Karlsruhe Institute of Technology

Aufruf externer Dienste mit Hilfe von aktiven Ontologien

Bachelorarbeit
von

Michael Jakob

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl.-Inform. Martin Blersch
Zweiter betreuender Mitarbeiter:	Dipl.-Inform.Wirt Mathias Landhäußer

Bearbeitungszeit: 01.07.2015 – 31.10.2015

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, Abgabedatum

. . . . . .

(**Michael Jakob**)

Kurzfassung

In dieser Arbeit wird ein Broker implementiert, der Webdienste und Webformulare automatisch mittels aktiver Ontologien anspricht. Dienstanfragen stammen von einer aktiven Ontologie als Eingabe.
Der Broker bietet einen einfachen Registrierungsprozess für existierende Webdienste und Webformulare. Es können gleichzeitig mehrere Dienste angesprochen und die Ergebnisse aggregiert werden. Störungen wie zum Beispiel die Nicht-Erreichbarkeit eines Dienstes werden behandelt. Zusätzlich werden Dienstgüteparameter unterstützt, die bei der Auswahl eines Dienstes herangezogen werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Automatische Spracherkennung erfreut sich unter Nutzern einer steigenden Beliebtheit. Sie dient als intuitive Eingabemodalität, die im Kontrast zum traditionellen Tippen-und-Klicken steht. Die Spracherkennung spielt besonders bei der Erstellung mobiler Assistenzsysteme eine wichtige Rolle.
Mobile Assistenzsysteme wie etwa *Siri* (von Apple) oder *Google Now* (von Google) können schon jetzt einfache Fragen und Aufforderungen erkennen und verarbeiten. Beispielsweise kann man alleine durch Spracheingabe Termine erstellen oder Nachrichten per E-Mail versenden. Apple nutzt zur Realisierung von Siri aktive Ontologien (AOs) [Bel14].
Eine Ontologie ist eine Datenstruktur zur formalen Repräsentation eines Wissensbereichs. Dieser wird über verschiedene Klassen, Attribute und Beziehungen zwischen den Klassen abgebildet. Bei einer aktiven Ontologie wird das Wissen im Faktenspeicher abgespeichert. Der Faktenspeicher ist eine Datenbank, welche Fakten enthält. Ein Fakt ist die fundamentale Datenstruktur um Informationen darzustellen, diese ähneln dabei der Prädikatenlogik erster Ordnung.
Beim Hinzufügen von Fakten in den Faktenspeicher können zuvor definierte Regeln ausgelöst werden, die den Faktenspeicher modifizieren, daher „aktiv“ [Guz08]. Die von Siri genutzten aktiven Ontologien haben den Nachteil, dass sie manuell erstellt werden müssen.
Das übergeordnete Ziel ist es, die Intention des Nutzers bei der Spracheingabe zu erkennen und die Anfrage mit Hilfe passender Online-Dienste automatisiert zu beantworten. Beispielhafte Anwendungen sind Flugbuchungen oder Reservierungen für Hotelzimmer.

Ziel dieser Arbeit ist es, eine service-orientierte Architekur auf der Basis einer aktiven Ontologie zu entwerfen, welche natürlich gesprochene Anfragen an passende Online-Dienste weiterleitet und deren Antworten zurück gibt. Der Broker sucht anhand der Eingabe-AO nach passenden Diensten im Faktenspeicher und kontaktiert diese. Die zurückgegebenen Antworten werden aggregiert und können als ein einziger Antwort-Fakt in den Faktenspeicher gelegt werden.
Onlinedienste umfassen sowohl Webdienste als auch Webformulare. In dieser Arbeit werden die Webdienste SOAP (WSDL), XML-RPC und REST unterstützt.
Durch das Zwischenschalten dieser service-orientierten Architektur entkoppeln wir die Klient-Seite (Spracheingabe) von der externen Dienst-Seite (externe Online-Dienste). Diese Architektur wird „Broker“ genannt.

2. Grundlagen

In diesem Kapitel werden zunächst die wichtigsten Begriffe und Fachtermini erklärt. Beginnend mit einem Abschnitt über dienstorientierte Architekturen wird dann die fundamentale Technik erklärt, die hinter dem Broker steckt: Aktive Ontologien.

Anschließend wird ein Überblick über die verwendeten Webdienste, Webformulare und Internet-Protokolle präsentiert, welche für die hier entworfene Broker-Architektur relevant sind.

2.1. SOA

Die dienstorientierte Architektur (engl. *service-oriented architecture*) ist ein verbreiteter Architekturstil, der besonderen Wert auf voneinander unabhängige Dienste legt. Ein Dienst kapselt dabei Funktionalität, also ausführbaren Programmcode, mit definierten Ein- und Ausgabeschnittstellen [BEL+15].

Ein wichtiges Merkmal dieser Architektur ist die sogenannte „Blackbox"-Idee der einzelnen Dienste [BEL+15]. Gemeint ist, eine komplexe Struktur hinter einer vereinfachten Darstellung zu verbergen. Wie diese Struktur dabei aussieht ist für den Nutzer des Dienstes nicht ersichtlich und auch nicht relevant. Weil die Dienst-Schnittstelle zu Beginn festgelegt ist und nicht geändert wird, ist es möglich, dass die Funktionsweise eines Dienstes kontinuierlich verbessert werden kann, ohne dass der Nutzer sich darum kümmern muss, ob die Software auch weiterhin funktionieren wird.
Solange die gemeinsamen Schnittstellenstandards eingehalten werden, können sich Dienste auch in der Programmiersprache oder der Hardware unterscheiden.

Die entscheidenden Vorteile lassen sich in vier Punkten zusammen fassen [NL05, Seite 5ff].

- **Wiederverwendbarkeit** Dienste können in unterschiedlichen Anwendungen wiederverwendet werden
- **Effizienz** Entwickler können sich auf die Logik konzentrieren und müssen sich nicht mit Implementierungsdetails befassen
- **Lose Kopplung** Dienste sind plattformunabhängig und tauschen standardisierte Nachrichten aus, die jeder Dienst versteht
- **Teilung der Verantwortlichkeit** Entwickler mit wirtschaftlichen Kenntnissen programmieren die Firmenlogik, Entwickler mit technischem Detailwissen die Dienste. Beide nutzen die gemeinsame Dienstschnittstelle.

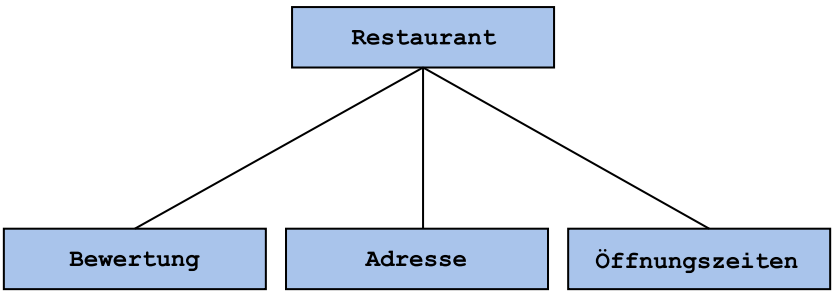

Abbildung 2.1.: Eine einfache Ontologie.

Der hier entworfene Broker orientiert sich an einer dienstorientierten Architektur durch die Nutzung von Online-Diensten. Webdienste und Webformulare registrieren/deregistrieren sich beim Broker. Klienten können über spezielle Befehle diese Online-Dienste finden und aufrufen.
Webdienste haben zudem den Vorteil, dass sie verbreitet, einfach zu integrieren und plattform-neutral sind. Aus diesem Grund sind Webdienste und dienstorientierte Architekturen gut kombinierbar [NL05, Seite 20ff].

2.2. Ontologien

Der Begriff „Ontologie“ ist schon sehr lange ein verbreiteter Begriff und wird bereichsübergreifend auch in anderen Disziplinen verwendet, nicht nur in der Informatik. Weil der Begriff Ontologie in den Disziplinen der Philosophie, Biologie und Informatik jeweils unterschiedliche Ideen bezeichnet [GOS09], wird hier genau beschrieben, was im Folgenden mit einer Ontologie gemeint ist.

In der Informatik nutzt man Ontologien, um Wissen in Form von Daten sowie Beziehungen zwischen den Wissenseinheiten untereinander abbilden zu können.

Eine Ontologie ist aus den beiden Grundbausteinen „Konzept“ (*concept*) und „Beziehung“ (*relationship*) aufgebaut. Das Konzept stellt eine Wissenseinheit dar. In Abbildung 2.1 ist eine einfache Ontologie dargestellt. Die Konzepte sind hier `Restaurant` sowie die zu dem Restaurant in Beziehung gesetzten Eigenschaften `Bewertung`, `Adresse` und `Öffnungszeiten`.

In Kontrast zu Datenbanksystemen wird in Ontologien auch die kontextuelle Bedeutung der gespeicherten Daten abgebildet. Diese Informationen ermöglichen es gemeinsam mit definierten Regeln einerseits Unstimmigkeiten im Wissensbestand zu erkennen (zum Beispiel Duplikate) und andererseits logische Schlüsse ziehen zu können. Damit ist es in Ontologien möglich, implizites Wissen aus den existierenden Konzepten und Beziehungen abzuleiten.

Es gilt zu beachten, dass streng genommen eine Differenzierung zur Taxonomie stattfindet, welche lediglich baumartige Hierarchien durch Klassendefinitionen und Ableitung abbilden [Gru92]. Ontologien ermöglichen es, weitaus komplexere Beziehungen zu projezieren, beispielsweise Graphen. Die zugrundeliegende Datenstruktur einer Ontologie ist also allgemeiner gefasst als die einer Taxonomie [VR03]. Es bleibt zu bemerken, dass die Ontologie mit der Taxonomie in manchen Veröffentlichungen vermengt oder gleichgesetzt werden. In dieser Arbeit wird durchgängig am Begriff Ontologie festgehalten.

2.3. EASIER Active Server

EASIER Active Server ist eine Laufzeitumgebung, welche alle aktiven Ontologien sowie den gemeinsamen Faktenspeicher verwaltet. Er stellt das Umfeld dar, in dem sich der Bro-

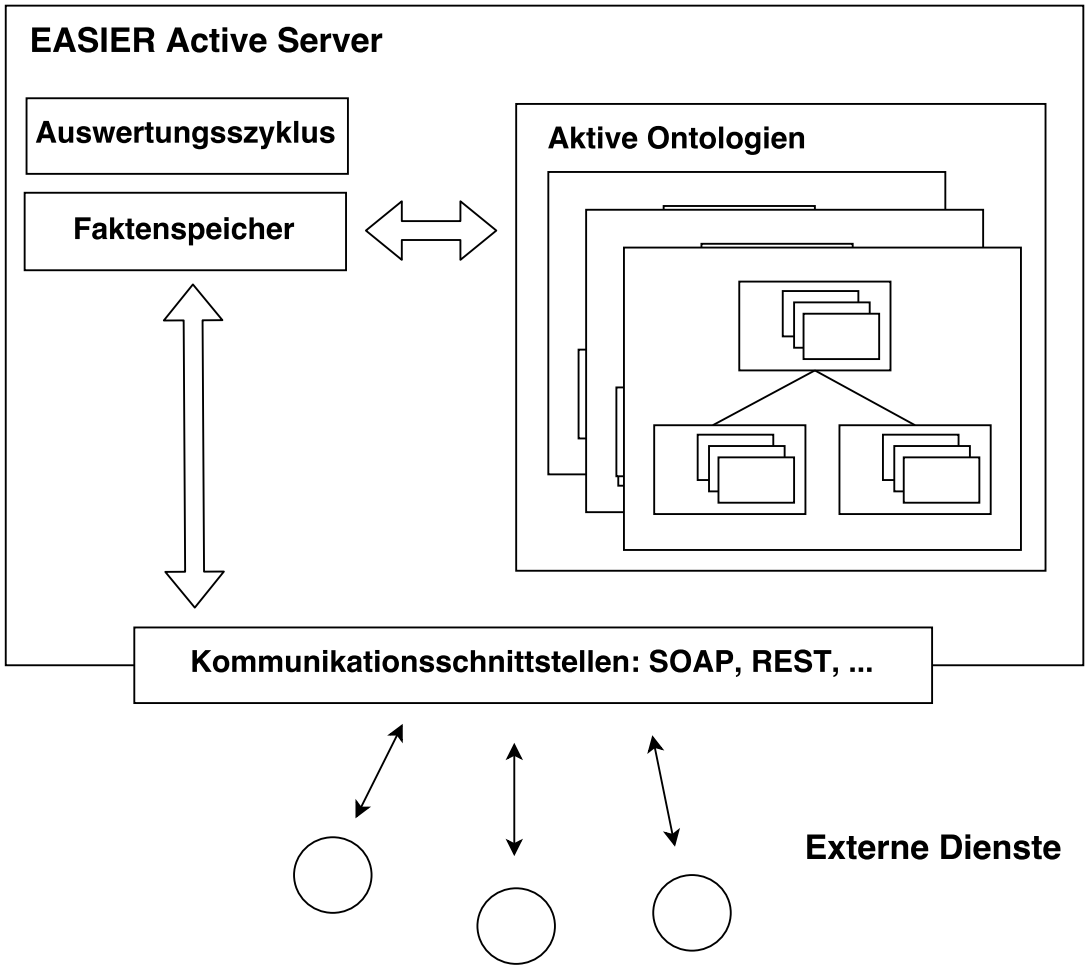

Abbildung 2.2.: EASIER Active Server

ker befindet. Alle 100 Millisekunden[1] wird der Faktenspeicher auf Änderung überprüft — dieser Zyklus heißt Auswertungszyklus. Eine Übersicht des EASIER Active Server ist in Abbildung 2.2 gegeben.
Aktive Ontologien können sich beim EASIER Active Server registrieren, um untereinander über den gemeinsamen Faktenspeicher zu kommunizieren. Der Broker ist eine registrierte, aktive Ontologie, die Dienstanfragen von anderen aktiven Ontologien entgegen nimmt, diese verarbeitet und die Ergebnisse zurück in den Faktenspeicher schreibt.
Durch den Faktenspeicher erreicht man die vollständige Entkopplung der einzelnen aktiven Ontologien und insbesondere die Entkopplung von aktiven Ontologien und externen Online-Diensten.
Der Faktenspeicher ist im Arbeitsspeicher abgelegt, eine Auslagerung in das Dateisystem ist jedoch einfach möglich.

2.4. Aktive Ontologien

Aktive Ontologien können als eine Weiterentwicklung der Ontologien gesehen werden — sie vereinen die beiden Gedanken der Informationsspeicherung und der Informationsverarbei-

[1] Dieser Zeitraum ist frei wählbar. 100 Millisekunden hat sich als praxistauglich in der Implementierung eines Mitarbeiters am IPD herausgestellt.

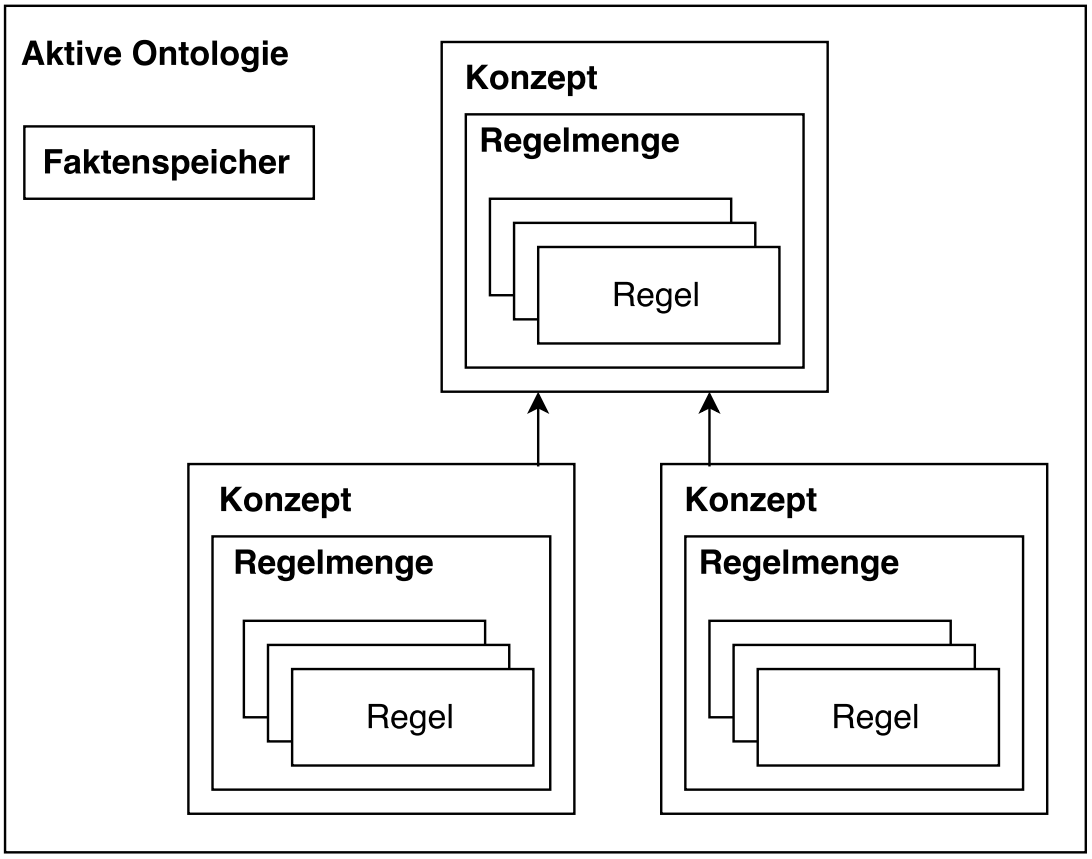

Abbildung 2.3.: Die Bestandteile einer aktiven Ontologie: Mehrere Konzepte und der Faktenspeicher.

tung in einer Softwarekomponente. Informationen werden gemeinsam mit einer Regelmenge als ein Konzept abgespeichert. Mit den sogenannten Regeln als Ausführungselementen wird es möglich, den Datenbestand zu manipulieren. Der Datenbestand wird durch den Faktenspeicher repräsentiert. In Abbildung 2.3 ist der strukturelle Aufbau einer aktiven Ontologie dargestellt. Die Eigenschaften von aktiven Ontologien decken sich weitgehend mit denen herkömmlicher Ontologien. Es gibt jedoch einige Unterschiede in der Konstruktion.
Der erste Unterschied ist, dass alle Konzepte einen eindeutigen Namen haben; Es dürfen keine zwei Konzepte mit gleichem Namen in der gleichen Ontologie vorkommen.
Beziehungen sind stets gerichtet und verbinden genau zwei Konzepte, nämlich ein Quell-Konzept mit einem Ziel-Konzept. Jeder Beziehung ist ein Typ zugewiesen, zum Beispiel eine `hat`-Beziehung zwischen einer `Bewertung` und einem `Restaurant`. Jedem Typ wiederum ist eine Menge von Attributen zugewiesen, welche die Beziehung genauer beschreibt. Der Typ `hat` besitzt das Attribut `istEinzeln`. Im Beispiel ist die Eigenschaft `Bewertung` auf `falsch` gesetzt, da mehrere Bewertungen für ein Restaurant erlaubt sind. Die aktive Ontologie ist in Abbildung 2.4 dargestellt.

2.4.1. Faktenspeicher

Der Faktenspeicher ist der gemeinsame Speicherort für alle Fakten. Da der Faktenspeicher als Einzelstück konzipiert ist, wird zur Ausführungszeit sichergestellt, dass es immer nur eine einzige Instanz des Faktenspeichers gibt, nicht mehrere. Die wichtigste Funktion des

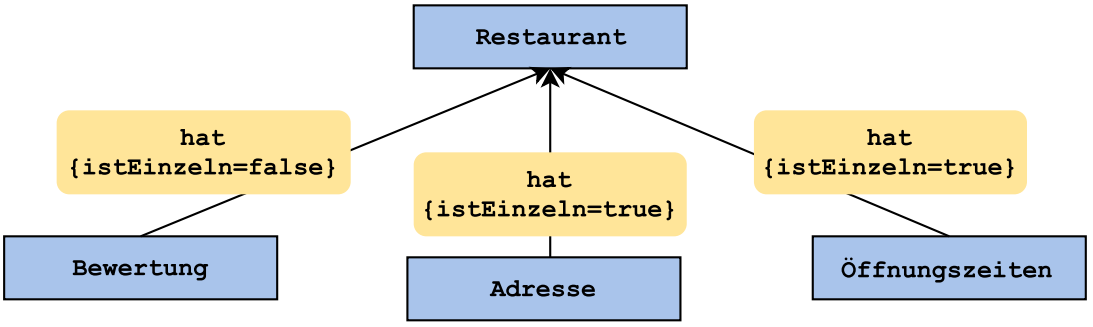

Abbildung 2.4.: Beispiel einer aktiven Ontologie. Konzepte sind blau hervorgehoben, Beziehungen gelb. Allen gerichteten Beziehungen ist ein eindeutiger Typ zugewiesen.

Faktenspeichers ist es, einfache Schnittstellen und Methoden anzubieten, um Fakten kontrolliert abzulegen und wiederauffinden zu können. Auf dem EASIER Active Server laufen mehrere unterschiedliche Ontologien, welche unabhängig voneinander Fakten schreiben und lesen können. Der Faktenspeicher stellt somit das „Kommunikationsmedium“ dar, da eine erste Ontologie einen Fakt ablegt und dieser dann von einer zweiten Ontologie gelesen oder manipuliert werden kann.

Gleichzeitig wird durch eine vorgegebene Schnittstelle sichergestellt, dass im Faktenspeicher nur gültige Fakten abgelegt sind. Wie gültige Fakten definiert sind ist im Abschnitt 2.4.3 beschrieben. Um diesen Anforderungen gerecht zu werden, offeriert der Faktenspeicher eine standardisierte Zugriffsschnittstelle, welche im folgenden Abschnitt kurz erläutert wird.

Schreiben- und Lesen

Da die komplette Schnittstelle durchaus sehr umfangreich ist, sind hier nur die sieben wichtigsten öffentlichen Methoden skizziert.

- `static FactStore FactStore.getInstance()` Diese statische Methode gibt das einzige Exemplar des Faktenspeichers zurück.
- `void writeFact(Fact factToWrite)` Diese Methode schreibt einen Fakt in den Faktenspeicher.
- `Map<Fact, Map<Variable, Fact>> readFacts(Fact factPattern, FACT_TYPE type)` Durch Unifikation werden alle abgespeicherten Fakten zurückgegeben, die dem Muster entsprechen. Mit `FACT_TYPE` kann zwischen Ereignis-Fakten und persistenten Fakten unterschieden werden.
- `void removeFacts(Fact factPattern)` Alle Fakten des Faktenspeichers, die mit dem Fakt `factPattern` unifizierbar sind, werden aus diesem permanent entfernt.
- `void overwriteFact(Fact factToWrite, Fact factPattern)` Alle Fakten, die mit dem Muster `factPattern` unifizierbar sind werden durch den Fakt `factToWrite` ersetzt.
- `void scheduleFact(Fact factToWrite, Date date)` Zu einem definierten Zeitpunkt in der Zukunft wird ein Fakt in den Faktenspeicher geschrieben.
- `Set<Fact> getFacts()` Gibt die Menge aller im Faktenspeicher enthaltenen Fakten zurück.

2.4.2. Auswertungszyklus

Jedes in einer aktiven Ontologie gespeicherte Konzept besitzt eine Menge von Regeln. Jede Regel hat zwei Bestandteile: Eine Bedingung und eine Aktion. Die Aktion kann hierbei mit regulären Java-Ausdrücken definiert werden. Interessanter sind die Regelbedingungen: Sie sind immer boolesche Ausdrücke und werden immer zu entweder `wahr` oder `falsch` ausgewertet.

Die Definition von Bedingungen wird aus [Guz08, Seite 45] übernommen. Die zugehörige EBNF-Grammatik ist im Quelltextausschnitt 1 dargestellt.

Quelltextausschnitt 1: EBNF-Grammatik für Regelbedingungen

```
<rule_condition> ⇒ <Boolean_expression>
<Boolean_expression> ⇒ <term> | <term> <add_op> <
   Boolean_expression>
<term> ⇒ <constant> | ( <expression> ) | <store_check>
<add_op> ⇒ OR | '||'
<mult_op> ⇒ AND | '&&'
<not_op> ⇒ NOT | '!'
<store-check> ⇒ FactStore.checkFact(PATTERN) | FactStore.
   checkEvent(PATTERN)
<constant> ⇒ true | false
```

Innerhalb eines Auswertungszyklusses werden alle Regelbedingungen einmal ausgewertet und — sofern die Auswertung positiv ausfällt — die definierte Aktion der Regel ausgeführt.

Der Faktenspeicher einer aktiven Ontologie wird in jedem Auswertungszyklus auf Veränderungen überprüft. Veränderungen entstehen, wenn neue Fakten hinzugefügt oder ersetzt aber auch wenn vorhandene Fakten entfernt werden. Die zeitliche Länge eines Auswertungszyklusses ist für jede aktive Ontologie frei wählbar. Exemplarisch wird eine Zeitspanne von 100 Millisekunden genommen.

Innerhalb eines Auswertungszyklusses gibt es wie in Abbildung 2.5 dargestellt drei ausgewiesene Schritte:

1. Vorverarbeitung
2. Regelauswertung und Aktion
3. Aufräumen

In dem ersten Schritt, dem Vorverarbeitungsschritt, wird überprüft ob sich der Inhalt des Faktenspeichers verändert hat. Nur wenn das auch tatsächlich der Fall ist, gelangt man in den zweiten Schritt: den Auswertungsschritt. In dieser Phase werden alle Regelbedingungen kontrolliert. Die definierten Aktionen werden ausgeführt, wenn die Regelbedingungen zu *wahr* auswerten.
Im dritten Schritt, dem Aufräumen, wird der Faktenspeicher von überflüssigen Fakten bereinigt. Unter anderem werden hier alle Ereignisse (sprich: Ereignis-Fakten) aus dem Faktenspeicher gelöscht. Ein einzelner Auswertungszyklus endet hier, und der nächste beginnt.

Im Folgenden werden die Regelbedingungen genauer betrachtet. Man kann grob zwischen drei verschiedenen Arten unterscheiden:

1. Einfache Regelbedingungen
2. Kombinierte Regelbedingungen
3. Regelbedingungen mit Variablen

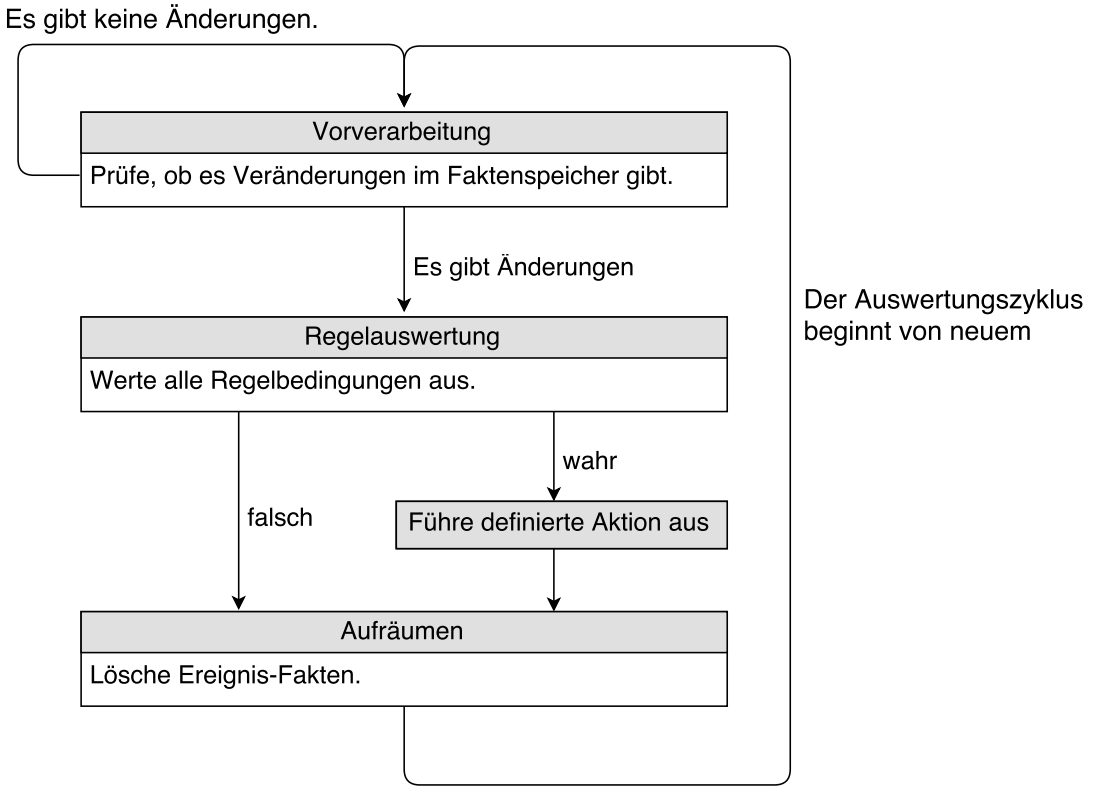

Abbildung 2.5.: Der Auswertungszyklus

Name	BenachrichtigeNeuePizza
Bedingung	`FactStore.checkEvent(product(pizza, $, $))`
Aktion	`System.out.println("Es gibt neue Pizzen");`

Tabelle 2.1.: Regel, die über neue Pizzen benachrichtigt

Einfache Regelbedingungen

Einfache Regelbedingungen sind Bedingungen, welche aus nur einem einzigen Ausdruck bestehen. Der Faktenspeicher bietet zwei Möglichkeiten, um nach Bedingungen zu prüfen. Diese sind:

- `FactStore.checkFact(Fact pattern)`
- `FactStore.checkEvent(Fact pattern)`

Es ist wichtig anzumerken, dass obige Methoden die beiden einzigen direkten Werkzeuge sind, um Bedingungen prüfen zu können.
Möchte man beispielsweise die Bedingung konstruieren, dass ein bestimmter Zahlen-Fakt einen gewissen Wert überschreitet, muss man wie in der Regel in Tabelle 2.2 vorgehen.

Regel 2.1 prüft, ob es im Faktenspeicher Produkte der Kategorie `pizza` gibt. Wenn dies der Fall ist, so wird *„Es gibt neue Pizzen“* auf der Konsole ausgegeben.

Regel 2.2 gibt alle Personen aus, die älter als 40 Jahre sind. Diese Regel produziert durch den gezielten Einsatz benannter Variablen eine aussagekräftigere Ausgabe. In der Aktion kann beliebiger Java-Code verwendet werden.

Name	PersonenÜberVierzig
Bedingung	`FactStore.checkFact(person($age, $name))`
Aktion	`if ($age > 40) { System.out.println("$name ist älter als 40."); }`

Tabelle 2.2.: Regel, die alle Personen über 40 ausgibt.

Name	BenachrichtigeNeuePizzaMitBeschreibung
Bedingung	`FactStore.checkFact(product($produkt, $name, $preis))` AND `FactStore.checkEvent(print($produkt))`
Aktion	`System.out.println("Wir haben das Produkt $produkt mit dem Namen $name zum Preis von $preis Euro im Sortiment.");`

Tabelle 2.3.: Bedingungen können mit AND, OR und NOT verknüpft werden.

Kombinierte Regelbedingungen

Es ist in der aktiven Architektur möglich, mehrere einfache Regelbedingungen zu komplexeren zu kombinieren. Nach der oben eingeführten EBNF-Grammatik für Regelbedingungen werden die Operationen Disjunktion (OR, ||), Konjunktion (AND, &&) und Negation (NOT, !) unterstützt.

Regelbedingungen mit Variablen

Eine Aktion ist ausführbahrer Code in der Zielsprache, hier Java. Sie kann eine Ausgabe tätigen, mit anderen Klassen kommunizieren oder den Faktenspeicher manipulieren.
Die Regel in Tabelle 2.3 enthält zwei mit `AND` verknüpfte Bedingungen. Die Variablen `$produkt`, `$name` und `$preis` werden hier instanziiert.

2.4.3. Fakten

Die fundamentale Datenstruktur der Wissensabbildung sind Fakten. Als Fakten sind alle Informationen modelliert, die für den Broker wichtig sind. Zusätzlich wird auch die Suchanfrage und deren Parameter über einen Fakt im Faktenspeicher repräsentiert.
Fakten ähneln in ihrem Aufbau sehr der Prädikatenlogik erster Ordnung. Fakten sind in vier verschiedene Klassen aufgeteilt (gemäß [Guz08, Seiten 41ff]):

- *Einfache Fakten*: Unteilbare, konstante Wert-Typen wie Zahlen oder Text. Im Text-Typ sind alle verfügbaren Zeichen erlaubt (UTF-8).
  Beispielfakten: **`1, 128, a, Beruf, Alan Turing`**
- *Komplexe Fakten*: Komplexe Fakten sind benannte Prädikate mit einem oder mehreren Parametern der Form *Name(Wert)*. Dadurch ist es möglich, dem Parameter-Fakt eine Bedeutung zuzuordnen oder mehrere Fakten in fester Reihenfolge zu gruppieren.
  Beispielfakten: **`Person(Alan Turing), Kunde(003, Alan, 23)`**
- *Listenfakten*: Listenfakten sind eine unsortierte Ansammlung von Fakten. Die Fakten müssen dabei im Typ nicht einheitlich sein.
  Beispielfakten: **`[id(1), name(Alonso)], [Informatik, Tennis, Skifahren], [Beruf, 5]`**
- *Variablen*: Variablennamen beginnen immer mit dem **$**-Zeichen. Anonyme Variablen ohne Instanziierung gibt es auch, sie werden durch das einfache **$** ohne nachfolgende

Tabelle 2.4.: Unifikation von einfachen Fakten

Fakt 1	Fakt 2	Unifizieren diese?
`42`	`42`	ja
`einfacher Fakt`	`einfacher Fakt`	ja
`ein Auto`	`ein Zug`	nein
`Name`	`Vorname`	nein

Zeichen dargestellt. Aus dem Englischem übernommen bezeichnet man solche anonyme Variablen oft als Wildcard-Variablen.
Beispielfakten: **`$alter, $priorität, $`** (wildcard)

Man unterscheidet überdies zwischen Ereignis-Fakten und persistenten Fakten. Ereignis-Fakten (engl. *event-facts*) sind vergänglich und werden nach dem aktuellen Auswertungszyklus aus dem Faktenspeicher entfernt. Persistente Fakten hingegen verbleiben so lange im Faktenspeicher, bis der Faktenspeicher die explizite Aufforderung erhält, diese zu löschen. Wenn im Folgenden die Rede von „Fakten“ ist, so sind immer persistente Fakten gemeint. Ereignis-Fakten werden mit „Ereignissen“ abgekürzt.
Im Folgenden wird oft vom Faktentyp (`FACT_TYPE` bzw. `EVENT_TYPE`) gesprochen. Es wird immer aus dem Kontext ersichtlich sein, ob die Unterscheidung zwischen Ereignis-Fakten und persistenten Fakten oder die Unterscheidung nach obiger Auflistung gemeint ist.

2.4.4. Unifikation

In diesem Abschnitt wird der Prozess der Unifikation genauer beschrieben. Bei der Unifikation geht es darum, zu einem gegebenem Fakt einen anderen äquivalenten Fakt zu finden. Man sagt dann, die zwei Fakten lassen sich unifizieren. Anhand der vier verschiedenen Fakt-Typen wird im Nachfolgenden darauf eingegangen, wann genau zwei Fakten als äquivalent und damit als unifizierbar angesehen werden.

Einfache Fakten Die Unifikation einfacher Fakten ist unkompliziert, da diese genau dann unifizieren, wenn sie Zeichen für Zeichen identisch sind. Zwischen Zahlen und Buchstaben wird hierbei nicht unterschieden, zwischen Groß- und Kleinschreibung aber sehr wohl. Ein Beispiel findet sich in Tabelle 2.4.

Komplexe Fakten Die Unifikation komplexer Fakten ist etwas umfangreicher. Damit komplexe Fakten unifizieren, müssen alle der folgenden Bedingungen zutreffen.

- Beide Fakten sind komplexe Fakten
- Die Funktoren (Namen) der Fakten sind identisch
- Die Anzahl der Parameter ist identisch
- Alle Parameter unifizieren in korrekter Reihenfolge

Ein Beispiel findet sich in Tabelle 2.5.

Listenfakten Die Unifikation von Listenfakten kann auf drei verschiedene Weisen erfolgen [Guz08, Seite 42]. Es gibt die Unifizierungsstrategien *strikt*, *teilweise strikt* und *teilweise kommutativ*. Wenn es im Text nicht anders beschrieben ist, wird von *teilweise kommutativ* als Standardstrategie ausgegangen.

Für *strikte* Listenunifikation müssen laut Guzzoni zwei Voraussetzungen erfüllt sein: Erstens müssen beide Listen die gleiche Anzahl an Elementen besitzen und zweitens

Tabelle 2.5.: Unifikation von komplexen Fakten

Fakt 1	**Fakt 2**	**Unifizieren diese?**
`person(alan)`	`person(alan)`	ja
`person(alan)`	`person(noam)`	nein
`person(alan, turing, 24)`	`person(alan, turing, 53)`	nein
`person(name(Alan), age(40))`	`person(name(Alan), age(40))`	ja

Tabelle 2.6.: Unifikation von Listenfakten

Fakt 1	**Fakt 2**	**Strikt?**	**Teilweise Strikt?**	**Teilweise Kommutativ?**
`[alan]`	`[alan]`	ja	ja	ja
`[a]`	`[b]`	nein	nein	nein
`[a, b]`	`[b, a]`	nein	ja	ja
`[a, b, c]`	`[a, b]`	nein	nein	ja
`[a(2), b(4)]`	`[a(2), b(3)]`	nein	nein	nein
`[]`	`[]`	ja	ja	ja

müssen alle Elemente in korrekter Reihenfolge unifizierbar sein. Nur wenn beide Bedingungen erfüllt sind, unifizieren die Listenfakten. Listenfakten unifizieren nie mit einfachen oder komplexen Fakten.

Bei der Strategie *teilweise strikt* ist nur gefordert, dass beide Listen gleich lang sind, also die gleiche Anzahl an Elementen kapseln. Im Unterschied zu *strikt* dürfen Listenelemente in beliebiger Reihenfolge unifizieren.

Geht man von *teilweise kommutativ* aus, dann unifizieren zwei Listen genau dann, wenn jedes Element der kürzeren Liste mit mindestens einem Element der längeren Liste unifiziert.

Zusammenfassend folgt also, dass zwei Listen, die unter *strikt* unifizieren, dies auch unter *teilweise strikt* tun. Unifizieren zwei Listen wiederum *teilweise strikt*, dann garantiert auch *teilweise kommutativ*. *teilweise kommutativ* ist in diesem Bild also die schwächste Unifikationsanforderung. Ein Beispiel findet sich in Tabelle 2.6.

Variablen Variablen können zunächst als Platzhalter gesehen werden, welche immer mit ihrem Gegenstück unifizieren. Wenn eine Variable mit einem anderen Fakt unifiziert, so wird die Variable mit diesem Fakt instanziiert. Die Variable nimmt also genau den Wert des unifizierenden Fakts an und wird an diesen gebunden. Ab diesem Zeitpunkt ist die Variable nicht mehr frei belegbar. Ein Beispiel findet sich in Tabelle 2.7.
Für den weiteren Ablauf der Unifikation ist der Wert dieser Variablen dann fest und kann nicht mehr geändert werden. Im Gegensatz zu benannten Variablen (z.B. `$a`, `$name`) können anonyme Variablen (ausschließlich der Spezial-Platzhalter `$`) nicht an einen Wert gebunden werden bzw. instanziiert werden [Guz08, Seite 42f]. Der Vorteil anonymer Variablen ist, dass sie mit allen Fakten (beliebig oft) unifizieren und gleichzeitig nie instanziiert werden. Der Nachteil ist, dass wegen einem fehlenden Namen der Zugriff auf den Wert der Variablen unmöglich wird. Eine anonyme Variable kann als Platzhalter für beliebige Fakten genutzt werden, die für den weiteren Ablauf nicht weiter von Bedeutung sind. Durch die Verwendung von Variablen wird der Unifikationsprozess deutlich aufwändiger, aber auch gleichermaßen mächtiger. Benannte Variablen ermöglichen es, komplexe Beziehungen auszudrücken und gezielte Abfragen an den gespeicherten Wissensbestand stellen zu können.
Angenommen im Faktenspeicher befindet sich eine Liste aller Angestellten. Für jeden Angestellten existiert ein Fakt folgender Struktur:

Tabelle 2.7.: Unifikation benannter und anonymer Variablen

Fakt 1	**Fakt 2**	**Unifizieren diese?**	**Instanziierungen**
`Turing`	`$var1`	ja	$var1 = Turing
`Turing`	`$`	ja	-
`a(b, c)`	`a($fst, c)`	ja	$fst = b
`a(b, c)`	`a($fst, d)`	nein	-
`[name(Alan), age(24)]`	`[name($n), age($a)]`	ja	$n = Alan, $a = 24

```
employee($firstname, $lastname, $jobtitle, $employee_id)
```

Mit dem Einsatz von Variablen sind unter anderem folgende Anfragen möglich:

- Gib eine Liste aller Manager zurück:
  `employee($firstname, $lastname, manager, $employee_id)`
- Zwei oder mehr Angestellte, wobei der Vorname des einen dem Nachnamen des anderen entsprechen soll:
  `employee($firstname, $, $, $employee_id)`
  `employee($, $firstname, $, $employee_id)`

Wie man sieht, ermöglicht die Variablen-Instanziierung den Einsatz konstruktiver Wissensabfragen, die über boolesche Ja-Nein Anfragen weit hinausgehen (diese wären etwa: Das Wissen (ein Fakt) ist exakt so abgespeichert: Ja oder Nein.)
Unifikation ist der zugrundeliegende Mechanismus des EASIER Active Servers und insbesondere des Brokers, um auf das im Faktenspeicher abgelegte Wissen zugreifen zu können.

2.5. Webdienste

Ein Webdienst ist eine Softwareanwendung, die auf einem (entfernten) Server kontinuierlich läuft, Anfragen entgegennimmt und beantwortet. Die Anfragen werden im Gegensatz zum üblichen Stöbern im Netz nicht von Menschen, sondern von anderen Softwareanwendungen gestellt. Es handelt sich damit um eine klassische Maschine-zu-Maschine Kommunikation [HB04].

Webdienste haben eine vordefinierte Programmierschnittstelle (engl. *Application Programming Interface, API*), welche ausschließlich für Entwickler entworfen wurde, um auf den angebotenen Dienst einfach zugreifen zu können. In der Schnittstelle wird genau definiert, welche Methoden unterstützt werden. Es ist auch spezifiziert, welche Parameter übergeben werden müssen um bestimmte Ergebnisse zu erzielen.

2.6. Webformulare

Webformulare funktionieren ähnlich wie Webdienste: Es läuft permanent eine Software auf einem Server, welche Anfragen entgegennimmt und beantwortet. Im Kontrast zu Webdiensten ist die Schnittstelle für die Bedienung durch den Menschen vorgesehen. Deshalb gibt es ähnlich wie in einem Papierformular Textflächen, Eingabefelder und Auswahlboxen. In Abbildung 2.6 ist das Schema eines Webformulars dargestellt.

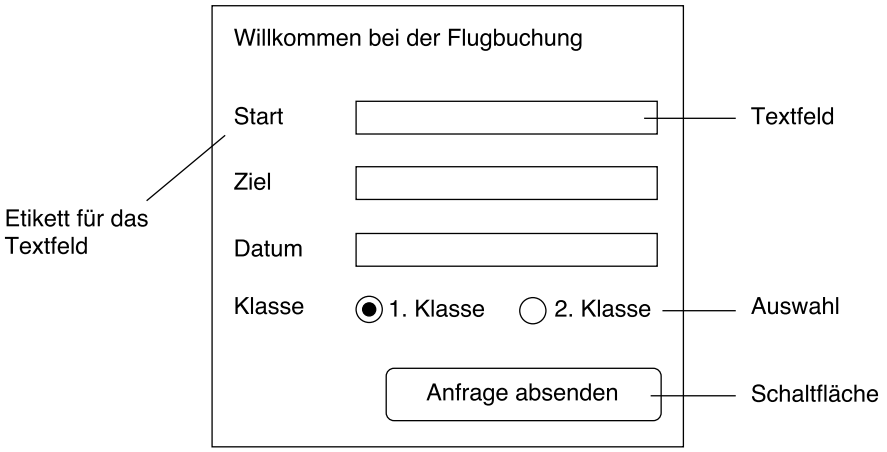

Abbildung 2.6.: Schematischer Aufbau eines Webformulars

Ein weiterer markanter Unterschied zu Webdiensten ist, dass sich Anfragen an Webformulare häufig wie in einem Dialog über mehrere Seitenaufrufe hinweg ziehen. Das ist bei Webdiensten zwar prinzipiell auch möglich, aber nicht der Standardfall. Beispielsweise könnte ein Nutzer in der ersten Anfrage nach Restaurants in seiner Umgebung suchen. Nachdem in der ersten Antwort eine Liste aller Restaurants angezeigt wird, stellt der Nutzer fest, dass er heute asiatisch essen gehen möchte und präzisiert seine Anfrage.

Webseiten und auch Webformulare werden zur Zeit hauptsächlich in der Auszeichnungssprache HTML bzw. der neueren Version HTML5 erstellt.

HTML

Die Hypertext-Auszeichnungssprache (engl. *Hypertext Markup Language*) ist die primäre Sprache zur Beschreibung von Webseiten und damit auch Webformularen. Die Sprache basiert auf XML, wird ebenfalls vom *W3C* standardisiert und befindet sich momentan in der Version 5 (HTML5) [HBF+14]. Zu beachten ist, dass HTML lediglich für den (informativen) Inhalt einer Webseite verantwortlich ist, nicht jedoch für die Repräsentation des Inhalts. Damit sind beispielsweise Farbgebung und Schriftart nicht Teil eines HTML5-Dokumentes und werden stattdessen in einer dafür geschaffenen CSS-Datei abgelegt. Obwohl in der Praxis in unsauberer Programmierung zahlreiche Webseiten CSS-Code mit HTML-Code vermengen, so wird im Folgenden angenommen, dass HTML-Dateien ausschließlich aus HTML5-Komponenten gemäß dem W3C-Standard aufgebaut sind [HBF+14].

Ein Mitarbeiter am IPD erstellt aus den Webformularen eine aktive Ontologie. Die aktive Ontologie registriert sich wie ein Webdienst beim Broker. Potenzielle Dienstanfragen an den Broker werden dann durch einen Webdienst oder ein Webformular (repräsentiert durch die aktive Ontologie) befriedigt.

2.7. Protokolle

In dieser Arbeit werden mehrere Internet-Protokolle verwendet, um auf das Potenzial von Webdiensten und Webformularen zugreifen zu können.
Es gibt einerseits Protokolle wie WSDL, die einem Dienst dazu verhelfen, sich selbst zu

beschreiben und andererseits Kommunikationsprotokolle, die für den direkten Datenaustausch mit den Diensten genutzt werden. Zu den Kommunikationsprotokollen zählen unter Anderem XML-RPC, SOAP und REST.
Mit WSDL können sich Dienste beim Broker registrieren. Der Broker verwendet die Protokolle XML-RPC und SOAP sowie die REST-Technik, um Online-Dienste aufzurufen.

2.7.1. WSDL

WSDL (engl. *Web Services Description Language*) ist eine vom World Wide Web Consortium standardisierte Schnittstellenbeschreibungssprache um Funktionsumfang, Schnittstellen, Datentypen und Daten von Webdiensten zu beschreiben. Mit WSDL wird die von Webdiensten ausgehende Kommunikation in strukturierter Art und Weise genau spezifiziert.
Der Entwickler eines Dienstes kann ein WSDL-Dokument schreiben oder generieren lassen und dieses dann an einen Dienstnutzer weiterreichen. Der Dienstnutzer ist durch die genaue Spezifikation der Dienstschnittstelle in der Lage, den Dienst gezielt anzusprechen [CDK+02].

Ein WSDL-Dokument (.wsdl) enthält die folgenden sieben Attribute:

- **types** Die Beschreibung aller verwendeten Datentypen.
- **message** Eine abstrakte, typisierte Definition der übertragenen Daten.
- **operation** Eine abstrakte Beschreibung einer Aktion.
- **portType** Eine Menge von unterstützten Aktionen.
- **binding** Die Spezifikation für ein konkretes Protokoll und ein konkretes Datenformat für einen bestimmten Datentyp.
- **port** Ein Paar aus einem *binding* und einer Adresse (URI).
- **service** Die Menge aller definierten Ports.

Die klare Trennung der abstrakten Definitionen (*types*, *messages*, *portType*) von konkreten Protokollen und Datenformaten (*binding*, *port*, *service*) erlaubt uns die Wiederverwendung ersterer, selbst wenn sich die Implementierung in der Zukunft ändern sollte.
Da die Struktur dank dieser sauberen Aufteilung wegen zahlreichen Referenzen schnell unübersichtlich ist, gibt Abbildung 2.7 den strukturellen Aufbau wieder. Ein WSDL-Beispieldokument befindet sich im Anhang A.1. Das Beispiel beschreibt einen zur Zeit des Schreibens funktionstüchtigen Webdienst, der die lokale Zeit anhand der Postleitzahl zurück gibt.

WSDL bietet keine Möglichkeit zur Angabe von Dienstgüteparametern[2] oder die Klassifizierung von Diensten an, zum Beispiel anhand von vorgegebenen Kategorien, Taxonomien oder Ontologien. Ein Dienstnutzer kann also nicht ohne weiteres in einer Menge von WSDL-Dokumenten passende Dienste finden, wenn ihm nur die Dienstkategorie zur Verfügung steht, noch kann er Dienste anhand von Güteparametern unterscheiden.
Aus diesem Grund muss der Verzeichnisdienst sowohl Kategorie als auch mögliche QoS-Parameter (engl. *Quality of Service*) verwalten. WSDL ist lediglich zur Bekanntmachung der Dienstschnittstelle geeignet, nicht jedoch für den tatsächlichen Datenaustausch zwischen Dienstanbieter und Klient. Aus diesem Grund wird WSDL oft mit einer Datenaustauschsprache wie XML-RPC oder SOAP kombiniert. Im WSDL-Dokument ist definiert, über welches andere Protokoll der Datenaustausch stattfindet.

[2] Dienstgüteparameter sind bei Webdiensten zum Beispiel die Latenz, die Korrektheit der Resultate und die Ausfallwahrscheinlichkeit. Im Deutschen findet sich mittlerweile häufig auch der vom Englischen entliehene Ausdruck *Quality of Service* (QoS).

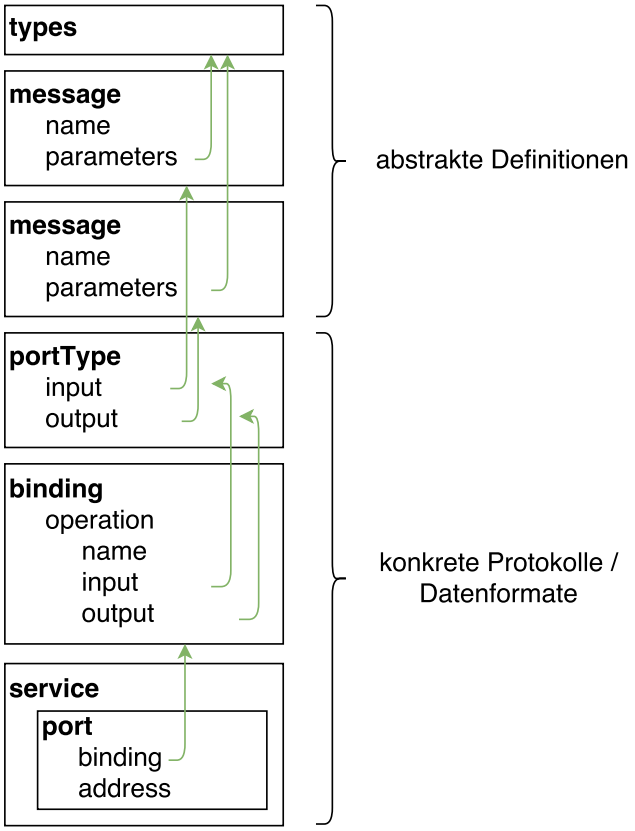

Abbildung 2.7.: Der strukturelle Aufbau einer WSDL-Datei. Grüne Pfeile symbolisieren namentliche Referenzen auf andere Elemente des gleichen Dokumentes.

Tabelle 2.8.: Datentypen in XML-RPC.

XML-RPC Typ	**Bedeutung**
int, i4	Integer (4 Byte)
double	Gleitkommazahl
boolean	Wahr/Falsch Variable
string	Zeichenkette
dateTime.iso8601	Datum und Uhrzeit
base64	Kodierte Binärdaten

2.7.2. XML-RPC

XML-RPC[3] ist die Spezifikation für den Aufruf einer fernen Prozedur (*Remote Procedure Call*). Zwischen Rechnern eines verteilten Systems werden XML-Nachrichten ausgetauscht, welche zur Ausführung von Programmcode auf einem anderen als dem Ursprungsrechner führen. Der Standard ist plattformunabhängig [Win99].
XML wird zur Darstellung verwendet und HTTP zur Datenübertragung. Das Ziel ist es, möglichst einfache und bekannte Datenformate zu wählen, die auf vielen Plattformen unterstützt werden um die Plattformunabhängigkeit zu erhöhen. Außer XML und HTTP sind keine weiteren Protokolle möglich.

Eine XML-RPC Nachricht ist eine HTTP-Post Anfrage. Eine Beispielanfrage ist in Quelltextausschnitt 2 zu sehen. Die Anfrage fragt nach dem Namen des 41. US-Staat.
Die Nachricht enthält immer einen Prozedurnamen (`methodName`) und die Aufrufparameter (`params`). XML-RPC unterstützt sechs verschiedene Datentypen. In der Beispielanfrage ist der Datentyp des Parameters Integer (`i4`) und der Wert `41`. Eine Liste der unterstützten Datentypen ist in Tabelle 2.8 ersichtlich. Wie eine mögliche Antwort des Servers aussehen kann, demonstriert Quelltextausschnitt 3.

Quelltextausschnitt 2: Beispiel einer XML-RPC Anfrage [Win99].

```
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
      </param>
   </params>
</methodCall>
```

Quelltextausschnitt 3: Beispiel einer XML-RPC Antwort [Win99].

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
```

[3] engl. *Extensible Markup Language Remote Procedure Call.*

```
</methodResponse>
```

XML-RPC ist schlank und schnell, jedoch in vielen Bereichen unflexibel bezüglich der Datenformate und -inhalte. Zusätzlich werden nur Aufrufe entfernter Prozeduren unterstützt, nicht jedoch die Übertragung von Nutzdaten. Um diese Mängel zu beheben, wurde das historische Nachfolgerprotokoll SOAP entwickelt.

2.7.3. SOAP

SOAP (engl. *Simple Object Access Protocol*) ist ein Netzwerkprotokoll zum Austausch von Daten, vorrangig unter Webdiensten. SOAP entstand aus der Weiterentwicklung von XML-RPC und unterstützt sowohl den strukturierten Austausch von Nutzdaten als auch ferne Prozeduraufrufe.
Als Anwendungsschichtprotokoll basiert es auf anderen Protokollen der gleichen Schicht, meistens auf HTTP[4], SMTP[5], FTP[6] und JMS[7] [CDK+02]. Auch HTTPS ist möglich, um eine verschlüsselte Punkt-zu-Punkt Datenübertragung zu ermöglichen. Für eine Ende-zu-Ende Verschlüsselung ist der WS-Security Standard empfohlen [OAS06]. SOAP kommt häufig dann zum Einsatz, wenn eine Kommunikation über mehrere Hops geschieht. Dabei können bei unterschiedlichen „Hops“ auch unterschiedliche unterliegende Anwendungsschicht- und Transportprotokolle verwendet werden, zum Beispiel HTTP für den ersten Hop, HTTPS für den zweiten und SMTP für den dritten.

Eine SOAP-Nachricht ist ein gewöhnliches XML-Dokument, welches aus vier Elementen zusammengesetzt ist, siehe auch Abbildung 2.8:

- **Envelope** Dieses Wurzelelement identifiziert ein XML-Dokument als eine SOAP-Nachricht.
- **Header** (optional) Der Header enthält Meta-Informationen zum Routing (nächste Hops), der Verschlüsselung und der Transaktion (z.B. um eine SOAP-Antwort eindeutig einer Anfrage zuordnen zu können).
- **Body** Im Body sind entweder die Nutzdaten oder die Anweisung für einen entfernten Prozeduraufruf. Der Empfänger ist dafür verantwortlich, den Inhalt entsprechend zu interpretieren.
- **Fault** (optional) Das Fault-Element beschreibt Fehler, die während der Übertragung entstanden sind; zum Beispiel den Versursacherknoten bei der Übertragung über mehrere Hops.

Ein Unterschied zu XML-RPC ist, dass Daten nicht mehr notwendigerweise über XML versendet werden müssen — alternative Datenformate wie zum Beispiel CSV[8] und JSON sind erlaubt.
SOAP wird vom *World Wide Web Consortium (W3C)* für den Nachrichtenaustausch mit Webdiensten explizit empfohlen [ML07].

2.7.4. REST

REST (engl. *Representational State Transfer*) ist ein Programmierparadima zur Kommunikation mit einem Webserver. Dabei wird eine spezielle URI mit definierten GET oder

[4] HyperText Markup Language
[5] Simple Mail Transfer Protocol
[6] File Transfer Protocol
[7] Java Message Service
[8] Comma-Separated Values oder auch Character-Separated Values

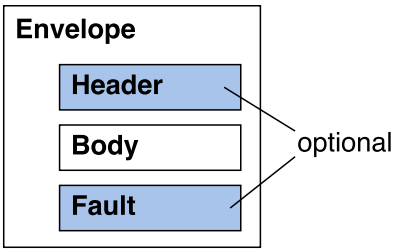

Abbildung 2.8.: Der strukturelle Aufbau einer SOAP-Nachricht. Das Header- und das Fault-Element sind optional.

POST Parametern aufgerufen und der Webseiten-Inhalt heruntergeladen und interpretiert. Hierbei wird das Protokoll HTTP zum Datenaustausch verwendet. Die Darstellung der Nutzdaten kann in beliebigen Protokollen erfolgen. REST ist vom Darstellungsprotokoll des Webseiten-Inhalts unabhängig, der Aufrufer muss jedoch wissen, worum es sich handelt.

3. Verwandte Arbeiten

Viele Arbeiten haben sich bereits mit dem Modularisieren externer, wiederverwendbarer Dienste und der einheitlichen Kommunikation mit selbigen auseinandergesetzt. Einige Arbeiten zielen dabei auf Agentensysteme ab, bei der alle Teilnehmer strukturell gleich gestellt sind, um ein gemeinsames Problem zu lösen. Dieser Ansatz ist sehr interessant, da man dort auf ähnliche Herausforderungen stößt wie bei dem Broker. Die dort verwendeten Herangehensweisen waren Inspiration für den Aufbau des Brokers.

Die verwandte Arbeit, die wohl am engsten mit dieser zusammenhängt, ist *Active: A unified platform for building intelligent applications* von Didier Guzzoni [Guz08]. Er war einer der ersten, der das Potential aktiver Ontologien erkannt hat.
Aus dieser Dissertation stammt die strukturelle Organisation der Fakten und des Faktenspeichers als Grundlage dieser Arbeit.

3.1. Broker-Architekturen

Broker sind Vermittler zwischen Dienstanbietern und Entwicklern, die einen bestimmten Dienst suchen. Im ersten Schritt ermöglicht der Broker das Auffinden von passenden Diensten und im zweiten Schritt kontaktiert er den Dienstanbieter. Die Ergebnisse werden dann an den Aufrufer zurückgegeben [DHC13].

3.1.1. Active Framework

Guzzoni beschreibt in [Guz08] ein Rahmenwerk für intelligente Assistenzsysteme, das Active Framework. Endnutzer können in ein Mobiltelefon hineinsprechen, um einen Flug zu buchen. Anschließend werden online Flugbuchungsdienste zur Erfüllung der Anfrage gesucht und kontaktiert, woraufhin verfügbare Flüge dem Nutzer präsentiert werden.

In Active ist Wissen in aktiven Ontologien abgespeichert, wie sie in den Grundlagen beschrieben sind.
Active nutzt zur Verwaltung von Diensten ähnlich wie der hier beschriebene Broker einen Faktenspeicher. Dienste registrieren sich durch das Ablegen von Fakten. Bei einer Dienstanfrage wird versucht, den Anfrage-Fakt mit einem registrierten Dienst-Fakt zu unifizieren und den Dienst dann aufzurufen.
Im ersten Teil ist beschrieben, wie Konzepte der aktiven Ontologie untereinander kommunizieren können. Anschließend ist die Dienstverwaltung bei Active geschildert.

Regelname	Ziel1HörtAufNachrichten
Bedingung	`FactStore.checkEvent(pipe($quelle, Ziel1, $daten, $optional))`
Aktion	`System.out.println("In Ziel1 kam die Nachricht $daten an");`

Tabelle 3.1.: Regel, um auf Nachrichten beliebiger Quellen und beliebigem Inhalt zu hören.

Kommunikation

Beim Active Framework setzt Guzzoni auf Pipes als Kommunikationskanäle zwischen Konzepten einer Ontologie und asynchrone, entfernte Prozeduraufrufe. Eine Pipe besteht aus einer Quelle, einem Ziel, dem Dateninhalt und einer Liste an optionalen Parametern.
Pipes funktionieren wie Regeln. Sie werden als komplexe Fakten mit genannten Attributen von der Ziel-Ontologie angelegt und in den Faktenspeicher geschrieben:

```
pipe($quelle, $ziel, $daten, $optionale_parameter)
```

Broadcasts sind möglich, in dem das $ziel-Argument auf die anonyme Variable $ gesetzt wird [Guz08, Seite 60ff] . Mit einer Regel wie in Tabelle 3.1 kann ein Empfänger definieren, dass er alle an ihn gerichteten Nachrichten empfangen möchte.

Entfernte Prozeduren werden über einen standardisierten `invoke`-Fakt aufgerufen, welchen den Prozedurnamen sowie die Parameter enthält. Der Aufgerufene hat eine Regel für alle seine öffentlichen Prozeduren angelegt, in der Aktion der Regel ist definiert, was bei der entsprechenden Prozedur zu tun ist.
Nach Abarbeitung legt der Aufgerufene einen standardisierten Ergebnis-Fakt (`invoked`) ab. Der Aufrufer wiederum legt eine Regel an, welche auf den Ergebnis-Fakt wartet.
In der Active-Architektur sind alle Ereignisse asynchron, weshalb ein Timeout-Mechanismus beim Aufrufer erforderlich ist. Der Timeout wird realisiert, in dem ein Ergebnis-Fakt mit dem Status „Timeout“ verzögert in den Faktenspeicher geschrieben wird
(durch `FactStore.scheduleFact(Fact factToWrite, Date date)`).

Ich finde dieses Vorgehen sehr elegant, da keine neuen Methodiken zum Faktenspeicher oder den Ontologien hinzugefügt werden, sondern der Kommunikationsmechanismus der Pipes vollständig auf Fakten und Regeln aufbaut.

Dienstverwaltung

Wie in der hier beschriebenen Architektur existiert auch bei Active ein Broker als Vermittler zwischen Klienten und Dienstanbietern.
Klienten delegieren ihre Dienstaufrufe an den Broker. Der Broker implementiert die notwendige Logik, um folgende Schritte auszuführen [Guz08, Seite 100ff]:

1. Liste alle passenden Dienste
2. Wähle den passendsten Dienst aus
3. Rufe den Dienst auf
4. Sortiere die Antworten und gib diese zurück an den Aufrufer

Der Klient muss lediglich beschreiben, welche Dienstkategorie er wünscht, und nicht mehr welchen konkreten Dienst er ansprechen möchte. Die angebotenen Dienstkategorien sind normiert, das heißt es existiert eine feste Menge an eindeutig unterscheidbaren Kategorien. Neue Dienste müssen genau einer Kategorie bei der Registrierung im Dienstverzeichnis zugeordnet werden.

Bei der Dienstregistrierung werden drei Parameter angegeben:

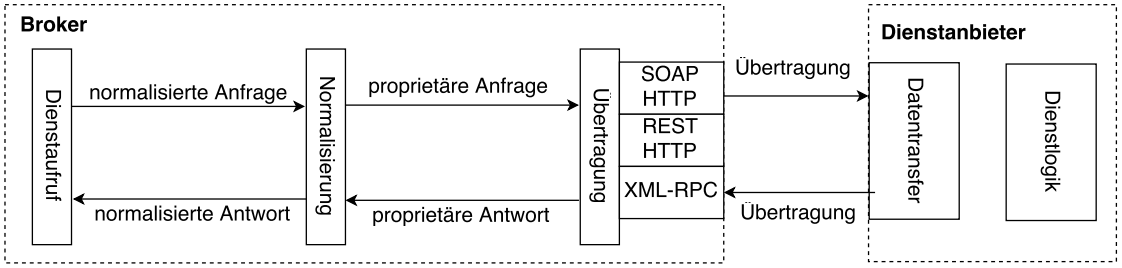

Abbildung 3.1.: Datennormalisierung und Übertragung vom Broker zum Dienstanbieter [Guz08, Seite 104].

- Die eindeutige Kategorie und eine optionale Liste an Attributen, die den Dienst genauer definieren, zum Beispiel Dienstgüteparameter oder eine Verfeinerung der Kategorie. Optional kann der Aufrufer auch angeben, ob lediglich ein einziger Dienst aufgerufen werden soll oder mehrere gleichzeitig. Ersteres ist nur dann sinnvoll, wenn beispielsweise eine Email abgesendet wird. Das parallele kontaktieren mehrerer Dienste der gleichen Kategorie kann sinnvoll sein, wenn man die Ergebnisse mehrerer Anbieter vergleichen möchte.
- Eine Transformation der Schnittstelle des proprietären Dienstes in die Schnittstelle der Kategorie, um eine Kommunikation zu ermöglichen.
- Das Kommunikationsprotokoll, mit dem der Dienst anzusprechen ist. Wenn ein Dienst beispielsweise SOAP nutzt, muss eine WSDL Datei angegeben werden. Bei CORBA oder REST sind die Adresse und das verwendete Protokoll wichtig. Bei der Präsentation der Ergebnisse ist es möglich, dass Dienst-Empfehlungen dritter sowie Dienstgüteparameter berücksichtigt werden.

Die proprietäre Schnittstelle des Dienstanbieters ist höchstwahrscheinlich nicht so definiert wie die Schnittstelle des Brokers, weshalb eine Datennormalisierung notwendig ist. Hierfür wird eine sogenannte *fact transformation language* (FTL) genutzt, um einen Ursprungsfakt in einen Ziel-Fakt zu konvertieren. Die *fact transformation engine* (FTE) führt diese Sprache aus.
Dienstanbieter müssen zwei FTL-Prozeduren angeben: Eine, um normalisierte Parameter der Dienstanfrage in proprietäre umzuwandeln und eine zweite Prozedur, um die Antworten des Dienstanbieters zu normalisieren [Guz08, Seite 104]. Eine Veranschaulichung des Ablaufs ist in Abbildung 3.1 dargestellt.

Ich finde den modularen Umgang mit den Übertragungs-Protokollen gut, bin jedoch der Meinung, dass man auch Module unterstützen sollte, die nicht unmittelbar mit Webdiensten zusammenhängen, zum Beispiel Webformulare.

3.1.2. UDDI

UDDI[1] ist ein standardisierter, plattform-unabhängiger Verzeichnisdienst für Webdienste, welcher auf XML basiert [OAS04] [CDK+02].

Die Struktur von UDDI lässt sich in drei Komponenten aufgliedern:

- **White Pages** enthalten Informationen über das Unternehmen oder die Entwickler, die einen Dienst publizieren.

[1] Universal Description, Discovery and Integration

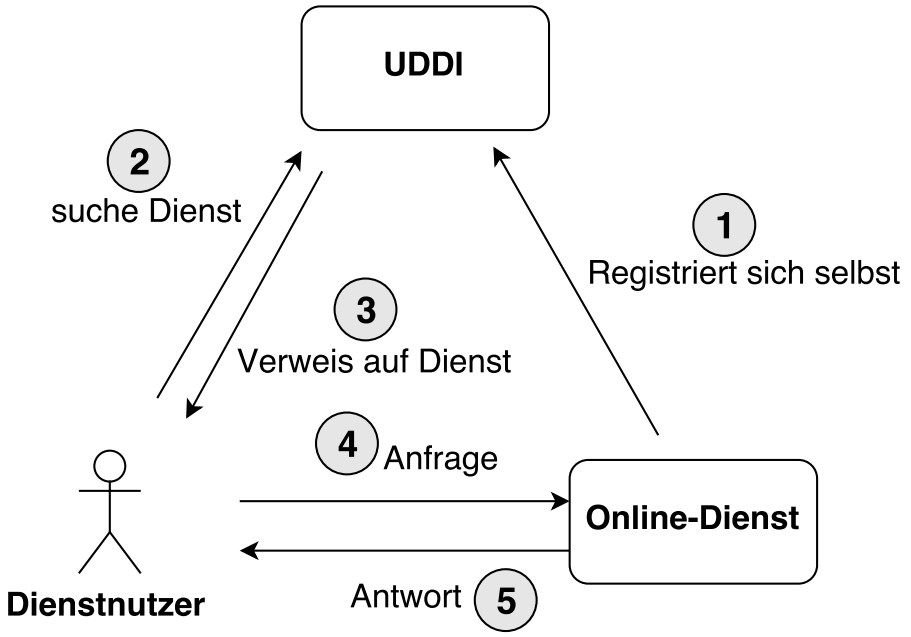

Abbildung 3.2.: Funktionsweise von UDDI

- **Yellow Pages** nehmen eine Klassifizierung der Dienste anhand eines internationalen Standards[2] vor. Diese Einteilung ist wichtig, um Dienste später im Verzeichnis einfach auffinden zu können.
- **Green Pages** enthalten technische Informationen. In diesen ist beschrieben, wie man auf den entsprechenden Dienst genau zugreifen kann: Dienstadresse, Parameter, Schnittstellen.

Sogenannte tModels[3] speichern in UDDI eindeutige Dienstanbieter ab. Es werden beispielsweise Dienst-Typen, die Spezifikation eines Dienstes, verwendete Protokolle und Namensräume abgespeichert [OAS04, Abschnitt 1.6.4]. Zudem werden auch die Protokolle WSDL oder XSD[4] in tModels verwendet, um Verträge und Verhalten zu spezifizieren und damit die Interoperabilität gewährleisten zu können. Ein tModel muss mindestens die folgenden Attribute definieren [OAS07]:

- Name, formattiert als URL
- Kurzbeschreibung
- Eine ausführlichen Beschreibung bzw. einen Link darauf
- Alle referenzierten tModels haben einen Namen und einen Typen

In Abbildung 3.2 ist die Funktionsweise des Dienstverzeichnisses UDDI dargestellt.

1. Online-Dienste registrieren sich bei einer dafür ausgelegten Datenbank und machen sich damit publik. Hier wird üblicherweise das Protokoll WSDL verwendet.

[2] Zum Beispiel: North American Industry Classification System (NAICS) und United Nations Standard Products and Services Code (UNSPSC) [CDK+02].
[3] Technical Model. Im Appendix A.2 ist ein Beispiel dargestellt.
[4] XML Schema Definition

2. Andere Entwickler (hier als Dienstnutzer bezeichnet) können die Datenbank nach für sie passenden Diensten durchsuchen.
3. Existiert ein passender Dienst in der Datenbank, so können die Details eines Dienstes wie die Aufrufmethodik und die URL eingesehen werden. Alle in der Datenbank registrierten Dienste beschreiben sich selbst im Datenformat WSDL.
4. Der Dienstnutzer kann den Online-Dienst nun autonom ohne weitere Intervention durch die Datenbank ansprechen.
5. Die Online-Dienste antworten spezifikationsgemäß.

UDDI bleibt durch die Verwendung von tModels unabhängig, was (zukünftige) Webdienstprotokolle anbelangt. Gleichzeitig kann auch hier die Dienstkategorie auf verschiedene Arten angegeben werden und auch Unternehmensinformationen werden je Dienst gespeichert. Durch das Aufteilen in die drei Kategorien White Pages, Yellow Pages und Green Pages werden die Informationen zu Unternehmen, Diensteigenschaften und Dienstschnittstelle ordentlich separiert.
Ein Unterschied zu meinem Broker ist, dass dieser nicht nur passende Dienstanbieter mit Dienstsuchenden vermittelt, sondern zusätzlich auch die konkreten Dienstanfragen delegiert und Anfrageergebnisse zurückliefert. Klienten von UDDI müssen sich selbst darum kümmern, den erhalten Dienstanbieter mit dem korrekten Protokoll anzusprechen.

3.1.3. CORBA

CORBA[5] ist ein von der OMG[6] entwickelter, objektorientierter Standard zur plattformunabhängigen Kommunikation verteilter Systeme [OMG12].

Der Object Request Broker (ORB) ist der Vermittler, welcher Plattform- und Programmiersprachenunabhängigkeit innerhalb des verteilten Systems ermöglicht. CORBA definiert ORBs für einige Programmiersprachen[7], für viele weitere Sprachen gibt es von Dritten erstellte ORBs. Der ORB verbirgt Detailwissen über Programmiersprachen und Protokolle wie beispielsweise die Bytereihenfolge [Min97].

Die wichtigste Komponente eines ORBs ist der IDL-Übersetzer. Schnittstellen werden in der generischen Interface Definition Language (IDL) spezifiziert. Ein IDL-Übersetzer generiert aus IDL den Quelltext in der Zielsprache mit Stubs und Skeletons, so dass entfernte Aufrufe wie lokale Aufrufe für den Programmierer erscheinen.
Die Komplexität des IDL-Übersetzers hängt hauptsächlich von der Komplexität der Zielsprache ab, so ist der IDL-Übersetzer nach Java im Vergleich zu dem C-Äquivalent sehr einfach aufgebaut. Insbesondere hängt es auch davon ab, ob die Zielsprache Objektorientierung nativ unterstützt, oder diese nachgebildet werden muss.

Zur Kommunikation zwischen den ORBs können herstellerspezifische Protokolle verwendet werden. Um die Kommunikation zwischen Knoten verschiedener Hersteller zu ermöglichen, wurde in CORBA 2.0 das General Inter-ORB Protocol (GIOP) spezifiziert [OMG08]. GIOP ist ein abstraktes Protokoll, für welches mehrere konkrete Protokolle existieren, IIOP ist dabei mit Abstand am weitesten verbreitet:

- IIOP — Bildet GIOP-Nachrichten auf die TCP/IP-Schicht ab.
- SSLIOP — IIOP über SSL, um Verschlüssenung und Authentifizierung anzubieten.
- HTIOP — IIOP über HTTP.

[5] Common Object Request Broker Architecture
[6] Object Management Group
[7] Ada, C, C++, C++11, COBOL, Java, Lisp, PL/I, Python, Ruby und Smalltalk

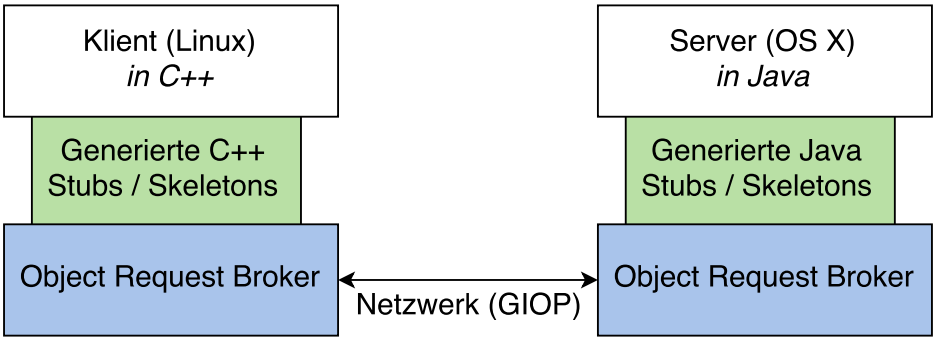

Abbildung 3.3.: Die Funktionsweise von CORBA.

Tabelle 3.2.: Eine Auswahl an CORBA-Diensten mit definierter IDL-Schnittstelle

Dienstname	**Funktion**
Naming Service	Auffinden eines CORBA-Objektes anhand eines eindeutigen Namen
Trading Service	Auffinden eines CORBA-Objektes anhand seiner Eigenschaften
Event Service	Ereignisbasierte N:M Kommunikation durch Push oder Pull
Life Cycle Service	Kopieren, Verschieben und Löschen von CORBA-Objekten

- ZIOP — Komprimierte Version (gezippt), um Bandbreite zu sparen.

In Abbildung 3.3 ist die Funktionsweise von CORBA illustriert. Unterschiedliche ORBs (blau) generieren Programmiersprachenspezifischen Code (grün) für unterschiedliche Plattformen und Programmiersprachen. Die Kommunikation zwischen Klient und Server erfolgt ausschließlich über die Object Request Broker.

Ein Vorteil gegenüber Webdiensten ist, dass CORBA eine vordefinierte IDL-Schnittstelle anbietet, um häufig auftauchende Anwendungsfälle abzudecken. Dadurch können sich Entwickler stärker auf die Umsetzung der Firmenlogik konzentrieren. Eine Auswahl der vordefinierten Dienste ist in Tabelle 3.2 ersichtlich.

Neben CORBA existieren noch weitere verteilte Architekturen, um komplexe Online-Dienste zu erstellen. Viele davon sind proprietär und nur mit Komponenten des eigenen Systems kompatibel. Die bekanntesten finden sich in Tabelle 3.3.
Diese Technologien unterscheiden sich großteils nur in der Implementierung, in vielen Fällen sind lediglich Methodennamen unterschiedlich [Raj98].
Java/RMI unterscheidet sich von den anderen Rahmenwerken und auch von UDDI/WSDL/SOAP durch die Besonderheit, dass auch Code ausgeliefert werden kann, d.h. die lokale Ausführung von entferntem Code oder die entfernte Ausführung von lokalem Code. Die mangelnde Unterstützung ist auf Sicherheitsrisiken und Programmiersprachenunabhängigkeit zurückzuführen [GKS02]. So eine Funktion könnte bei unserem Broker in der Kommunikation mit Webformularen interessant sein, zum Beispiel das Ausliefern von Javascript.

Eine Gegenüberstellung zwischen dem verteilten Rahmenwerk CORBA und Webdienst-Technologien (UDDI/WSDL/SOAP) ist in Tabelle 3.4. Der offensichtlichste Unterschied ist, dass CORBA eine vollständig objekt-orientierte Architektur repräsentiert, wohingegen Webdienste primär auf den Nachrichtenaustausch abzielen. Das Design von CORBA ist sowohl in den Endknoten als auch der Nachrichtenübertragung außerst generisch gehalten, was bei einfachen Umsetzungen zu erheblichen, relativen Verwaltungskosten führt. Gokhale et al. beschreibt in [GKS02], dass CORBA und Webdienste nicht komplementär, sondern orthogonal gesehen werden sollten. Beide Technologien konkurrieren also nicht gegeneinander, sondern ergänzen sich. Ich sehe das auch so, weshalb zu betrachten ist,

Tabelle 3.3.: Verteilte Rahmenwerke zur Erstellung von Online-Diensten

Technologie	**Plattform**
COM	Microsoft
DCOM	Microsoft (.NET)
COM+	Microsoft
EJB	Java
Java/RMI	Java, mit CORBA-Unterstützung
CORBA	plattform- und programmiersprachenunabhängig

wie der EASIER Active Broker ein CORBA-Modul parallel zu den weiteren Schnittstellen (Webdienste, Webformulare) anbieten soll.

3.1.4. Weitere

Al-Masri und Mahmoud haben in ihrer Arbeit [AMM08] einen etwas anderen Ansatz gewählt. Statt einen zentralen Ort, an dem Entwickler ihren Webdienst registrieren können, verfolgen sie die Idee, eine Suchmaschine WSCE[8] für Webdienste zu konstruieren. Die Autoren ziehen eine Analogie zu den ersten Kategorie-basierten Verzeichnissen für HTML-Webseiten, die nach der rapide zunehmenden Anzahl an Webseiten nicht mehr effizient waren — Suchmaschinen entstanden.

Existierende Suchmaschinen zeigen bereits WSDL-Dokumente in den Suchergebnissen an, können mit den Daten jedoch wenig anfangen, da sich die Struktur von WSDL und HTML (das Dateiformat, für welches die Suchmaschine optimiert ist) unterscheiden. WSCE durchsucht die Ergebnisse von HTML-orientierten Suchmaschinen und die UDDI Business Registry (UBI), eine Implementierung der UDDI-Spezifikation [CW04].

Ich sehe darin den Schritt in die richtige Richtung, aber es fehlt meiner Meinung nach noch an einer zuverlässigen Quelle der semantischen Informationen. Insbesondere kann man sich beim automatisierten durchstöbern von HTML oder WSDL nicht sicher sein, wie der Inhalt semantisch korrekt zu interpretieren ist.
Ding et al. schlagen vor, mit Semantic Web Documents[9] zu Beschreiben, welche Dokumente oder Dienste welche Bedeutung haben. Die Verbreitung und eine einheitliche Standardisierung fehlt jedoch noch [DFJ+04]. Ein Crawler kann beispielsweise nach einmaligem Finden eines WSDL-Dokumentes nicht feststellen, wie sich die Dienstgüteparameter zusammensetzen. Al-Masri et al. hat gezeigt, dass sich manche Dienstgüteparameter bereits beim Finden eines WSDL-Dokumentes berechnen lassen, etwa die Antwortzeit. Für die meisten Parameter müssen jedoch entweder Daten über mehrere Tage verfügbar sein und gespeichert werden oder es muss der Dienstanbieter die Information angeben [AMM07]. Genannte Beispiele sind etwa die Nutzbarkeit oder die Integrität. Bei WSCE ist diese Information nur verfügbar, wenn der Dienst in UBI gefunden wurde und der Veröffentlicher sie angegeben hat [AMM08].
Der Crawler muss ohne semantische Information durch Interpretation von umliegendem HTML herausfinden, welcher Kategorie ein Dienst zuzuordnen ist. Webdienste, die nicht mit SOAP angesprochen werden und demnach kein WSDL-Dokument (oder ein Dokument ähnlicher Funktion) generieren, können nicht gefunden werden.

Al-Masri und Mahmoud kommen zu dem Ergebnis, dass 63 % aller gefundenen Webdienste aktiv sind. Da WSCE nur in UBI und den bereits vorgefilterten Suchergebnissen von

[8] Web Server Crawler Engine
[9] Die Autoren nennen RDF und OWL als geeignete Semantic Web Documents.

Tabelle 3.4.: Vergleich von CORBA mit Webdiensten [GKS02].

	CORBA	**Webdienste**
Kommunikation	Objektorientiert	Nachrichtenaustausch
Standort-Transparenz	Keine. Klienten erhalten nur Objekt-Referenzen	Implizit in der URL codiert
Verzeichnis	Interface Repository (IR)	UDDI
Dienst-Entdeckung	Naming Service	UDDI
Firewall-Überwindung	CORBA Firewall Traversal	Der HTTP-Port ist gewöhnlich erlaubt.
Sicherheit	CORBA Security Service	Keine standardisierte Mechanismen. Ausweichung auf HTTPS und SSL nötig
Lebensdauer	CORBA Persistent State Service	Das Halten von Zustand über einen längeren Zeitraum ist Aufgabe der Anwendung
Einfachheit der Bereitstellung	CORBA Component Model	SOAP, HTTP und XML sind vertraute Technologien
Plattformunabhängigkeit	ja	ja
Verwaltungskosten	Minimum CORBA specification.	Stark Anwendungsabhängig. In speziellen Fällen ist ein sehr einfacher XML-Parser ausreichend

HTML-Suchmaschinen sucht, ist unklar ob ein so hoher Prozentsatz auch aufrecht erhalten wird, wenn das gesamte Web autonom durchstöbert wird. Auch die Frage nach der Interpretation von Webformularen bleibt bei diesem Ansatz offen.

In den letzten Jahren werden vor Allem semantische Werkzeuge herangezogen, um Webdienste genauer beschreiben und einordnen zu können. Einerseits wird dies mit Semantic Web Documents versucht, wie Al-Masri et al. beschreibt. Andererseits werden auch Ontologien herangezogen [MM03], welche zum Beispiel mit den Sprachen OWL[10] oder DAML-S[11] beschrieben werden. Semantic Web Documents erleichtern das automatisierte Finden von Informationen im Web erheblich, da ein Interpreter nicht mehr erraten muss, welche Textstücke einer Webseite welche Bedeutung haben könnten — der Webentwickler kann die Semantik (sprich Bedeutung) der Information ganz einfach selbst angeben [DHC13].
Ich denke, dass das Nutzen semantischer Daten ein Schritt in die richtige Richtung ist, um einerseits Informationen über Webdienste eindeutig wahrnehmen zu können und andererseits Eingabefelder von Webformularen schnell kategorisieren zu können.

Garofalakis et al. hebt in [GPST04] hervor, dass in bisherigen Broker-Architekturen zu wenig Wert auf QoS- bzw. QoWS[12]-Parameter gelegt wird. Eine Dienstkategorie alleine reiche nicht aus, um einen guten Dienst finden zu können. Sicherheit, Integrität, Verfügbarkeit, Genauigkeit, Authentizität, Latenz, Geschwindigkeit, Skalierung des Dienstes bei großen Datenmengen oder Dienstkosten[13] sind oftmals die wichtigsten Anforderungen an einen Dienst.
Ich sehe das ähnlich, da Dienstsuchende in der Wahl des Protokolls oft frei sind — die Entwickler können die Anfragen mit einem beliebigen Protokoll senden, wichtig ist hauptsächlich, dass die Dienstgüteparameter stimmen. In einem modifizierten tModel[14]-Format von UDDI können solche Argumente von den Dienstanbietern bereits angegeben werden [GPST04].

Der Autor spricht zusätzlich an, dass sich nicht nur Webdienste beim Broker registrieren können sollen, sondern auch andere weborientierte Komponenten. Diese Meinung teile ich, da auch der hier entworfene Broker nicht nur Webdienste sondern auch Webformulare unterstützt. Ich sehe es jedoch auch als schwierig an, eine gemeinsame Schnittstelle für stark heterogene Komponenten zu finden.

3.2. Agentensysteme

Bei Agentensystemen ist genau wie beim Broker relevant, wie das Problem der schnittstellengerechten Kommunikation unter den Diensten gelöst wurde und wie an die Herausforderungen Datensicherung, Zurücksetzung, Protokollierung und Umgang mit Legacy-Technologien herangegangen wurde. Interessant sind auch die vorgebrachten Ansätze, wie und wo sich neue Dienste an- und abmelden können sowie die Verwaltung der eingetragenen Dienste.

3.2.1. Open Agent Architecture

Das Ziel der *Open Agent Architecture* (OAA) ist es, eine Agentenarchitektur für vernetzte Desktop- und Mobilgeräte zu entwickeln. Wie beim Broker können Anfragen transparent in das System eingeschleust werden — der Aufrufer braucht also nicht zu wissen, welcher andere konkrete Dienst seine Anfrage beantworten kann.

[10] Web Ontology Language
[11] DARPA agent markup language for service, welche auf OWL aufbaut.
[12] Quality of Web Service
[13] Bei kommerziell angebotenen Diensten relevant.
[14] Ein tModel ist die generische Repräsentation eines Dienstes.

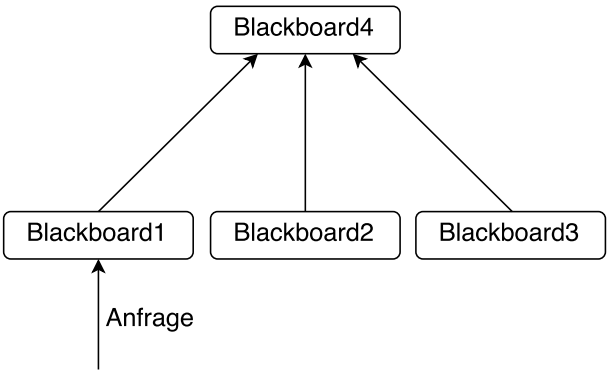

Abbildung 3.4.: Kann ein Blackboard eine Anfrage nicht selbst beantworten, so wird diese an ein hierarchisch höheres Blackboard weitergeleitet.

OAA ist im Blackboard-Architekturmuster aufgebaut, Agenten registrieren sich bei einem von mehreren Blackboards.
Bei OAA werden Dienstanfragen in das Blackboard geschrieben, woraufhin der Blackboard Server versucht, den passenden Dienst zu finden. Im Kontrast zum Faktenspeicher existiert jedoch nicht nur ein Blackboard, sondern mehrere hierarchisch angeordnete. Wenn ein Blackboard eine Anfrage nicht befriedigen kann, so wird die Anfrage an das hierarchisch darüberliegende Blackboard wie in Abbildung 3.4 weitergeleitet. Erst wenn das Wurzel-Element erreicht ist und dennoch kein passender Agent gefunden wurde, kommt es zu einer Fehlernachricht.

Der Ansatz eines Faktenspeichers gegenüber mehreren Blackboards bietet den Vorteil, dass der Unifikationsprozess am Stück abgearbeitet wird und unnötige Kommunikation bzw. Synchronisation zwischen mehreren Faktenspeichern vermieden wird. Der Nachteil ist, dass bei einer sehr hohen Anzahl gleichzeitiger Dienstanfragen und registrierten Diensten zum einen der Unifikationsprozess besonders lange dauern kann und zum anderen eine korrekte Synchronisierung parallel gestellter Anfragen aufwändiger wird. Mehrere Faktenspeicher sind in der Ausprägung denkbar, dass jeder Ontologie ein fester, eigener Faktenspeicher zugeordnet ist. Dadurch könnte man die Modularisierung und Unabhängigkeit der einzelnen Ontologien verstärken, das Gesamtsystem wird durch das zusätzliche hin- und herkopieren von Fakten komplexer und es entstehen höhere Verwaltungskosten.

Zur Kommunikation bedient sich die OAA der *Interagent Communication Language (ICL)*, in welcher Nachrichten ähnlich zu Fakten in einem eigenen Prolog-Dialekt dargestellt werden. So eine Darstellung begünstigt die Unifikation. Hier kann man betonen, dass der Faktenspeicher auch dieses Softwarebauteil mit abdeckt, da die Kommunikation ausschließlich über den Faktenspeicher abläuft. Das Einführen einer eigenen Kommunikationssprache ist daher nicht notwendig.

Im Ausblick der Arbeit wird auf den Wunsch eines Transaktionssystems eingegangen. Damit ist gemeint, dass Momentaufnahmen des aktuellen Gesamtzustands möglich sind: registrierte Dienste, eingetragene Anfragen, Status der Anfragen. Da der Faktenspeicher ja gerade alle Informationen über eingetragene Dienste, Anfragen sowie den aktuellen Status der Kommunikation darstellt, ist das beim hier entwickelten Broker einfach möglich. Der Faktenspeicher ist als einfache Datei oder Abbildung im Arbeitsspeicher realisiert, so ist das Festhalten des Gesamtzustands durch Kopieren der Daten möglich. Man muss allerdings darauf achten, dass keine ungültige Kopie durch einen Synchronisierungsfehler entsteht.

3.2.2. RETSINA MAS Infrastruktur

Die *RETSINA MAS Infrastruktur* (kurz *Retsina*) geht einen Schritt weiter und unterstützt die Aufzeichnung von Transaktionen, Management-Werkzeuge zur Visualisierung der Aktivität und Dienstgüteparameter pro Agent.

Zu jedem Agent wird die entsprechende Ausfallwahrscheinlichkeit und Verzögerung abgespeichert. Dadurch können zuerst solche Dienste angesprochen werden, die besonders reaktiv sind, um so eine geringere insgesamte Latenz des Systems zu erreichen.
Man muss meiner Meinung nach berücksichtigen, dass die Latenz und die Ausfallwahrscheinlichkeit nicht unbedingt gleichmäßig über einen längeren Zeitraum verteilt sein müssen, weshalb Messwerte einer exponentiellen Glättung[15] unterzogen werden sollten. Weiterhin liegt es dann nahe, auch die Glaubwürdigkeit eines Dienstes zu protokollieren [SPVVG03]. Es könnte beispielsweise sein, dass ein Dienst sehr schnell reagiert aber immer falsche Ergebnisse zurück liefert. Solche Dienste sollten gegebenenfalls als unglaubwürdiger eingestuft werden und bei einer Aufstellung aller verfügbaren Dienste weiter unten gelistet sein.
Die Frage, ob ein Dienst falsche Ergebnisse liefert, ist in meinen Augen jedoch nicht so einfach zu beantworten. Eine Herangehensweise ist die Beobachtung des Nutzerverhaltens. Wird ein Dienst permanent an oberster Stelle angezeigt, jedoch nie ausgewählt, so liegt die Vermutung nahe, dass der Nutzer nicht an diesem Dienst interessiert ist. Daraus lässt sich nicht direkt schließen, dass die Ergebnisse falsch sind, da der Nutzer den Dienst auch aus anderen Gründen prinzipiell ablehnen könnte, es ist jedoch ein Anhaltspunkt.

Ein weiteres Merkmal der RETSINA MAS Infrastruktur ist ihr dezentraler Aufbau — weder ein Blackboard noch ein Faktenspeicher existiert. Dienste finden sich untereinander durch das Absenden[16] spezieller Datenpakete.
Ich sehe ein System ohne zentrale Kontrolle als besonders reizvoll, da das Ausfallen einzelner Komponenten das Gesamtsystem nicht oder nur geringfügig beeinträchtigt. Im Kontrast dazu ist keine Kommunikation mehr möglich, wenn der Faktenspeicher ausfällt. Gleichzeitig werden Flaschenhälse über die gleichmäßigere Aufteilung der Last vermieden, da sich diese nicht an einer zentralen Komponente bündelt. Der Preis der Dezentralität ist, dass durch die Multicast-Kommunikation ein hoher Kommunikationsaufwand besteht.

[15] Ältere Messwerte werden dabei weniger stark gewichtet als aktuellere. Üblicherweise dargestellt mit der Formel $s_t = \alpha \cdot x_t + (1 - \alpha) \cdot s_{t-1}$. α bezeichnet dabei die Alterungsgeschwindigkeit, x_t den neuen Messwert und s_t bzw. s_{t-1} den exponentiell gewichteten Mittelwert zum Zeitpunkt t bzw $t - 1$.

[16] Zur Bekanntmachung wird Multicast verwendet. Beim Senden an eine anfangs festgesetzte Multicast-Adresse wird die gleiche Nachricht an mehrere vorher definierte Empfänger gleichzeitig gesendet, allerdings nicht an alle im Netz befindlichen (Broadcast). Innerhalb von RETSINA MAS verhält es sich aber tatsächlich wie ein Broadcast, da alle Agenten auf die gleiche Multicast-Adresse eingestellt sind.

4. Analyse und Entwurf

Aufbauend auf den vorgestellten Verfahren des dritten Kapitels wird eine Broker Architektur für den EASIER Active Server erstellt. Abbildung 4.1 Grenzt den Umfang dieser Arbeit grafisch von der Gesamtarchitektur ab und verdeutlicht den abstrakten Ablauf.

Die Hauptaufgabe des Brokers ist die Zusammenführung von Dienstanfragen des Dialogmanagers mit passenden Onlinediensten (Webdienste und Webformulare), siehe Abbildung 4.2.

Der Dialogmanager[1] schreibt die Dienstanfrage in den Faktenspeicher (1). Der Broker verarbeitet die Anfrage und sucht nach bereits registrierten Onlinediensten (2). Gefundene, passende Onlinedienste werden vom Broker aufgerufen. Sämtliche Kommunikation zwischen den Komponenten Dialogmanager, Broker und Onlinediensten verläuft über den Faktenspeicher. Abbildung 4.3 zeigt das Datenflussdiagramm bei der Anfrage.
Webdienste und Webformulare, welche sich zuvor beim Broker registriert haben, werden vom Broker bei der Suche nach passenden Diensten in Betracht gezogen. Ist ein passender Dienstanbieter gefunden, so wird dieser vom Broker kontaktiert.
Aufgerufene Dienste schreiben ihre Ergebnisse in den Faktenspeicher (4). Der Broker kann mehrere Dienstergebnisse zu einem Anfrageergebnis aggregieren (5). Eine vom Dialogmanager erstellte Regel wird ausgelöst, sobald das Anfrageergebnis im Faktenspeicher vorliegt (6). Der Dialogmanager visualisiert das Ergebnis und stellt gegebenfalls Rückfragen an den Broker. Der genaue Ablauf ist in Sequenzdiagramm 4.4 dargestellt.

[1] Dieser wird von einem Masterstudenten am IPD entworfen [Sai16]. Dieser stellt eine Kommunikationsschnittstelle zum Broker dar.

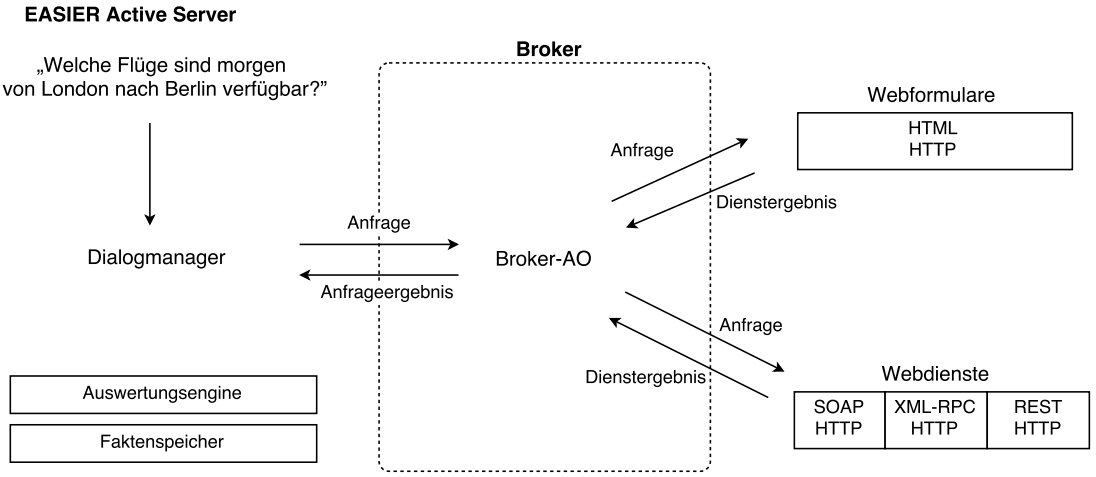

Abbildung 4.1.: Die EASIER Active Server Architektur und der schematische Broker darin.

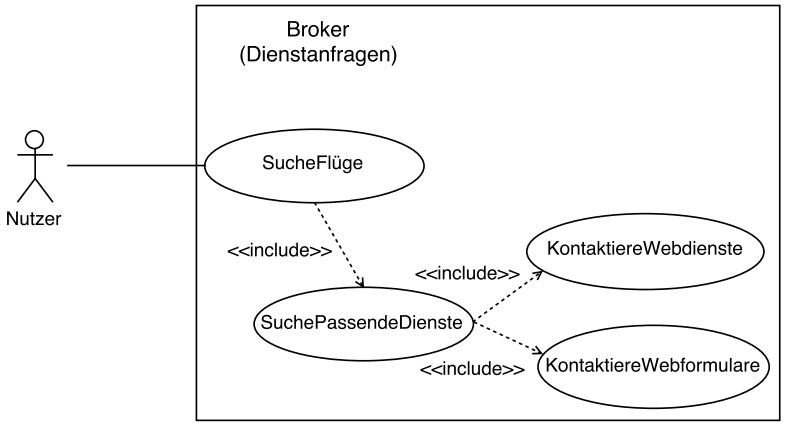

Abbildung 4.2.: Die Aufgabe des Brokers: Vermittlung von Dienstanfragen

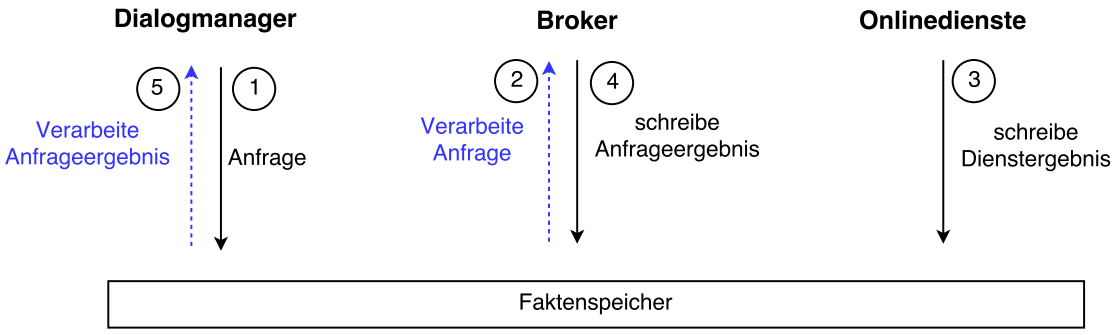

Abbildung 4.3.: Die Kommunikation zwischen Dialogmanager, Broker und Onlinediensten entsteht nur durch den Faktenspeicher. Blaue Pfeile symbolisieren feuernde Regeln.

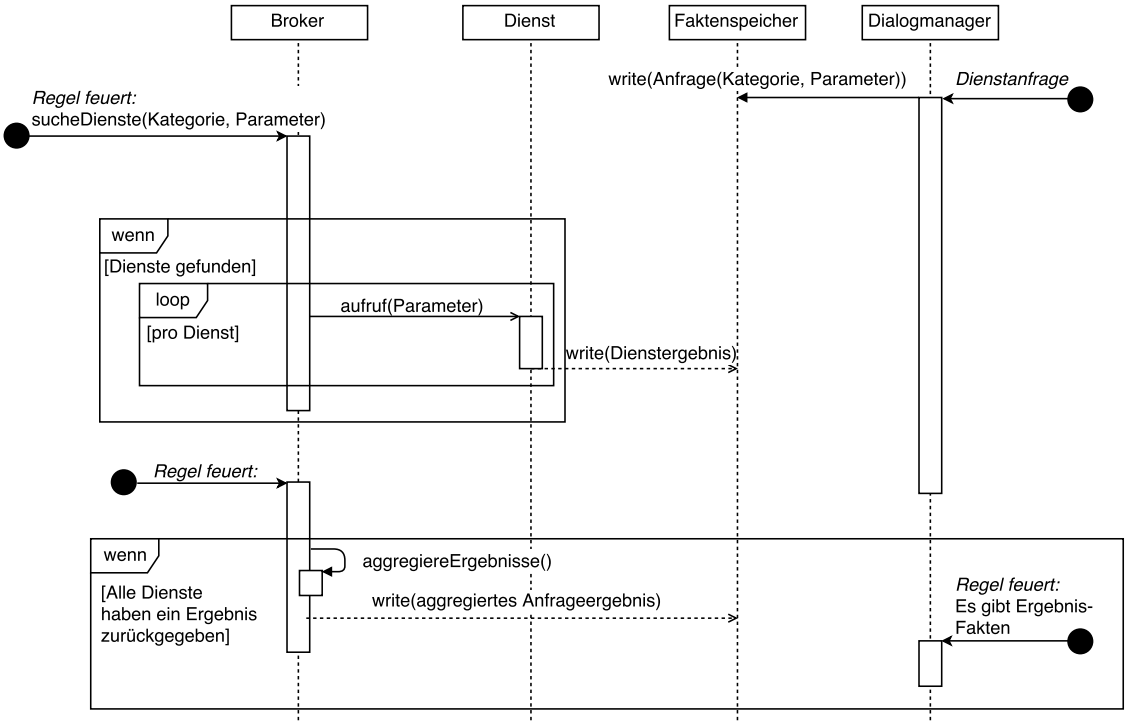

Abbildung 4.4.: Der Broker bearbeitet eine Dienstanfrage

4.1. Onlinedienste

Onlinedienste werden in zwei Typen untergliedert: Es gibt Webdienste und Webformulare. Wie auch im Klassendiagramm in Abbildung 4.5 ersichtlich, lassen sich Webdienste weiter in SOAP, XML-RPC und REST unterteilen.

Ein generischer Onlinedienst verfügt bereits über alle Eigenschaften, die für den Broker wichtig sind: Er kann vom Broker durch eine definierte Schnittstelle aufgerufen werden und versteckt alle dienst-spezifischen Details vor dem Broker.

Onlinedienste haben eine feste Kategorie, zum Beispiel „Flug“, eine Url und eine Priorität. Die Priorität ist wichtig, falls mehrere Onlinedienste der gleichen Kategorie zur Auswahl stehen, aber nur einer davon aufgerufen werden soll.

Jeder Onlinedienst kann darüber hinaus ein Muster als regulären Ausdruck definieren, der genau einer Ergebniszeile in der vom Dienst erhaltenen Antwort entspricht[2]. Bei Webformularen ist dieses Muster zwingend, sofern nicht die gesamte Webseite in HTML für den Aufrufer informationstragend ist. Bei Webdiensten kann der Ausdruck optional verwendet werden, um erhaltene Daten in die gewünschte Form zu bringen.
Alternativ kann auch eine eigene Methode definiert und an den Dienst übergeben werden, die die Rohergebnisse verarbeitet.

Allen Onlinediensten ist gemein, dass diese auf einem weiteren Anwendungsschichtprotokoll aufbauen. Hier wird das Strategie-Entwurfsmuster angewandt: Das von einem Onlinedienst verwendete Anwendungsschichtprotokoll ist frei austauschbar. Es werden momentan die Anwendungsschichtprotokolle HTTP und HTTPS jeweils im GET und POST Modus unterstützt. Es ist ohne weiteres möglich, weitere Protokolle, etwa SMTP oder FTP, hinzuzufügen.

4.2. Kategorie und Dienstparameter

Jeder existierende Dienst ist genau einer Dienstkategorie zugeordnet. Eine Dienstkategorie zeichnet sich aus durch einen eindeutigen Namen und einer Liste von verpflichtenden Parametern. Verpflichtende Parameter müssen dem Dienst mindestens übergeben werden, damit eine Anfrage überhaupt sinnvoll formuliert werden kann.
Kategorien sowie deren Parameter befinden sich im Faktenspeicher. Neue Kategorien können auch zur Laufzeit eingetragen werden. Es ist nicht möglich, einen Onlinedienst unbekannter Kategorie zu registrieren.

Im Beispiel eines Flugdienstes lautet die Kategorie `Flug`. Die verpflichtenden Parameter für diese Kategorie sind `von`, `nach` und `datum`. Darüber hinaus unterstützt dieser Dienst auch noch die Möglichkeit, optional WLAN hinzuzubuchen (Ja/Nein). Dieser Parameter kann als optionaler Parameter bei der Instanziierung eines Onlinedienstes übergeben werden. Andere optionale Parameter für diesen Dienst existieren nicht.

Man kann nicht davon ausgehen, dass die Parameter aller Onlinedienste einer Kategorie standardisiert sind. Bei unserem Fluganbieter-Dienst heißen die Parameter `von`, `nach` und `datum`. Bei einem anderen Webformular eines Flugbuchungsportals heißen die gleichen Parameter möglicherweise `from`, `to` und `date`.
In [Guz08, Seite 101] müssen alle Dienste bei der Registrierung eine Abbildung angeben, die diese Parameterkonvertierung beschreibt. Das müssen Dienste bei dieser Architektur zwar auch, die Abbildung ist jedoch automatisch generiert worden. Die Erstellung einer solchen Abbildung ist Teil der Arbeit eines Masterstudenten am IPD [Sai16].

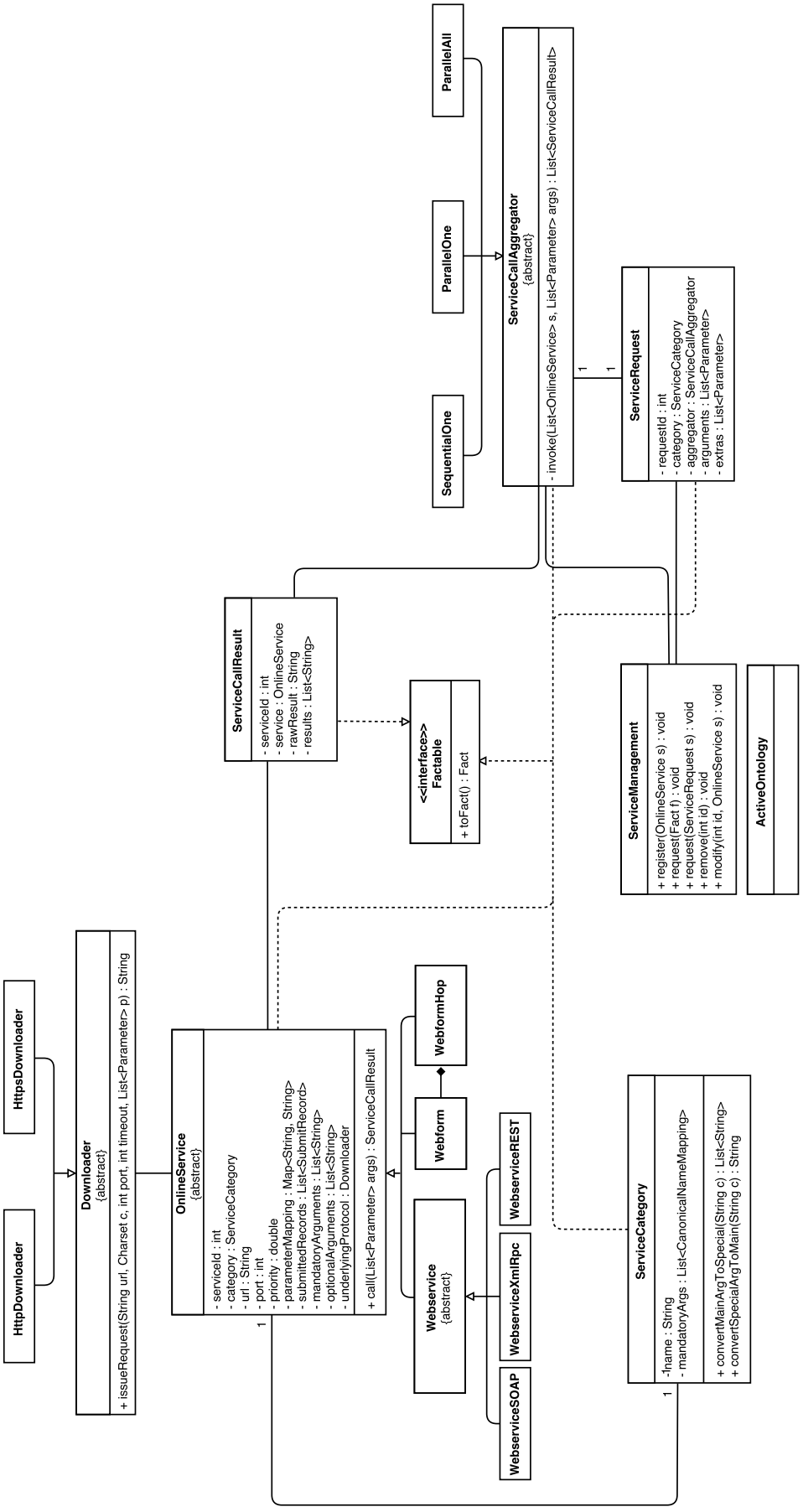

Abbildung 4.5.: Das Klassendiagramm zur Broker-Architektur.

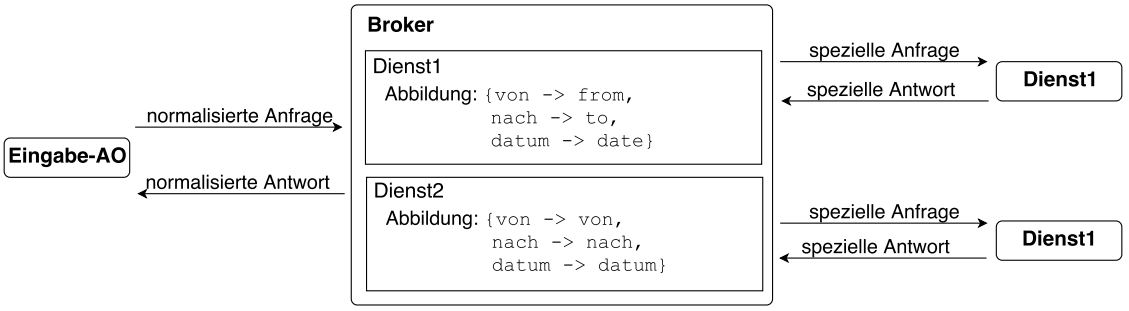

Abbildung 4.6.: Durch eine automatisch generierte Abbildung werden normalisierte in spezifische Parameter transformiert.

Die Abbildung wird jedem Onlinedienst bei seiner Erstellung mitgegeben und beim Dienstaufruf angewandt.

Es gibt auch Fluganbieter, welchen die per Kategorie zugeordneten verpflichtenden Parameter nicht ausreichen und es müssen weitere Parameter in der Anfrage enthalten sein, bevor sich der Dienst sinnvoll ansprechen lässt.
Aus diesem Grund kann der Dienst auch eigene, für ihn speziell verpflichtende Parameter angeben. Nur wenn alle für die Kategorie notwendigen Parameter und alle selbst als verpflichtend eingetragenen Parameter bei einer Dienstanfrage an der Broker übergeben werden, wird der Dienst betrachtet und aufgerufen.

Es wird zusammenfassend zwischen drei Parametertypen eines Dienstes unterschieden:

- Verpflichtend durch Kategorie — Diese Parameter werden von der Kategorie vorgegeben und müssen in jedem Fall verfügbar sein.
- Verpflichtend durch Dienst — Diese Parameter werden von einem spezifischen Dienst definiert und gelten nur für diesen zusätzlich zu denen der Kategorie.
- Optional durch Dienst — Rein optionale Parameter, die für eine feinere Filterung der Dienste genutzt werden können.

Da der Broker eine AO ist, besitzt dieser keine öffentlichen Methoden und kommuniziert ausschließlich über den Faktenspeicher.
Die Hilfsklasse ServiceManagement bietet die öffentlichen Methoden zur Dienstregistrierung und Dienstanfrage an.

4.3. Registrierung

Das Ziel bei der Dienstregistrierung ist es, dass verschiedene Onlinedienste dem Broker (automatisiert) hinzugefügt werden. Ein Masterstudent am IPD hat sich damit befasst, Webformulare im Internet zu finden und zu kategorisieren [Lin15]. Die gefundenen Dienste können automatisiert dem Broker übergeben und registriert werden.

Ein Onlinedienst wird dem Dienstverwalter per Methodenaufruf übergeben, um den Onlinedienst in den Faktenspeicher aufzunehmen. Der Dienstverwalter erzeugt einen Fakt, der genau diesem Onlinedienst entspricht, und fügt diesen dem Faktenspeicher hinzu. Dabei wird der Dienst auch mit einer eindeutigen Identifikationsnummer versehen.

Die Broker-AO selbst wird bei der Dienstregistrierung noch nicht aktiv, da es für diesen noch keine Aufgaben zu erledigen gibt. Erst bei einer Dienstanfrage kommt die Broker-AO ins Spiel.

[2] Alternativ kann auch eine eigene Funktion zur Extrahierung der Nutzdaten angegeben werden.

Abhängig davon, ob ein Webdienst oder ein Webformular registriert wird, müssen bestimmte Voraussetzungen erfüllt sein.

4.3.1. Webdienste

Für Webdienste wurden drei Ausprägungen umgesetzt: SOAP, XML-RPC und REST. Bei der Instanziierung eines Webdienstes müssen mindestens drei Parameter angegeben werden:

- Dienstkategorie
- Eine Url, die den Dienst identifiziert. Bei SOAP-Diensten ist dies der Link zu einer WSDL-Datei, bei XML-RPC und REST die Adresse des Dienstservers.
- Einen Operationnamen des Dienstes. Oft bieten Webdienste mehrere unterschiedliche Operationen an. Es muss eine Operation ausgewählt werden, die vom Broker aufgerufen werden soll. Die Operationsnamen sind dienstspezifisch.

Sobald diese drei Parameter zur Verfügung stehen, kann der Webdienst registriert werden. Die Parameter der Operation (die im dritten Argument angegeben wird) sind bereits durch die Kategorie bestimmt. Wenn keine Parametertransformation angegeben wird, geht man davon aus, dass der Dienst die normalisierten Parameternamen annimmt und keine spezifischen verlangt.
Zusätzliche verpflichtende oder optionale Parameter für einen konkreten Dienst können angegeben werden.

Als Beispiel nehmen wir einen Fluganbieter, welcher sich als SOAP-Webdienst registriert. Dazu werden die Parameter `Kategorie: Flug`, `Url: http://server.de/beschreibung.wsdl` und `Operationsname: findeFlüge` übergeben. Der Dienstverwalter nimmt das Objekt entgegen und schreibt es in den Faktenspeicher als verfügbaren Dienst. Die WSDL-Datei wird erst beim Eintritt einer Anfrage betrachtet und ausgewertet. So ist es möglich, dass bereits registrierte SOAP-Dienste vom Dienstbetreiber entkoppelt aktualisiert werden können – Der Broker muss dazu nicht ein weiteres Mal aktualisiert werden.

Alle Dienste können jederzeit über den Dienstverwalter aktualisiert werden.

4.3.2. Webformulare

Ein Webformular ist definiert durch eine Kategorie, eine Basis-Url und eventuell mehrere Hops. Ein Hop existiert aus Sicht des Brokers immer dann, wenn Webinhalte nachgeladen werden. Zum Beispiel, wenn eine neue Webseite aufgerufen oder ein Unterformular geladen wird. Dadurch werden Formulare unterstützt, die sich über mehrere Seiten hinweg erstrecken, sogenannte mehrstufige Formulare. Es ist ebenfalls möglich, eine einzelne Webseite mit mehreren Formularen auszuwerten.

Jeder Hop hat dabei eine eigene Url, sowie spezifische (verpflichtende und optionale) Parameter, die angegeben werden können.

Zu jedem Hop existiert weiterhin ein regulärer Ausdruck, welcher die Ergebnisseite auf die tatsächlichen Ergebnisdaten reduziert.
Da die Ergebnisseiten von Webformularen für Endnutzer konzipiert sind, befinden sich auf diesen Seiten viele HTML-Elemente, die nicht zu den unmittelbaren Nutzdaten beitragen, den tatsächlichen Ergebnisdaten. Es seien Werbeflächen und eingebettete Multimediainhalte als Beispiele genannt. Mit einem regulären Ausdruck werden die Ergebnisse einer Anfrage aus dem HTML-Code extrahiert.

4.4. Anfrage

Bei der Dienstanfrage an den Broker wird nicht mehr zwischen Webdiensten und Webformularen unterschieden. Für alle im Faktenspeicher gefundenen und zur Anfrage passenden Dienste wird lediglich die abstrakte `invoke`-Methode eines Onlinedienstes aufgerufen.

Wenn ein Anfrage-Ereignis im Faktenspeicher platziert wird, feuert eine Regel des Brokers. Daraufhin sucht dieser nach passenden Diensten, kontaktiert diese und schreibt das zusammengefasste Ergebnis in den Faktenspeicher.
Eine Anfrage enthält folgende Informationen:

- Anfragenummer
- Dienstkategorie
- Anfragemodus
- Parameter für den Dienst
- Qualitätsparameter über die Anfrage

Die Anfragenummer dient der eindeutigen Zuordnung der Anfrage zu den Dienstergebnissen, welche ebenfalls in den Faktenspeicher geschrieben werden.

Die Dienstkategorie gibt an, welche Art von Dienst der Anfrager wünscht. Die Kategorie muss mit der Kategorie übereinstimmen, die ein Onlinedienst bei seiner Registrierung angegeben hat.

Es gibt drei verschiedene Anfragemodi, von denen einer ausgewählt werden muss. Das erste Wort gibt im Folgenden an, ob die Dienste vom Broker sequenziell oder parallel kontaktiert werden. Der zweite Term gibt an, ob das Ergebnis eines Dienstes oder eine Ergebnisliste aller passenden Dienste zurückgegeben wird:

- **Sequenziell-Einer**: Unter allen passenden Diensten wird das Ergebnis genau eines Dienstes zurückgegeben. Es wird so lange ein passender Dienst nach dem anderen kontaktiert, bis der erste Dienst eine gültige Antwort zurückgibt. Dienste mit höherer Priorität werden zuerst kontaktiert.
- **Parallel-Einer**: Es werden alle passenden Dienste parallel kontaktiert. Die erste erhaltene Antwort wird in den Faktenspeicher geschrieben und die anderen Verbindungen werden abgebrochen. Diese Methodik ist sinnvoll, wenn eine hohe Bandbreite zur Verfügung steht und ein schnelles Ergebnis wichtig ist. Es wird keine Rücksicht auf die Priorität der Dienste genommen.
- **Parallel-Alle**: Es werden alle passenden Dienste parallel kontaktiert. Die Ergebnisse werden zusammengefasst und als Liste in den Faktenspeicher geschrieben.

Die Parameter der Anfrage werden dem Broker in normalisierter Form übergeben. Die bei der Dienstregistrierung erstellte Abbildung wird vom Broker genutzt, um die normalisierten Parameternamen in seine spezifischen zu konvertieren (gemäß dem Diagramm 4.4). Von einem Mitarbeiter am IPD wird sichergestellt, dass alle übergebenen Parameter in normalisierter Form übergeben werden und die Menge der übergebenen Parameter eine Obermenge der von der Dienstkategorie geforderten sind.

Qualitätsparameter der Anfrage dienen dazu, dem Broker zusätzliche (optionale) Informationen zu vermitteln. Zum Beispiel kann die anfragende AO übergeben werden.

Bei der Anfrage kann optional ein Timeout angegeben werden. Wenn ein Dienst gefunden und kontaktiert wird, so kann es dazu kommen, dass keine Antwort von diesem für einen definierten Zeitraum kommt. Ist das der Fall, so wird die Anfrage abgebrochen und ein

Ergebnis-Fakt mit einer Fehler-Id in den Faktenspeicher geschrieben.
Der Dienstanfrager wird über eine Regel, welche er zuvor erstellt hat, über das Ergebnis informiert. Diese Regel feuert, wenn zu der Anfrage passende Ergebnis-Fakten in den Faktenspeicher geschrieben werden. Ergebnis-Fakten enthalten entweder die Resultate der Dienstanfragen oder eine Fehlermeldung mit Begründung, warum keine Resultate geliefert werden konnten.

Da das Anfrage-Ereignis nach dem Auswertungszyklus gelöscht wird, erstellt der Broker einen Anfragekontext-Fakt, wodurch die Anfrage-Id und den Anfragemodus permanent im Faktenspeicher sind.

4.5. Ergebnisse

Die Dienstergebnisse der einzelnen Onlinedienste werden von diesen direkt in den Faktenspeicher geschrieben, so bald sie verfügbar sind. Daraufhin feuert eine Regel des Brokers. Der Broker aggregiert die Ergebnisse wie bei der Anfrage angegeben (die Informationen stammet zu diesem Zeitpunkt aus dem Anfragekontext-Fakt) und erzeugt einen Anfrageergebnis-Fakt. Der Anfrageergebnis-Fakt wird wiederum in den Faktenspeicher geschrieben.

Ein Anfrageergebnis-Fakt hat dabei mindestens folgende Parameter:

- Anfragenummer
- Status
- Ergebnisliste

Durch die Existenz der Anfragenummer kann der Auslöser der Anfrage eine Regel erstellen, die auf den zugehörigen Ergebnis-Fakt feuert.

Der Status gibt an, ob eine Anfrage erfolgreich war oder ob ein Fehler aufgetreten ist. Im Fehlerfall kann die Ursache aus dem Status gelesen werden, zum Beispiel ein Timeout oder dass es keine passenden Dienste gibt.

Im dritten Parameter befinden sich die über alle angesprochenen Dienste hinweg aggregierten und aufbereiteten Ergebnisse in Form einer Liste.

Die Ergebnisse von Webformularen werden in der Regel nicht roh zurückübergeben. Ein bei der Diensterstellung angegebener regulärer Ausdruck wird auf die Rohergebnisse angewendet. Da dieses Verhalten von allen Onlinediensten unterstützt wird, können auch alle Webdienste ihre Ergebnisse durch einen regulären Ausdruck filtern. Das kann sinnvoll sein, wenn die Ergebnisse eines Webdienstes noch nicht in der gewünschten Formatierung vorliegen.

Alternativ ist es möglich, für einen bestimmten Onlinedienst eine eigene Methode zu definieren, die die erhaltenen Rohergebnisse aufbereitet. Durch die Implementierung einer existierenden Schnittstelle ist dies möglich. Statt dem regulären Ausdruck wird dann diese Methode auf die Rohergebnisse angewendet.

Zusätzlich zu den aufbereiteten, unmittelbar an die Eingabe-AO zurückgegebenen Ergebnisse werden auch noch die unverarbeiteten Ergebnisse abgespeichert. Bei einem Webformular ist das unverarbeitete Ergebnis zum Beispiel die vollständige Ergebniswebseite;
Bei SOAP- und XML-RPC-Webdiensten ist es die vollständige XML-Datei.
Im Fall eines mehrstufigen Webformulars kann die Ergebniswebseite zur Erstellung einer neuen AO genutzt werden, in dem das sich dort befindliche Webformular interpretiert wird. Diese AO kann dann wiederum eine Anfrage an den Broker stellen.

Der herausragende Vorteil dieser Vorgehensweise ist, dass dadurch mehrstufige Webformulare mit dynamisch nachgeladenen Folgeformularen (Hops) vom Broker verarbeitet werden können. Es handelt sich also um Webformulare über mehrere Hops, bei denen nicht von Anfang an klar ist, wie das zweite, dritte oder n-te Webformular aussehen wird, denn das ist möglicherweise von der konkreten Eingabe abhängig.

5. Implementierung

In dieser Arbeit wurde eine Broker-Architektur implementiert, die das im Entwurf beschriebene Verhalten umsetzt. Es wurde dazu die Programmiersprache Java herangezogen, in welcher auch die anderen Teile des EASIER Active Servers implementiert sind.

Die wichtigste Methode eines `OnlineServices` ist `call(List<Parameter<T>>)`, wie in Quelltextausschnittt 4 dargestellt. Sie konvertiert übergebene Standardparameter in dienstspezifische Parameter und wird von Unterklassen überschrieben, um die Funktionalität des Dienstes zu implementieren.

Quelltextausschnitt 4: `call`-Methode von `OnlineService`

```
/**
 * Rufe den Dienst auf.
 *
 * @param args
 *            Übergebene Parameter an den Dienst
 */
public ServiceCallResult call(List<Parameter<?>> args) {
    convertArguments(args);

    return null;
}
```

Ein `Parameter<T>` kapselt dabei einen Parameter an den Dienst. Dieser hat einen Namen als Text und einen Wert von generischem Typ `T`.
Bei manchen Diensttypen muss der Typ des Parameters angegeben werden, deshalb wird nicht ein einfaches (`String`, `String`)-Paar verwendet. Auf diese Weise wird der Parametertyp gespeichert.
Das ist zum Beispiel der Fall bei XML-RPC: Hier muss für jeden Parameter der genaue Typ bekannt sein. Die Schnittstelle der verwendeten XML-RPC Bibliothek nimmt Parameterwerte vom Typ `Object` entgegen und generiert daraus eine XML-Anfrage wie in den Grundlagen beschrieben.
Bei den anderen Webdiensten und beim Webformular muss der Typ nicht bekannt sein, sondern lediglich eine Text-Repräsentation des Parameters verfügbar sein.

Standardwerte des Brokers werden in einer .properties Datei verwaltet und sind damit sauber vom Quelltext getrennt. Dazu gehören zum Beispiel Ports für eine HTTP-Verbindung.

5.1. Kategorie und Dienstparameter

Eine Dienstkategorie wird durch die Klasse `ServiceCategory` abgebildet. Zur Erstellung ist der Name der Kategorie und eine Menge an Parametern notwendig. Die Parameter haben einen Standardnamen und eine Menge an Synonymen. Die Parameter heißen `CanonicalNameMapping`. Der erste an `CanonicalNameMapping` übergebene Name repräsentiert den Standardnamen.
Im folgenden Beispiel wird eine Dienstkategorie für Aktienwerte realisiert. Der Name der Kategorie ist daher „stocks“. Diese Kategorie erwartet einen Parameter „ticker“, der das eindeutige Kürzel eines Wertpapieres angibt, durch welches dieser Wert an der Börse notiert ist (zum Beispiel „GOOG“ für Google).

```
List<String> parameter = new List<>(); parameter.add("ticker"); ServiceCategory
STOCK = new ServiceCategory("stocks", parameter);
```

Die Klasse `ServiceCategory` implementiert die Schnittstelle `Factable`. Klassen, die diese Schnittstelle implementieren, besitzen eine Fakt-Darstellung für den Faktenspeicher, an welche man über die Methode `toFact()` gelangt.
Im Faktenspeicher wird die Kategorie „stocks“ in folgender Form abgelegt:

```
category(stocks, [ticker])
```

Der erste Parameter gibt den Namen der Kategorie an. Es folgt eine Liste von verpflichtenden Parametern.

Nachdem nun die Grundbegriffe über Dienstkategorien sowie die Schnittstelle zum Aufruf eines Onlinedienstes gezeigt wurde, wird im folgenden Abschnitt die Registrierung eines Onlinedienstes beschrieben.

Quelltextausschnitt 5: `convertArguments`-Methode von `OnlineService`

```
/**
 * Konvertiert uebergebene Standardparameter
 * in dienstspezifische. Parameter ohne existierendes
 * Mapping werden nicht ersetzt.
 *
 * @param args
 *            Die an den Dienst uebergebenen Parameter.
 */
protected void convertArguments(List<Parameter> args) {
    Iterator<Parameter> it = args.iterator();
    while (it.hasNext()) {
        Parameter a = it.next();
        if (mParameterMapping.containsKey(a.getName())) {
            a.setName(mParameterMapping.get(a.getName()));
        }
    }
}
```

5.2. Registrierung

Für die Registrierung eines Dienstes wird zunächst eine Instanz eines Webdienstes oder Webformulares erstellt. Wie diese genau erstellt wird, hängt von dem konkreten Dienst

ab.
Bei einem Webformular ist die Registrierung wie in Quelltextausschnitt 6 abgebildet. Zuerst wird eine Liste `actionOne` mit den Parametern für den ersten und einzigen Hop definiert. In diesem Fall gibt es nur einen übergebenen Parameter, nämlich „ticker“. Danach wird die Url des Webformulares definiert und anschließend wird ein regulärer Ausdruck definiert, um die Nutzdaten von dem gesamtem Quelltext der Webseite trennen zu können. Zum Schluß wird eine Liste von Hops erstellt, der einzige Hop hinzugefügt und ein neues Webformular erstellt. Dank der `toFact()`-Methode von `OnlineService` kann die erstellte Instanz sehr einfach dem Faktenspeicher hinzugefügt werden. Man beachte, dass die Dienstkategorie `STOCK` wie oben definiert verwendet wird.

Quelltextausschnitt 6: Registrierung eines Webformulares

```
// Parameter fuer den 1. Hop
List<String> args = new ArrayList<>();
args.add("ticker");

// Informationen ueber das Webformular
String url = "http://aktienkurse.com";
Downloader dl = new HTTPDownloader();
String regEx = "<h3>.*</h3>";

// Erstellung der Hops
List<WebformHop> hops = new ArrayList<>();
hops.add(new WebformHop(STOCK, url, dl, args, regEx));

// Eintragung in den Faktenspeicher
Webform form = new Webform(STOCK, url, hops).toFact();
form.setPriority(1);

// form.setOptionalParameters();

factStore.write(form);
```

Der Dienst wird ab diesem Moment von Dienstanfragen berücksichtigt. Der in `toFact` generierte Fakt ist für alle Onlinedienste gleich aufgebaut:

```
service(ID, Kategorie, VerpflichtendeArg, OptionaleArg, Priorität, Json)
```

- **ID** (einfacher Fakt: Zahl): Die eindeutige Identifikationsnummer des Dienstes. Diese wird automatisch generiert.
- **Kategorie** (einfacher Fakt: Text): Die Kategorie des Dienstes
- **VerpflichtendeArg** (ListenFakt): Ohne diese Argumente ist ein Aufruf des Dienstes nicht möglich. Alle dienstspezifischen und von der Kategorie geforderten Parameter sind in diesem Parameter enthalten.
- **OptionaleArg** (ListenFakt): Optionale Argumente des Dienstes
- **Priorität** (einfacher Fakt: Zahl): Die Priorität des Dienstes als Gleitkommazahl zwischen 0 (niedrigste Priorität) und 1 (höchste Priorität).

- **Json** (einfacher Fakt: Text): Ein Json-Text, der den Onlinedienst abbildet. Dieser erlaubt eine besonders elegante und effiziente Abbildung des Faktes zurück zu einer Java-Instanz.

Darüber hinaus bleibt der Faktenspeicher klar von einer bestimmten Programmiersprache getrennt, da Json-Anbindungen für zahlreiche Programmiersprachen existieren.

Der Typ des Dienstes, etwa SOAP-Webdienst oder Webformular, wird nur implizit im Json-Text abgespeichert. Zur Auffindung eines passenden Dienstes ist diese Information jedoch nicht wichtig und wird deshalb nicht separat gespeichert.

Der Fakt des hier definierten Webformulares hat folgende Form[1]:

```
service(3, stocks, [ticker], [], 0, ...)
```

5.2.1. Webdienste

Für die Instanziierung eines Webdienstes werden mindestens Kategorie, Url und Operation an eine Subklasse von `WebService` übergeben. Das besondere Merkmal an Webdiensten ist der Operationsname, welcher zusätzlich angegeben wird. Webdienstschnittstellen bieten oft mehrere Operationen an, die sie ausführen können. In WSDL-Dokumenten ist es selten, dass nur eine Operation definiert ist.
Solche Dienste müssen (mit jeweils korrekter Kategorie) mehrmals beim Broker registriert werden, denn jede Operation entspricht genau einer Funktion des Dienstes. Große Dienstanbieter wie die Expedia-Schnittstelle bieten beispielsweise die Suche nach Hotels und die Suche nach Flügen an – Es registrieren sich beim Broker zwei verschiedene Dienste.

Bei Webformularen hingegen taucht dieses Problem nicht auf, da pro Hop nur eine einzige „Formular absenden“-Schaltfläche existiert. Der nachfolgende Hop ist immer eindeutig durch das `target`-Attribut definiert.

5.2.2. Webformulare

Eine neue `Webform` wird mit einer Kategorie, einer Url und einer Liste an Hops erstellt.
Jeder Hop wiederum hat eine Url, eine Liste von Argumenten und einen regulären Ausdruck. Der reguläre Ausdruck filtert die Nutzinformationen aus dem HTML. Alternativ kann auch eine eigene Funktion angegeben werden, die diese Aufgabe übernimmt. Dafür wird das Interface `RawdataProcessor` implementiert und dem Onlinedienst mit `setRawdataProcessor()` übergeben.

Die Argumente werden pro Hop angegeben, damit in `Webform` geprüft werden kann, ob alle für die Kategorie notwendigen Parameter existieren. Möglicherweise sind diese über mehrere Hops verteilt.

Da `Hop` ebenfalls von `OnlineService` erbt, kann jeder Hop auch ein eigenes Anwendungsschichtprotokoll verwenden, zum Beispiel HTTP (GET / POST), HTTPS (GET / POST) oder SMTP.

Ob die GET oder POST-Methode bei HTTP(S) verwendet wird, ist für den Broker unerheblich. Beide können von Webentwicklern für beliebige Anwendungen gewählt werden. Die Empfehlung ist es jedoch, GET für idempotente Anfragen zu verwenden. Das sind Anfragen, die keine Daten auf dem Server permanent manipulieren (zum Beispiel Suchanfragen). POST-Anfragen werden im anderen Falle benutzt und können mit Seiteneffekten einhergehen.

[1] Da der Json-Text zu lange ist, wurde er hier ausgelassen.

Tabelle 5.1.: Mögliche HTML Formularattribute

Attribut	Beschreibung
`name`	Der Name identifiziert das Formular auf einer Webseite eindeutig
`target`	Das Dokument, welches die Formulardaten verarbeitet
`action`	Die HTTP-Methode `GET` oder `POST`
`accept-charset`	Der für die Eingabedaten verwendete Zeichensatz
`enctype`	Zeichenenkodierung der übertragenen Daten

Die freie Vor- und Rückwärtsnavigierbarkeit bei mehreren Hops wird von GET-Formularen unterstützt, bei POST-Formularen muss das wegen Seiteneffekten nicht der Fall sein[2].

In Tabelle 5.1 sind alle HTML-Formularattribute angegeben, die für `Webform` relevant sind und unterstützt werden. Es existieren noch weitere Formularattribute wie zum Beispiel `hidden`, welches angibt, ob ein Eingabefeld für den Benutzer sichtbar ist. Diese Felder sind aber ausschließlich für das Anzeigemedium, sprich dem Browser, interessant. Versteckte Eingabefelder sind meistens mit einem Standardwert belegt, dieser Parameter wird behandelt wie alle anderen.
Bei mehreren Hops werden Informationen der vorhergehenden an nachfolgende Hops weiter gereicht.

5.3. Anfrage

Nun wird der Ablauf beschrieben, der bei einer eintreffenden Dienstanfrage durch eine Eingabe-AO ausgeführt wird. Am leichtesten lässt sich dieser verstehen, wenn man zunächst einen Blick auf das Aktivitätsdiagramm 5.1 wirft.

Zur Modellierung von Anfragen in Java gibt es die `ServiceRequest`-Klasse.
Die Klasse implementiert das Interface `Factable` und ist somit in einen Fakt konvertierbar.

Ein `ServiceRequest`-Fakt hat folgende Form:

```
request(AnfrageId, Kategorie, Aggregator, ParameterListe, MetaParameter)
```

- **AnfrageId** (einfacher Fakt: Zahl): Mit der AnfrageId kann ein Antwort-Fakt erstellt werden, der sich genau auf diese Anfrage bezieht
- **Kategorie** (einfacher Fakt: Text): Die Dienstkategorie wird genutzt, um passende Dienste zu finden
- **Aggregator** (einfacher Fakt: Text): Der Aggregationsmodus: Sequenziell-Einer, Parallel-Einer oder Parallel-Alle
- **ParameterListe** (ListenFakt): Eine Liste aller Fakten, die von der Eingabe-AO als Parameter für den Aufruf vorgesehen sind. Darin sind zwangsweise alle für die Kategorie als verpflichtend gesehenen Parameter enthalten.
- **MetaParameter** (ListenFakt): Eine Liste von Metaparametern, zum Beispiel eine Timeout-Angabe oder eine Referenz auf die anfragende AO.

Nach der Eintragung des Anfrage-Fakts wird ein zugehöriger Timeout-Fakt erstellt. Wenn die Anfrage einen Timeout in den Metaparametern angibt, so wird dieser verwendet. Ist das nicht der Fall, so wird ein Standardwert verwendet, der in einer .properties-Datei definiert ist. Da dieser Wert in eine Konfigurationsdatei ausgelagert ist, kann dieser auch zur Laufzeit verändert werden.

[2]Es wird davon ausgegangen, dass Entwickler sich an diese W3C-Empfehlung gehalten haben

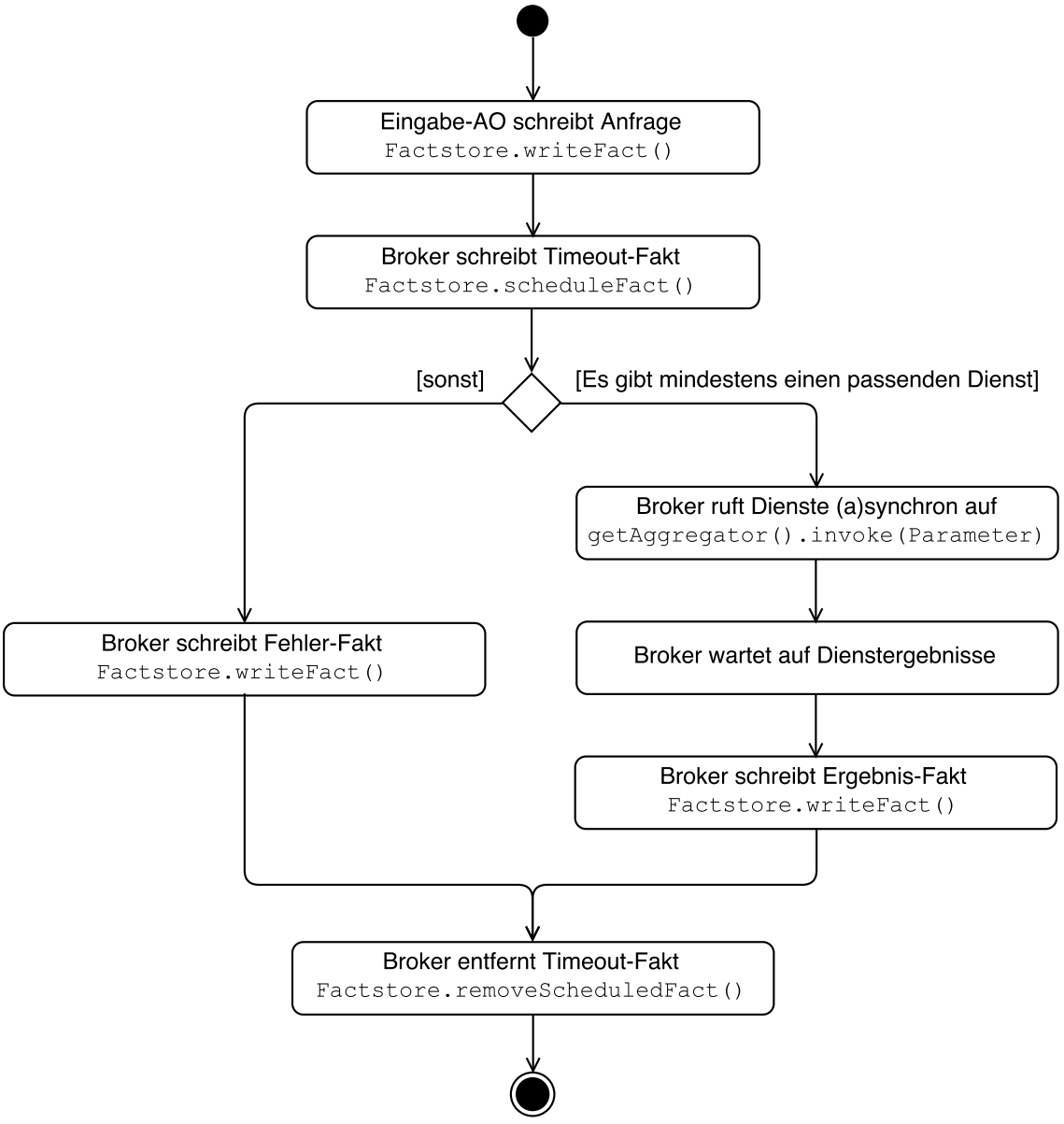

Abbildung 5.1.: Die Abarbeitung einer Dienstanfrage

Der Timeout-Fakt wird nicht unmittelbar in den Faktenspeicher geschrieben, sondern erst nach der definierten Timeout-Zeit. Der Faktenspeicher bietet dazu die Methode `scheduleFact(Fact, Date)` an. Wenn wir Ergebnisse vor dieser Zeit erhalten, wird der Timeout-Fakt mit `removeScheduledFact(Fact)` entfernt.
Wenn keine Ergebnisse vor Ablaufen dieser Zeit geliefert werden, so wird dieser Fakt in den Faktenspeicher geschrieben und alle noch aktiven Verbindungen zu Onlinediensten werden abgebrochen.

Als nächstes wird eine Liste aller passenden Onlinedienste erstellt. Es wird zuerst ein Fakt mit der Struktur eines Onlinedienstes aus der Anfrage erstellt, um die Unifizierung zu ermöglichen. Dieser Fakt wird der Methode `checkFact(Fact)` des Faktenspeichers übergeben, siehe Quelltextausschnitt 7.
`VariableIdentifier` dienen dazu, beliebige Fakten zuzulassen. Da es sich um benannte Variablen handelt, kann bei den Übereinstimmungen auf die Werte eindeutig zugegriffen werden.

Quelltextausschnitt 7: `findMatchingService`-Methode des Brokers

```
/**
 * Suche passende Dienste fuer eine Anfrage
 * @param  request Anfrage, fuer welche Dienste gefunden werden
 * @return Eine Menge an unifizierenden Fakten.
 *         In Map<Variable, Fact> ist angegeben, wie
 *         Platzhalter belegt wurden
 */
private Map<Fact, Map<Variable, Fact>> findMatchingServices(
   ServiceRequest request) {
    FactList matcherArgs = new FactList();
    matcherArgs.addFact(new VariableIdentifier("id"));
    matcherArgs.addFact(new SimpleFact(request.getCategory()));
    matcherArgs.addFact(new FactList(request.getArgumentNames()
       ));
    matcherArgs.addFact(new VariableIdentifier("opts"));
    matcherArgs.addFact(new VariableIdentifier("prio"));
    matcherArgs.addFact(new VariableIdentifier("json"));
    ComplexFact matcher = new ComplexFact(Broker.SERVICE,
       matcherArgs);
    Map<Fact, Map<Variable, Fact>> matching = getActiveServer()
       .getFactStore().checkFact(matcher);
    return matching;
}
```

Mit der `fromFact(Fact)` Methode von `OnlineService` wird im Anschluss eine Liste von selbigen aus der `Map` von Übereinstimmungen erstellt.

Nun unterscheidet man zwei Fälle: Es wurden passende Dienste gefunden oder nicht. Wenn keine passenden Dienste gefunden wurden, so wird ein entsprechender Fehler-Fakt in den Faktenspeicher geschrieben. Gleichzeitig wird auch der Timeout-Fakt entfernt.

Bei gefundenen Diensten wird der in der Anfrage übergebene Aggregationsmodus verwendet, um die Onlinedienste zu kontaktieren. Dafür reicht ein Aufruf der Methode `invoke`. Die Umsetzung dieser Schritte findet sich in Quelltextausschnitt 8.

Quelltextausschnitt 8: Ausschnitt der `request`-Methode

```
// kontaktiere Dienste, aggregiere Ergebnisse
val[a] matchings = findMatchingServices(request);
List<OnlineService> services = toServicesList(matchings);
List<ServiceCallResult> results = request.getAggregator().
   invoke(services, request.getArguments());

// in den FS werden nur die verarbeiteten Ergebnisse
// geschrieben
List<String> processedResults = getProcessedResults(results);

// entferne Timeout-Fakt
getActiveServer().getFactStore().removeFacts(timeoutFact);

// schreibe Ergebnisse in den Faktenspeicher
writeFact(createResponse(request.getId(), ResultStatus.SUCCESS,
    processedResults));
```

[a] automatische Typinferenz durch die Bibliothek „Project Lombok“

Bei allen implementierten Anfragemodi wird ein `ScheduledExecutorService` mit darunterliegendem `Threadpool` verwendet. Bei den parallelen Modi entspricht die Anzahl der erstellten Threads genau der Anzahl der verfügbarer Prozessoren, da dies laut Dokumentation die schnellsten Ergebnisse liefert.
`ScheduledExecutorService` unterstützt außerdem das kontrollierte Abbrechen von Threads mit `shutdownNow()`. Diese Methode wird von der `cancel()` Methode des Aggregators aufgerufen, wenn ein Timeout abgelaufen ist. Besonders bei `ParallelOne` ist diese Funktion auch von Vorteil: Sobald das erste Ergebnis verfügbar ist, werden alle anderen Verbindungen abgebrochen.

5.4. Ergebnisse

Vom `invoke`-Aufruf des `ServiceCallResultAggregators` wird eine Liste von `ServiceCallResults` zurückgegeben. Ein `ServiceCallResult` speichert die Information, von welchem Dienst es stammt, das unverarbeitete Ergebnis dieses Dienstes und das Ergebnis nach Anwendung des (dienstspezifisch angewendeten) regulären Ausdrucks. In den Faktenspeicher wird in dieser Implementierung nur letzteres geschrieben.

Im dritten Schritt wird mit `createResult` ein Ergebnis-Fakt erzeugt und in den Faktenspeicher gelegt (Quelltextausschnitt 9).

Quelltextausschnitt 9: Ausschnitt der `createResponse`-Methode des Brokers

```
/**
 * Erstellt einen Antwortfakt
 *
 * @param requestid
 * @param status
 * @param results
 * @return
 */
```

```
private Fact createResponse(int requestid, ResultStatus status,
    List<String> results) {
    FactList args = new FactList();
    args.addFact(new SimpleFact(requestid));
    args.addFact(new SimpleFact(status));
    args.addFacts(results);
    ComplexFact f = new ComplexFact(RESULT, args);

    return f;
}
```

In der Ergbnisliste befinden sich nur die verarbeiteten Resultate. Der Ergebnisfakt ist, wie man auch dem Methodenkörper entnehmen kann, immer nach diesem Schema aufgebaut:

`result(AnfrageId, Status, Ergebnisliste)`

Die AnfrageId ermöglicht es der Eingabe-AO eine Regel zu erstellen, die eindeutig auf diese Antwort feuert (`result(id, $, $)`). Dadurch wird die Eingabe-AO sofort benachrichtigt[3], wenn Ergebnisse vorliegen.

Es sind drei mögliche Status-Codes für ein Ergebnis definiert:

- Erfolg (`ResultStatus.SUCCESS`)
- Keine passenden Dienste gefunden (`ResultStatus.NO_SERVICES_FOUND`)
- Zeitüberschreitung (`ResultStatus.TIMEOUT`)

Im Erfolgsfall wird eine Liste der aggregierten Ergebnisse aller kontaktierten Dienste in den Faktenspeicher geschrieben, zum Beispiel:

`result(1, SUCCESS, [24, 15, 199])`

Wenn ein Dienst angefragt wird, für den keine passenden Dienste im Faktenspeicher gefunden werden, wird ein Ergebnis-Fakt mit dieser Information in den Faktenspeicher gelegt:

`result(2, NO_SERVICES_FOUND, [])`

Wenn der `invoke` Aufruf länger benötigt, als die Anfrage es per `timeout(Millisekunden)` erlaubt hat, findet eine Zeitüberschreitung statt.
Die Methode `scheduleFact`, welche bei der Dienstanfrage aufgerufen wurde, schreibt einen Ergebnis-Fakt mit dem Status `TIMEOUT` in den Faktenspeicher:

`result(3, TIMEOUT, [])`

Gleichzeitig werden alle laufenden Verbindungen zu Onlinediensten abgebrochen[4].

Wenn Onlinedienste gefunden werden und mit einem Fehlercode antworten, so können keine Nutzdaten aus dem Ergebnis extrahiert werden (bzw. es wird ein leerer Text extrahiert). Wenn alle Dienste auf diese Weise reagieren, wird ein `NO_SERVICES_FOUND`-Fakt generiert.

[3] Die maximale Verzögerung ist abhängig von der definierten (aber frei wählbaren) Länge des Auswertungszyklusses.
[4] Der zum Aufruf verwendete `ScheduledExecutorService` unterstützt diese Operation.

6. Evaluation

Die Bewertung meiner Bachelorarbeit wird anhand der Norm ISO/IEC 9126 für Softwarequalität vorgenommen. Die folgenden Qualitätsmerkmale wurden dabei gemäß der Norm herangezogen:

- Funktionalität (Tabelle 6.1)
- Zuverlässigkeit (Tabelle 6.2)
- Benutzbarkeit (Tabelle 6.3)
- Effizienz (Tabelle 6.4)
- Wartbarkeit (Tabelle 6.5)
- Übertragbarkeit (Tabelle 6.6)

Tabelle 6.1.: Teilmerkmale der Funktionalität

Angemessenheit	Der Broker eignet sich zur Erfüllung der Anfragen mittels Webdienste und Webformulare
Richtigkeit	Die Richtigkeit wurde mit Unit-Tests sichergestellt
Interopabilität	Als aktive Ontologie gliedert sich der Broker reibungslos in den EASIER Active Server ein
Sicherheit	Die Sicherheit des Brokers basiert auf der Sicherheit des Faktenspeichers des EASIER-Systems
Ordnungsmäßigkeit	Die Ordnungsmäßigkeit kann von Diensten erfüllt werden, die ein WSDL-Dokument bereitstellen
Konformität	Webstandards wurden ausführlich beachtet und bei der Implementierung berücksichtigt

Tabelle 6.2.: Teilmerkmale der Zuverlässigkeit

Reife	Die Versagenshäufigkeit hängt maßgeblich von den angesprochenen Diensten ab.
Fehlertoleranz	Durch Regressionstests wird die Fehlertoleranz gering gehalten.
Wiederherstellbarkeit	Nicht implementiert. Denkbar ist ein protokollierter Faktenspeicher.
Konformität	Unit-Tests sichern die Konformität.

Tabelle 6.3.: Teilmerkmale der Bedienbarkeit

Verständlichkeit	Der Broker ist einfach verständlich. Es müssen gegebene Klassen lediglich mit den konkreten Dienst-Parametern befüllt werden
Erlernbarkeit	Für die Registrierung eines neuen Dienstes muss lediglich eine `Webservice-` bzw. eine `Webformular`-Instanz erstellt werden.
Bedienbarkeit	Eingabe-AOs können Anfragen in den Faktenspeicher schreiben. Regeln des Brokers werden aktiviert und die Anfrage wird bearbeitet.
Attraktivität	-
Konformität	Die vom Broker generierten Fakten sind konform mit dem Fakten-Layout, welches in der Einleitung dargestellt wurde.

Tabelle 6.4.: Teilmerkmale der Effizienz

Zeitverhalten	Timeouts setzen eine Obergrenze für Antwortzeiten. Timeouts können von der Eingabe-AO gesetzt werden oder in einer externen .properties-Datei definiert sein.
Verbrauchsverhalten	Der Ressourcenverbrauch hängt hauptsächlich von den kontaktierten Diensten und der Formatierung der Dienstergebnisse ab. Vor allem bei Webformularen kann der Ressourcenverbrauch im Vorhinein schwer abgeschätzt werden.
Konformität	Bei der Entwicklung des Brokers wurde besonders Wert auf die Verwendung aktueller und effizienter Bibliotheken gelegt. Java 7 und die Bibliotheken Jsoup, Apache-XMLRPC, Google-Collections kommen zum Einsatz.

Tabelle 6.5.: Teilmerkmale der Wartbarkeit

Analysierbarkeit	Der Code ist ausführlich dokumentiert und in eine einfach verständliche Paket-Struktur aufgeteilt: `broker`, `broker.connect`, `broker.forms` , `broker.webservices` und `broker.rules`.
Modifizierbarkeit	Zwischen den Paketen herrscht eine geringe Kopplung, weshalb abgekapselte Klassen ohne unvorhersehbare Nebeneffektemodifiziert werden können. Dies gilt zum Beispiel für die Klassen der Aufrufmodi: `SequentialOne`, `ParallelOne` und `ParallelAll`.
Stabilität	Unerwartete Zustände sind durch ausführliches Testen unwahrscheinlich
Testbarkeit	Es existiert ein eigenes Java-Paket für Broker-Testfälle. Darin befinden sich bereits 11 Test-Klassen, die den Broker testen.
Konformität	-

Tabelle 6.6.: Teilmerkmale der Übertragbarkeit

Anpassbarkeit	Standardwerte werden extern abgelegt (`defaults.properties`). Vom Faktenspeicher kann mit beliebigen Programmiersprachen gelesen und geschrieben werden. Ebenso ist das verwendete Serialisierungsformat JSON unabhängig von der Programmiersprache.
Installierbarkeit	Der Broker wurde als Teil des Eclipse-Projektes des EASIER Active Server entwickelt und muss nicht separat installiert werden. Der Broker wird durch das hinzufügen einer neuen AO zu dem EASIER Active Server installiert.
Koexistenz	Neben dem Broker können beliebig viele weitere AOs ohne Reibungen koexistieren, die eine ähnliche Aufgabe übernehmen.
Austauschbarkeit	Der Broker kann ausgetauscht werden, in dem statt der Broker-AO eine andere Vermittler-AO im EASIER Active Server eingetragen wird.
Konformität	-

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine erweiterbare Broker-Architektur für den EASIER Active Server entworfen und implementiert. Externe Dienste (Webdienste und Webformulare) registrieren sich automatisiert oder manuell beim Broker. Sobald die Dienste beim Broker (und damit im Faktenspeicher) als existent gemeldet sind, lassen sie sich von einer oder mehreren aktiven Ontologien ansprechen.
Der Broker findet passende Dienste für eine Anfrage-AO, kontaktiert diese und gibt die aggregierten Ergebnisse zurück.
Der Broker ist selbst auch eine aktive Ontologie und gliedert sich somit reibungslos in das Konzept des EASIER Active Servers als unabhängige Komponente ein. Die Kommunikation und Datenhaltung findet über den einzigen Faktenspeicher statt.

Die Vorteile gegenüber [Guz08] spiegeln sich vor allem in zwei Punkten wieder: Webformulare und Automatisierbarkeit. Es werden die Webdienste SOAP, XML-RPC und REST unterstützt. Zusätzlich werden auch dynamische Webformulare unterstützt. In Guzzonis Arbeit werden solche nicht genannt. Webformulare sind dabei wesentlich komplexer, da sie keine fest definierte Schnittstelle bieten und sich der HTML-Code jederzeit ändern kann. Darüber hinaus sind bei Webformularen oft mehrere Aufrufe (hier als Hops bezeichnet) notwendig, um an die Ergebnisse zu gelangen. Weiterhin sind die Nutzdaten, welche die Anfrageergebnisse verpacken, auf einer Webseite wesentlich schwieriger auszumachen als dies bei einer Antwort eines Webdienstes der Fall ist. Auch hier hat unser Ansatz Schwierigkeiten, wenn sich die Webformulare grundsätzlich ändern.
Im hier beschriebenen Ansatz sind weiterhin die im OSI-Schichtenmodell unter einem Onlinedienst liegenden Protokolle austauschbar, exemplarisch wurden hier HTTP und HTTPS implementiert.

Durch ein automatisiertes und kontinuierliches Aktualisieren der Kategorien ist es möglich, dass der Broker und das Kategoriensystem fortlaufend aktuell bleiben und eine beachtliche Menge an verfügbaren Onlinediensten abgedeckt ist.
Ich unterstütze dabei die These von [AMM08] und [DFJ+04]: Bei einer rapiden Zunahme von öffentlich zugänglichen Webdiensten und Webformularen ist es nicht mehr praktikabel, ein manuell verwaltetes Verzeichnis zu nutzen. Ab dem Jahr 1994 wurden per Hand verwaltete Onlineverzeichnisse für Webseiten durch Suchmaschinen abgelöst, da die Anzahl der Webseiten rasch anwuchs. Hier sehe ich die Analogie zu Onlinediensten, da es hier ähnlich sein könnte.

Es werden verschiedene Anfragemodi und erste Ansätze zu Dienstgüteparametern unterstützt. Je nach Anwendungsfall haben die drei vorgestellten Anfragemodi Vor- und Nach-

teile. Wenn beispielsweise ein besonders verlässliches Ergebnis wichtig ist, dann sollte man Ergebnisse von mehreren Diensten in Betracht ziehen.
Da jedem Dienst eine Priorität zugewiesen ist, kann auch eine Sortierung der Dienste erfolgen, bevor der erste Dienst tatsächlich kontaktiert wird. In die Priorität können mehrere, unterschiedlich gewichtete Dienstgüteparameter einfließen. In dieser Implementierung wird zum Beispiel die Ausfallrate für jeden Dienst protokolliert.

Es wurde davon ausgegangen, dass sich Webseiteninhalte bei Webformularen nicht oder nur geringfügig nach der Registrierung ändern. Eine getroffene Einschränkung ist, dass sich die Ergebnisse aus dem HTML-Inhalt eindeutig mit einem regulären Ausdruck oder einer frei definierten Funktion extrahieren lassen. Bei starken HTML-Änderungen im Bereich der Ergebnisse kann dies zu einem Versagen der Extrahierung und damit zu unerwarteten oder schlechten Resultaten führen.
Da Webformulare automatisiert aufgefunden werden, sollte man sicherstellen, dass Webformulare, die bereits einmal erfolgreich kontaktiert werden konnten, in einen Index aufgenommen werden. Dieser Index wird regelmäßig aktualisiert, die Webformulare also neu interpretiert und möglicherweise mit leichten Änderungen beim Broker aktualisiert.

Der Broker ist um weitere Webdienste erweiterbar. Zum Beispiel JSON-RPC und WPS könnten hinzugefügt werden, um eine breitere Masse an existierenden Internetdiensten abzudecken. Zusätzlich ist eine CORBA-Schnittstelle interessant, da auch dieser Kommunikationsmechanismus von vielen Webseiten angeboten wird.

Eine weitere Eigenschaft, die dem Broker noch hinzugefügt werden kann, ist die regelmäßige Protokollierung des Faktenspeichers. Es werden in definierten Abschnitten Schnappschüsse des aktuellen Faktenspeichers erstellt. Dadurch wird es einerseits ermöglicht, den Broker in einen vorigen Zustand zurücksetzen zu können und andererseits kann die Protokollierung als wertvolles Überwachungs- und Analysewerkzeug verwendet werden.
Beispielsweise können Flaschenhälse leichter identifiziert werden und eine Analyse des Faktenspeicherinhalts über gewisse Zeiträume ließe sich visualisieren.
Wenn in einer aktiven Ontologie andere Programmiersprachen als Java zum Einsatz kommen, kann es sich lohnen den Faktenspeicher vom Arbeitsspeicher auf eine Festplatte zu verlagern. Man spart es sich dadurch, JVM-abhängige Java-Objekte im Arbeitsspeicher interpretieren zu müssen. Zudem ist der Faktenspeicher dann unabhängig von der Arbeitsspeicherverwaltung des Betriebssystems und kann zwischen heterogenen Systemen geteilt werden.

Es ist zu erwähnen, dass der vom Broker verwendete Mechanismus, um Anfragen und Dienste zusammenzuführen, die Unifikation, bei sehr großen Datenmengen vergleichsmäßig langsam werden kann. Große Datenmengen können jedoch schnell auftreten, wenn mit automatisiert gefundenen Diensten gearbeitet wird und viele Anfragen parallel eintreffen. Ein möglicher Lösungsansatz ist die Aufteilung des einzigen Faktenspeichers in mehrere Faktenspeicher. Vorstellbar ist, dass jeder Faktenspeicher nur Dienste einer festen Kategorie beherbergt.

Um noch bessere Anfrageergebnisse zu erhalten, kann es sich lohnen zusätzliche Dienstgüteparameter zu berücksichtigen, zum Beispiel Sicherheit, Latenz und Authentizifierung der Dienste durch Dritte. Manche Webdienste sind nur gegen Bezahlung nutzbar, was ebenfalls als QoS-Parameter modelliert werden kann. Der Broker kann bereits alle durchgeführten Paare von konkreten Anfragen mit konkreten Antworten speichern. Diese könnten noch weiter evaluiert werden.
In diesem Kontext könnte man bei den Anfragen auch noch angeben wollen, dass ein bestimmter Dienst verwendet werden soll. Gerade dies zu verbergen ist zwar eine der

Hauptaufgaben des Brokers, es kann aber durchaus sinnvoll sein, etwa wenn der Nutzer bei einem bestimmten Fluganbieter einen Preisnachlass erhält.

Literaturverzeichnis

[AMM07] Al-Masri, Eyhab ; Mahmoud, Qusay H.: Qos-based discovery and ranking of web services. In: *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on* IEEE, 2007, S. 529–534

[AMM08] Al-Masri, Eyhab ; Mahmoud, Qusay H.: Investigating web services on the world wide web. In: *Proceedings of the 17th international conference on World Wide Web* ACM, 2008, S. 795–804

[Bel14] Bellegarda, Jerome R.: Natural Interaction with Robots, Knowbots and Smartphones. 2014, Kapitel Spoken Language Understanding for Natural Interaction: The Siri Experience, S. 3–14

[BEL+15] Brown, Peter ; Estefan, Jeff A. ; Laskey, Ken ; McCabe, Francis G. ; Thornton, Danny: *Service Oriented Architecture : What Is SOA?* http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition. Version: 2015. – zuletzt abgerufen am 26.08.2015

[CDK+02] Curbera, Francisco ; Duftler, Matthew ; Khalaf, Rania ; Nagy, William ; Mukhi, Nirmal ; Weerawarana, Sanjiva: Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. In: *IEEE Internet computing* (2002), Nr. 2, S. 86–93

[CW04] Chatterjee, Sandeep ; Webber, James: *Developing Enterprise Web Services: An Architect's Guide*. Prentice Hall Professional, 2004. – 126–128 S. http://books.google.com/books?id=LEpPzQ5mRDoC&pg=PA126. – ISBN 978–0–13–140160–0. – zuletzt abgerufen am 26.08.2015

[DFJ+04] Ding, Li ; Finin, Tim ; Joshi, Anupam ; Pan, Rong ; Cost, R S. ; Peng, Yun ; Reddivari, Pavan ; Doshi, Vishal ; Sachs, Joel: Swoogle: a search and metadata engine for the semantic web. In: *Proceedings of the thirteenth ACM international conference on Information and knowledge management* ACM, 2004, S. 652–659

[DHC13] Dong, Hai ; Hussain, Farookh K. ; Chang, Elizabeth: Semantic Web Service matchmakers: state of the art and challenges. In: *Concurrency and Computation: Practice and Experience* 25 (2013), Nr. 7, S. 961–988

[GKS02] Gokhale, Aniruddha ; Kumar, Bharat ; Sahuguet, Arnaud: Reinventing the wheel? CORBA vs. Web services. In: *Proceedings of international world wide Web conference*, 2002

[GOS09] Guarino1, Nicola ; Oberle2, Daniel ; Staab3, Steffen: *What Is an Ontology?* http://userpages.uni-koblenz.de/~staab/Research/Publications/2009/handbookEdition2/what-is-an-ontology.pdf. Version: 2009. – zuletzt abgerufen am 26.08.2015

[GPST04] Garofalakis, John ; Panagis, Yannis ; Sakkopoulos, Evangelos ; Tsakalidis, Athanasios: Web service discovery mechanisms: Looking for a needle in a haystack. In: *International Workshop on Web Engineering* Bd. 38, 2004

[Gru92] Gruber, Tom: *What Is an Ontology?* http://www-ksl.stanford.edu/kst/what-is-an-ontology.html. Version: 1992. – zuletzt abgerufen am 26.08.2015

[Guz08] Guzzoni, Didier: *Active: A Unified Platform for Building Intelligent Applications*, EPFL Lausanne, Diss., 2008. http://biblion.epfl.ch/EPFL/theses/2008/3990/3990_abs.pdf. – zuletzt abgerufen am 26.08.2015

[HB04] Haas, Hugo ; Brown, Allen: *Web Services Glossary.* http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice. Version: 2004. – zuletzt abgerufen am 26.08.2015

[HBF+14] Hickson, Ian ; Berjon, Robin ; Faulkner, Steve ; Leithead, Travis ; Navara, Erika D. ; O'Connor, Edward ; Pfeiffer, Silvia: *HTML5.* http://www.w3.org/TR/2014/REC-html5-20141028/. Version: 2014. – zuletzt abgerufen am 26.08.2015

[Lin15] Lingel, Philipp: *Clustering von Internetdiensten für aktive Ontologien*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master-Thesis, 2015

[Min97] Minton, Gabriel: IIOP specification: A closer look. In: *Unix Review* 15 (1997), Nr. 1, S. 7

[ML07] Mitra, Nilo ; Lafon, Yves: *SOAP Version 1.2 Part 0: Primer (Second Edition).* http://www.w3.org/TR/2007/REC-soap12-part0-20070427/. Version: 2007. – zuletzt abgerufen am 26.08.2015

[MM03] McIlraith, Sheila A. ; Martin, David L.: Bringing semantics to web services. In: *Intelligent Systems, IEEE* 18 (2003), Nr. 1, S. 90–93

[NL05] Newcomer, Eric ; Lomow, Greg: *Understanding SOA with Web services.* Addison-Wesley, 2005

[OAS02] OASIS: *UDDI Registry tModels, Version 2.04.* http://www.uddi.org/taxonomies/UDDI_Registry_tModels.htm. Version: 2002. – zuletzt abgerufen am 27.08.2015

[OAS04] OASIS: *UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, Dated 20041019.* https://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm. Version: 2004. – zuletzt abgerufen am 26.08.2015

[OAS06] OASIS: *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004).* http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf. Version: 2006. – zuletzt abgerufen am 26.08.2015

[OAS07] OASIS: *tModels.* http://uddi.xml.org/tmodels. Version: 2007. – zuletzt abgerufen am 26.08.2015

[OMG08] OMG: *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2: CORBA Interoperability.* http://people.inf.elte.hu/toke/CORBA%20distr-object-management/OMG-current-doc-CORBA3.1/CORBA-08-01-07-part2.pdf. Version: 2008. – zuletzt abgerufen am 26.08.2015

[OMG12] OMG: *Documents Associated With CORBA, 3.3.* http://www.omg.org/spec/CORBA/3.3/. Version: 2012. – zuletzt abgerufen am 26.08.2015

[Raj98] Raj, Gopalan S.: A detailed comparison of CORBA, DCOM and Java/RMI. In: *Object Management Group (OMG) whitepaper* (1998)

[Sai16] Said, Wasim: *Abbildung von Webformularen auf aktive Ontologien*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master-Thesis, in Bearbeitung, voraussichtliches Abgabedatum 16.02.2016

[SPVVG03] Sycara, Katia ; Paolucci, Massimo ; Van Velsen, Martin ; Giampapa, Joseph: The retsina mas infrastructure. In: *Autonomous agents and multi-agent systems* 7 (2003), Nr. 1-2, S. 29–48

[VR03] Van Rees, Reinout: Clarity in the usage of the terms ontology, taxonomy and classification. In: *CIB REPORT* 284 (2003), Nr. 432, S. 1–8

[Win99] Winer, Dave: *XML-RPC Specification.* http://xmlrpc.scripting.com/spec.html. Version: 1999. – zuletzt abgerufen am 24.08.2015

Anhang

A. Beispiele der Protokolle

A.1. WSDL

Quelltextausschnitt 10: Beispiel einer WSDL-Datei (http://www.ripedevelopment.com/webservices/LocalTime.asmx).

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:tm="http://microsoft.com/wsdl/mime/
   textMatching/" xmlns:soapenc="http://schemas.xmlsoap.org/
   soap/encoding/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/
   mime/" xmlns:tns="http://www.ripedev.com/" xmlns:soap="http
   ://schemas.xmlsoap.org/wsdl/soap/" xmlns:s="http://www.w3.
   org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org
   /wsdl/soap12/" xmlns:http="http://schemas.xmlsoap.org/wsdl/
   http/" targetNamespace="http://www.ripedev.com/" xmlns:wsdl=
   "http://schemas.xmlsoap.org/wsdl/">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/
     wsdl/">Provides local time for the supplied zip code</wsdl
     :documentation>
  <wsdl:types>
    <s:schema elementFormDefault="qualified" targetNamespace="
       http://www.ripedev.com/">
      <s:element name="LocalTimeByZipCode">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="
               ZipCode" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="LocalTimeByZipCodeResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="
               LocalTimeByZipCodeResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
```

```
</wsdl:types>
<wsdl:message name="LocalTimeByZipCodeSoapIn">
  <wsdl:part name="parameters" element="tns:
     LocalTimeByZipCode" />
</wsdl:message>
<wsdl:message name="LocalTimeByZipCodeSoapOut">
  <wsdl:part name="parameters" element="tns:
     LocalTimeByZipCodeResponse" />
</wsdl:message>
<wsdl:portType name="LocalTimeSoap">
  <wsdl:operation name="LocalTimeByZipCode">
    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.
       org/wsdl/">Returns local time for a given zip code.</
       wsdl:documentation>
    <wsdl:input message="tns:LocalTimeByZipCodeSoapIn" />
    <wsdl:output message="tns:LocalTimeByZipCodeSoapOut" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="LocalTimeSoap" type="tns:LocalTimeSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/
     http" />
  <wsdl:operation name="LocalTimeByZipCode">
    <soap:operation soapAction="http://www.ripedev.com/
       LocalTimeByZipCode" style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="LocalTimeSoap12" type="tns:LocalTimeSoap"
   >
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/
     http" />
  <wsdl:operation name="LocalTimeByZipCode">
    <soap12:operation soapAction="http://www.ripedev.com/
       LocalTimeByZipCode" style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="LocalTime">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/
     wsdl/">Provides local time for the supplied zip code</
     wsdl:documentation>
```

```
    <wsdl:port name="LocalTimeSoap" binding="tns:LocalTimeSoap"
       >
      <soap:address location="http://www.ripedevelopment.com/
         webservices/LocalTime.asmx" />
    </wsdl:port>
    <wsdl:port name="LocalTimeSoap12" binding="tns:
       LocalTimeSoap12">
      <soap12:address location="http://www.ripedevelopment.com/
         webservices/LocalTime.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

A.2. tModel Beispiel

Quelltextausschnitt 11: Beispiel eines tModels [OAS02].

```
<tModel tModelKey="uuid:AC104DCC-D623-452f-88A7-F8ACD94D9B2B">
  <name>uddi-org:inquiry_v2</name>
  <description xml:lang="en">UDDI Inquiry API Version 2
     - Core Specification</description>
  <overviewDoc>
    <description xml:lang="en">
      This tModel defines the inquiry API calls for interacting
          with a V2 UDDI
      node.
    </description>
    <overviewURL>
      http://www.uddi.org/wsdl/inquire_v2.wsdl
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39
       B756E62AB4"
      keyName="types"
      keyValue="specification"/>
    <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39
       B756E62AB4"
      keyName="types"
      keyValue="xmlSpec"/>
    <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39
       B756E62AB4"
      keyName="types"
      keyValue="soapSpec"/>
    <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39
       B756E62AB4"
      keyName="types"
      keyValue="wsdlSpec"/>
   </categoryBag>
</tModel>
```