der Folge dass Daten mit einer Gültigkeitsdauer von mehreren Veröffentlichungsvorgängen, z.B. UDDI-Parameter, getrennt von den Daten gehalten werden, die nur für einen einzelnen Veröffentlichungsprozess gültig sind. Abbildung 5.2 auf Seite 75 zeigt den Zugriff auf die unterschiedlichen Datenquellen am Beispiel des `InterfaceRegistrar`.

WSDL-Schnittstelle

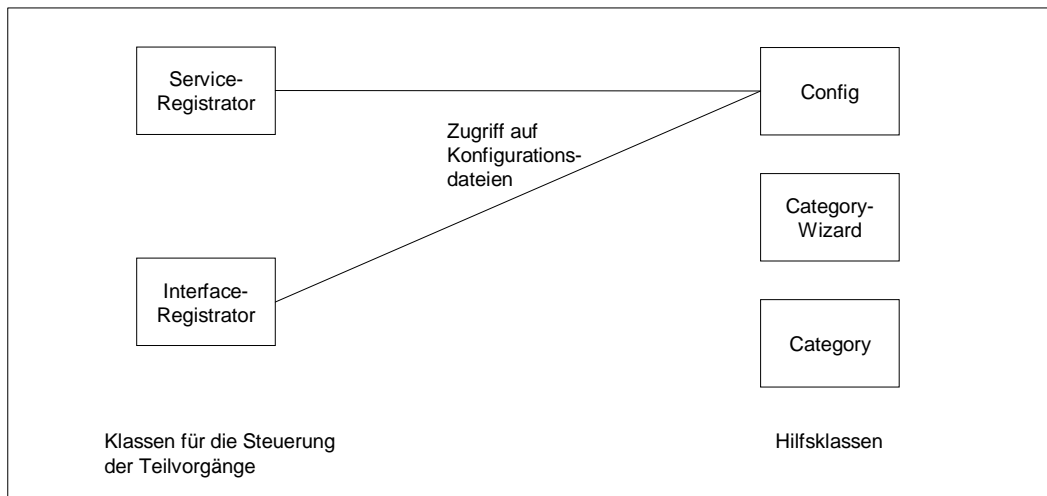


Abbildung 5.1: Klassen für die Steuerung des UDDI-Zugriffs

Die WSDL-Schnittstelle wird in UDDI als TModel-Objekt gespeichert. Für den einfachen Zugriff auf UDDI und dessen Datenstrukturen kann die Java-Bibliothek UDDI4J benutzt werden, das im WSTK enthalten ist. Zum Zeitpunkt dieser Arbeit liegt die Bibliothek in der Version 1.0 vor. Bei Nutzung von UDDI4J kann das TModel-Objekt lokal erzeugt und über eine Proxy-Klasse an UDDI übertragen werden.

Abbildung 5.3 auf Seite 76 zeigt ein Klassendiagramm für das Veröffentlichen der WSDL-Schnittstelle in UML-Notation.

Beschreibung der Klassen

UDDIProxy fungiert als Schnittstelle zu UDDI. TModel-Objekte können mit Hilfe der `<save_tmodel>`-Anweisung an UDDI übertragen werden. Für die Nutzung der Klasse zum Speichern von Objekten müssen die Sicherheitserweiterungen der Java-Umgebung aktiviert werden, damit für die Übertragung das HTTPS-Protokoll benutzt werden kann.

tModel Java-Repräsentation der UDDI-TModel-Struktur. Stellt die notwendigen `get/set`-Methoden zur Verfügung.

InterfaceRegistrator steuert den Vorgang der Veröffentlichung. Diese Klasse erzeugt ein TModel-Objekt, ermittelt die Kategorisierungsinformationen und übergibt diese zusammen mit dem TModel-Objekt an eine Hilfsklasse zur Erzeugung der Kategorien. Anschließend wird das TModel-Objekt mit Hilfe der UDDIProxy-Klasse an UDDI übertragen. Die Einstellungen für den UDDIProxy werden aus einer Konfigurationsdatei gelesen. Weiter ist die **ServiceRegistrator**-Klasse verantwortlich für das Speichern des beim Registrieren zurückgegebenen TModel-Schlüssels.

cfgParser Klasse aus den Web-Service-Erweiterungen zu MQWF, die benutzt wird, um textbasierte Konfigurationsdateien einzulesen. Der Aufbau der Konfigurationsdatei kann der Dokumentation zum WSPMTK entnommen werden.

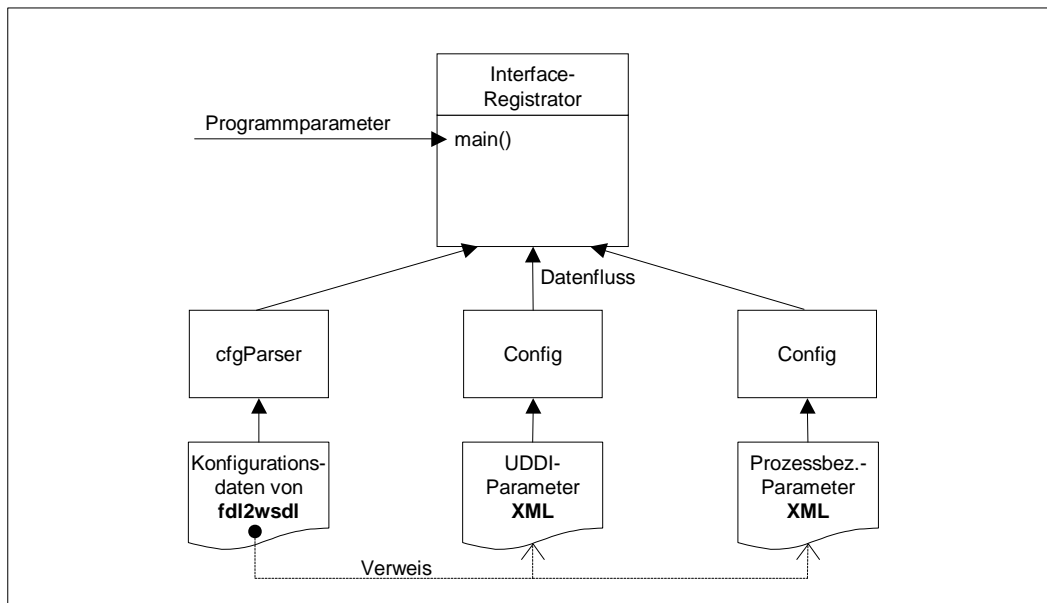


Abbildung 5.2: Datenquellen für den Veröffentlichungsprozess

Config Klasse zum Einlesen von Konfigurationsinformationen im XML-Format. Diese wird später genauer erläutert.

CategoryWizard Hilfsklasse, die das Registrieren von Kategorien bei UDDI-Objekten vereinfacht. Diese Klasse soll verschiedene Wege zur Speicherung von einzelnen oder mehreren Kategorien eröffnen, z.B. können mehrere **Category**-Objekte, die in einem `java.util.Vector`-Objekt enthalten sind, zur Kategorisierung eines `tModel` übergeben werden.

Category Hilfsklasse, die notwendige Informationen für eine Kategorie enthält. Diese Klasse wird vor allem beim Einlesen von Kategorisierungsinformationen aus einer Datei benutzt.

Schnittstellenbeschreibung

Die Schnittstelle zur Klasse **InterfaceRegistrator** besteht aus einer Methode, `publish()`. Dieser Methode werden zwei Parameter übergeben: der URL für die WSDL-Schnittstelle und der Name einer Konfigurationsdatei, aus der weitere Parameter gelesen werden können. Dazu gehören folgende UDDI-Parameter: URL für die UDDI-Schnittstelle, an die Objekte geschickt werden können, der Benutzername und das Kennwort für den UDDI-Zugriff. Weiter werden Kategorisierungsinformationen aus einer Datei gelesen und beim `TModel` registriert. Diese Kategorisierungsinformationen bestehen aus einem oder mehreren Tripeln aus `TModel`-Schlüssel, der die Spezifikation einer Taxonomie repräsentiert, ein Schlüssel, der einen einzelnen Taxonomie-Eintrag identifiziert und eine erläuternde Beschreibung des Taxonomie-Schlüssels. Bei erfolgreicher Speicherung der WSDL-Schnittstelle in UDDI wird der neu erzeugte `TModel`-Schlüssel zurückgegeben. Daneben wird dieser in der Konfigurationsdatei mit den prozessbezogenen Parametern gespeichert. Daneben besitzt

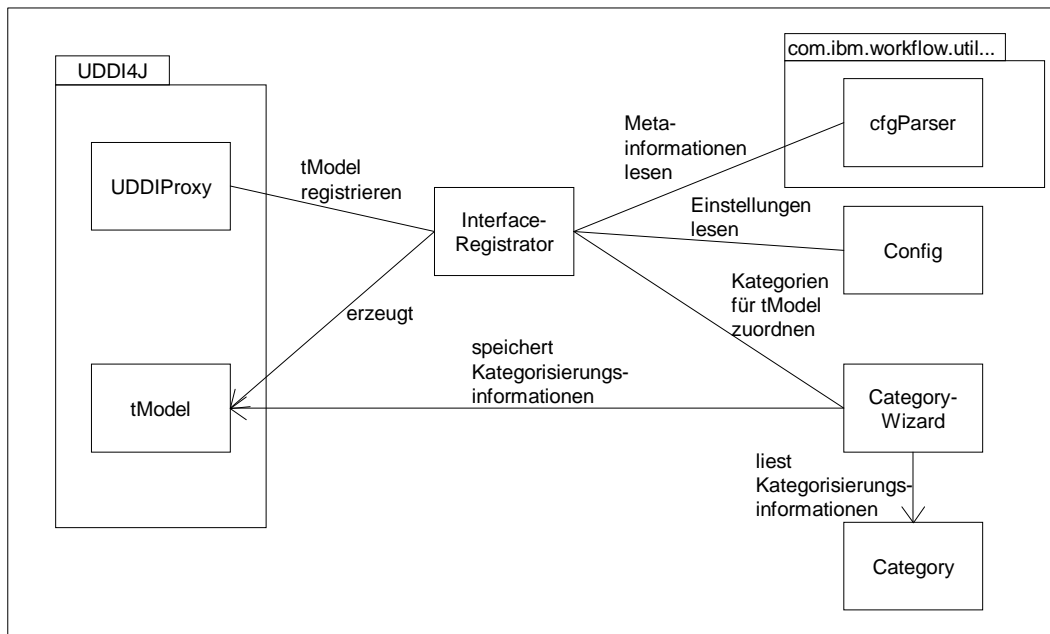


Abbildung 5.3: Klassendiagramm: Veröffentlichung der WSDL-Schnittstelle

die Klasse eine `main()`-Methode mit den gleichen Parametern, damit die Klasse autonom beim Start von Java ausgeführt werden kann.

Die Schnittstelle zur Hilfsklasse `CategoryWizard` besteht aus mehreren Methoden, mit denen Kategorisierungsinformationen in die `<tModelBag>` eines UDDI-Elementes eingefügt werden kann. Für jede in UDDI integrierte Taxonomie steht jeweils eine eigene Methode zur Verfügung, der jeweils nur der Schlüssel und die Schlüsselbeschreibung übergeben werden muss. Die weiteren Informationen werden automatisch erzeugt. Für andere Taxonomien steht eine einfach überladene Methode `addCategory()` zur Verfügung, mit der die Informationen aus einem `Category`-Objekt gelesen und beim UDDI-Element registriert werden. Dieser Methode kann auch ein mit `Category`-Objekten gefüllter `java.util.Vector` zum Registrieren übergeben werden. Bei allen Methoden muss das Objekt, bei dem die Kategorien registriert werden sollen, als Parameter übergeben werden. Außerdem steht noch eine Methode ohne Parameter zur Verfügung, die ein `CategoryBag`-Objekt zurückgibt, das die Kategorie „WSDL Document“ enthält.

Die Schnittstelle für das `Category`-Objekt besteht aus einer Anzahl von `get-` bzw. `set-`Methoden, mit denen die Werte von Datenfeldern gelesen oder gesetzt werden können. Das Objekt selbst dient als Speicher für die Daten einer einzelnen Kategorie und besitzt darüber hinaus keine weitere Funktionalität.

Ablauf

Der Ablauf des Veröffentlichungsvorganges für eine WSDL-Schnittstelle ist in Abbildung 5.4 auf der nächsten Seite als UML-Sequenzdiagramm dargestellt.

Zunächst wird ein `InterfaceRegistrator`-Objekt über die `main()`-Methode gestartet.

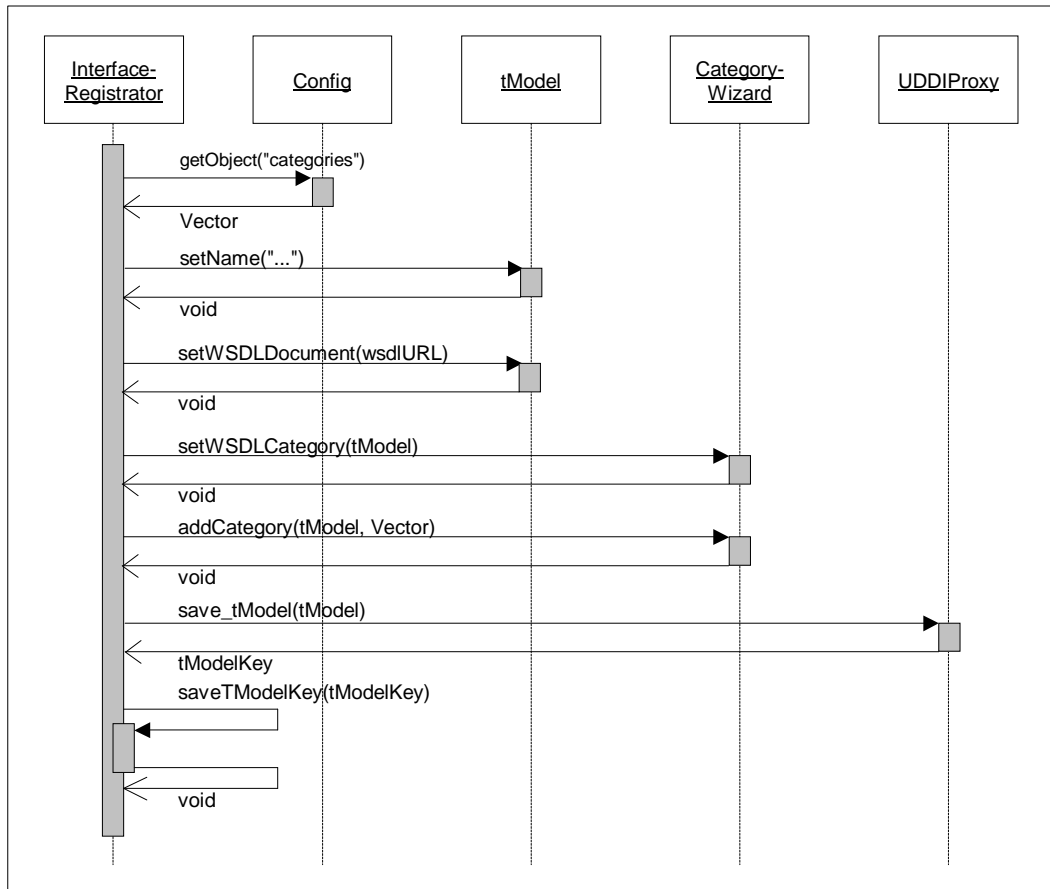


Abbildung 5.4: Veröffentlichung einer WSDL-Schnittstelle

Dabei werden zwei Parameter übergeben:

wsdlURL URL, über die auf die WSDL-Schnittstelle zugegriffen werden kann,
configFileName Name der Konfigurationsdatei, die auch das Software-Werkzeug `fdl2wsdl` benutzt. Diese Konfigurationsdatei benötigt zwei Einträge in der Sektion [UDDI]: ein Eintrag `uddiConfig`, der auf eine Datei mit UDDI-Konfigurationsdaten verweist und einen Eintrag `interfaceConfig`, der auf eine Datei zeigt, in der die Kategorisierungsinformationen für die WSDL-Schnittstelle enthalten sind.

Im nächsten Schritt werden die Namen der beiden Konfigurationsdateien für UDDI und die WSDL-Schnittstelle ermittelt. Dazu wird mit Hilfe eines `cfgParser`-Objektes auf die als Parameter übergebene Konfigurationsdatei zugegriffen. Die UDDI-Konfigurationsdaten werden benötigt, um die `UDDIProxy`-Klasse zu erzeugen und Daten in UDDI einzutragen. In der zweiten Datei sind die Kategorisierungsinformationen gespeichert.

Nun kann eine `tModel`-Klasse erzeugt und der URL, der auf die WSDL-Schnittstelle zeigt, gespeichert werden. Um den Namen des TModels gemäß den Empfehlungen

für die Nutzung von WSDL in UDDI, vgl. [CER01], richtig zu setzen, muss dieser aus dem WSDL-Dokument ermittelt werden, z.B. über DOM, vgl. [ABK⁺00]. Die Kategorisierungsinformationen werden aus der Konfigurationsdatei mit Hilfe eines `Config`-Objektes gelesen und dem `CategoryWizard` zusammen mit dem `tModel` zur Speicherung übergeben. Nach diesem Schritt kann das `tModel` an UDDI übertragen werden.

Die `UDDIProxy`-Klasse stellt die notwendigen Methoden zum Registrieren eines `TModel`-Objektes in UDDI zur Verfügung. Die notwendigen Informationen, wie URL für den UDDI-Zugriff, Benutzername, Kennwort, werden aus der UDDI-Konfigurationsdatei, ebenfalls mit Hilfe eines `Config`-Objektes gelesen. Der Zugriff auf UDDI beim Veröffentlichen erfolgt über HTTPS, also eine gesicherte HTTP-Verbindung. Bei erfolgreicher Speicherung des `TModel`-Objektes in UDDI wird der neu generierte `tModelKey`, also der UDDI-Schlüssel für das `TModel`-Objekt zurückgeliefert. Dieser Schlüssel wird für die Veröffentlichung der WSDL-Implementierung benötigt und wird deshalb in der Konfigurationsdatei für die WSDL-Schnittstelle gespeichert. Beim Speichern erfolgt der Zugriff nicht über das `Config`-Objekt, denn dieses ist für den lesenden Zugriff konzipiert, s.u. Für das Speichern müssen sowohl die Parameternamen als auch die Struktur der Konfigurationsinformationen bekannt sein. Nach der erfolgreichen Speicherung des `tModelKey` ist die Veröffentlichung der Schnittstelle abgeschlossen.

WSDL-Implementierung

Bei der Veröffentlichung der WSDL-Implementierung muss ein `BusinessService`-Objekt mit einem enthaltenen `BindingTemplate`-Objekt erzeugt und in UDDI registriert werden. Dieser Vorgang erfordert im Gegensatz zur Registrierung der Schnittstelle eine Analyse des WSDL-Dokumentes. Für die Generierung eines `BusinessService`-Objektes stellt das Web Services Toolkit (WSTK) von IBM entsprechende Klassen zur Verfügung. Da das WSTK eine abstrakte Sicht auf UDDI anbietet, müssen andere Verfahren gewählt werden, um z.B. Kategorien zuzuordnen oder ein Objekt an UDDI zu übertragen. Abbildung 5.5 auf der nächsten Seite zeigt ein Klassendiagramm für das Veröffentlichen einer WSDL-Implementierung in UML-Notation.

Beschreibung der Klassen

ServiceRegistrar steuert den Ablauf des Registrierungs Vorganges für die Web-Service-Implementierung. Zunächst wird aus den Konfigurationsinformationen ein `BusinessService`-Objekt erzeugt und in einem weiteren Schritt an UDDI übertragen. Dafür werden die Klassen aus dem Web Service Toolkit (WSTK) verwendet.

ServiceRegistryProxy Schnittstelle des WSTK zu UDDI. Verfügt über die notwendigen Methoden zum Veröffentlichen von Web Services.

SOAPServiceDefinition kapselt die UDDI-Objekte zur Aufnahme eines Web Services mit SOAP Transportprotokoll. Zum Erzeugen des Objektes wird eine WSDL-Beschreibung, ein `CategoryList`-Objekt (s.u.), der SOAP-Deployment-Descriptor und die SOAP-Adresse benötigt.

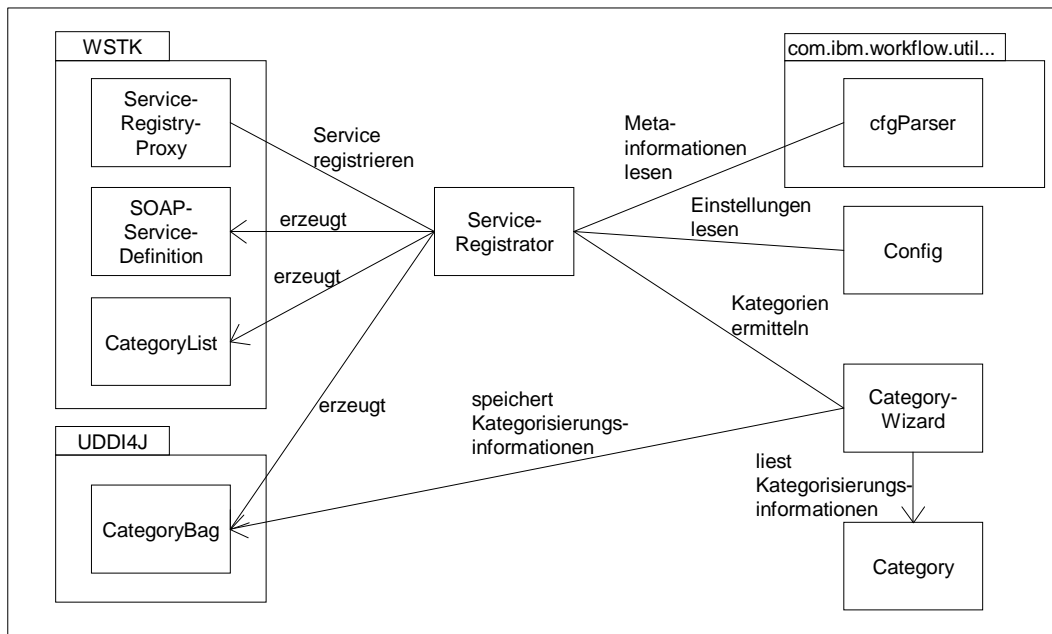


Abbildung 5.5: Klassendiagramm: Veröffentlichung der WSDL-Implementierung

CategoryList repräsentiert die Kategorisierungsinformationen im WSTK. Dieses Objekt kann aus einer UDDI **categoryBag** erzeugt werden.

categoryBag repräsentiert die UDDI Datenstruktur, die Kategorisierungsinformationen in Form von **keyedReference**-Objekten enthält.

cfgParser Klasse aus den Web-Service-Erweiterungen zu MQWF, die benutzt wird, um textbasierte Konfigurationsdateien einzulesen. Der Aufbau der Konfigurationsdatei kann der Dokumentation zum WSPMTK entnommen werden.

Config Klasse zum Einlesen von Konfigurationsinformationen im XML-Format. Diese wird später genauer erläutert.

CategoryWizard Hilfsklasse, die das Registrieren von Kategorien bei UDDI-Objekten vereinfacht. Diese Klasse soll verschiedene Wege zur Speicherung von einzelnen oder mehreren Kategorien eröffnen, z.B. können mehrere **Category**-Objekte, die in einem **java.util.Vector**-Objekt enthalten sind, zur Erzeugung einer **categoryBag** übergeben werden.

Category Hilfsklasse, die notwendige Informationen für eine Kategorie enthält. Diese Klasse wird vor allem beim Einlesen von Kategorisierungsinformationen aus einer Datei benutzt.

Schnittstellenbeschreibung

Das **ServiceRegistrator**-Objekt besitzt eine Schnittstelle in Form einer **publish()**-Methode, der drei Parameter übergeben werden müssen: der URL der WSDL-Implementierung, der SOAP-DeploymentDescriptor für den registrierten Web Service und der Name einer Konfigurationsdatei. Aus dieser Konfigurationsdatei werden

die Kategorisierungsinformationen ermittelt, die beim BusinessService-Objekt gespeichert werden sollen. Als Ausgabe registriert die Klasse die WSDL-Implementierung in UDDI. Der Schlüssel dieses UDDI-Objektes wird in der Datei mit den prozessbezogenen Informationen gespeichert. Daneben existiert noch eine `main()`-Methode mit den gleichen Parametern für den autonomen Aufruf beim Start der Java-Umgebung.

Ablauf

In Abbildung 5.6 ist der Veröffentlichungsvorgang für die WSDL-Implementierung als UML-Sequenzdiagramm dargestellt.

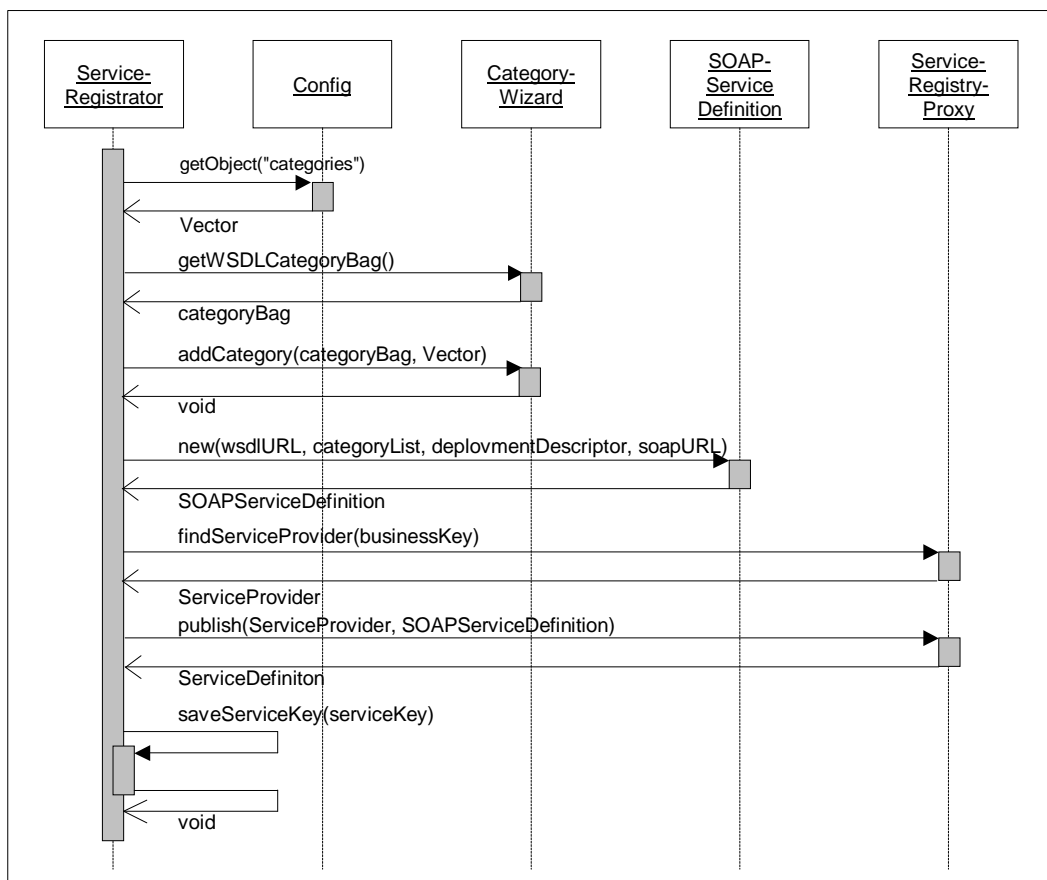


Abbildung 5.6: Veröffentlichung einer WSDL-Implementierung

Der Vorgang wird gestartet, in dem die `main()`-Methode der `ServiceRegistrator`-Klasse aufgerufen wird. Dabei werden drei Parameter übergeben:

serviceURL URL, über die auf die WSDL-Implementierung zugegriffen werden kann,

deploymentDescriptor Datei, die alle notwendigen Informationen zur Registrierung eines SOAP-Service beim SOAP-Server enthält,

configFileName Name der Konfigurationsdatei von `fd12wsdl`. Zwei Einträge in der Sektion `[UDDI]` dieser Konfigurationsdatei sind für den weiteren Ablauf

wichtig: ein Eintrag `interfaceConfig`, der auf die Datei verweist, in der der `tModelKey` der WSDL-Schnittstelle gespeichert ist und ein Eintrag `serviceConfig`, die Kategorisierungsinformationen für die WSDL-Implementierung enthält.

Im ersten Schritt werden die Einträge aus der Konfigurationsdatei gelesen, anhand derer die weiteren Informationsquellen erschlossen werden können. Der Zugriff erfolgt über die `cfgParser`-Klasse.

Anschließend wird die Kategorisierung des `BusinessService`-Objektes in UDDI vorbereitet. Dazu werden die Kategorisierungsinformationen aus der entsprechenden Konfigurationsdatei gelesen und in einem `categoryBag`-Objekt gespeichert. Aus dieser `categoryBag` wird ein `categoryList`-Objekt erzeugt, das die Kategorisierungsinformationen im WSTK repräsentiert.

Aus dem WSDL-Dokument, dem `DeploymentDescriptor` und der `categoryList` wird ein `SOAPServiceDefinition`-Objekt erzeugt, das mit Hilfe der `ServiceRegistryProxy` an UDDI übertragen wird. Vorher muss jedoch eine Referenz auf das `BusinessEntity`-Objekt, bei dem die Web-Service-Implementierung gespeichert wird, ermittelt werden. Dazu wird der aus der Konfigurationsdatei ermittelte `businessKey` an die `findServiceProvider`-Anfrage des `ServiceRegistryProxy` übergeben. Das Ergebnis dieser Anfrage ist ein `ServiceProvider`-Objekt, das zusammen mit der `SOAPServiceDefinition` an die `publish()`-Methode zur Veröffentlichung übergeben wird.

Der UDDI-Schlüssel zu dieser Web-Service-Implementierung wird in einem letzten Schritt ermittelt und in der Konfigurationsdatei für die Service-bezogenen Informationen gespeichert.

Beim Veröffentlichen von WSDL-Implementierungen wird nicht wie bei den Schnittstellen auf UDDI4J zurückgegriffen, sondern auf die Klassen des WSTK. Der Grund dafür ist, dass im WSTK Methoden zur Erzeugung der entsprechenden UDDI-Strukturen aus einem WSDL-Dokument und dem `DeploymentDescriptor` enthalten sind, nicht jedoch in UDDI4J.

5.3 Zugriff auf Parameter in Dateien

Der größte Teil der Parameter für den Veröffentlichungsvorgang wird aus Dateien gelesen. Für diesen Zweck benutzt das Software-Werkzeug `fd12wsdl` die Klasse `cfgParser`. Die Möglichkeiten zur Speicherung von strukturierten Informationen, wie sie zur Veröffentlichung von Web Services in UDDI notwendig sind, sind in dieser Klasse leider zu eingeschränkt. Um Kollektionen und Objekte aus Dateien auszulesen muss deshalb das Dateiformat geändert und eine neue Zugriffsmöglichkeit implementiert werden.

Bislang werden die Konfigurationsdaten in folgendem Format gespeichert:

```
[Sektion1]
name1 = wert1
name2 = wert2
...
[Sektion2]
name1 = wert1
name2 = wert2
...
```

Listing 5.1: Struktur der existierenden Konfigurationsdatei

Bei der Speicherung von Kategorisierungsinformationen ist jedoch die in Listing 5.2 dargestellte Struktur notwendig.

```
Schnittstellen-Kategorien
Kategorie
  TModelKey
  Schluesselwert
  Schluesselbeschreibung
Kategorie
  TModelKey
  Schluesselwert
  Schluesselbeschreibung
...
Implementierungs-Kategorien
Kategorie
  TModelKey
  Schluesselwert
  Schluesselbeschreibung
...
```

Listing 5.2: Struktur der Kategorisierungsinformationen

In Listing 5.1 wird deutlich, dass für jede Sektion ein unterschiedlichen Bezeichner benutzt werden muss. Außerdem besitzt die dargestellte Konfigurationsdatei einen zweidimensionalen Aufbau. Somit muss jeder Parameterwert durch die Angabe von Sektionsname und Parametername identifiziert werden. Dazu steht die Methode `getParameter(String section,String key)` zur Verfügung.

Die zwei Dimensionen reichen jedoch nicht aus, um die Kategorien für die Schnittstelle und die Implementierung in derselben Datei zu speichern. Zwar könnte der Sektionsname durch bestimmte Namenskonventionen auch für eine Unterscheidung herangezogen werden. Um jedoch eine beliebige Anzahl von Kategorien mit Hilfe einer solchen Namenskonvention über die `getParameter()`-Methode einzulesen, muss über alle Sektionsnamen der Datei iteriert und die Zuordnung durch Zeichenkettenvergleich ermittelt werden.

5.3.1 Spezifikation

Für die Neuentwicklung der Konfiguration gelten folgende Anforderungen:

- Für die Syntax der Konfigurationsdatei wird XML benutzt.
- Die Schnittstelle der `cfgParser`-Klasse bleibt mit gleicher Semantik erhalten.
- Die Definition von Name-Wert-Paaren soll auf jeder Hierarchieebene der Konfigurationsdatei möglich sein.
- Mehr als zwei Hierarchieebenen müssen abgebildet werden können.
- Selbstdefinierte Objekte müssen eingelesen werden können.
- Kollektionen von Objekten müssen eingelesen werden können.
- Nur lesender Zugriff auf die Konfigurationsdatei.

Für die Darstellung der vorhandenen Konfigurationsdaten in der neuen XML-Syntax ist in Listing 5.3 dargestellt. Dabei werden Name-Wert-Paare durch den Elementnamen identifiziert.

```
<Sektion1>
  <name1>wert1</name1>
  <name2>wert2</name2>
  ...
</Sektion1>
<Sektion2>
  ...
</Sektion2>
```

Listing 5.3: Darstellung einer zweistufigen Hierarchie in XML-Notation

Schnittstellenbeschreibung

Der lesende Zugriff auf die Konfigurationsdatei im „alten“ Format wird durch die `cfgParser`-Klasse realisiert. Die Schnittstelle umfasst folgende Methoden:

`java.util.Hashtable` `getTopics()` Gibt alle Werte der Konfigurationsdatei in einem `Hashtable`-Objekt zurück. Diese Hash-Tabelle enthält weitere `Hashtable`-Objekte, in denen schließlich die Name-Wert-Paare aus der Konfigurationsdatei enthalten sind.

`java.util.Hashtable` `getParameters(String key)` Gibt alle Name-Wert-Paare einer Sektion in einem `Hashtable`-Objekt zurück. Wird keine Sektion mit diesem Namen gefunden, wird `null` zurückgegeben.

`String` `getParameter(String section, Stringkey)` Gibt einen einzelnen Wert zurück der durch die Kombination von Sektionsname Parametername identifiziert wird. Wird kein entsprechender Wert gefunden, wird `null` zurückgegeben.

`String getParameter(String section, String key, String defaultValue)` Gibt einen einzelnen Wert zurück der durch die Kombination von Sektionsname Parametername identifiziert wird. Wird kein entsprechender Parameterwert gefunden, wird der als Parameter übergebene Wert zurückgegeben.

Die erweiterte Schnittstelle benötigt neben den vorhandenen auch die nachfolgend dargestellten Methoden, um den Zugriff auf Vektoren bzw. Objekte zu ermöglichen.

`Object getObject(String key)` Gibt das durch den Parameter identifizierte Objekt zurück. Das zurückgelieferte Objekt ist auf der obersten Hierarchieebene der Konfigurationsdatei gespeichert. Falls kein Objekt mit dem entsprechenden Schlüssel gefunden wird, wird `null` zurückgegeben.

`Object getObject(String section, String key)` Gibt das durch die Kombination der Parameter identifizierte Objekt zurück. Wird kein entsprechendes Objekt gefunden, wird `null` zurück gegeben.

`Object getObject(String section, String key, Object defaultObject)` Gibt das durch die Kombination der Parameter identifizierte Objekt zurück. Wird kein entsprechendes Objekt gefunden, wird der als dritter Parameter übergebene Wert zurückgegeben.

Hinsichtlich der einzulesenden Objekte müssen folgende Einschränkungen berücksichtigt werden. Für das Einlesen kommen nur solche Objekte in Frage, für die ein Standardkonstruktor ohne Parameter zur Verfügung steht. Außerdem muss der Zustand der Objekte über `set`-Methoden verändert werden können. Voraussetzung für das Einlesen ist auch, dass die in der Konfiguration angegebene Klassendefinition im `CLASSPATH` der rufenden Java-Anwendung liegen.

Kollektionen können auf ein `java.util.Vector`-Objekt beschränkt werden. Ein solcher Vektor ist in Bezug auf die zu speichernde Anzahl als auch auf die Art an Objekten flexibel.

Das Einlesen von Kollektionen und Objekten bezieht sich in erster Linie auf die Kategorisierungsinformationen. Dabei muss eine wechselnde Anzahl gleichartiger Konstrukte, z.B. Kategorien in einer Gesamtheit eingelesen werden. Dazu kann ein `java.util.Vector` erzeugt werden, der eine entsprechende Anzahl von `Category`-Objekten enthält.

Format der Konfigurationsinformationen

Für Konfigurationsdateien werden folgende Schlüsselwörter bzw. Schlüsselemente benutzt: `<Configuration>` bezeichnet das Top-Level-Element, die umschließende Klammer um alle Konfigurationsdaten, `class` ist eine Attributsbezeichnung, die die korrespondierende Klassenbezeichnung eines einzulesenden Objektes kennzeichnet, `<Vector>` kennzeichnet ein einzulesendes `java.util.Vector`-Objekt, `keyName` wird verwendet, um bei `<Vector>`-Elementen einen Zugriffsschlüssel zu definieren. Der Aufbau wird anhand der folgenden Beispiele erläutert:

Das `<Configuration>`-Element muss das erste Element in der Datei sein. Alle Informationen werden innerhalb dieses Elementes gespeichert.

```

<Configuration>
  ...
</Configuration>

```

Listing 5.4: Top-Level-Element der Konfigurationsinformationen

Name Wert Paare werden als Text-Elemente gespeichert. Beim Einlesen werden alle Whitespace-Zeichen entfernt. Über den Elementnamen kann auf den Parameterwert zugegriffen werden. Name-Wert-Paare können an beliebiger Stelle innerhalb der `<Configuration>` auftreten. Innerhalb von Objekten werden diese jedoch als Werte von Datenfeldern interpretiert, s.u.

```

...
<name1>Wert</name1>
...

```

Listing 5.5: Name-Wert-Paare im XML-Format

Die aus der vorhandenen Konfiguration bekannten Sektionen werden als Elemente dargestellt, die keine Attribute und nur Elemente als Inhalt besitzen. Sektionen können an beliebiger Stelle innerhalb der `<Configuration>` auftreten, nicht jedoch innerhalb von Objekten.

```

...
<sektion1>
  <name2>wert2</name2>
  ...
</sektion1>
...

```

Listing 5.6: Sektionen im XML-Format

Um auf einen Vektor zuzugreifen muss dieser mit einem Namen qualifiziert werden. Vektoren können an beliebiger Stelle innerhalb der `<Configuration>` auftreten, nicht jedoch innerhalb von Objekten. Sektionen, innerhalb eines Vektors werden mit ihrem gesamten Inhalt als `java.util.Hashtable`, Name-Wert-Paare als `java.lang.String` und Objekte bzw. Vektoren als Java-Objekte gespeichert.

```

...
<Vector keyName="name3">
  ...
</Vector>
...

```

Listing 5.7: Darstellung eines Vektor-Objektes

Objekte werden durch das Attribut `class` gekennzeichnet. Voraussetzung für das Erzeugen von Objekten ist, dass die Klassendefinition im Attribut über den `CLASSPATH` eingebunden werden kann. Außerdem gelten für die Darstellung von Objekten folgende Restriktionen: die Klasse benötigt einen Standardkonstruktor ohne Parameter. Außerdem muss für jedes Name-Wert-Paar innerhalb des Objektes eine `set`-Methode zur Verfügung stehen.

```
<name4 class="com.ibm.uddi.Category">
  <key>123</key>
  <keyName>Kreditkartenpruefung</keyName>
  <tModelKey>456789</tModelKey>
</name4>
```

Listing 5.8: Darstellung von Objekten

5.3.2 Entwurf

Das Klassendiagramm für die Konfiguration ist in Abbildung 5.7 dargestellt.

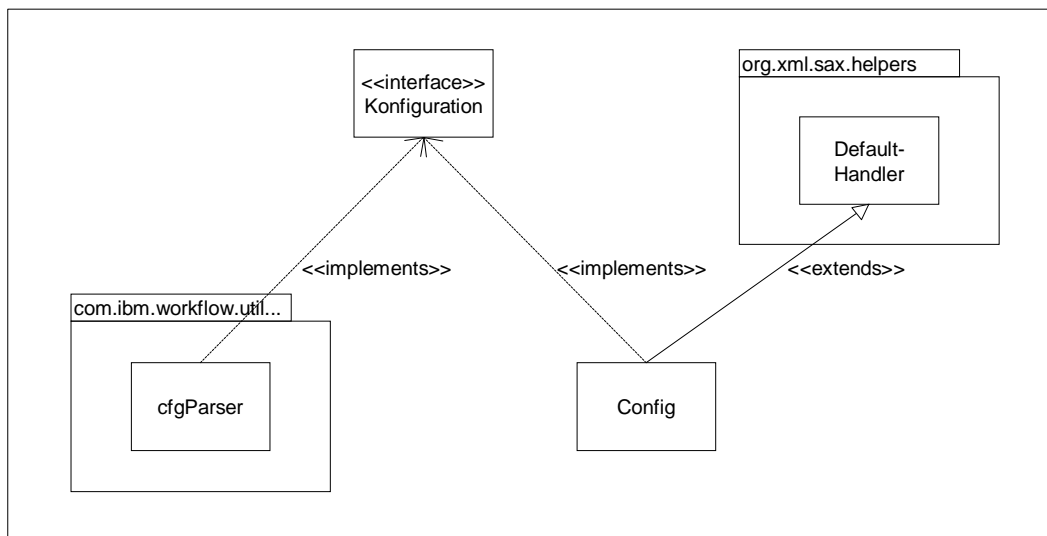


Abbildung 5.7: Klassendiagramm: Klasse zum Lesen der Konfigurationsdatei

Die gemeinsame Schnittstelle gewährleistet die Kompatibilität zwischen der bestehenden und der neuen Variante der Konfigurations-Klasse. Dabei können jedoch maximal zwei Hierarchieebenen und ausschließlich Name-Wert-Paare abgefragt werden. Für die erweiterten Möglichkeiten muss die Schnittstelle um zusätzliche Methoden erweitert werden. Die neuen Methoden liefern jeweils einen Rückgabewert vom Typ `java.lang.Object`, der von der Anwendung auf den gewünschten Typ angepasst werden muss.

Zur Analyse der Konfigurationsdatei wird ein XML-Parser benutzt, der dem Simple API for XML (SAX) -Standard entspricht, vgl. auch [ABK⁺00]. Dies ermöglicht den Einsatz beliebiger SAX-Parser von unterschiedlichen Herstellern. Eingesetzt wird ein SAX-Parser aus dem Java API for XML Processing (JAXP), eine Java-Bibliothek, die einen Standard für die Nutzung von XML unter Java definiert.

Da der erwähnte SAX-Parser ausschließlich die XML-Syntax erkennt, muss dieser für die Erkennung selbst definierter Strukturen erweitert werden. Hierzu müssen die Methoden des `DefaultHandler` neu implementiert werden. Das bedeutet, dass für die selbst definierte Grammatik, die in der Konfigurationsdatei angewendet wird, der SAX-Parser erweitert werden muss. Die für den Aufbau der Datei zu Grunde liegende, kontextfreie Grammatik kann mit Hilfe einer Stack-Maschine implementiert werden.

5.4 Steuerung des Gesamtvorganges

5.4.1 Spezifikation

Zum Veröffentlichen eines Workflow-Prozesses als Web Service in UDDI soll auf Basis der Anforderungen aus Abschnitt 5.1 eine Lösung entwickelt werden, mit der alle notwendigen Schritte zusammenhängend ausgeführt werden können. Diese Schritte sind

1. Erzeugen der Proxy-Klassen für die Web-Service-Schnittstelle aus der Prozessdefinition.
2. Installation der Klassen und Registrierung in der SOAP-Umgebung.
3. Generierung der Web-Service-Beschreibung, getrennt nach WSDL-Schnittstelle und WSDL-Implementierung.
4. Web-Zugriff für WSDL-Dokumente ermöglichen.
5. Veröffentlichen der WSDL-Schnittstelle in UDDI.
6. Veröffentlichen der WSDL-Implementierung in UDDI.

Eingabe

Die Eingaben für den Veröffentlichungsprozess bestehen aus den Dateien, die von den eingesetzten Werkzeugen benötigt werden:

- Datei mit der Prozessdefinition im FDL-Format. Diese Datei wird vom Werkzeug `fdl2wsdl` benötigt.
- Datei mit Konfigurationsinformationen für `fdl2wsdl`.
- Datei mit Einstellungen für UDDI im XML-Konfigurationsformat.

- Datei mit Kategorisierungsinformationen für die WSDL-Schnittstelle im XML-Konfigurationsformat.
- Datei mit Kategorisierungsinformationen für die WSDL-Implementierung im XML-Konfigurationsformat.

Daneben wird als Eingabe noch ein Verzeichnis benötigt, in dem Dokumente für den Web-Zugriff gespeichert werden.

Ausgabe

Am Ende des Veröffentlichungsprozesses liegen folgende Resultate vor:

- Die Proxy-Klassen für den Web Service wurden generiert und der Web Service ist in der SOAP-Umgebung registriert.
- Eine WSDL-Beschreibung für den Web Service steht für den Web-Zugriff zur Verfügung. Diese liegt getrennt nach WSDL-Schnittstelle und WSDL-Implementierung vor.
- In UDDI ist ein TModel-Objekt registriert, das die WSDL-Schnittstelle repräsentiert. Bei diesem TModel-Objekt sind die in der Konfigurationsdatei angegebenen Kategorien registriert und zusätzlich die Kategorie „WSDL Document“.
- In UDDI ist ein BusinessService-Objekt für die WSDL-Implementierung registriert. Die Registrierung erfolgt bei dem in der Konfigurationsdatei angegebenen BusinessEntity-Objekt. Beim BusinessService-Objekt sind die in der Konfigurationsdatei angegebenen Kategorien registriert.

Werkzeuge für Einzelschritte

Zur Realisierung von Einzelschritten sollen folgende Werkzeuge bzw. Programme eingesetzt werden:

- das Werkzeug `fdl2wsdl` des WSPMTK zur Generierung der Web-Service-Schnittstelle und der WSDL-Beschreibungen,
- die Java-Klasse `InterfaceRegistrar` zur Veröffentlichung der WSDL-Schnittstelle,
- die Java-Klasse `ServiceRegistrar` zur Veröffentlichung der WSDL-Implementierung.

5.4.2 Entwurf

Für die Realisierung des Gesamtvorgangs zum Veröffentlichen wird das Java-Werkzeug ANT der APACHE Group eingesetzt. Mit ANT können die Einzelschritte des Vorgangs

wie mit einer Skriptsprache zu einem Gesamtvorgang zusammengefasst werden. Zur Definition eines Vorganges wird eine XML-Datei mit ANT-Befehlen für die Einzelschritte erstellt und beim Aufruf von ANT als Parameter übergeben.

Der gesamte Veröffentlichungsvorgang wird durch zwei Build-Dateien realisiert, von denen eine zur Ausführungszeit generiert wird. Zur Generierung der Build-Datei wird das Werkzeug `fdl2wsdl` so modifiziert, dass statt einer MS-DOS-Stapelverarbeitungsdatei für die Registrierung der Proxy-Klassen eine erweiterte ANT-Build-Datei erstellt wird. Das Prinzip dieses zweistufigen Vorganges ist in Abbildung 5.8 dargestellt.

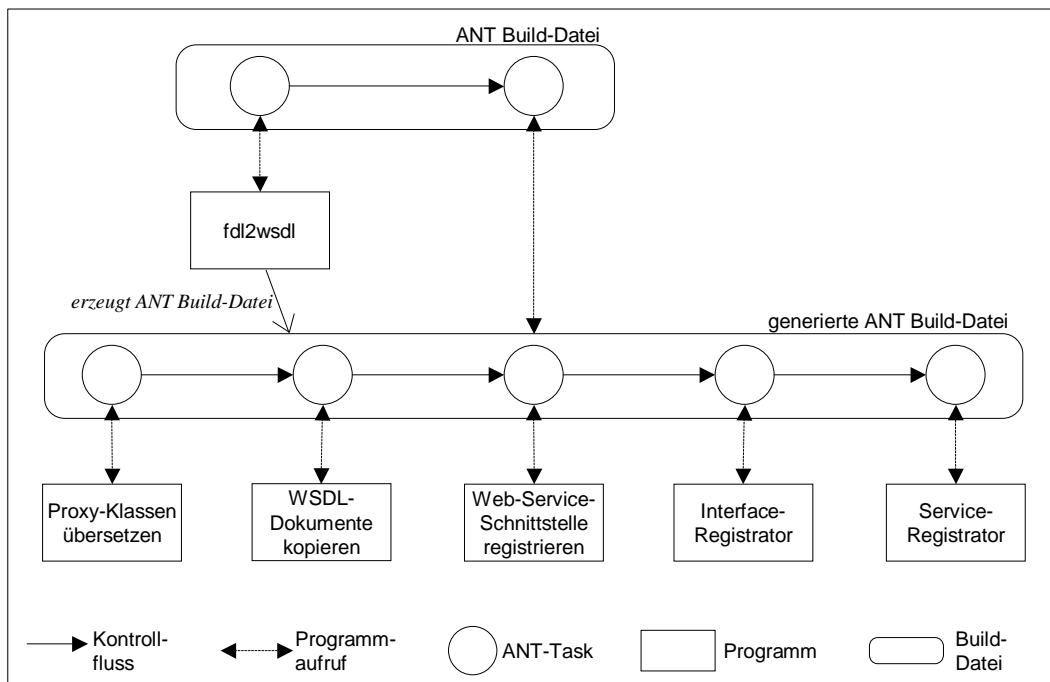


Abbildung 5.8: Zweistufiger Veröffentlichungsprozess

Die erste Build-Datei umfasst nur zwei sog. ANT-Tasks, einer für den Start von `fdl2wsdl` und einer für den Aufruf der zweiten Build-Datei. Bei diesem zweiten Aufruf von ANT bleibt die gesamte Umgebung mit allen Variablen erhalten.

Im der zweiten Build-Datei befinden sich die Tasks für die Übersetzung und Installation der Proxy-Klassen, der Kopierbefehl für die WSDL-Dokumente und die Java-Klassen zum Veröffentlichen der WSDL-Beschreibung in UDDI.

Ablauf

Der Ablauf des Veröffentlichungsprozesses ist in den Abbildungen 5.9 auf der nächsten Seite und 5.10 auf Seite 91 als UML-Sequenzdiagramme dargestellt.

Als erster ANT-Task wird das Werkzeug `fdl2wsdl` aufgerufen. Dabei wird aus der FDL-Datei mit der Prozessdefinition die WSDL-Beschreibung, getrennt nach Schnittstelle und Implementierung, der Quellcode für die Java-Proxy-Klassen, ein Deploy-

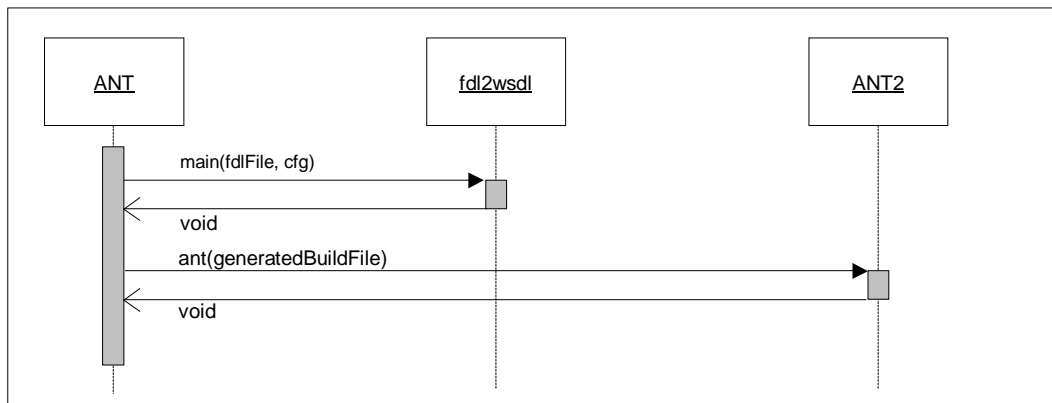


Abbildung 5.9: Erster Teil des Veröffentlichungsprozesses

mentDescriptor für die SOAP-Umgebung und eine ANT-Build-Datei für die weiteren Schritte erzeugt. Der zweite Task ruft ANT mit der generierten Build-Datei auf.

Neben den Tasks werden in der ersten Build-Datei auch die Variablen für den Gesamtprozess definiert:

fd1File Name der Datei für die Workflow-Prozessdefinition,

cfg Dateiname für die `fd12wsdl`-Konfigurationsdatei,

generated Name der generierten Build-Datei

soapAdress Adresse des SOAP-Umgebung

wSDLLocation Verzeichnis, in das die WSDL-Beschreibungen für den Web-Zugriff kopiert werden,

httpsUrlStreamHandler Name der Java-Handler-Klasse für das HTTPS-Protokoll.

In der zweiten Build-Datei wird zunächst der Java-Quellcode für die Proxy-Klassen übersetzt. Dabei wird von `fd12wsdl` für jede Quelldatei ein separater Eintrag als Parameter für den Übersetzungs-Task erzeugt. Nach der Übersetzung der Klassen wird die Web-Service-Schnittstelle bei der SOAP-Umgebung registriert. Dafür wird der von `wSDL2fdl` erzeugte `DeploymentDescriptor`, eine XML-Datei mit Informationen über den Web Service, an die `ServiceManagerClient`-Klasse übergeben. Im nächsten Schritt werden die WSDL-Beschreibungen in ein Verzeichnis mit HTTP-Zugriffsmöglichkeit kopiert. Dafür stehen in ANT Kopier-Tasks zur Verfügung. Anschließend wird zunächst die WSDL-Schnittstelle in UDDI veröffentlicht und darauf die WSDL-Implementierung.

Modifikation von wSDL2fdl

Durch die Modifikation von `wSDL2fdl` wird erreicht, dass eine ANT-Build-Datei für die nachfolgenden Veröffentlichungsschritte erzeugt wird. Dabei werden alle variablen Parameter, z.B. Klassen- oder Dateinamen direkt als Aufrufparameter in den ANT-Tasks eingesetzt. Der Vorteil dieser Vorgehensweise besteht darin, dass diese Namen nicht von späteren Schritten neu berechnet werden müssen.

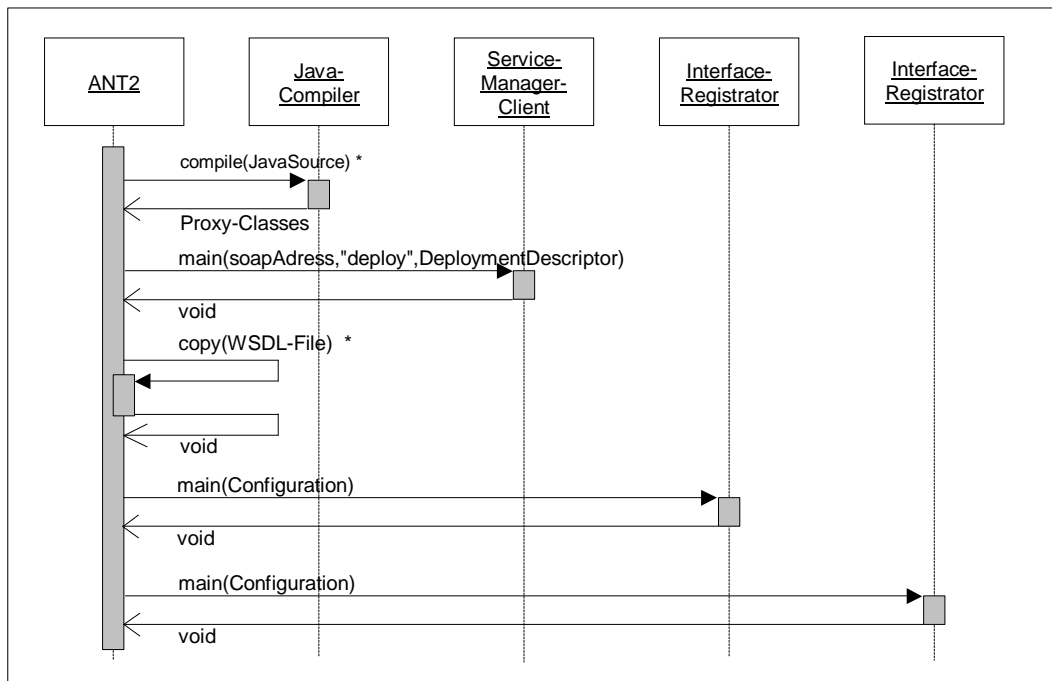


Abbildung 5.10: Zweiter Teil des Veröffentlichungsprozesses

Intern verwendet `wsdl2fdl` XSLT-Stylesheets zur Generierung der MS-DOS-Stapelverarbeitungsdatei. Dieses Stylesheet muss so abgeändert werden, dass statt der Kommandozeilenbefehle für MS-DOS die entsprechenden ANT-Aufrufe ausgegeben werden. Darüber hinaus müssen in der Ausgabedatei die ANT-Kommandos für das Kopieren der WSDL-Dateien und die Aufrufe der Java-Klassen zur Veröffentlichung zusätzlich erzeugt werden.

6 Lösung für das Einbinden

Im Rahmen dieser Arbeit soll eine Lösung für das dynamische Einbinden von in UDDI veröffentlichten Web Services erarbeitet werden. Dabei sind folgende Teilaspekte zu beachten:

- Erstellung von Klassen für das Auffinden von Web Services in UDDI,
- Steuerung des Einbindungsvorgangs,
- Integration vorhandener Werkzeuge.

In den folgenden Abschnitten wird ein Lösungsansatz für das dynamische Einbinden vorgestellt.

6.1 Anforderungen

Voraussetzungen

Für die Implementierung des konkreten Lösungsansatzes werden folgende Voraussetzungen angenommen:

- die Schnittstelle zum Web Service wird bereits statisch eingebunden, insbesondere werden die Datenzuordnungen in der Prozessdefinition statisch festgelegt,
- der UDDI-Schlüssel zu dieser Schnittstelle ist bekannt und steht der Implementierung als Parameter zur Verfügung,
- Teile der Suchanfragen werden bereits statisch festgelegt,
- für den Web-Service-Aufruf wird der im WSPMTK enthaltenen Web Service UPES verwendet,
- für den Aufruf wird eine Java-Proxy-Klasse benutzt.
- Da die vorhandenen Web-Service-Erweiterungen zu MQWF statisch ausgerichtet sind, was den Import von Teilen der Prozessdefinition erschwert, soll unterstellt werden, dass nur eine Instanz der Prozessdefinition gleichzeitig aktiv ist.

Anforderungen an den Gesamtprozess

Neben den dargestellten technischen Voraussetzungen, sollen folgende Anforderungen bezüglich der Umsetzung eines automatisierten Gesamtvorganges beachtet werden:

- Der Vorgang muss ohne menschliche Interaktion auszuführen sein.

- Vorhandene Werkzeuge sollen in den Prozess integriert werden, insbesondere der Proxy-Generator `proxygen` aus dem WSTK und das unveröffentlichte Werkzeug `wsdl2fdl` zur Generierung der Programmaktivität.

Anforderungen an die UDDI-Komponente zum Auffinden

Für die UDDI-Komponente zum Auffinden gelten folgende Anforderungen:

- Die Komponente ist flexibel zu entwerfen, damit das Einbinden in verschiedene, übergeordnete Kontrollstrukturen, z.B. Java-Programme, Skriptsprachen oder Workflow-Prozesse, möglich ist.
- Die Implementierung der Selektionskomponente soll an die Möglichkeiten angepasst werden, die von UDDI aktuell zur Verfügung gestellt werden. Zusätzlich soll eine Möglichkeit vorgesehen werden, mit der verschiedene, benutzerdefinierte Selektions-Komponenten auf einfache Weise eingebunden werden können.
- Da die WSDL-Schnittstelle bereits statisch eingebunden wird, kann die Suche nach Web Services in UDDI auf die `find_business`-Anfrage eingeschränkt werden, bei der zusätzlich eine `tModelBag` übergeben wird.
- Alle Komponenten werden auf Basis der Java 2 Plattform implementiert.

6.2 Zugriff auf UDDI

6.2.1 Spezifikation

Für den Schritt zum Auffinden ist ein zweistufiges System zu entwickeln, mit dem eine Menge von geeigneten Web-Service-Implementierungen in UDDI gefunden und daraus der einzubindende Web Service selektiert werden kann.

Eingabe

Als Eingabe für das Auffinden dient neben dem `TModel`-Schlüssel, der die WSDL-Schnittstelle in UDDI identifiziert, eine Reihe optionaler Parameter:

- Namen für `BusinessEntity`-Objekte in UDDI,
- Kategorien für `BusinessEntity`-Objekte in UDDI,
- Schlüssel für `BusinessEntity`-Objekte in UDDI
- Schlüssel für `BusinessService`-Objekte in UDDI

Neben diesen anwendungsbezogenen Parametern müssen auch technische Parameter, wie URL für den UDDI-Betreiber, Benutzername und Passwort für das UDDI-Konto etc. übergeben werden.

Die Eingabe erfolgt in Form von einer oder mehreren Dateien mit dem in Abschnitt 5 dargestellten Format für Konfigurationsdateien.

Falls das System zum Auffinden in einen übergeordneten Prozess eingebunden wird, das diese Form der Parameterübergabe nicht unterstützt, ist ein System zur Anpassung der Parameterübergabe vorzuschalten.

Ausgabe

Die Ausgabe des Auffindvorganges ist ein URL, der die WSDL-Beschreibung des einzubindenden Web Service lokalisiert. Die Form der Ausgabe ist flexibel zu gestalten, damit das System in verschiedene Prozesse eingebunden werden kann. Folgende Ausgabeformen sind vorzusehen:

- Speicherung der WSDL-URL in einer Textdatei,
- Speicherung der WSDL-URL als Systemeigenschaft in der Java-Umgebung,
- WSDL-URL als Rückgabewert vom Typ String einer Methode.

Daneben können weitere Formen der Ausgabe implementiert werden.

Flexibler Auswahlmechanismus

Die Komponente für die Auswahl eines geeigneten Web Service ist so ins System zu integrieren, dass sie ohne Neu-Übersetzung des Systems ausgetauscht werden kann. Die Schnittstelle ist dabei einerseits stabil andererseits offen für zukünftige Erweiterungen zu gestalten.

6.2.2 Entwurf

Der Kern für die UDDI-Komponente für das Einbinden besteht aus drei Java-Klassen: Die Klasse **ServiceWizard** übernimmt die Rolle des Koordinators für den zweistufigen Auffindvorgang. Der **ServiceFinder** ist für die Suche in UDDI verantwortlich, der **ServiceSelector** für die Auswahl des geeigneten Web Service. Für den flexiblen Austausch der **ServiceSelector**-Klasse sorgt ein Plugin-Mechanismus, vgl. Klassendiagramm in Abbildung 6.1. Abbildung 6.2 auf der nächsten Seite zeigt den Auffindvorgang als UML-Sequenzdiagramm.

Daneben werden noch eine Reihe von Hilfsklassen benötigt, z.B. für das Einlesen von Parametern oder die Ausgabe von Binde-Informationen. Ein Klassendiagramm aller am Auffindvorgang beteiligter Klassen zeigt Abbildung 6.3 auf Seite 96. Für den Zugriff auf UDDI werden die Schnittstellen und Klassen der Bibliothek UDDI4J verwendet.

Beschreibung der Klassen

ServiceWizard steuert den Auffindvorgang durch Aufruf der Klassen zum Suchen und Finden. Außerdem ist der **ServiceWizard** verantwortlich für die flexible Ausgabe des Resultates.

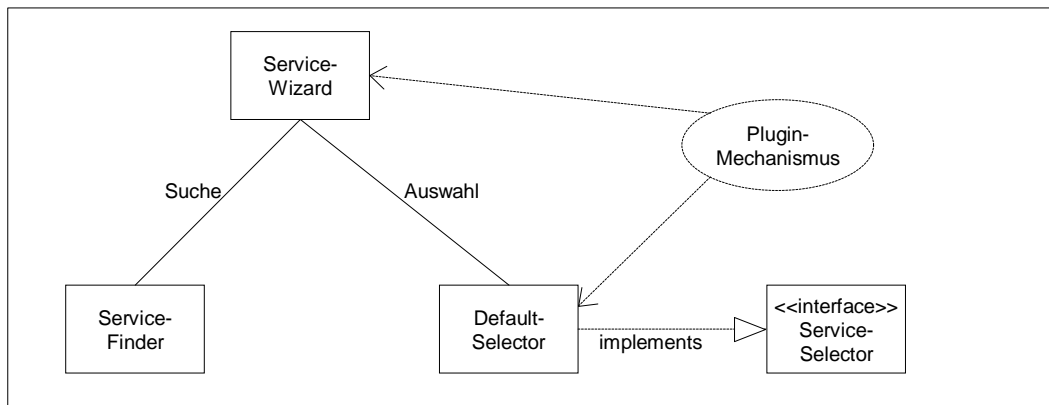


Abbildung 6.1: Kernklassen für den Auffindvorgang

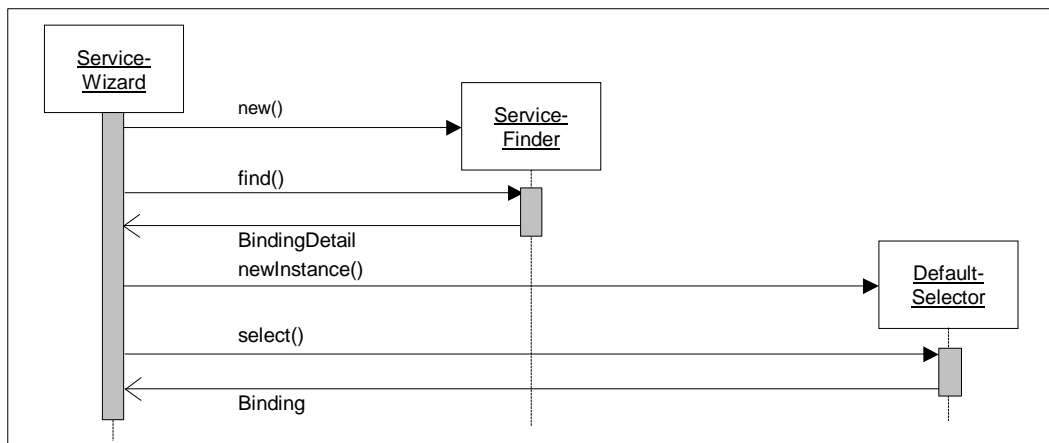


Abbildung 6.2: Sequenzdiagramm: Auffinden eines Web Service

ServiceFinder ist die Komponente für die Suche nach geeigneten Web Services in UDDI. Dazu wird auf die Klassen in der Bibliothek UDDI4J zurückgegriffen.

ServiceSelector ist ein Java-Interface, das die Schnittstelle zur Komponente für die Auswahl repräsentiert. Der Zugriff auf die Methoden der implementierenden Klasse erfolgt auf Basis dieser Schnittstelle.

DefaultSelector Implementierung der **ServiceSelector**-Schnittstelle. Diese Klasse ist verantwortlich für die Auswahl des geeigneten Web Service aus einer Menge von möglichen Web Services.

Binding alternative Möglichkeit zur Ausgabe von Resultaten. In einem Binding Objekt werden die technischen Informationen einer UDDI-BindingDetails-Datenstruktur in kompakter Form gekapselt.

PortInfo enthält Informationen zu einem einzelnen Web-Service-Port. Diese Klasse ist Teil der **Binding**-Informationen. In **Binding**-Objekten können mehrere **PortInfo**-Objekte enthalten sein.

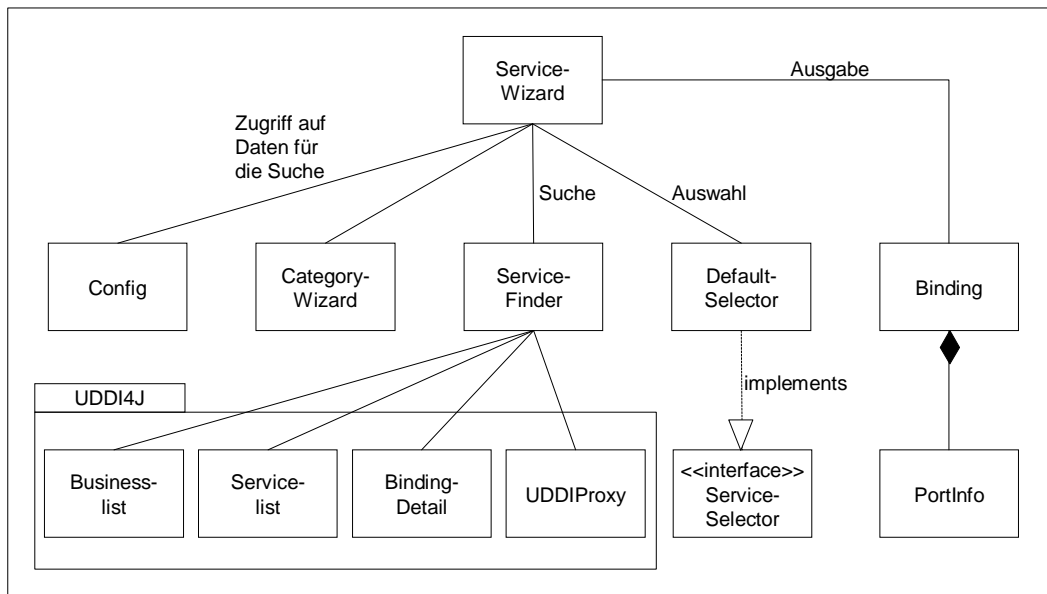


Abbildung 6.3: Klassendiagramm: Auffindvorgang

Die Hilfsklassen `CategoryWizard` und `Config` werden im Abschnitt 5 beschrieben.

Schnittstellenbeschreibung

Die Schnittstelle zum `ServiceWizard` besteht aus drei Methoden: durch Aufruf der `main()`-Methode kann das Auffinden autonom gestartet werden. Dabei müssen drei Parameter übergeben werden: eine Konfigurationsdatei für allgemeine Einstellungen, eine Datei mit den Parametern für den Auffindvorgang und das gewünschte Ausgabeformat für das Resultat. Damit kann festgelegt werden, ob das Resultat in einer Textdatei gespeichert, als Java-Systemeigenschaft zurückgeliefert oder als serialisiertes `Binding`-Objekt ausgegeben werden soll. Durch Aufruf der `find()`-Methode wird der Suchvorgang in UDDI gestartet. Die Parameter dieser Methode sind die beiden Dateien für allgemeine bzw. Suchparameter. Bei dieser Methode wird als Resultat eine UDDI `BindingDetail`-Datenstruktur in Form eines `BindingDetail`-Objektes von UDDI4J. Dieses Objekt kann die Binde-Informationen für mehrere Web Services unterschiedlicher Anbieter, jedoch alle für die gleiche Schnittstelle enthalten. Durch Aufruf der `select()`-Methode wird der Auswahlvorgang gestartet. Als Parameter wird die Konfigurationsdatei mit den Auffindparametern, ein `BindingDetail`-Objekt und ein `java.lang.Object` übergeben. Letzteres wird bei der aktuellen Implementierung nicht benötigt, kann jedoch für spätere Erweiterungen genutzt werden. Das Resultat dieser Methode besteht aus einem `Binding`-Objekt, das die Binde-Informationen zu dem ausgewählten Web Service enthält.

Die Schnittstelle zum `ServiceFinder` besteht aus zwei Methoden: zunächst muss die `updateSettings()`-Methode aufgerufen werden, um alle Einstellungen für den UDDI-Zugriff vorzunehmen. Dazu muss als Parameter eine Konfigurationsdatei mit den UDDI-Einstellungen übergeben werden. Ohne Aufruf dieser Methode kann nicht auf UDDI zugegriffen werden. Der eigentliche Suchvorgang wird durch die `find()`-

Methode gestartet. Diese benötigt als Parameter eine Datei im XML-Konfigurations-Format (vgl. Abschnitt 5), mit folgenden Inhalten:

Obligatorischer TModel-Schlüssel für die WSDL-Schnittstelle in UDDI

```
<Configuration>
  <Interface>
    <TModelKey>456789</TModelKey>
  </Interface>
  ...
</Configuration>
```

Listing 6.1: TModelKey als obligatorischer Suchparameter

Optional Einstellungen: Name eines Unternehmens, Vektor mit Unternehmensnamen, Unternehmenskategorien oder UDDI-Schlüssel von Unternehmen

```
<Configuration>
  ...
  <Business>
    <Name></Name>
    <BusinessKey></BusinessKey>
    <Vector keyName="names">
    </Vector>
    <Vector keyName="businessKeys">
    </Vector>
    <Vector keyName="categories">
      <Category class="Category">
        <CategoryType>unspsc:3-1</CategoryType>
        <Key>84141602</Key>
        <KeyDescription>
          Credit card service providers
        </KeyDescription>
      </Category>
    </Vector>
  </Business>
  ...
</Configuration>
```

Listing 6.2: Optionale Suchparameter

Optionale Parameter: Vektor mit UDDI-Schlüsseln für BusinessService-Objekten

```
<Configuration>
  ...
  <Service>
```

```

        <Vector keyName="keys">
            <key>13579</key>
            ...
        </Vector>
    </Service>
    ...
</Configuration>

```

Listing 6.3: Optionale Suchparameter für BusinessServices

Das Interface `ServiceSelector` gibt die Schnittstelle für das `DefaultSelector`-Objekt vor. Die `select()`-Methode startet den Auswahlvorgang. Dazu müssen drei Parameter angegeben werden: Konfigurationsdatei mit Auswahlparametern, `BindingDetail`-Objekt und ein `java.lang.Object`. Dieser dritte Parameter ist als zukünftige Erweiterung der Schnittstelle vorgesehen und erlaubt die Übergabe beliebiger weiterer Informationen für die Auswahl. Als Resultat der Methode wird ein `BindingTemplate`-Objekt zurückgegeben, das die Informationen zur ausgewählten Web-Service-Implementierung enthält.

Die Schnittstelle zum `Binding`- und `PortInfo`-Objekt besteht aus einer Reihe von `set`- und `get`-Methoden, mit denen die Werte der Felder gesetzt bzw. abgerufen werden können.

Ablauf

In Abbildung 6.2 auf Seite 95 ist ein Sequenzdiagramm für den Auffindvorgang dargestellt. Dabei ist der zweistufige Aufbau des Auffindvorganges erkennbar. zunächst wird die Suche an die Klasse `ServiceFinder` delegiert.

Der `ServiceFinder` analysiert zunächst die Datei mit den Suchparametern und erzeugt daraus drei verschiedene Anfragetypen für die Suche in UDDI. Mit Unternehmens-Namen und -Kategorien werden `find_business()`-, mit Unternehmens-Schlüsseln `find_service()`- und mit `BusinessService`-Schlüsseln eine `getBindingDetail()`-Anfrage formuliert. Dazu wird bei jeder Anfrage auch eine `tModelBag`, die den Schnittstellen-Schlüssel enthält, integriert.

Zuerst werden die `find_business`-Anfragen an UDDI geschickt. Bei jeder neuen Anfrage werden die Resultate in Form eines `BusinessList`-Objektes mit den früheren Resultaten vereinigt. Am Ende der Suchanfragen liegt somit ein vereinigtes `BusinessList`-Objekt vor.

Darauf werden die `find_service()`-Anfragen an UDDI geschickt, deren Resultate analog zu einem `ServiceList`-Objekt vereinigt werden.

In zwei getrennten Schritten werden nun aus der `BusinessList` und aus der `ServiceList` die `BusinessService`-Schlüssel ermittelt und mit den `BusinessService`-Schlüsseln aus der Konfigurationsdatei vereinigt. Bei der Vereinigung werden gleichzeitig Duplikate eliminiert.

Nun wird für jeden `BusinessService`-Schlüssel eine `getBindingDetail()`-Anfrage formuliert und an UDDI geschickt. Die Resultate der Einzelanfragen in Form eines

`BindingDetail`-Objektes werden wiederum vereinigt. Damit liegt das Ergebnis der Suche vor und wird in Form des `BindingDetail`-Objektes an den `ServiceWizard` zurückgegeben. Die Suche nach geeigneten Web Services in UDDI ist in Abbildung 6.4 auf der nächsten Seite als UML-Sequenzdiagramm dargestellt.

Das `BindingDetail`-Objekt wird zur Auswahl an die implementierende Klasse der `ServiceSelector`-Schnittstelle zur Auswahl übergeben. Das Resultat der Auswahl in Form eines `BindingDetail`-Objektes wird vom `ServiceWizard` in ein `Binding`-Objekt transformiert. Die Ausgabe des Resultates des Auffindvorganges wird schließlich in der gewünschten Form ausgegeben.

Plugin-Mechanismus

In Abschnitt 4.2 wird dargestellt, dass die aktuellen Möglichkeiten von UDDI zur Unterstützung der automatischen Auswahl nicht befriedigend sind. Um das System offen für neue Verfahren zur Auswahl zu gestalten, wird eine Möglichkeit vorgesehen, mit der die Klasse, die die Auswahl implementiert, ohne Programmieraufwand ausgetauscht werden kann. Dazu wird eine Java Systemeigenschaft definiert, deren Wert den Namen der implementierenden Klasse enthält. Der `ServiceWizard` liest diese Systemeigenschaft und lädt die entsprechende Klasse zur Instanzierung. Voraussetzung für den Aufruf der Auswahl-Methode ist, dass die Klassen für die Auswahl eine gemeinsame Schnittstelle implementieren. Diese gemeinsame Schnittstelle wird im Java-Interface `ServiceSelector` definiert. Die Java-Systemeigenschaft hat dabei den Schlüssel `"com.ibm.workflow.util.uddi.ServiceSelector"`.

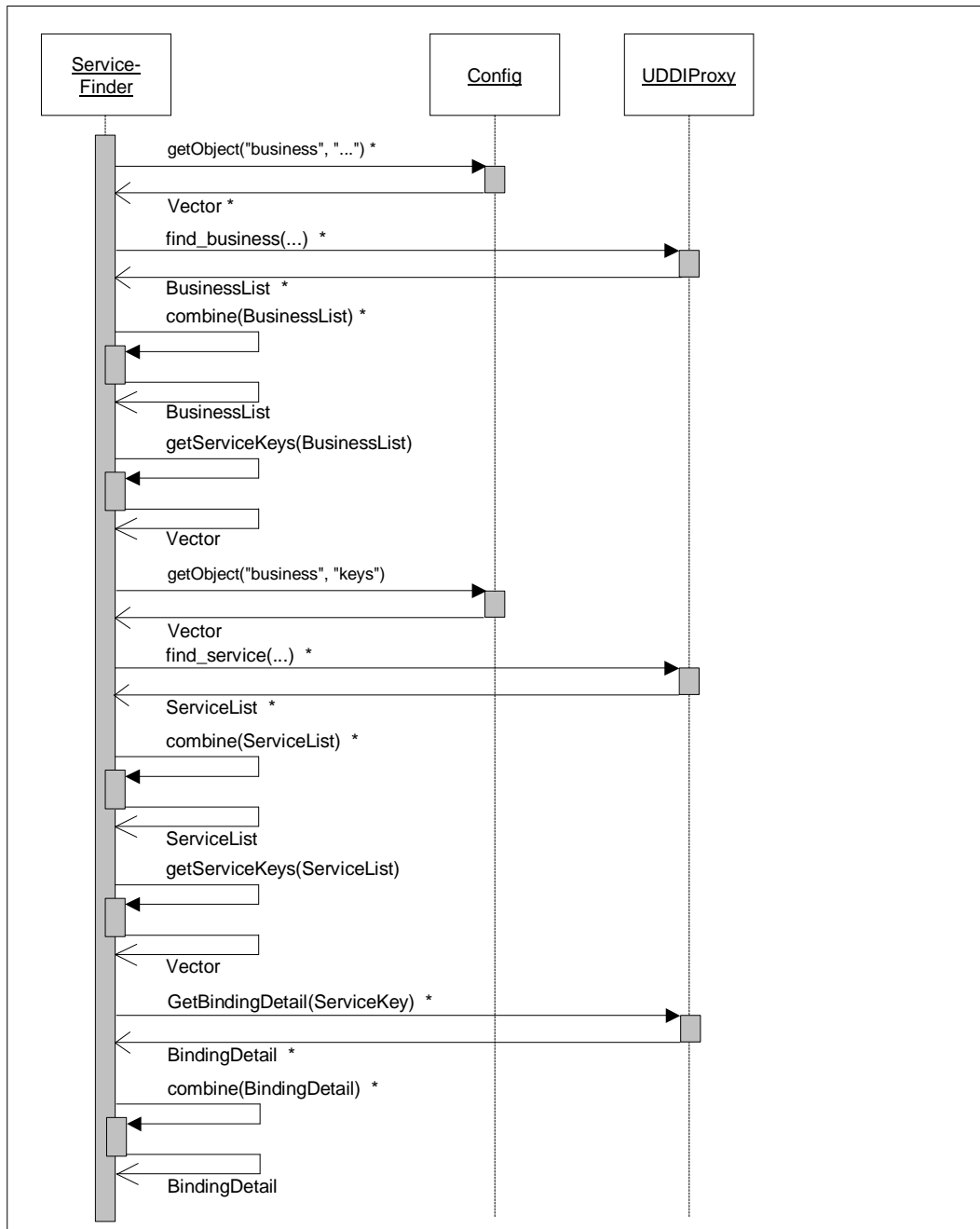


Abbildung 6.4: Sequenzdiagramm: Suche nach geeigneten Web Services in UDDI

6.3 Einbindungsprozess

6.3.1 Spezifikation

Auf Basis der in Abschnitt 6.1 dargestellten Anforderungen soll eine Lösung für das dynamische Einbinden von Web Services in Workflow-Prozesse entwickelt werden. Der Vorgang wird von einer Programmaktivität in einem Workflow aktiviert. Nach Abschluss des Prozesses soll das WFMS in der Lage sein, einen Web Service als Implementierung einer Aktivität aufzurufen. Die Schnittstelle für den Web Service ist bei Prozessbeginn in Form eines UDDI-Schlüssels für ein TModel bekannt.

Eingabe

Als Eingabe für den Prozess werden dynamische Parameter, die zur Laufzeit festgelegt werden, und statische Parameter, die bereits zum Zeitpunkt der Modellierung bekannt sind, erwartet. Als dynamische Parameter können eine Reihe von Unternehmensnamen übergeben werden, bei denen nach einem Web Service gesucht wird, der zur Schnittstelle passt.

Zu den statischen Parametern gehören neben dem TModel-Schlüssel auch die optionalen Parameter Unternehmensnamen, Unternehmens-Kategorien, Unternehmens-Schlüssel oder Service-Schlüssel. Bei den optionalen Parametern können jeweils mehrere unterschiedliche Werte angegeben werden. Die statischen Parameter für die Generierung der Proxy-Klassen und der Programmdefinition sind zwei Ausgabeverzeichnisse, Name der Programmdefinition sowie weitere Einstellungen für die MQWF-Umgebung, vgl. Dokumentation zum WSPMTK.

Daneben werden UDDI-relevante Einstellungen, z.B. URL für die Anfragen, Benutzername und Kennwort, für die Suche benötigt, vgl. 6.2.1.

Die dynamischen Parameter befinden sich zum Startzeitpunkt des Einbindungsprozesses in einem Datencontainer des Workflow-Prozesses. Erwartet wird ein Feld von Strings, in dem mehrere Unternehmensnamen gespeichert werden können. Das folgende Listing zeigt eine Workflow-Datenstruktur für fünf Unternehmensnamen:

```
STRUCTURE 'BusinessNames'
  'name': STRING(5);
END 'BusinessNames'
```

Listing 6.4: Workflow-Struktur der dynamischen Parameter

Die statischen Parameter werden zum Modellierungs-Zeitpunkt festgelegt und in einer Datei, z.B. im XML-Konfigurationsformat, vgl. 6.2.2, gespeichert.

Ausgabe

Die Ausgabe des Einbindungsprozesses besteht darin, die Voraussetzungen dafür zu schaffen, dass das WFMS den gewünschten Web Service aufrufen und die Resultate weiter verarbeiten kann. Dazu müssen die notwendigen Proxy-Klassen für den

UPES-Aufruf erzeugt und in den CLASSPATH eingefügt werden. Daneben muss auch eine Programmdefinition für diesen UPES-Aufruf erzeugt und in MQWF importiert werden.

Werkzeuge für Einzelschritte

Zur Realisierung des Einbindungsprozesses sollen folgende Software-Werkzeuge verwendet werden:

- für den Auffindvorgang die in Abschnitt 6.2.2 dargestellte `ServiceWizard`-Klasse,
- zur Generierung der Proxy-Klassen für den Aufruf des Software-Werkzeug `proxygen` des Web Services Toolkit (WSTK),
- zur Generierung der Programmdefinition das Werkzeug `wsdl2fdl` in modifizierter Form.
- für den Import der Programmdefinition das Hilfsprogramm `fmcibie` von MQWF.

6.3.2 Entwurf

Für die Steuerung des Gesamtprozesses wird das Werkzeug `ANT` der APACHE Group verwendet. Mit `ANT` können die Einzelschritte wie bei einer Skriptsprache in einer eigenen Syntax definiert und zu einem Gesamtprozess zusammengefasst werden. Dieser Gesamtprozess wird dann in einer Datei, der sog. Build-Datei gespeichert. Mit `ANT` können heterogene Programme, z.B. Systemprogramme und Java-Programme, integriert werden. Innerhalb der `ANT`-Umgebung können Variablen definiert werden, auf die innerhalb von Java-Klassen wie auf Systemeigenschaften zugegriffen werden kann. In der Build-Datei werden die Programmaufrufe der Einzelschritte in sequentieller Reihenfolge definiert. Innerhalb der Build-Datei können auch statische Parameter des Einbindungsprozesses direkt festgelegt werden, z.B. die Ausgabeverzeichnisse für die Proxy-Klassen und die FDL-Datei mit der Programmdefinition.

Der Einbindungsprozess wird dadurch gestartet, dass MQWF das `ANT`-Programm aufruft. Als Parameter wird dabei der Name der Build-Datei angegeben. Dieser Programmaufruf wird bei der Modellierung des Workflow-Prozesses statisch festgelegt.

In der Build-Datei werden folgende Einzelschritte definiert:

1. Zugriff auf den Datencontainer zur Ermittlung der dynamischen Parameter. Dazu muss eine Java-Klasse implementiert werden, mit der über das Workflow-API auf die Daten zugegriffen werden kann. Die ermittelten Daten müssen mit den statisch definierten Suchparametern kombiniert werden. Dazu ist der Name der Datei mit den statischen Informationen als Parameter zu übergeben.
2. Auffinden des Web Service mit Hilfe der `ServiceWizard`-Klasse. Dazu werden drei Parameter benötigt: Konfigurationsdatei, Datei mit Suchparametern und die Ausgabeoption „ant“.

3. Generierung der Proxy-Klassen mit Hilfe des Proxy-Generators `proxygen`. Dabei werden zwei Parameter benötigt: zum einen die URL der WSDL-Beschreibung, zum anderen ein Ausgabeverzeichnis für die erzeugten Klassen. Dieses Ausgabeverzeichnis ist sinnvollerweise in dem `CLASSPATH` des Web-Service-UPES enthalten, so dass dieser auf die Proxy-Klassen zugreifen kann.
4. Generierung der fehlenden Programmdefinition für den Workflow-Prozess mit Hilfe des Werkzeuges `wsdl2fdl`. Für den Aufruf sind zwei Parameter notwendig: die URL der WSDL-Beschreibung und eine weitere Konfigurationsdatei. Die notwendigen Änderungen an `wsdl2fdl` werden weiter unten beschrieben.
5. Import der Programmdefinition in die Runtime-Datenbank von MQWF. Dazu kann das Werkzeug `fmcibie` von MQWF benutzt werden mit den Parametern `FDL-Dateiname` und Option „import“.

Neben den Schritten werden folgende Variablen für Aufrufparameter in der Build-Datei gespeichert:

Konfiguration Name der Konfigurationsdatei für allgemeine Einstellungen.

Suchkonfiguration Name der Datei mit den Suchparametern.

Proxy-Verzeichnis Ausgabeverzeichnis für die Proxy-Klassen.

FDL-Datei Datei, die die generierte Programmdefinition enthält.

WSDL-URL Variable zur Speicherung des URL der WSDL-Beschreibung.

Ablauf

Der Ablauf des Einbindungsprozesses ist in den Abbildungen 6.5 und 6.6 auf der nächsten Seite als UML-Sequenzdiagramme dargestellt.

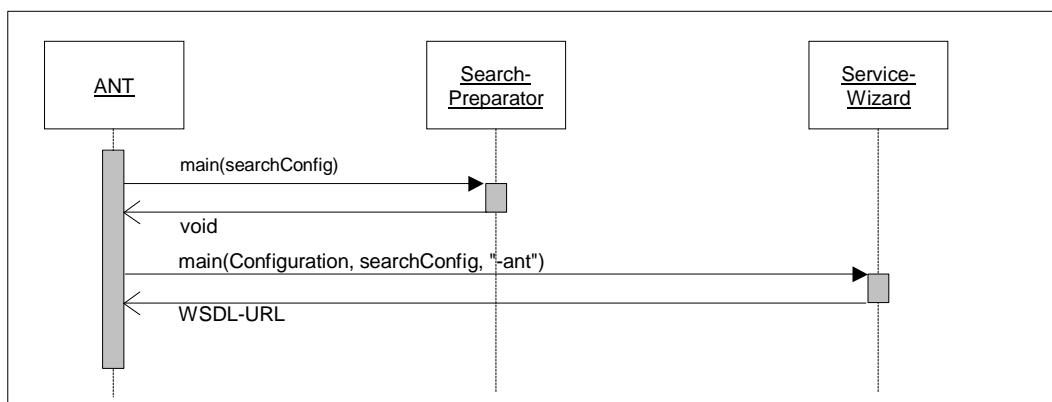


Abbildung 6.5: Einbindungsprozess: Zugriff auf UDDI

Im ersten Schritt werden die dynamischen Suchparameter aus den Workflow-Datenstrukturen gelesen und mit den statischen Suchparametern kombiniert. Mit den vereinigten

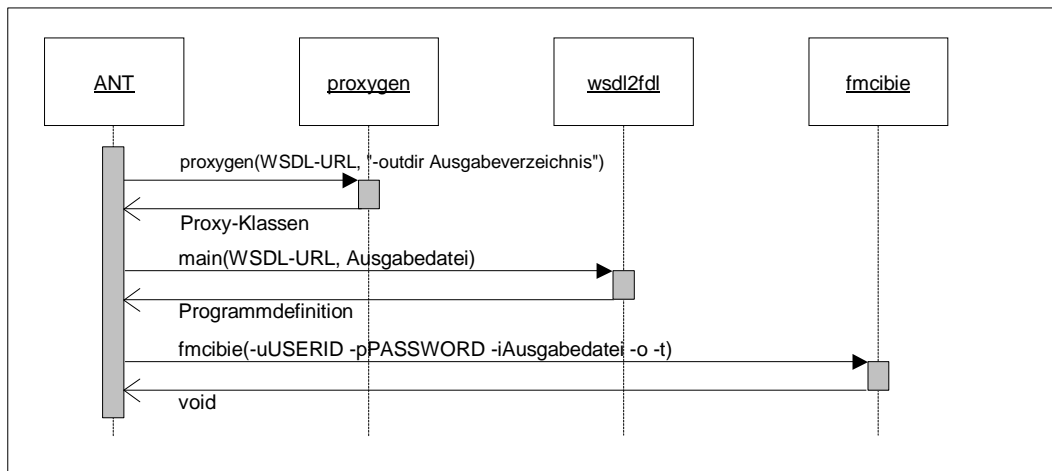


Abbildung 6.6: Einbindungsprozess: Generierung der Proxy-Klassen und der Programmdefinition

Suchparametern wird dann ein geeigneter Web Service in UDDI ermittelt. Als Zwischenergebnis liegt nun der URL der WSDL-Beschreibung vor.

Im nächsten Schritt wird aus der URL bzw. dem WSDL-Dokument die notwendigen Proxy-Klassen für den Web-Service-Aufruf generiert. Anschließend wird eine FDL-Programmdefinition erzeugt, die den Web-Service-Aufruf über MQWF ermöglicht. Diese Programmdefinition wird in einem letzten Schritt in die Laufzeitumgebung von MQWF importiert und kann dann im Workflow-Prozess genutzt werden.

Container-Zugriff

Für den Zugriff auf die dynamischen Suchparameter muss eine Java-Klasse entwickelt werden, die über die Programmierschnittstelle von MQWF Daten aus den Workflow-Datenstrukturen ermittelt und mit den vorhandenen statischen Suchparametern kombiniert, vgl. Abbildung 6.7.

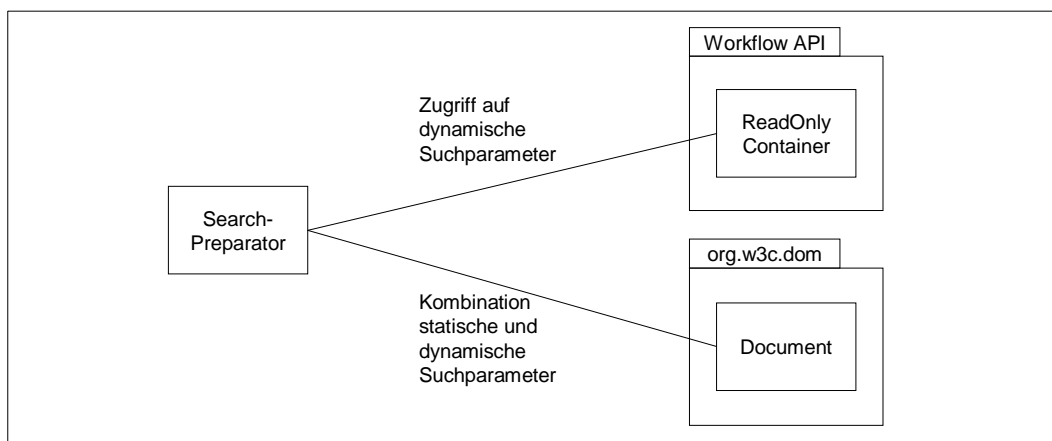


Abbildung 6.7: Klassendiagramm: Zugriff auf dynamische Suchparameter

Die `SearchPreparator`-Klasse hat eine Schnittstelle in Form einer `main`-Methode mit einem Parameter: Name der Datei mit den statischen Suchparametern im XML-Konfigurations-Format. Die über die Workflow-Schnittstelle ermittelten Parameter werden dann mit den vorhandenen Suchparametern kombiniert. Dafür kann das Document Object Model (DOM) benutzt werden. Nach dem Einfügen wird das DOM-Dokument zurück in die übergebene Datei geschrieben.

Bei jeder Änderung der Suchparameter im Workflow-Prozess muss die `SearchPreparator`-Klasse an die neue Situation angepasst werden.

Änderungen an `wsdl2fdl`

Das in Abschnitt 2.2 vorgestellte Software-Werkzeug `wsdl2fdl` ist ohne Modifikation für den Einsatz beim dynamischen Einbinden nicht geeignet, da die Eingabe von Parametern zu unflexibel ist und die Resultate nicht genau den Anforderungen entsprechen. So müssen folgende Änderungen an `wsdl2fdl` vorgenommen werden:

- Die einzige Ausgabe ist eine Datei, die nur eine Programmdefinition in FDL enthält,
- Als Eingabeparameter muss der Dateiname für die Ausgabedatei angegeben werden.
- Der Name für die Proxy-Klasse wird nicht, wie bisher statisch in der Konfigurationsdatei festgelegt, sondern aus dem WSDL-Dokument abgeleitet.

Dazu sind Modifikationen an der Java-Klasse `com.ibm.workflow.util.wsdl.make.FDL` als auch an den XSLT-Stylesheets für die Generierung der FDL-Programmdefinition notwendig.

6.4 Praktische Erfahrungen beim dynamischen Einbinden

Erste praktische Erfahrungen konnten in Laufversuchen gewonnen werden. Dabei wurden mehrere Web Services in einem Test-UDDI-Verzeichnis im Internet eingetragen, um das Auffinden in UDDI zu testen. Bei diesen Tests fällt zunächst die lange Wartezeit bei den UDDI-Anfragen auf. In Testszenarien mit drei bis fünf UDDI-Anfragen wurde eine effektive Wartezeit von etwa einer Minute beobachtet, was eine erhebliche Verlangsamung des Workflow-Prozesses nach sich zieht.

Daneben wurde beobachtet, dass manche UDDI-Anfragen nicht die in der UDDI-Spezifikation definierten Resultate erzeugten, z.B. wurde bei einer `find_business`-Anfrage mit `tModelBag` auch Business-Service-Objekte aufgelistet, die nicht die erforderliche Schnittstelle implementieren. Als Resultat dieser Beobachtung wurden im Auffindprozess Prüfroutinen integriert, die unerwünschte Resultate aus den Anfrageergebnissen herausfiltern.

Bereits in Abschnitt 4 wird deutlich, dass die vorhandenen Web-Service-Erweiterungen zu MQWF das dynamische Einbinden nicht direkt unterstützen. Insbesondere die Modifikation des Prozessmodells zur Laufzeit kann ohne Modifikation von

`wsdl2fdl` nicht realisiert werden. Diese Modifikation ist explizit nicht Teil dieser Arbeit. Eine Lösung des Problems ist eher im Rahmen einer überarbeiteten Aufrufkomponente zu suchen und weniger in der Modifikation der vorhandenen Werkzeuge.

7 Zusammenfassung und Ausblick

Universal Description, Discovery and Integration (UDDI) als globales Web-Service-Verzeichnis spielt eine zentrale Rolle im Zusammenspiel zwischen Web Services und IBM MQ Series Workflow (MQWF). Durch die klare Definition eines Informationsmodells und einer entsprechenden Programmierschnittstelle ermöglicht UDDI sowohl Anbietern als auch Nachfragern von Web Services eine einheitliche Sicht und einen einheitlichen Zugriff auf technische und beschreibende Informationen zu den angebotenen Software-Dienstleistungen. Dieser einheitliche Zugriff macht eine automatisierte Nutzung von UDDI durch andere Anwendungen möglich.

Web-Service-fähig wird MQWF durch die in Abschnitt 2 vorgestellten Web-Service-Erweiterungen in Form eines generischen Web-Service-Provider und einem generischen Web-Service-UPES. Diese Erweiterungen schränken die Menge der nutzbaren Transport-Mechanismen bzw. Nachrichtenformate auf SOAP über HTTP ein. Zur einfacheren Benutzung dieser Erweiterungen werden auch Software-Werkzeuge zur Verfügung gestellt. Noch nicht realisiert ist die automatisierte Nutzung von UDDI zur Veröffentlichung eigener und für die Suche nach in Workflows einzubindende Web Services.

Für das Veröffentlichen muss die Prozessdefinition, die in Flow Definition Language (FDL) formuliert vorliegt, in eine Web-Service-Beschreibung in Web Services Description Language (WSDL) transformiert und schließlich an UDDI zur Speicherung übertragen werden. Um den Vorgang zu automatisieren müssen vorhandene Werkzeuge mit neu zu entwickelnden Anwendungen integriert werden. Zur Integration kann ein Workflow Management System, ein Java-Programm oder eine Skriptsprache verwendet werden.

Doch nicht nur die technische Seite spielt bei der Veröffentlichung eine Rolle, sondern auch die Frage, wie Informationen in UDDI dargestellt werden müssen, damit Nachfrager diese schnell und gezielt auffinden können. UDDI besitzt zwar Möglichkeiten zur Klassifizierung von veröffentlichten Web Services durch eingebaute und benutzerdefinierte Taxonomien, doch liegt der Schwerpunkt auf unstrukturierten, textbasierten Beschreibungen, die letztendlich von jedem Benutzer selbst gestaltet werden können. Die Klassifizierung von Web Services in UDDI ist jedoch notwendig für die automatisierte Interaktion. Somit muss ein automatisierter Veröffentlichungsvorgang so gestaltet werden, dass solche Klassifizierungsinformationen berücksichtigt werden können.

Auf der anderen Seite kann UDDI dazu benutzt werden, um geeignete Web Services aufzufinden, die in Workflow-Prozesse eingebunden werden sollen. Dabei wird zwischen statischem und dynamischen Einbinden unterschieden. Beim statischen Ansatz wird UDDI während der Modellierung eines Prozesses konsultiert, um den Web Service zu finden. Die Interaktion kann dabei über eine geeignete Anwendung oder über die Web-Schnittstelle zu UDDI erfolgen. Nach der Modellierung kann der Web Service durch aus dem Workflow heraus aufgerufen werden. Weitere Zugriffe auf UDDI sind nicht notwendig.

Dagegen wird beim dynamischen Einbinden zur Ausführungszeit des Workflow-Prozesses auf UDDI zugegriffen, d.h. der Web Service wird erst zur Laufzeit gesucht und eingebunden. Dazu ist eine Interaktion mit UDDI über die Programmierschnittstelle notwendig und zwar ohne menschliche Interaktion. Hier wird auch deutlich, wie wichtig die Kategorisierung von Web Services für die Automatisierung der Suche ist.

Beim dynamischen Ansatz werden Schwachstellen von UDDI deutlich: Informationen die bei der Auswahl eines Web Service wichtig sind, wie z.B. Verfügbarkeit, Sicherheit, Zuverlässigkeit, Preis, etc., können in UDDI nicht abgefragt werden, weil für solche Informationen kein einheitliches Informationsmodell und keine Taxonomien entwickelt wurden. Daneben ist der hierarchische Aufbau der Suchanfragen, die von der Programmierschnittstelle zur Verfügung gestellt werden, hinderlich bei der Suche. Wünschenswert wäre hier eine SQL-ähnliche Anfrageformulierung, damit z.B. Verbunde nicht durch eigene Algorithmen nachgestellt werden müssen.

Die Werkzeuge und der Aufrufmechanismus der Web-Service-Erweiterungen zu MQWF sind auf das statische Einbinden von Web Services ausgerichtet. Der Einsatz zum dynamischen Einbinden ist dabei in vielfacher Hinsicht problematisch. Probleme bereitet vor allem der Datenfluss zwischen Workflow-Prozess und Werkzeugen sowie das dynamische Einbinden von eigentlich statischen Prozess-Komponenten. Dabei bietet sowohl UDDI als auch WSDL alle Voraussetzungen dafür, dass der Aufwand zur Laufzeit minimiert werden kann. Dazu werden alle Vorkehrungen zum Aufruf des Web Service statisch aus der WSDL-Schnittstelle abgeleitet. Der UDDI-Schlüssel zu dieser Schnittstelle wird zur Laufzeit für eine vereinfachte Suche nach geeigneten Web Services benutzt, bei der die fehlenden Informationen ermittelt und einem veränderten Aufrufmechanismus als dynamische Parameter übergeben werden.

In der aktuellen Version stellt UDDI einen interessanten Ansatz für ein globales Verzeichnis für Web Services dar. Die dargestellten Mängel können bei einer Nutzung von UDDI in einem Intranet kompensiert werden, in dem eine gewisse Disziplin und interne Standards, z.B. für Taxonomien leichter etabliert werden können. Der globale Einsatz im Internet kann dagegen als problematisch bezeichnet werden, da weltweite Standardisierungsprozesse eher langsam vonstatten gehen und „Quality of Services“ eine größere Bedeutung besitzen.

A Code-Beispiele

A.1 WSDL

Die folgenden Listings zeigen eine Web-Service-Beschreibung in WSDL, aufgeteilt in Schnittstelle und Implementierung. Zunächst wird die Schnittstelle vorgestellt:

```
<definitions
  name="CreditCardChecker_Service"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace=
    "http://www.plasticmoney.de/CreditCardChecker-interface"
  xmlns:tns="http://www.plasticmoney.de/CreditCardChecker"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="CheckCreditCardRequest">
    <part name="creditCardNumber" type="xsd:string"/>
  </message>
  <message name="CheckCreditCardResponse">
    <part name="checkResult" type="xsd:int"/>
  </message>
  <portType name="CreditCardChecker_Service">
    <operation name="checkCreditCard">
      <input message="tns:CheckCreditCardRequest"/>
      <output message="tns:CheckCreditCardResponse"/>
    </operation>
  </portType>
  <binding name="CreditCardChecker_ServiceBinding"
    type="tns:CreditCardChecker_Service">
    <soap:binding
      style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="checkCreditCard">
      <soap:operation soapAction="urn:creditcardchecker-service"/>
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:creditcardchecker-service" use="encoded"/>
      </input>
    </operation>
  </binding>
</definitions>
```

```
<output>
  <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:creditcardchecker-service" use="encoded"/>
</output>
</operation>
</binding>
</definitions>
```

Listing A.1: WSDL-Schnittstelle (CS-interface.wsdl)

Das folgende Listing zeigt das WSDL-Dokument mit der konkreten Service-Beschreibung:

```
<definitions
  name="CreditCardChecker_Service"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://www.plasticmoney.de/CreditCardChecker"
  xmlns:ins="http://www.plasticmoney.de/CreditCardChecker-interface"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <import
    location="http://www.plasticmoney.de/services/wsdl/CS-interface.wsdl"
    namespace="http://www.plasticmoney.de/CreditCardChecker-interface">
  </import>
  <service name="CreditCardChecker_Service">
    <documentation>...</documentation>
    <port name="CreditCardChecker_ServicePort"
      binding="ins:CreditCardChecker_ServiceBinding">
      <soap:address
        location="http://www.plasticmoney.de:8080/soap/servlet/rpcrouter"/>
      </port>
    </service>
  </definitions>
```

Listing A.2: WSDL-Implementierung (CS.wsdl)

A.2 UDDI-Strukturen

Das folgende Listing zeigt den Web Service für die Kreditkartenüberprüfung dargestellt als BusinessService in UDDI.

```

<businessEntity
  businessKey="123AAA-456B-789CCCC"
  operator="uddi.sample_uddi.org/publish">
  <discoveryURLs>...</discoveryURLs>
  <name lang="de">Plastic Money GmbH</name>
  <description>...</description>
  <contacts>...</contacts>
  <businessServices>

    <businessService
      businessKey="123AAA-456B-789CCCC"
      serviceKey="123AAA-999F-111DEF2">
      <name lang="en">CreditCardChecker_Service</name>
      <description>...</description>
      <bindingTemplates>

        <bindingTemplate
          serviceKey="123AAA-999F-111DEF2"
          bindingKey="123AAA-777A-345BBB7">
          <description>...</description>
          <accessPoint URLType="http">
            http://www.plasticmoney.de:8080/soap/servlet/rpcrouter
          </accessPoint>
          <tModelInstanceDetails>
          </tModelInstanceDetails>
        </bindingTemplate>

      </bindingTemplates>
      <categoryBag>
        <keyedReference>
      </categoryBag>
    </businessService>

  </businessServices>
  <categoryBag>
    <keyedReference>
  </categoryBag>
</businessEntity>

```

Listing A.3: Web Service In UDDI

Die folgenden Listings zeigen, wie der Web Service für die Kreditkartenüberprüfung, mit einer WSDL-Beschreibung in UDDI gespeichert wird.

Schnittstelle:

```
<tModel tModelKey="456EEE-A123-CD12BB4">
  <name>...</name>
  <description>...</description>
  <overviewDoc>
    <overviewURL
      http://www.plasticmoney.de/CS-interface.wsdl
    </overviewURL>
  </overviewDoc>
  ...
  <categoryBag>
    <keyedReference>
  </categoryBag>
</tModel>
```

Listing A.4: WSDL-Schnittstelle als UDDI-TModel

Implementierung

```
<businessEntity ...>
  ...
  <businessServices>
    <businessService>
      ...
      <bindingTemplates>
        <bindingTemplate ...>
          ...
          <accessPoint URLType="http">
            http://www.plasticmoney.de:8080/soap/servlet/rpcrouter
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo
              tModelKey="456EEE-A123-CD12BB4">
              <description>...</description>
              <instanceDetails>
                <overviewDoc>
                  <overviewURL
                    http://www.plasticmoney.de/CS.wsdl
                  </overviewURL>
                </overviewDoc>
                ...
              </instanceDetails>
              ...
            </tModelInstanceInfo>
          </tModelInstanceDetails>
        </bindingTemplate>
        ...
      </bindingTemplates>
    </businessService>
    ...
  </businessServices>
  ...
</businessEntity>
```

Listing A.5: WSDL-Implementierung als UDDI-BusinessService

A.3 Generierung von WSDL aus einer FDL-Prozessdefinition

Die folgenden Code-Beispiele zeigen die FDL-Definition und die daraus erzeugte WSDL-Beschreibung. Dabei wird aus Gründen der Übersichtlichkeit auf die Namespace-Deklarationen innerhalb der WSDL-Beschreibung verzichtet. Die verwendeten Namespace-Prefixes sind:

wsdl - WSDL-Namepace
tns - targetNamespace der WSDL-Definitionen
xs - XML-Schema-Namepace
sns - targetNamespace des XML-Schema soap - SOAP-Namepace

```
/*
 * STRUCTUREs
 */
STRUCTURE 'CreditCardInfo'
  DESCRIPTION "CreditCardInfo data structure"
  'CreditCardNumber': STRING
  'ValidityLevel': LONG
END 'CreditCardInfo'

/*
 * PROCESS CreditCardCheckerFlow
 */
PROCESS 'CreditCardCheckerFlow' ('CreditCardInfo', 'CreditCardInfo')
  DESCRIPTION "Checks the validity of a credit card"
  ...
END 'CreditCardCheckerFlow'
```

Listing A.6: Prozess zur Kreditkartenüberprüfung in FDL

Generiertes WSDL-Dokument:

```

<wsdl:definitions name="Checker"
  targetNamespace="http://www.plasticmoney.de/Checker">

<wsdl:types>
  <xs:schema
    targetNamespace="http://www.plasticmoney.de/Checker/schema">
    <xs:complexType name="CreditCardInfo">
      <xs:sequence>
        <xs:element name="CreditCardNumber" type="xs:string"
          maxOccurs="1" minOccurs="0"/>
        <xs:element name="ValidityLevel" type="xs:int"
          maxOccurs="1" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</wsdl:types>

<wsdl:message name="CreditCardCheckerFlowCallRequest">
  <wsdl:part name="Input" type="sns:CreditCardInfo"/>
</wsdl:message>

<wsdl:message name="CreditCardCheckerFlowCallResponse">
  <wsdl:part name="Output" type="sns:CreditCardInfo"/>
</wsdl:message>

<wsdl:portType name="CreditCardCheckerFlow">
  <wsdl:operation name="checkCreditCardNumber">
    <wsdl:input message="tns:CreditCardCheckerFlowCallRequest"/>
    <wsdl:output message="tns:CreditCardCheckerFlowCallResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="CreditCardCheckerFlowBinding"
  type="tns:CreditCardCheckerFlow">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="checkCreditCardNumber">
    <soap:operation soapAction="CreditCardCheckerFlow#call"/>
    <wsdl:input>
      <soap:body namespace="urn:Checker" use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body namespace="urn:Checker" use="encoded"

```

```
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

</wsdl:service>
  <wsdl:port name="CreditCardCheckerFlowPort"
    binding="tns:CreditCardCheckerFlowBinding">
    <soap:address
      location="http://www.plasticmoney.de/soap/servlet/rpcrouter"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Listing A.7: Kreditkartenüberprüfung - WSDL-Beschreibung

Literatur

- [ABK⁺00] Anderson, Richard; Birbeck, Mark; Kay, Michael et al.: *XML professionell*. 1. Auflage. Übersetzung aus dem Amerikanischen von Eduard Paul, Barbara Jaekel, Uwe Jaekel, Reinhard Engel. Bonn: MITP-Verlag 2000.
- [BCE⁺01] Brittenham, Peter; Curbera, Francisco; Ehnebuske, David et al.: *Understanding WSDL in a UDDI Registry*. Version 1.0. How to Publish and Find WSDL Service Descriptions. IBM August 2001.
- [BE⁺00] Box, Don; Ehnebuske, David et al.: *Simple Object Access Protocol (SOAP) 1.1*. W3C Note 8 May 2000. W3C May 2000.
- [Bri01] Brittenham, Peter: *Web Services Development Concepts*. IBM Software Group Mai 2001.
- [CER01] Curbera, Francisco; Ehnebuske, David; Rogers, Dan: *Using WSDL in a UDDI Registry 1.05*. UDDI Working Draft Best Practices Document. uddi.org June 2001.
- [CW⁺01] Curbera, Francisco; Weerawarana, Sanjiva et al.: *Web Services Description Language (WSDL)*. W3C Note 15 March 2001. Ariba, IBM, Microsoft March 2001.
- [DGM⁺01] Ducket, John; Griffin, Oliver; Mohr, Stephen et al.: *XML Schemas*. Birmingham: Wrox Press Ltd 2001.
- [ERvR01] Ehnebuske, David; Rogers, Dan; von Riegen, Claus: *UDDI Version 2.0 Data Structure Reference*. UDDI Open Draft Specification 8 June 2001. uddi.org June 2001.
- [Har00] Harold, Elliotte Rusty: *Java Network Programming*. 2nd edition. Sebastopol: O'Reilly 2000.
- [HR99] Härder, Theo; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Berlin, Heidelberg, New York: Springer 1999.
- [HU94] Hopcroft, John E.; Ullmann, Jeffrey: *Einführung in die Automaten-theorie, Formale Sprachen und Komplexitätstheorie*. 3., aktualisierte Auflage. Bonn, Paris, Mass.: Addison-Wesley 1994.
- [IBM01a] IBM, Hrsg.: *IBM MQ Series Workflow Conceptual Architecture*. 5th edition. Version 3.3.2. IBM 2001.
- [IBM01b] IBM, Hrsg.: *IBM MQ Series Workflow Getting Started with Buildtime*. 5th edition. Version 3.3.2. IBM 2001.
- [IBM01c] IBM, Hrsg.: *IBM MQ Series Workflow Programming Guide*. 5th edition. Version 3.3.2. IBM 2001.

- [Kay01] Kay, Michael: *XSLT: Programmer's Reference*. 2nd edition. Birmingham: Wrox Press Ltd 2001.
- [Kre01] Kreger, Heather: *Web Services Conceptual Architecture*. IBM Software Group Mai 2001.
- [LR00] Leymann, Frank; Roller, Dieter: *Production Workflow: Concepts and Techniques*. Upper Saddle River: Prentice-Hall PTR 2000.
- [M⁺00] uddi.org, Hrsg.: *UDDI Technical Whitepaper*. uddi.org September 2000.
- [MER01] McKee, Barbara; Ehnebuske, David; Rogers, Dan: *UDDI Version 2.0 API Specification*. UDDI Open Draft Specification 8 June 2001. uddi.org Juni 2001.
- [Oes88] Oesterreich, Bernd: *Objektorientierte Softwareentwicklung - Analyse und Design mit der Unified Modeling Language*. 3., aktualisierte Auflage. München, Wien: Oldenbourg 1988.
- [Tea00] IBM Web Services Toolkit Development Team: *Web Services Toolkit Overview*. IBM 2000.
- [Tea01] IBM Web Services Toolkit Development Team: *Toolkit Client API*. IBM September 2001.
- [01] <http://www.uddi.org>.
- [02] <http://www.alphaworks.ibm.com/>.
- [03] <http://www.ibm.com/developerworks/webservices>.
- [04] <http://www.ibm.com/software/ts/mqseries/workflow>.
- [05] <http://www.w3.org/XML/>.
- [06] <http://www.w3.org/DOM/>.
- [07] <http://www.w3.org/XML/Schema>.
- [08] <http://www.w3.org/Style/XSL/>.
- [09] <http://www.w3.org/2002/ws/>.
- [10] <http://jakarta.apache.org/ant/index.html>.
- [11] <http://xml.apache.org/soap/index.html>.
- [12] <http://xml.apache.org/xerces2-j/index.html>.
- [13] <http://xml.apache.org/xalan-j/index.html>.
- [14] <http://www.theserverside.com>.
- [15] <http://www.webmethods.org>.
- [16] <http://java.sun.com>.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Andreas Fried)