T U M

I N S T I T U T F Ü R I N F O R M A T I K

Kontextadaptivität in dienstbasierten Softwaresystemen

Martin Deubler, Johannes Grünbauer, Andreas Holzbach,
Gerhard Popp, Guido Wimmel

TUM-I0511
Juli 05

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-07-I0511-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Kontextadaptivität in dienstbasierten Softwaresystemen[1]

Martin Deubler, Johannes Grünbauer,
Andreas Holzbach, Gerhard Popp, Guido Wimmel

Technische Universität München, Institut für Informatik,
Boltzmannstraße 3, 85748 Garching

{deubler,gruenbau,holzbach,popp,wimmel}@in.tum.de

20. Juli 2005

Kurzfassung:

Das Konzept der Dienste erweist sich in den frühen Phasen der Systementwicklung bei der Modellierung von verteilten, multifunktionalen Systemen durch die Strukturierung der Systemanforderungen als geeignet. Diese beziehungsorientierte Beschreibung der Systemfunktionen stellt eine zusätzliche Abstraktionsstufe im Entwicklungsprozess dar, bei der das Dienstverhalten aus Nutzersicht repräsentiert wird.

Während sich der klassische Dienstbegriff bereits etabliert hat, spielt die Dynamik von Diensten immer mehr eine bedeutende Rolle in neuen, hochkomplexen Softwaresystemen. So genannte adaptive Dienste müssen sich auf das Vorhandensein weiterer Dienste, auf aktuelle Belegungen von Konfigurationsparametern sowie auf Umgebungsparameter anpassen können. In diesem technischen Bericht stellen wir den Aspekt der Adaptivität von Diensten vor und präsentieren die damit eng verbundene Modellierung von Umgebungseigenschaften in Kontexten. An einer Fallstudie aus der Automotive Domäne werden die Konzepte exemplarisch vorgestellt.

[1] Diese Arbeit wird von der Bayerischen High-Tech-Offensive innerhalb des Projekts MEwaDis gefördert.

Inhaltsverzeichnis

1 Einleitung

Zur Modellierung verteilter, multifunktionaler Systeme eignet sich das Konzept der *Dienste* (engl. *Services*, siehe u.a. [Bro03]). Im Gegensatz zu herkömmlichen Entwicklungstechniken steht hier innerhalb der Entwicklung, insbesondere im Requirements Engineering, nicht die komponentenorientierte Modellierung im Vordergrund, sondern die Modellierung von Systemfunktionen (Diensten) und deren Beziehungen. Dies ermöglicht bereits in einer frühen Phase der Entwicklung die Strukturierung der Systemanforderungen, was insbesondere bei komplexen Abhängigkeiten der Dienste untereinander von Vorteil ist, wie sie in multifunktionalen Systemen (beispielsweise im Bereich der Automobilelektronik oder Telekommunikation) auftreten. Bei der dienstorientierten Modellierung wird die Funktionalität eines Systems entwickelt, ohne darauf zu achten, auf welcher Komponente eine Funktion später ausgeführt wird. Dienste können flexibel und dynamisch miteinander kombiniert werden, um eine Gesamtfunktionalität zu erbringen.

Das dienstbasierte Paradigma stellt mit der beziehungsorientierten Entwicklung von Diensten eine zusätzliche Abstraktionsstufe im Entwicklungsprozess multifunktionaler, verteilter Systeme dar. Das Dienstverhalten wird aus Nutzersicht modelliert, d.h. für jeden Dienst wird spezifiziert, wie sich der Dienst als Ganzes verhält. Diese Black-Box-Sichten werden mit der Spezifikation von Dienstbeziehungen und mit Diensteigenschaften (beispielsweise Quality-of-Service Attributen) angereichert. Im Gegensatz hierzu wird bei der komponentenorientierten Modellierung die Struktur der Komponenten als Unterteilung in Unterkomponenten (Glass-Box-Sicht) mit festgelegt. Im Anschluss an die dienstbasierte Modellierung, die eine neue Sicht auf den Prozess des Requirements Engineering darstellt, können Dienste auf Komponenten abgebildet werden.

Neben expliziten Interaktionen mit anderen Diensten sollten Dienste ihr Verhalten auch an relevante Umgebungseigenschaften anpassen können. Hierzu wird ein gemeinsamer *Kontext* modelliert und die Kontextabhängigkeit der Dienste über *implizite Schnittstellen* berücksichtigt. Dienste, die auf Kontextinformationen reagieren, werden als *adaptive Dienste* bezeichnet. Die Adaptivität von Diensten spielt bei der *Dienstkombination* eine bedeutende Rolle, da sich die Dienste über ihre impliziten Schnittstellen zum Kontext beeinflussen können.

Eine Anwendungsdomäne für adaptive dienstbasierte Systeme ist die Fahrzeugelektronik. In Fahrzeugen werden typischerweise die Funktionen auf verschiedene ECUs (Electronic Control Units) verteilt, die über verschiedene Bussysteme (MOST, CAN) miteinander verbunden sind [DGP+04b]. Wie in anderen Applikationen können Funktionen voneinander abhängig sein und miteinander interagieren. In der multifunktionalen Fahrzeugelektronik müssen sich jedoch die Systemfunktionen sehr stark auf Umgebungs-

eigenschaften anpassen, wie z.B. die aktuelle Fahrzeuggeschwindigkeit oder ankommende Telefonanrufe. Die Mensch-Maschine-Schnittstelle (engl. *Man Machine Interface*, MMI) entscheidet beispielsweise in Abhängigkeit von der aktuellen Fahrzeuggeschwindigkeit über die Anzeige der Telefonnummer des Anrufers in einem Fahrzeugdisplay und stoppt bei der Annahme des Gesprächs das Autoradio.

In diesem Bericht stellen wir zunächst die klassische Sichtweise der dienstorientierten, modellbasierten Softwareentwicklung und deren formale Grundlagen vor (Kapitel 2). Kapitel 3 beschäftigt sich mit der Eigenschaft der *Adaptivität* bei der dienstorientierten Modellierung und in Kapitel 4 wird der damit eng in Beziehung stehende Begriff des *Kontexts* untersucht. Insbesondere wird in Hinblick auf die Austauschbarkeit des Kontextmodells eine Trennung zwischen Kontextwert, Kontextmodell und adaptivem Dienst eingeführt. Eine Fallstudie aus der Anwendungsdomäne Fahrzeugelektronik wird in Kapitel 5 vorgestellt.

2 Klassische Sichtweise auf Dienste

In diesem Kapitel werden die formalen Grundlagen erklärt, die notwendig sind, um einen Dienst zu beschreiben. Hierzu wird die Spezifikationssprache FOCUS verwendet. Um Dienste einheitlich beschreiben zu können, müssen sie bestimmten Konventionen genügen. Diese Kapitel zeigt die Betrachtungsweise auf Dienste im Projekt MEWADIS. Am Ende des Kapitels wird auf die modellbasierte Entwicklung von dienstbasierten Systemen eingegangen.

2.1 Formale Grundlagen

Als formale Grundlage zur Spezifikation von Diensten verwenden wir FOCUS [BS01]. FOCUS ist eine Spezifikationssprache, die auf Strömen basiert. Im Folgenden werden wir die Notation kurz erklären.

Für eine Menge M werden mit M^* die Menge der endlichen Sequenzen über die Elemente von M bezeichnet. Mit M^∞ werden die Mengen der unendlichen Sequenzen aller Elemente von M bezeichnet. Diese können durch die Funktion $\mathbb{N} \to M$ beschrieben werden. M^ω bezeichnet die Menge der endlichen und unendlichen *ungezeiteten* Ströme und steht für $M^* \cup M^\infty$.

Für eine Menge von Nachrichten M definiert die Funktion $s : \mathbb{N} \to M^\infty$ einen gezeiteten Strom. Zu einer Zeit t bezeichnet die Sequenz $s(t)$ die Sequenz aller Nachrichten, die zum Zeitpunkt t im Strom s vorhanden sind.

Kanäle sind Bezeichner für Ströme. Sei I die Menge der Eingabekanäle und O die Menge der Ausgabekanäle. Zu jedem Kanal c in der Menge der Kanäle $I \cup O$ gibt es einen Datentyp $Type(c)$.

Im Weiteren werden folgende Notationen für gezeitete Ströme s verwendet (sei S eine Menge von Nachrichten, $k \in \mathbb{N}$):

$z \frown s$	Konkatenation einer Sequenz oder eines Stromes z mit einem Strom s
$S©s$	Teilstrom von s, in dem sich nur Elemente aus S befinden.
$S\#s$	Anzahl der Elemente in s, die in S vorkommen
$s.k$	k-te Sequenz im Strom s
$s \downarrow k$	die ersten k Sequenzen im gezeiteten Strom s
$s \uparrow k$	Strom s ohne die ersten k Sequenzen
$\overline{s}$	endlicher oder unendlicher ungezeiteter Strom, der das Ergebnis der Konkatenation aller Sequenzen aus s darstellt.

2.2 Dienste als Modellierungseinheit

Ein Dienst ist eine funktionale, abstrakte Einheit zur Modellierung von Systemen und steht im Mittelpunkt des Entwicklungsprozesses für dienstbasierte Systeme. Ein Dienst kann im Gegensatz zu einer Komponente partiell spezifiziert sein. Das bedeutet, dass das Verhalten nur für einen Teil der möglichen Eingaben festgelegt werden muss, während es für andere Eingaben offen gelassen werden kann. Beispielsweise kann die Modellierung des Fehlverhaltens auf der aktuell betrachteten Abstraktionsebene noch irrelevant sein. Ein Dienst ist partiell definiert bezüglich der angenommenen Umgebung. Speziell bei adaptiven Diensten (siehe Kapitel 3) sollten lediglich die relevanten Umgebungseigenschaften (Kontext) betrachtet werden.

In MEwaDis wird folgende Sichtweise auf einen Dienst verwendet. Ein Dienst besteht aus:

(i) Einer syntaktischen Schnittstelle. Durch diese werden die Nachrichten spezifiziert, die über die Schnittstelle laufen.

(ii) Einer Verhaltensspezifikation (semantische Schnittstelle). Das Verhalten wird durch eine stromverarbeitende, partielle Funktion beschrieben, die gültige Sequenzen von Eingabenachrichten auf Mengen von Sequenzen von Ausgaben abbildet.

(iii) Einer Menge von Eigenschaften. Dies sind Annotationen am Dienst und können verschiedene Aufgaben haben, z. B. die Beschreibung des Dienstes oder von Quality-of-Service-Attributen.

(iv) Beziehungen zu anderen Diensten. Dies schließt auch die Beschreibung der Arten der Beziehungen ein, die auf verschiedenen Abstraktionsebenen beschrieben werden können.

Dienste können in Focus spezifiziert werden (siehe hierzu [Bro05]). Folgend wird lediglich die zum Verständnis notwendige Notation präsentiert.

2.2.1 Die syntaktische Schnittstelle

Als syntaktische Schnittstelle (vgl. [Bro03]) eines Dienstes werden die Nachrichten bezeichnet, welche über die einzelnen Ports/Kanäle des Dienstes laufen können.

Sei I eine Menge von getypten Eingabekanälen und O eine Menge von getypten Ausgabekanälen eines Dienstes. Die syntaktische Schnittstelle eines Dienstes wird durch das Paar (I, O) charakterisiert und durch $(I \blacktriangleright O)$ bezeichnet.

Die Verbindung eines Dienstes zu seiner Umgebung (im Allgemeinen zu weiteren Diensten) geschieht exklusiv durch Kanäle, über die verschiedene Nachrichten verschiedener Typen ausgetauscht werden können. Die syntaktische Schnittstelle enthält keine Beschreibung des Dienstverhaltens.

2.2.2 Die semantische Schnittstelle

Das Verhalten eines Dienstes mit der syntaktischen Schnittstelle (I, O) ist durch die Funktion $F : \overrightarrow{I} \to \wp(\overrightarrow{O})$ gegeben. F muss die *„Timing Property“* nur für die nichtleeren Ausgaben erfüllen. Sei $x, z \in \overrightarrow{I}, y \in \overrightarrow{O}, t \in \mathbb{N}$:

$$F.x \neq \varnothing \neq F.z \wedge x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t + 1 : y \in F(x)\} = \{y \downarrow t + 1 : y \in F(z)\}$$

Die *Timing Property* (vgl. [BS01]) stellt eine (strenge) Kausalität des Verhaltens sicher, in dem Sinne, dass Ausgaben bis zum Zeitintervall $t + 1$ nur von Eingaben bis zum Zeitintervall t abhängen dürfen. Die Menge $\mathrm{dom}(F) = \{x : F.x \neq \varnothing\}$ heißt *Dienstdomäne*. Die Menge $\mathsf{rng}(F) = \{y \in F.x : x \in \mathrm{dom}(F)\}$ ist der *Wertebereich* des Dienstes. Mit

$$\mathbb{F}[I \blacktriangleright O]$$

wird die Menge aller Dienstschnittstellen mit den Eingabekanälen I und den Ausgabekanälen O bezeichnet. Mit $\mathbb{F}$ wird die Menge aller semantischen Schnittstellen für beliebige Mengen von Kanälen I und O bezeichnet.

2.2.3 Eigenschaften eines Dienstes

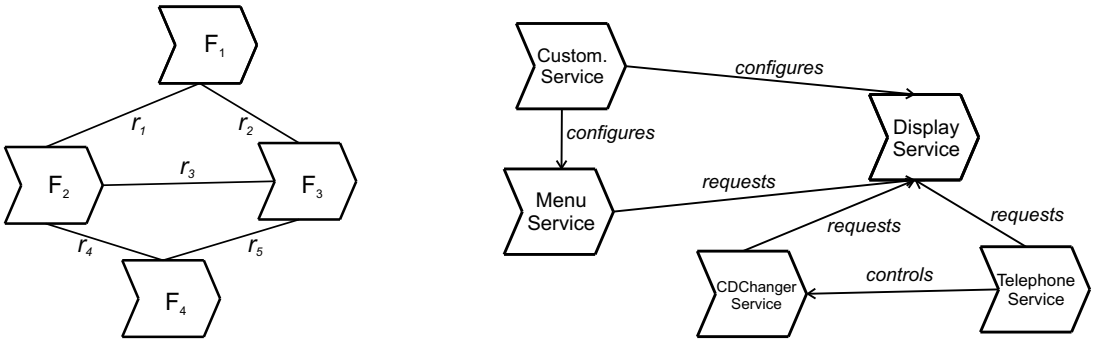
Mit einem Dienst können verschiedene Eigenschaften verbunden werden. Dies dient zur Beschreibung und Identifikation des Dienstes und geschieht beispielsweise mit Identifikatoren, textuellen Beschreibungen etc. Beispiele für Eigenschaften können QoS-Attribute sein. Beispiele für Eigenschaften, speziell für adaptive Dienste, werden in Kapitel 3.2 genauer beschrieben.

2.2.4 Beziehungen zwischen Diensten

Da Dienste eine Funktionalität bieten, interagieren sie mit anderen Diensten im System oder der Umgebung. Dienste beeinflussen sich gegenseitig oder sind voneinander abhängig. Diese Tatsache spiegelt sich in der Modellierung wieder, d.h. Dienste werden miteinander in Beziehung gesetzt. Diese Beziehungen besitzen je nach Typ eine unterschiedliche Semantik. Dies soll in einer frühen Entwicklungsphase helfen, fehlende Abhängigkeiten zu erkennen, verbotene Abhängigkeiten aufzuspüren und versteckte Abhängigkeiten sichtbar zu machen.

In Abbildung 2.1(a) sind beispielhaft einzelne Beziehungen zwischen Diensten zu erkennen: Dienst F_3 steht in Beziehung zu den Diensten F_1, F_2 und F_4. Die Beziehungen werden durch die Relationen r_2, r_3 sowie r_5 beschrieben.

Um die Interaktionen genauer zu beschreiben, können die Relationen verfeinert werden. Die Beziehung mit dem höchsten Abstraktionsgrad ist „interacts“. Auf deren Basis können nun weitere Beziehungen verfeinert werden, wie z. B. „configures“, „requests“ oder „controls“. In Abb.2.1(b) wird dies anhand einer Grafik aus [DGP+04a] ersichtlich. Einen Vorschlag zu einem Satz von Beziehungen beschreibt [Deu05]. Die Gesamtheit der in Beziehung stehenden Dienste wird als *logische Dienstarchitektur* bezeichnet.



(a) Beziehungen (b) Nutzungssicht

Abbildung 2.1: Beziehungen zwischen Diensten

2.2.5 Unterschied zwischen Dienst und Komponente

Eine Komponente ist definiert als eine wieder verwendbare Einheit von einem bestimmten Verhalten, die durch ihr Verhalten und Struktur definiert ist (siehe [BS01, SS03]). Die Struktur einer Komponente ist dadurch definiert, dass Subkomponenten zusammen mit ihrem Verhalten zu einer Komponente hinzugefügt werden. Eine Komponente spezifiziert somit Attribute über ihr Verhalten und ihre Struktur. Ein Dienst hingegen enthält eine abstraktere Spezifikation seines Verhaltens.

Der komponentenbasierte Ansatz weist so genannte „Features“ individuellen Komponenten zu und versucht, mit diesen Systeme zu erstellen. Der dienstbasierte Ansatz geht die andere Richtung. Hier wird beginnend vom Requirements Engineering das System in seine Funktionalitäten (Dienste) aufgeteilt und in Beziehung gesetzt (Nutzungssicht). Erst zu einem späteren Zeitpunkt werden die Dienste den Komponenten zugewiesen (*Deployment*).

Sowohl Dienste als auch Komponenten besitzen Schnittstellen (*Interfaces*), welche die Typen von Nachrichten, die über diese Schnittstelle laufen, definieren. Eine Komponente ist *total* definiert. Sie gibt für jede Eingabe eine Ausgabe zurück. Somit sind bei einer Komponente grundsätzlich alle Fälle abgedeckt. Anders hingegen ist es bei einem Dienst. Dieser ist lediglich *partiell* definiert und gibt nur für gültig Eingaben, also für Werte die in der Eingabedomäne sind, auch Ausgaben zurück. Mit anderen Worten: Bei einem Dienst werden nur die Fälle spezifiziert, die auch interessant sind. Diese Unterspezifikation hilft, sich am Anfang der Modellierung nicht in Details zu verlieren und sämtliche Fälle abdecken zu müssen.

2.3 Modellbasierte Entwicklung

Abgestützt auf Modelle, wie die oben beschriebenen, lassen sich dienstbasierte Systeme nach dem Paradigma der *modellbasierten Entwicklung* erstellen. Modellbasierte Entwick-

lung ist charakterisiert durch die durchgängige Verwendung explizit definierter Modelle innerhalb des Entwicklungsprozesses. Modelle beschreiben hierbei allgemein Abstraktionen zur Konzentration auf die für die verschiedenen Entwicklungsaktivitäten relevanten Aspekte eines Systems. In frühen Phasen ist die Modellierung plattform- und sprachunabhängig. Eine *Sicht* ist ein Ausschnitt aus einem Modell zur Darstellung eines bestimmten Teilaspekts, beispielsweise Struktur, Verhalten, oder auch die in Abbildung 2.1(b) gezeigte Nutzungssicht auf ein dienstbasiertes System, und wird mit Hilfe einer (in der Regel graphischen) Beschreibungstechnik dargestellt. Die Sichten sind über Beziehungen untereinander integriert, beispielsweise durch die Zuordnung von Anforderungen in einer Anforderungssicht zu Diensten in einer Nutzungssicht. Da eine Sicht selbst wieder ein Modell darstellt, wird im Folgenden nicht weiter zwischen diesen Begriffen unterschieden.

Die explizite Modellierung ermöglicht es bereits von frühen Phasen an, Modelle auf Konsistenz zu überprüfen, zu simulieren, zu verifizieren, Code oder Testfälle zu generieren. Zur Simulation, Verifikation, Code- oder Testfallgenerierung muss hierbei ein unterliegender Formalismus (Semantik) definiert werden. Solche Entwicklungsschritte und auch die Modellierung selbst können durch geeignete Entwicklungswerkzeuge unterstützt werden.

Schließlich kann auch der Entwicklungsprozess selbst durch ein explizites Modell beschrieben werden, üblicherweise basierend auf Entwicklungsartefakten und -aktivitäten. Der Schwerpunkt dieses Berichts liegt auf den Aspekten Adaptivität und Kontext im Rahmen einer dienstbasierten Modellierung. Der zugrunde liegende dienstbasierte Prozess ist in [DGP+04b] beschrieben.

Eine ausführlichere Beschreibung der Konzepte der modellbasierten Entwicklung gibt [SPHP02].

3 Adaptive Dienste

Im Rahmen der Darstellung der klassischen Sichtweisen wurde der Begriff des Dienstes eingeführt. In heutigen hochgradig verteilten, multifunktionalen Systemen spielt die dynamische Anpassung von Dienten eine weit reichende Rolle. Hierzu stellen wir in diesem Kapitel *adaptive Dienste* vor. Neben einer Definition führen wir unter anderem Adaptionsstufen und eine grafische Notation ein, diskutieren die Eigenschaften adaptiver Dienste und vergleichen dynamische und konfigurierbare Eigenschaften adaptiver Dienste.

3.1 Begriffsbildung

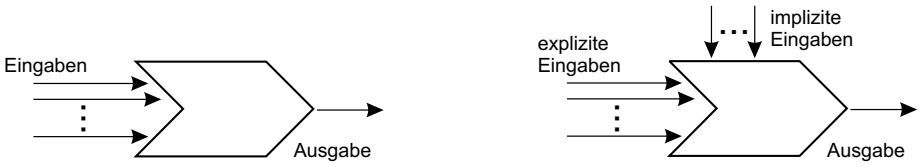
Ein adaptiver Dienst unterscheidet sich zu einem herkömmlichen Dienst insofern, als er zwei unterschiedliche Arten von Eingabekanälen besitzt. Über die *expliziten* Eingabekanäle bekommt der adaptive Dienst Eingaben vom *System*, d.h. üblicherweise von anderen Diensten bzw. Benutzereingaben. Die *impliziten* Eingabekanäle geben Werte aus dem *Kontext* an den Dienst weiter.

Definition 3.1 (Eingaben eines adaptiven Diensts) *Das syntaktische Interface eines adaptiven Diensts ist formal definiert durch* $(I \blacktriangleright O)$*, wobei* I *die Menge der Eingabekanäle und* O *die Menge der Ausgabekanäle ist. Die Menge der Eingabekanäle* I *ist die Vereinigung der Menge der expliziten Eingabekanäle* I_E *mit der Menge der impliziten Eingabekanäle* I_I*, also* $I = I_E \cup I_I$*.*

Explizite Eingaben *sind die Menge der Nachrichtenströme auf den Eingabekanälen* I_E*.*
Implizite Eingaben *sind die Menge der Nachrichtenströme auf den Eingabekanälen* I_I*.* □

Für die grafische Notation eines adaptiven Dienstes wird das Pfeil-Symbol verwendet (siehe Abbildung 3.1(a)). Die expliziten Eingabekanäle werden als waagerecht eingehende Pfeile repräsentiert. In der Erweiterung zu einem adaptiven Dienst werden die impliziten Eingabekanäle durch senkrecht eingehende Pfeile repräsentiert (siehe Abbildung 3.1(b)).

Zwischen dem System, das die expliziten Eingaben an den Dienst sendet und dem Kontext, der für die impliziten Eingaben zuständig ist, gibt es keine im Allgemeinen gültige Grenze. Diese kann je nach Anwendung und Entwurf unterschiedlich gezogen werden. Für die Modellierung eines adaptiven Dienstes und dessen Kontext ist es jedoch hilfreich, die Grenze so zu entwerfen, dass der Dienst auch ohne Einbeziehung von impliziten Eingaben, also Kontextinformation, ein wohldefiniertes Verhalten zeigt. Damit kann das Verhalten des Dienstes sowohl mit als auch ohne Einbeziehung des Kontexts spezifiziert werden.



(a) Dienst (b) adaptiver Dienst

Abbildung 3.1: Erweiterung eines Dienstes zu einem *adaptiven* Dienst

Definition 3.2 (Kontextneutralität eines adaptiven Diensts) *Ein adaptiver Dienst besitzt ein wohldefiniertes Verhalten auch ohne Vorhandensein von relevanter Kontextinformation (z. B. Ausfall eines Sensors). Dies ist der Standardfall in einer Spezifikation.* □

In Anlehnung an die Definition des „Splicing“ in [Bro05] kann dies auch formal wie folgt ausgedrückt werden. Sei ad ein adaptiver Dienst, gegeben durch

$$ad \in \mathbb{F}[I_E \cup I_I \blacktriangleright O]$$

wobei I_E die Menge der expliziten Eingabekanäle und I_I die Menge der impliziten Eingabekanäle ist. Der adaptive Dienst ad besitzt auch ohne implizite Eingaben ein wohldefiniertes Verhalten, das durch die Dienstfunktion ad'

$$ad' = ad \dagger (I_E \blacktriangleright O) \text{ mit } ad'.x' = \{y : \exists\, x : x' = x \mid I_E \wedge y \in ad.x\}$$

definiert ist und als „Splicing von ad“ bezeichnet wird. In Abbildung 4.2 entspricht dies dem Ausblenden der impliziten Eingaben. Der Dienst ad' ist ein *nicht-adaptiver Dienst*. Die Spezifikation von ad' beinhaltet ausschließlich die „ungestörte“ Funktionalität, also die Abbildung von (expliziten) Eingabeströmen auf Ausgabeströme [BS01].

Bevor wir untersuchen, was adaptive Dienste sind beziehungsweise wie das Verhalten adaptiver Dienste charakterisiert werden kann, soll zunächst die Frage beantwortet werden, warum ein Dienst überhaupt adaptiv sein sollte.

In so genannten *multifunktionalen Systemen*, in denen eine Funktionalität simultan von einer Menge anderer Funktionen benutzt werden kann, passt diese sich dem gegebenen Nutzungskontext an [DGP+04b], d.h. es findet eine situationsbedingte Anpassung von Funktionalität statt. Dadurch soll erreicht werden, dass in möglichst vielen Situationen oder Anwendungsfällen (sich ändernder Kontext) bestimmte Funktionalitäten (optimal) genutzt werden können (Ubiquität).

Adaptivität im Zusammenhang mit Kontext ist eine automatisierte oder teilautomatisierte Anpassung an bestimmte Kontextbedingungen, wie zum Beispiel die Verfügbarkeit bestimmter Kontextinformation oder Werte von Umgebungseigenschaftstypen (siehe auch Definition 3.3). Die Anpassung kann in der Bedeutung sowohl von einer einfachen Transformation von Informationen (Berechnung, Übersetzung) reichen, bis hin zu geändertem

(angepassten) Verhalten. Bei der Nutzung eines Systems ohne Adaptivität müssten notwendige Anpassungen ausschließlich vom Nutzer selbst vorgenommen werden. Der Nutzer selbst ist hier der so genannte *Mediator*. Im Falle eines menschlichen Nutzers werden Anpassungen stets *manuell* vorgenommen, sei es bewusst oder unbewusst.

Beispiel 3.1 Adaptivität eines menschlichen Nutzers
Auf dem Tachometer eines Fahrzeugs sei die einzige Einheit (mph). Die Information „Geschwindigkeit: maximal 50“ hat im Kontext „Deutschland“ eine andere Bedeutung als etwa im Kontext „US“. Der menschliche Nutzer wird in der Regel sein Verhalten im Kontext „Deutschland“ hier selbständig auf ca. 30 mph adaptieren. □

Adaptive Dienste übernehmen durch automatisierte oder teilautomatisierte Anpassung ganz oder teilweise die Funktion des *Mediator*. Dazu nimmt ein adaptiver Dienst gemäß Definition 4.3 stets die Rolle des *Aktuators* ein. In den Fällen, in denen er die Ergebnisse seiner Berechnungen nicht nur unmittelbar dem Dienstnutzer, sondern sie als (neue) Kontextinformation zur Verfügung stellt, ist er auch *Sensor* zugleich. Die Interpretation beliebiger Umgebungseigenschaften, die durch eine komplexe Entscheidungslogik zu bestimmten Schlussfolgerungen führt, ist nicht die Aufgabe eines adaptiven Diensts. Dies ist die Aufgabe von Kontextmodellen. Verschiedene Kontextmodelle werden in Kapitel 4 diskutiert. Ein adaptiver Dienst trifft nur die Entscheidung, ob eine als Umgebungseigenschaft zur Verfügung stehende Information als Auslöser seiner Aktionen herangezogen wird. Umgebungseigenschaften können hier sowohl nicht-interpretierte Informationen (zum Beispiel durch Hardware-Sensoren), als auch interpretierte Werte (Schlussfolgerungen durch Kontextmodelle) sein (siehe hierzu Abschnitt 4.2). Oberflächlich ausgedrückt ist die Sicht eines adaptiven Diensts auf eine interpretierte Kontextinformation eine *Empfehlung* zur Durchführung Aktionen.

Ob das Verhalten eines Diensts adaptiv oder nicht adaptiv ist, kann auch als eine Qualitätseigenschaft des Diensts betrachtet werden. Um darüber hinaus die Klasse der adaptiven Dienste hinsichtlich ihrer Adaptivitätseigenschaft vergleichbar zu machen, quantifizieren wir diese Eigenschaft. In der folgenden Definition kommt dabei das Konzept des Kontextmodells zur Anwendung:

Definition 3.3 (Adaptivitätsstufe von Diensten) *Der Einfluss von Umgebungseigenschaften auf das Verhalten eines Diensts wird durch folgende fünf Adaptivitätsstufen charakterisiert:*

Stufe 0: *Das Verhalten des Diensts ist unabhängig von Umgebungseigenschaften. Der Dienst ist* nicht adaptiv.

Stufe 1: *Das Verhalten des Diensts ist abhängig von* konstanten Umgebungseigenschaften. *Dies beinhaltet insbesondere auch eine statische Konfigurierbarkeit des Diensts.*

Stufe 2: *Das Verhalten des Diensts ist abhängig von sich* ändernden Umgebungseigenschaften. *Der Dienst stützt sich dabei auf höchstens* ein Kontextmodell *ab.*

Stufe 3: *Das Verhalten des Diensts stützt sich auf* ein beliebiges Kontextmodell *ab, welches statisch konfiguriert ist.*

Stufe 4: *Das Verhalten des Diensts stützt sich auf ein vom Dienst selbst gewählten beliebigen Kontextmodell ab.*

□

Die Adaptivität eines Diensts vereinfacht sich damit auf die Konfigurierbarkeitseigenschaft. Nach Definition 3.3 reicht dies von der (statischen) Definition von Schwellwerten bis hin zur Festlegung eines geeigneten Kontextmodells.

Definition 3.4 (Adaptivität eines Diensts) *Das Verhalten eines Diensts wird beeinflusst durch mindestens eine Umgebungseigenschaft als implizite Eingabe. Das heißt, die Adaptivitätsstufe des Diensts ist ungleich 0. Mit der Berücksichtigung der Kontextinformationen ändert sich insbesondere der Zustandsraum des Diensts.* □

3.2 Eigenschaften adaptiver Dienste

Adaptive Dienste unterscheiden sich zu herkömmlichen Diensten durch eine Reihe von zusätzlichen Eigenschaften. Abbildung 3.2 gibt einen Überblick über eine Reihe zusätzlicher Eigenschaften adaptiver Dienste. Zu diesen zusätzlichen Eigenschaften sei noch

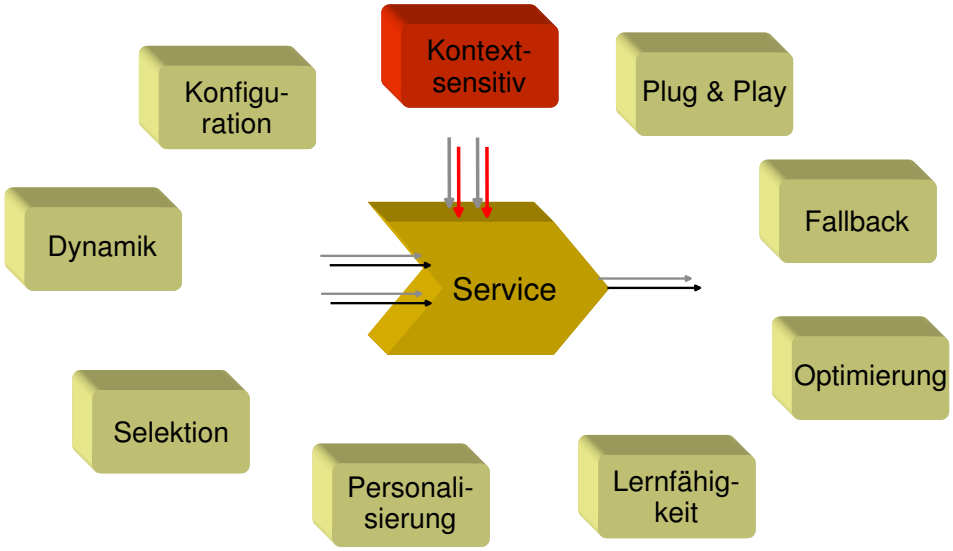


Abbildung 3.2: Adaptivitätseigenschaften

anzumerken, dass diese Liste nicht den Anspruch auf Vollständigkeit erhebt und dass die Eigenschaften nicht völlig disjunkt voneinander sind.

Konfiguration

Die Eigenschaft Konfiguration eines Dienstes ermöglicht es dem Nutzer eines Dienstes spezielle Attribute in einem Dienst zu verändern. Dies ist notwendig, um beispielsweise Benutzerpräferenzen setzen zu können. Hierfür ist eine spezielle Schnittstelle am Dienst erforderlich, die das Setzen von Attributwerten ermöglicht. Eine derartige Schnittstelle wird häufig als *set-Methode* bezeichnet.

Dynamik

Die Eigenschaft der Dynamik eines Dienstes ermöglicht es, Attribute abzufragen, die sich im Laufe der Zeit verändern können. Beispielsweise können die Anzahl der Einträge in einer Datenbank oder ein Attribut, das die Umwelt beschreibt (Regen, Nebel, etc.) abgefragt werden. Dynamische Eigenschaften können nur ausgelesen werden. Der Dienst besitzt hierfür eine Schnittstelle, die das Auslesen von Attributwerten ermöglicht. Die Schnittstelle wird als *get-Methode* bezeichnet.

Selektion

Die Selektion von Diensten ermöglicht das dynamische Suchen und Auffinden von geeigneten Diensten. Als Suchkriterien können Attributbelegungen, Namen der Dienste, das Dienstverhalten oder Dienstbeziehungen verwendet werden. Wird bei der Selektion ein geeigneter Dienst gefunden, so kann dieser anschließend mit den bestehenden Diensten in Beziehung gebracht und verwendet werden.

Personalisierung

Die Eigenschaft der Personalisierung von adaptiven Diensten erlaubt es, dass Dienste bei ihrer Ausführung über die implizite Schnittstelle auf Profile zugreifen können, wie zum Beispiel auf Nutzerprofile oder Fahrzeugprofile. Es können in den Profilen zusätzliche Regeln definiert werden, die das Dienstverhalten beeinflussen.

Lernfähigkeit

Lernfähigkeit ist die Eigenschaft adaptiver Dienste, dass sie sich bestimmte Verhaltensmuster einprägen und bei der wiederholten Ausführung der Dienste ihr Verhalten an diese Verhaltensmuster anpassen. Für die Lernfähigkeit von Diensten müssen Schwellwerte definiert werden, ab wann ein System auf Änderungen reagieren muss, d.h. ab wann ein System das Nutzerverhalten erlernt haben muss. Bei einem zu kleinen Schwellwert nimmt das System auch ein eventuelles Fehlverhalten mit an, wie etwa ein versehentliches Drücken eines daneben liegenden Schaltknopfes. Bei einem zu großen Schwellwert dauert der Lernprozess zu lange. Die Lernfähigkeit ist häufig an die Personalisierung gekoppelt. Ein System merkt sich in diesem Fall von verschiedenen Nutzern die Verhaltensmuster und stellt diese Werte wieder ein, sobald sich ein Nutzer am System anmeldet.

Fallback

Adaptive Dienste haben die Eigenschaft, dynamisch auf den Ausfall oder Wegfall eines Dienstes reagieren zu können. Sie kompensieren die fehlende Verfügbarkeit durch Neu-

konfiguration des Dienstenetzwerks und durch Verändern von Eigenschaften und Attributen. Beispielsweise kann bei reduzierter Bandbreite der Netzanbindung die Videoqualität reduziert werden. Bei Fahrten durch einen längeren Tunnel schaltet zum Beispiel ein Radiodienst auf einen CD-Dienst automatisch um.

Optimierung

Die Optimierung des Dienstverhaltens auf die gegebenen Verhältnisse ist ebenfalls eine besondere Eigenschaft adaptiver Dienste. So erkennt das System von sich aus die Verfügbarkeit von neuen Diensten, höheren Bandbreiten, besseren Netzen, etc. Falls notwendig, schaltet das System automatisch auf die bessere Dienstqualität um und nutzt so selbständig die verbesserten Dienstmöglichkeiten.

Plug & Play

Der Plug & Play Mechanismus adaptiver Dienste ermöglicht es einerseits, dass Dienste dynamisch nachgeladen und gestartet werden. Hierzu werden nach dem Download zur Laufzeit die notwendigen Verbindungen zum Dienst erstellt und der Dienst wird anschließend ausgeführt. Andererseits ermöglicht der Plug & Play Mechanismus auch Änderungen an der Hardware und führt eine Konfiguration der Software durch. So erkennt z. B. ein Dienst nach der Installation eines neuen CD-Wechslers im Fahrzeug diesen, lädt von einer CD oder aus dem Internet den notwendigen Treiber auf die Fahrzeugelektronik und konfiguriert einen CD-Wechslerdienst im Fahrzeug.

Kontextsensitives Verhalten

Alle diese genannten Eigenschaften sowie die zusätzliche Eigenschaft eines adaptiven Dienstes, auf Werte von Sensoren, die im Kontext abgelegt sind, zu reagieren, verbergen sich hinter der Eigenschaft des kontextsensitiven Verhaltens. Insbesondere durch die implizite Schnittstelle der adaptiven Dienste ist es für einen Dienst möglich, auf Umgebungseigenschaften reagieren zu können. Wie im folgenden Kapitel 4 zum Themengebiet Kontext noch gezeigt wird, ermöglich die Kontextsensitivität die Anpassung der Dienste an die Umgebungseigenschaften.

3.3 Bewertung dynamischer und konfigurierbarer Eigenschaften

Dynamische Eigenschaften können nach Abschnitt 3.2 nur ausgelesen, konfigurierbare Eigenschaften nur gesetzt werden. Dies ist aber lediglich eine Frage des jeweiligen Blickwinkels, denn eine sich verändernde Eigenschaft, die nur gelesen werden darf, muss auch von einer Instanz gesetzt werden (sonst kann sie sich nicht verändern). Ebenso muss eine vom Benutzer gesetzte Eigenschaft auch vom System gelesen werden können, um Wirkung zeigen zu können.

So sind diese beiden Eigenschaften lediglich als theoretisches Modell anzusehen. Es ist notwendig, eine Unterscheidung zu treffen, *wer* Eigenschaften bei dynamischen Diensten setzen und bei konfigurierbaren Diensten lesen darf.

Folgende Tabelle gibt eine Lösungsidee:

	lesen	schreiben
dynamische Eigenschaften	Benutzer, System	System
konfigurierbare Eigenschaften	System	Benutzer, System

Tabelle 3.1: Ansatz für dynamische und konfigurierbare Eigenschaften

Ist eine detaillierte Unterscheidung der Ausführung zum Setzen und Lesen notwendig, so müssen die berechtigten Akteure weiter unterteilt werden. Beispielsweise können die Systemdienste in Basisdienste und erweiterte Dienste unterteilt werden (vgl. [Mer04]).

Adaptive Eigenschaften

Unser Eigenschaftsmodell mit Integration von adaptiven Eigenschaften kann als Vereinigung von dynamischen und konfigurierbaren Diensten gesehen werden. Zum einen müssen Schnittstellen zum Lesen von Attributen und Schnittstellen zum Schreiben von Attributen existieren. An obige Bewertung angelehnt, ist es somit notwendig, dass die Eigenschaften sowohl von Benutzern als auch vom System sowohl gelesen als auch geschrieben werden dürfen.

Wenn man den Kontext mit in das adaptive Eigenschaftsmodell mit einbezieht, ergibt sich folgende neue Sichtweise darauf:

Anpassbar: Das Verhalten eines Dienstes wird durch Eigenschaften konfiguriert. Dies kann durch den Benutzer erfolgen, der den Dienst nach seinen persönlichen Präferenzen anpasst. Weiterhin können die Eigenschaften eines Dienstes auch durch andere Dienste angepasst werden, die nicht zum Kontext gehören. Diese Unterscheidung ist dahingehend wichtig, um eine Abgrenzung zur nächsten Sichtweise, der Anpassung durch den Kontext, zu finden.

Angepasst: Das Verhalten des Dienstes wird durch den Kontext bestimmt. Die Veränderung von Eigenschaften erfolgt durch spezielle Kontextdienste, die die Umgebung repräsentieren. Beispiele hierfür sind Dienste wie ein Regenindikator (für den Scheibenwischer) oder ein Indikatordienst, der Nebel oder Glatteis meldet.

4 Kontext

In diesem Kapitel werden die Modellierung des Kontexts adaptiver Dienste sowie der Zusammenhang von Kontext und Dienst dargestellt. Dazu werden in Kapitel 4.1 die Konzepte *Kontext* und *Kontextmodell* eingeführt und die Beziehung untereinander und zu *adaptiven Diensten* untersucht. Kontextmodelle abstrahieren eine auf Umgebungseigenschaften angewandte Entscheidungslogik. Abschnitt 4.2 gibt einen Überblick über verschiedene mathematisch/logische Modelle, welche als Basis für die Entscheidungslogiken dienen. Der Unterschied zwischen System- und Dienstkontexten wird umrissen und die Eignung von Kontextmodellen wird diskutiert.

4.1 Grundlagen

Auf den ersten Blick scheint der Begriff des *Kontexts* eines Dienstes oder eines Systems leicht verständlich zu sein. Intuitiv ist es eine Umgebung und bei genauerer Betrachtung eine Menge von Umgebungseigenschaften, in die der Dienst oder das System eingebettet ist. Allerdings ist es sehr schwierig, eine klare Grenze zwischen Dienst und Kontext respektive System und Kontext zu ziehen. Nach [Fah05] ist eine allgemeingültige Definition dieser Grenze auch nicht möglich.

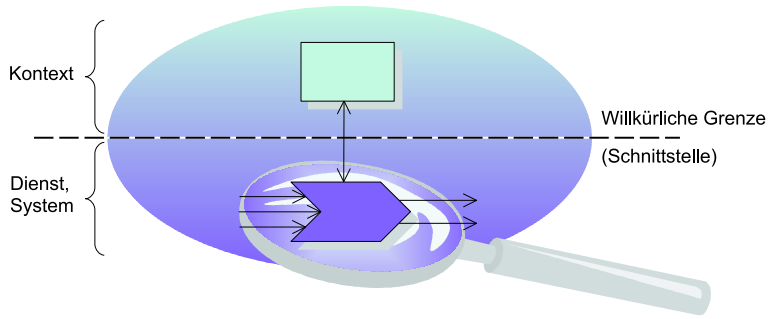


Abbildung 4.1: Kontextgrenze

Die Grenze zwischen System und Kontext ist willkürlich. Sie ist abhängig von der Sichtweise auf den Dienst respektive das System. Je nachdem wo die Grenze gezogen wird, kann der Kontext bereits wieder als Teil des Systems aufgefasst werden. Wie das in Abbildung 4.1 angedeutete Entwurfsmuster angewendet wird, kann nicht allgemein vorgegeben werden, dies ist eine reine Entwurfsentscheidung. Ob die Eingaben eines Dienstes oder

Systems implizit (über Kontextinformationen) oder explizit modelliert werden, ist für das Verhalten nicht entscheidend.

In der Literatur gibt es zahlreiche Versuche, insbesondere in Arbeiten zu *Ubiquitious Computing*, den Begriff „Kontext“ zu definieren. Je nach Anwendungsfall und Abstraktionsgrad bezieht sich dort Kontext zum Beispiel auf Orts- und Zeitinformationen, Personenidentitäten und Objektidentitäten sowie auch auf die Umgebung oder Situation.

Da zunächst der Adaptivitätsbegriff beziehungsweise der Kontextbegriff allgemein und unabhängig von einer bestimmten Domäne und Anwendung verwendet werden soll, wird „Kontext“ in Anlehnung an [DAS01] folgendermaßen definiert:

Definition 4.1 (Kontext allgemein) *Kontext ist jegliche Information, die dazu benutzt wird, den Zustand von Entitäten zu charakterisieren, die relevant für Interaktionen sind.* □

Im Folgenden bedeutet Kontext zunächst nichts anderes als eine Menge von Werten beliebiger Umgebungseigenschaftstypen. Da eine Entität (Dienst, System) Teil einer bestimmten Anwendungsdomäne ist, umfasst der Betrachtungshorizont jedoch nicht das ganze Universum, sondern wird auf die entsprechenden (geeigneten) Domänenkonzepte eingeschränkt.

Beispiel 4.1 Umgebungseigenschaften Automotive Domäne
In der Automotive Domäne sind Beispiele für solche Typen von Umgebungseigenschaften etwa die Innen- und Außentemperatur, Helligkeit, Luftfeuchtigkeit, Luftdruck, Beschleunigung, Fahrzeuggeschwindigkeit oder Batteriezustand. Auch Personalisierungsinformationen, gewünschte Sprache, Lautstärke des Entertainmentsystems, Fahrziel oder auch zur Verfügung stehende Dienste oder Geräte sind mögliche relevante Informationen. □

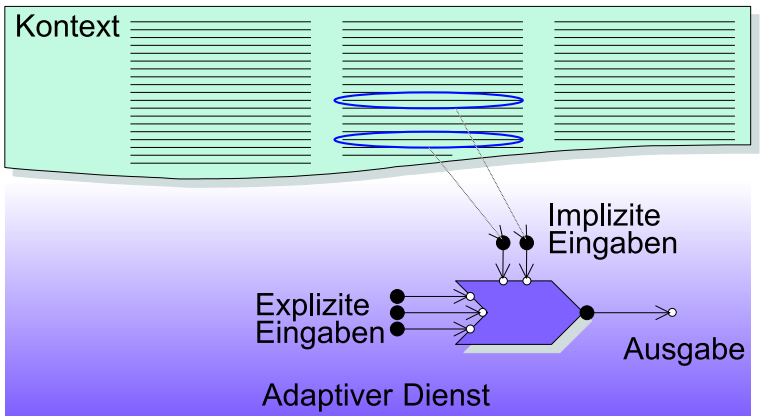


Abbildung 4.2: Adaptive Dienste und Kontext

Im oberen Teil von Abbildung 4.2 ist diese allgemeine Kontextdefinition in Form einer Tabelle von beliebigen Werten skizziert. Die Relevanz einer Kontextinformation für einen adaptiven Dienst ergibt sich erst aus dem Bezug als dessen implizite Eingabequelle. In der Skizze ist dies durch ellipsenförmige Markierungen angedeutet, wobei dieser Bezug zwischen Kontextinformation und adaptivem Dienst mit einem Pfeil dargestellt ist. Das Vorhandensein einer Kontextinformation führt dazu, dass sich ein adaptiver Dienst in einem Zustand befindet, der ihn zu anderem Verhalten veranlasst, als wenn diese Kontextinformation nicht zur Verfügung stehen würde. Wir differenzieren hier zwischen *explizitem* und *implizitem Kontext*. Die expliziten und impliziten Eingaben in einen Dienst, wie in Abbildung 4.2 gezeigt, wurden in Definition 3.1 eingeführt.

Die in Abbildung 4.2 dargestellten Konzepte *Kontext* und *Adaptiver Dienst* werden durch ein drittes ergänzt, und zwar durch das *Kontextmodell*. Damit wird eine Entkopplung von Dienstspezifikation und eingesetzter Entscheidungslogik erreicht. Zusammengefasst sind bei der Modellierung von kontextadaptiven Diensten folgende Konzepte beteiligt:

- Kontext,
- Kontextmodell und,
- adaptiver Dienst.

Der Zusammenhang dieser Konzepte wird in Abbildung 4.3 in einer an UML angelehnten Notation illustriert.

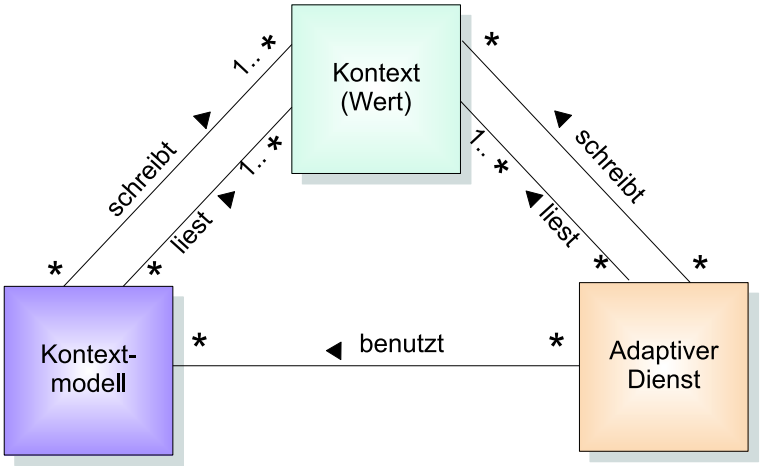


Abbildung 4.3: Konzepte

Nur über den *Kontext* stehen Umgebungseigenschaften zur Verfügung beziehungsweise im Kontext werden entsprechende Werte zur Verfügung gestellt. Kontextwerte werden nicht nur durch Sensoren bereitgestellt, sondern sind auch mithilfe von Entscheidungslogiken abgeleitete Werte. Auch Dienste können als Teil ihrer Ausgabe mit geänderten oder neuen Kontextwerten beitragen.

Das kontextsensitive Verhalten eines *adaptiven Diensts* wird von mindestens einer Umgebungseigenschaft beziehungsweise von mindestens einem Kontextwert beeinflusst. Eine Umgebungseigenschaft kann allerdings von beliebig vielen adaptiven Diensten herangezogen werden.

Ein *Kontextmodell* abstrahiert eine Entscheidungslogik, die auf Basis von mindestens einer Umgebungseigenschaft durch entsprechende Schlussfolgerungen mindestens eine weitere Umgebungseigenschaft im Kontext bereitstellt. Ein adaptiver Dienst hat konzeptionell zunächst über die Wahl von geeigneten (von Kontextmodellen zur Verfügung gestellten) Umgebungseigenschaften eine indirekte Beziehung zu Kontextmodellen.

Mit diesem Modell lässt sich nun in Anlehnung an Definition 4.1 der Kontext eines adaptiven Diensts folgendermaßen definieren:

Definition 4.2 (Kontext eines adaptiven Diensts) *Der Kontext eines adaptiven Diensts ist eine beliebige, über implizite Eingaben zur Verfügung stehende, Information, die den Zustand des Diensts charakterisiert und damit relevant für sein Verhalten (Interaktion) ist. Diese Information ist eine Menge von Umgebungseigenschaften, welche direkt von Sensoren ermittelten Werte und aus Entscheidungslogiken abgeleitete Werte sind.* □

Dienstrollen

Zur Veranschaulichung und Einordnung der vorgestellten Konzepte auch die in [Fah05] aus dem Bereich des „ubiquitious computing“ eingeführten Dienstrollen verwendet.

Definition 4.3 (Dienstrollen) *Das Verhalten von Diensten wird durch folgende vier Rollenkonzepte charakterisiert:*

Sensor:	*Legt Information im* Kontext *ab.*
Kontext:	*Bereitstellung von Daten (Lesen und Schreiben von Umgebungseigenschaften).*
Interpreter:	*Auswertung bestimmter Kontextinformation, die zu weiterer Kontextinformation führt.*
Aktuator:	*Ausführung konkreter Aktionen (stellvertretend für einen Benutzer) aufgrund geänderter Kontextinformation.*

□

4.2 Kontextmodelle

Der Zusammenhang von Kontext und adaptiven Diensten wird über *Kontextmodelle* hergestellt. In Abbildung 4.4 ist dies im linken Teil illustriert. Ein Kontextmodell ist eine

prinzipiell beliebige Entscheidungslogik (z. B. von Aussagenlogik bis hin zu einem komplexen Expertensystem), die auf Basis einer Menge von Umgebungseigenschaften (Ellipsen mit ausgehendem Pfeil) entsprechende Schlüsse zieht. Kontextmodelle entkoppeln die Entscheidungslogik von der Dienstspezifikation, wobei das Verhalten der Dienste sehr wohl von der gewählten Logik respektive den geschlussfolgerten Werten abhängig ist.

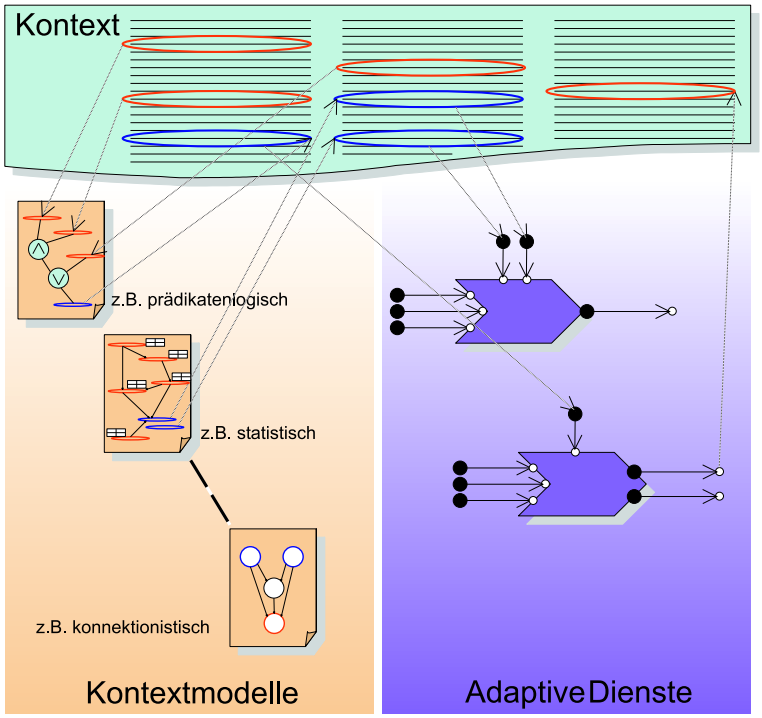


Abbildung 4.4: Zusammenhang Kontext und adaptive Dienste

Das Kontextmodell ist nach Definition 4.3 das einzige Konzept, welches die Rolle des *Interpreter* einnimmt. Da die gefolgerten Schlüsse wiederum im Kontext durch ein oder mehrere Umgebungseigenschaften zur Verfügung gestellt werden, ist ein Kontextmodell auch *Sensor* zugleich (Ellipsen mit eingehendem Pfeil).

Eine Umgebungseigenschaft kann von mehreren Entscheidungslogiken herangezogen werden. In der Regel sind also die Mengen der Umgebungseigenschaften für verschiedene Entscheidungslogiken nicht disjunkt. Dies soll auch nicht verhindert werden. Im Gegenteil, man erreicht dadurch ein noch größeres Maß an Flexibilität. Derselbe Satz an Umgebungseigenschaften kann durch unterschiedliche Entscheidungslogiken (zum Beispiel durch logische, statistische oder gar konnektionistische Modelle) interpretiert werden.

Über die Wahl einer für den adaptiven Dienst interessierenden Umgebungseigenschaft wendet dieser indirekt ein Kontextmodell an. Sein Verhalten ist abhängig von diesen als implizite Eingaben modellierten, interpretierten Umgebungseigenschaften. Umgebungseigenschaften, die nicht interpretiert werden müssen, entsprechen dabei dem Kontextmodell

Identität, d.h. diese können direkt verwendet werden und müssen vorab nicht innerhalb eines Kontextmodells berechnet werden.

Umgebungseigenschaften können durch unterschiedliche Beziehungstypen (kausal, numerisch, stochastisch, etc.) zusammenhängen. Kontextmodelle bilden diese Zusammenhänge ab. Aus diesen Umgebungseigenschaften ergeben sich insbesondere auch Anforderungen an einen Dienst oder ein System, deren Erfüllung mithilfe von Kontextmodellen verifiziert werden kann.

Kontextmodelle abstrahieren mithilfe von Entscheidungslogiken eine Menge von Umgebungseigenschaften. Im Folgenden wird ein Überblick über verschiedene zur Kontextmodellierung geeignete Logiken gegeben.

Diensterbringer als Umgebungseigenschaft

Ein adaptiver Dienst passt sich in der Regel mit seinem Verhalten speziellen, sich dynamisch ändernden Umgebungseigenschaften an. Beispiele für solche Typen von Umgebungseigenschaften sind, wie bereits in Beispiel 4.1 erwähnt, die Fahrzeuggeschwindigkeit oder etwa zur Verfügung stehende Dienste oder Geräte. Die für einen adaptiven Dienst relevanten Umgebungseigenschaften sind im Allgemeinen nicht unabhängig voneinander und beeinflussen sich sogar gegenseitig. Wenn sich ein Dienst auf die Funktionalitäten anderer Dienste abstützt, kann auch die Verfügbarkeit der so genannten *Diensterbringer*, als Umgebungseigenschaft aufgefasst werden.

Vorteile der Entkopplung des Kontextmodells

Ein Kontextmodell ist die Abstraktion einer Entscheidungslogik für einen adaptiven Dienst. Die Spezifikation eines Kontextmodells ist für sich genommen in der Regel bereits hinreichend komplex. Um nun das Verhalten eines adaptiven Diensts oder Systems zu beschreiben ist es nicht notwendig, die Entscheidungslogik in die Dienstspezifikation zu integrieren. Die oben vorgeschlagene Entkopplung ist deshalb eine erhebliche Vereinfachung. Ein adaptiver Dienst ist nur noch insofern bei der Spezifikation des Kontexts beteiligt, als es die Auswahl eines passenden Kontextmodells betrifft. Die Schlussfolgerungen sind so nicht mehr Teil der Dienstspezifikation und können sogar gegebenenfalls leicht modifiziert werden. Der adaptive Dienst benutzt geschlussfolgerte Kontextinformationen gleichsam als Empfehlung.

Über die durch die Entkopplung erreichte Vereinfachung der Spezifikationen hinaus gibt es noch weitere Gründe, weshalb es sich lohnt, Kontextmodelle zu entwerfen und sie insbesondere losgelöst von potentiell nutzenden adaptiven Diensten anzubieten:

- Das System soll den Regelsatz selbst herausfinden.
- Regeln ändern sich häufig.
- Die Regeln können nicht oder nur sehr schwierig exakt formuliert werden.
- Das Regel-Subjekt ist nicht konstant (zum Beispiel: Fahrer, Beifahrer, etc.).

4.2.1 Überblick über Entscheidungslogiken

Im Folgenden sind einige Entscheidungslogiken zur Kontextmodellierung übersichtsartig zusammengefasst. Neben *Aussagenlogik* und *Prädikatenlogik 1. Stufe* können beispielsweise Entscheidungsbäume, Bayes-Netze, Fuzzy Logik, künstliche neuronale Netze oder fallbasiertes Schließen zur Kontextmodellierung verwendet werden.

Entscheidungsbäume sind nach [CS93] eine grafische Methode zur Darstellung und Bearbeitung von Entscheidungsproblemen. Ein Entscheidungsbaum zu einem Entscheidungsproblem ist ein geordneter, gerichteter Baum. Jeder Knoten des Baumes mit Ausnahme der Blätter enthält seinerseits ein Entscheidungsproblem, das Teil des Gesamtproblems ist. Diese Knoten nennt man *Entscheidungsknoten*. Die Kanten zwischen einem Entscheidungsknoten und seinen Söhnen repräsentieren mögliche Entscheidungsalternativen für das zugehörige Entscheidungsproblem. Die Blätter enthalten die Ergebnisse der jeweiligen Entscheidungsfolgen. Dabei kann es sich um Situationen handeln, die man nach der entsprechenden Entscheidungsfolge erreicht, oder um Aktionen, die man befolgen muss. Die Auswertung eines Entscheidungsbaumes beginnt bei der Wurzel. Solange man noch kein Blatt erreicht hat, löst man das Entscheidungsproblem des gerade betrachteten Knotens durch Wahl einer der möglichen Alternativen und verzweigt zum entsprechenden Sohn. Erreicht man ein Blatt, so befolgt man gegebenenfalls die dort vorgeschriebene Aktion.

Bayes-Netze repräsentieren ein Netz von Zufallsvariablen und kausal bedingten Wahrscheinlichkeiten. Mit ihnen ist es möglich, Zusammenhänge über Wahrscheinlichkeiten, bedingte Wahrscheinlichkeiten und Unsicherheiten über den Wahrheitswerten von Aussagen zu repräsentieren. Bayes-Netze eignen sich insbesondere für diagnostische Aufgaben, insbesondere für das Schließen von Symptomen auf Ursachen.

Während Aussagen- und Prädikatenlogik nur auf bivalenten Werten operieren, steht bei der *Fuzzy Logik* die Multivalenz (Vielwertigkeit) im Vordergrund. Im Gegensatz zur Zweiwertigkeit mit nur zwei Extremen gibt es in der Fuzzy Logik drei, vier oder sogar ein unendliches Spektrum an Möglichkeiten. Mit Hilfe dieser Logik können Vagheit und unscharfe Prädikate ausgedrückt werden.

Ein *Neuronales Netz* ist die Verschaltung von gleichartigen, relativ einfachen Bausteinen zu einem Netzwerk und Informationsweiterleitung in diesem Netz nach Prinzipien, wie sie bei biologischen Nervenzellen (Neuronen) vorliegen [CS93]. Die Bausteine neuronaler Netze sind modifizierte Schwellenwertelemente. Die Weiterleitung von Werten geschieht längs der Verbindungen im Netz, wobei Werte in unterschiedlicher Weise gewichtet werden und sowohl aktivierend als auch hemmend auf andere Bausteine wirken können. Neuronale Netze lernen die Informationen durch Training und ermöglichen vage Schlussfolgerungen (an Stelle von exakten). Sie sind tolerant gegenüber Eingaberauschen und werden insbesondere dann zur Mustererkennung eingesetzt, wenn kein klarer Satz von Regeln angegeben werden kann.

Die hier knapp vorgestellten Entscheidungslogiken für Kontextmodelle sind in weiteren Arbeiten bezüglich ihrer Eignung zur Kontextmodellierung zu untersuchen.

4.2.2 Unterscheidung zwischen Systemkontext und Dienstkontext

Im Folgenden unterscheiden wir zwischen dem Kontext eines dynamischen Systems und dem Kontext eines adaptiven Dienstes. Da ein Dienst Teil eines Systems ist, ist auch dessen Kontext in den Systemkontext eingebettet. Jedoch sind in der Regel nicht alle Umgebungseigenschaften eines Systems auch relevant für einen adaptiven Dienst.

Unter *Systemkontext* versteht man die Verfügbarkeit von Diensten, im Sinne von *Existenz* von Diensten. Der Systemkontext kann als die *Menge der Diensterbringer*, welche die *Menge der Dienstnutzer* benötigt, betrachtet werden. Der Systemkontext ist Voraussetzung für die Anpassbarkeit von adaptiven Systemen (siehe hierzu Abschnitt 3.3).

Unter *Dienstkontext* verstehen wir Umgebungseigenschaften aus Domäne, relvante Eigenschaften, Abhängigkeiten und Schlussfolgerungen.

4.2.3 Eignung von Kontextmodellen

Kontextmodelle können als „zur Laufzeit genutztes Regelsystem“ implementiert werden. Selbstverständlich ist es auch möglich, Kontextmodelle als beliebig komplexe Systeme umzusetzen, beispielsweise auch als Expertensystem. Bei der Wahl eines Kontextmodells beziehungsweise dessen Umsetzung ist Folgendes zu beachten:

Wartung und Pflege. Die Wissensbasis eines Expertensystems ist unabhängig von anfragenden Entitäten. Die Pflege und Wartung der Wissensbasis muss vorab geklärt werden.

Nachvollziehbarkeit. Durch die Entkopplung von Kontextmodell und adaptiven Diensten ist die Nachvollziehbarkeit des Dienstverhaltens aufgrund bestimmter Umgebungseigenschaften gegeben. Die durch das Kontextmodell zur Verfügung gestellten Schlussfolgerungen können jedoch nicht mehr eindeutig nachvollziehbar sein. Ein Expertensystem ist in der Regel eine *Black Box*, dessen innere Struktur unbekannt ist. Aber auch konnektionistische Modelle, wie beispielsweise künstliche neuronale Netze, sind in der Regel ungeeignet, wenn die Nachvollziehbarkeit der Entscheidungslogik im Vordergrund steht.

5 Fallstudie

Die Fallstudie gibt ein Beispiel für die Entwicklung einer adaptiven Nutzerschnittstelle für den Bereich Automotive. Um eine Adaption der Nutzerschnittstelle durchführen zu können, benötigt man Informationen über den Kontext der Bedienung. Dazu modelliert man den Fahrer und die Fahrsituation. Diese Daten werden während der Fahrt von einem Interpreter ausgewertet. Je nach erkannter Situation kann dann die Nutzerschnittstelle angepasst werden.

Als konkretes Beispiel dieser Nutzerschnittstelle ist ein Tankassistent ausgewählt worden. Dieser soll den Fahrer auf die Möglichkeit bzw. die Notwendigkeit zu tanken aufmerksam machen und sich dabei bezüglich den Gewohnheiten des Fahrers adaptiv verhalten. Für die Auswertung der Adaptionslogik kommt dabei eine Regelmaschine (engl. *Rule Engine*) zum Einsatz.

5.1 Analyse des Fallbeispiels

Im Folgenden beschreiben wir kurz das Szenario, gehen auf die Problemstellung ein und stellen unseren Lösungsansatz vor.

5.1.1 Beschreibung des Szenarios

In diesem Szenario wird gezeigt, wie ein Tankassistent den Fahrer während der Fahrt unterstützen kann.

Herr Müller hat seit längerem ein Auto und musste schon öfters damit zum Tanken fahren. Der Tankassistent kennt dadurch schon einige Gewohnheiten des Fahrers. Herr Müller fährt nun los und wird nach einiger Zeit gefragt, ob er tanken will, da Herr Müller bisher gewöhnlich bei diesem aktuellen Tankstand zu einer Tankstelle gefahren ist. Da Herr Müller ungern erst zum tanken fährt, wenn bereits das „Lämpchen“ aufleuchtet, nimmt Herr Müller den Hinweis war und fährt zum Tanken.

Herr Müller hat von seiner Firma eine Tankkarte erhalten, mit der er bei Tankstellen des Konzerns Z günstiger tanken kann. Er findet es aber unangenehm bei jeder Tankstelle Z, die gerade in Sichtweite ist, auf den Tankstand zu schauen, um zu überprüfen, ob sich das Tanken schon lohnen würde. Andererseits ärgert es ihn eine andere Tankstelle anfahren zu müssen, obwohl er kurz vorher an einer bevorzugten Tankstelle vorbeigefahren ist, bzw. kurz darauf an einer vorbeifahren wird, zu der der Treibstoff noch gereicht hätte.

Der Tankassistent merkt durch die Häufigkeit der Tankvorgänge bei einer bestimmten Tankstelle, welcher Konzern bevorzugt wird und macht Herrn Müller, wenn eine dieser Tankstellen in der Nähe ist und es sich lohnen würde zu tanken, rechtzeitig darauf aufmerksam.

5.1.2 Problemstellung

Ein Tankassistent kann den Fahrer beim Tanken unterstützen, indem er ihn über Tankmöglichkeiten benachrichtigt. Dabei hängt das Verhalten der Dialogsteuerungen von verschiedenen Kontextwerten ab. Aus diesen Werten lassen sich Antworten auf die folgenden Fragen ableiten:

- Bei welchem Tankstand fährt der Fahrer zur Tankstelle?
- Gibt es eine Tankstelle die der Fahrer favorisiert?
- Hat der Fahrer einen Termin den er Erreichen muss? Dann will der Fahrer, wenn es nicht unbedingt sein muss, vorab nicht tanken.

Ebenfalls muss beachtet werden, dass das Auto von unterschiedlichen Personen gefahren wird. Das jeweilige Profil muss also personenbezogen gespeichert und geladen werden. Zusätzlich will z. B. nicht jeder Fahrer auf eine bevorzugte Tankstelle aufmerksam gemacht werden, d. h., der Tankassistent sollte bezüglich der Adaptivität konfigurierbar sein.

Da für die Verarbeitung der Adaptionslogik eine Rule Engine verwendet wird, sind geeignete Regeln zu finden, welche die Adaptivität des Tankassistenten erst ermöglichen und auf die unterschiedlichen Fakten geeignet reagieren.

5.1.3 Lösungsansatz

Um sich adaptiv verhalten zu können, muss der Tankassistent Informationen über den Fahrer sammeln. Dies geschieht jedes Mal, wenn der Fahrer zum Tanken fährt. Gesammelt werden Informationen wie der Tankstand, bei dem der Fahrer zum tanken fährt und bei welcher Tankstelle getankt wurde. Anhand dieser Informationen wird entschieden, welche Tankstelle der Fahrer bevorzugt, weil beispielsweise die Firma einen Vertrag mit der Tankstelle hat, und bei welchem durchschnittlichen Tankstand der Fahrer zum Tanken fährt. Diese Werte werden mit der aktuellen Situation verglichen, beispielsweise mit dem aktuellen Tankstand und mit welcher Tankstelle, die gerade in der Nähe ist. Je nach Situation wird nun entschieden, ob der Fahrer benachrichtigt werden soll. Er soll benachrichtigt werden, wenn

- der Tankstand einen kritischen Tiefpunkt erreicht hat (z. B. 10 % des Tankvolumens),
- der Tankstand sehr nah an dem durchschnittlichen Tankvolumen der Tankfahrten des Fahrers ist,
- eine vom Fahrer bevorzugte Tankstelle in der Nähe und der Tankstand nahe dem Durchschnittstankstand ist oder
- der Fahrer tanken muss, um zu einem Termin zu kommen.

5.2 Kontext und Kontextmodellierung

Um Informationen über die Umgebung zu verwerten, müssen diese zuerst aufbereitet und modelliert werden. Bei der Kontextmodellierung ist darauf zu achten, dass der Kontext nicht nur von der Umgebung um den Benutzer herum analysiert wird. Der Benutzerkontext, also das Verhalten und die Eigenschaften des Benutzers müssen ebenfalls beachtet und modelliert werden, da der Benutzer den Gesamtkontext mitgestaltet (siehe [Jam01]).

Um die Kontextinformationen besser zuordnen zu können, wurden in Abbildung 5.1 die verschiedenen Komponenten des Fallbeispiels und deren Informationsaustausch dargestellt.

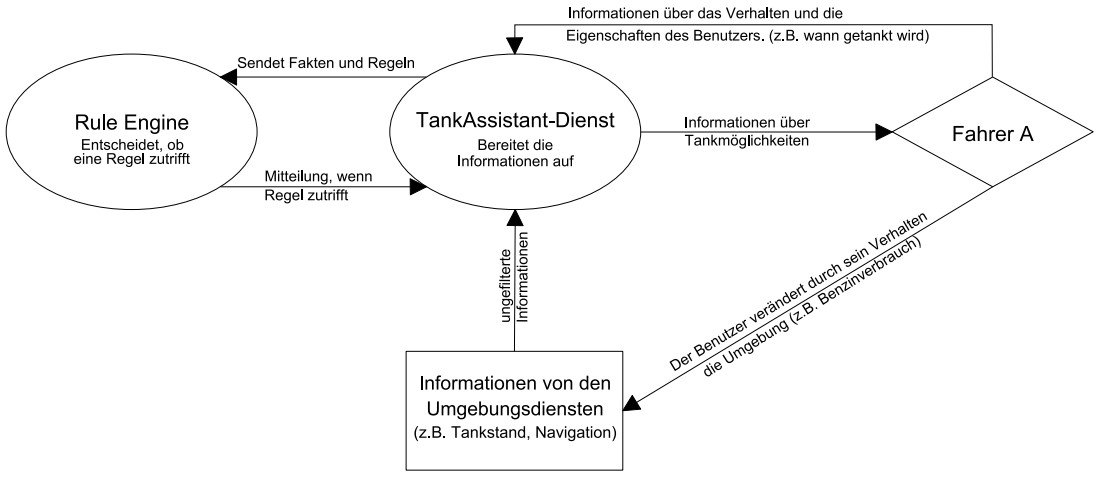


Abbildung 5.1: Informationsfluss

Mit *Informationen von den Umgebungsdiensten* sind die Informationen gemeint, welche von Sensoren aufgenommen und über bestimmte Dienste anderen Diensten bereitgestellt werden, wie zum Beispiel der aktuelle Tankstand. Der Fahrer selbst stellt indirekt über sein Verhalten und seine Eigenschaften ebenfalls Informationen bereit. Beispielsweise dann, wenn der Fahrer einen bestimmten Kraftstoffkonzern bevorzugt und dessen Tankstellen öfters angefahren werden als andere, muss dies vom Dienst des Tankassistenten registriert werden. Zusätzlich dazu wirkt der Fahrer auch auf den ihn umgebenden Kontext ein. Wenn der Fahrer zum Beispiel tendenziell schnell fährt, ist der Benzinverbrauch höher und folglich der Tank schneller leer.

Diese Informationen aus der Umgebung und über den Fahrer *sammelt* der Dienst des Tankassistenten, verpackt diese als Fakten und übergibt sie der Rule Engine. Die Rule Engine wird permanent mit Kontextinformationen beliefert. Diese enthält einen vom Dienst des Tankassistenten übergebenen Satz von Regeln, anhand derer sie entscheidet wann eine dieser Regeln zutrifft. Beispielsweise eine Regel, die bei einem Tankstand von 10 % aktiviert wird (*feuert*). Trifft eine Regel zu, so bekommt der Dienst des Tankassistenten darüber Bescheid, welcher wiederum dem Fahrer in geeigneter Form diese Information mitteilen kann.

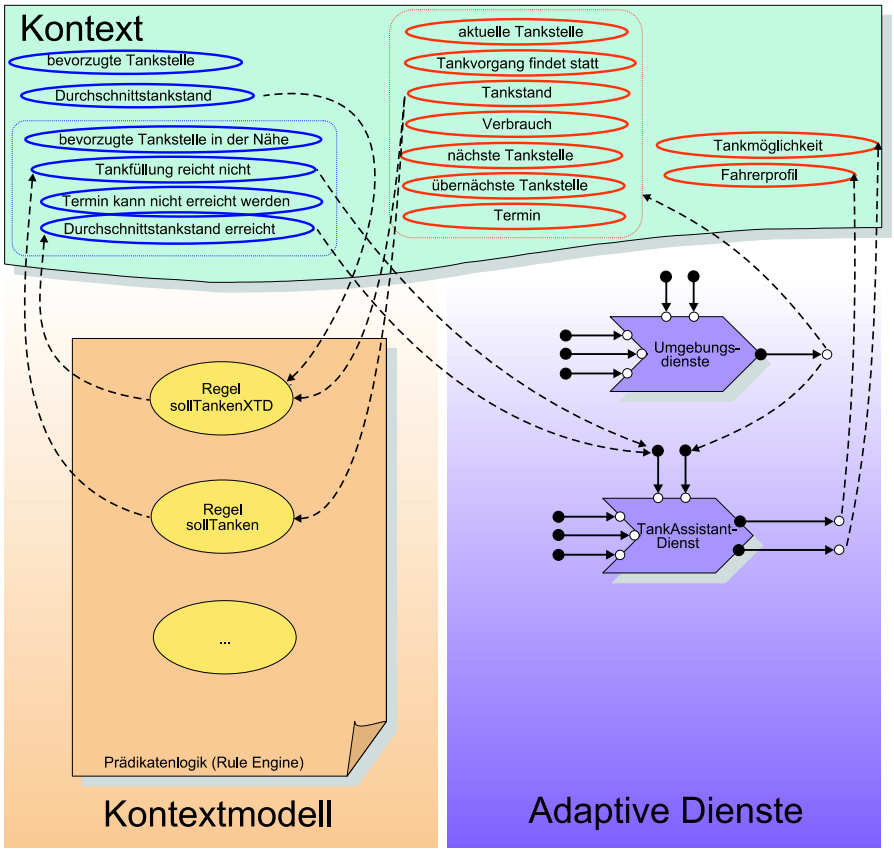


Abbildung 5.2: Ausschnitt aus dem Kontextmodell

In Abbildung 5.2 ist nun ein Ausschnitt des Kontextmodells dargestellt. Die schon erwähnten Umgebungsdienste stellen Informationen bereit wie z. B. den aktuellen Tankstand, den Verbrauch, welche die nächste und übernächste Tankstelle ist. Die in der Rule Engine enthaltenen Regeln benutzen den Kontext und schließen daraus neue Informationen, welche wiederum als Kontext zur Verfügung stehen.

5.2.1 Verwendete Regeln

Hier wird ein Ausschnitt der wichtigsten Regeln aus dem verwendeten Regelsatz gegeben. Als Rule Engine wurde in dem Fallbeispiel *Jess* [Jes] verwendet. Die in den folgenden Regeln verwendete Syntax ist diesbezüglich ebenfalls in Jess-Notation.

Die Regel „sollTanken“

```
(defrule sollTanken
  (aktuell (tanklevel ?x) (verbrauch ?y) (tankstelle ?z))
  (test (< ?x 10))
  =>  (printout t "Bitte␣tanken"))
```

Bei jedem neuen Fakt `aktuell` mit beliebigem Tanklevel, Verbrauch und nächster Tankstelle, wird überprüft, ob der aktuelle Tankstand kleiner als 10 % ist

```
((test (< ?x 10))).
```

Trifft dies zu, *feuert* die Regel und im Kontext ist die Information *Tankfüllung reicht nicht* verfügbar (siehe Abbildung 5.2). In Prädikatenlogik werden die Regeln folgendermaßen formuliert:

$$\forall x \forall y \forall z : (aktuell(x, y, z) \wedge tanklevel(x) \wedge verbrauch(y) \wedge tankstelle(z) \wedge (x < 10)) \rightarrow (printout)$$

Die Regel „sollTankenXTD“

Die Regel `sollTankenXTD` berücksichtigt nun den Tankstand, bei dem der Fahrer durchschnittlich am häufigsten zum Tanken fährt:

```
(defrule sollTankenXTD
  (aktuell (tanklevel ?x) (verbrauch ?y) (tankstelle ?z))
  (test (< ?x (+ (fetch tankDurchschnitt) 5)))
  => (printout t "Bitte␣tanken" ))
```

Bei jedem neuen Fakt `aktuell` wird überprüft, ob der aktuelle Tankstand kleiner dem Durchschnittstankstand plus 5 % ist

```
(test (< ?x (+ (fetch tankDurchschnitt) 5))).
```

Dadurch feuert die Regel, wenn der Tank nahe an dem durchschnittlichen Tankvolumens der Tankfahrten des Fahrers ist.
Prädikatenlogisch ausgedrückt:

$$\forall x \forall y \forall z : (aktuell(x, y, z) \wedge tanklevel(x) \wedge verbrauch(y) \wedge tankstelle(z) \wedge (x < (tankDurchschnitt + 5))) \rightarrow (printout)$$

Der Kontext des durchschnittlichen Tankvolumens wird von der folgenden Regel bereitgestellt.

Die Regel „werteErmitteln“

Diese Regel feuert immer dann, wenn getankt wird (ein Fakt `getankt` existiert) und berechnet daraufhin erneut den Durchschnitt des Tankstandes bei Tankvorgängen und die bevorzugte Tankstelle.

```
(defrule werteErmitteln
  (getankt (tanklevel ?x) (verbrauch ?y) (tankstelle ?z))
  => (store anzahlGetankt (+(fetch anzahlGetankt) 1))
  (store summeVerbrauch (+(fetch summeVerbrauch) ?x))
  (store tankDurchschnitt (/(fetch summeVerbrauch)(fetch anzahlGetankt)))
  (store ?z (+(fetch ?z) 1))
  (test (tsberechnen)))
```

Wenn diese Regel feuert, erhöht sich die Variable in der die Anzahl der insgesamten Tankvorgänge gespeichert sind um eins

```
(store anzahlGetankt (+ (fetch anzahlGetankt) 1)),
```

addiert den aktuellen Tankstand zu der Summe aller Tankstände

```
(store summeVerbrauch (+ (fetch  summeVerbrauch) ?x)), ´
```

berechnet den Durchschnitt daraus neu

```
(store tankDurchschnitt (/ (fetch summeVerbrauch)(fetch anzahlGetankt))
```

und erhöht die Variable der jeweiligen Tankstelle um eins

```
(store ?z (+ (fetch ?z) 1)).
```

Zusätzlich wird die Funktion `tsberechnen` aufgerufen

```
(test (tsberechnen)),
```

welche die am meisten besuchte Tankstelle ermittelt und diese als Fakt `bevorzugteTS` der Rule Engine wieder zur Verfügung stellt. Eine weitere Regel, welche hier nicht mehr vorgestellt wird, benutzt diesen Fakt und feuert, wenn der Tankstand entsprechend niedrig und die bevorzugte Tankstelle in der Nähe ist.

5.3 Wahl des Kontextmodells

Die Verwendung einer Rule Engine zur Realisierung eines prädikatenlogischen Kontextmodells hat einige Vorteile. Im Vergleich zu einem statischem Kontextmodell, welches z. B. durch *if-Abfragen* realisiert ist, ist das Anwenden von Fakten auf Regeln mit Hilfe einer Rule Engine wesentlich effizienter. Zudem wird bei schon wenigen *if-Abfragen* der Code sehr unübersichtlich, wohingegen Regeln und Fakten klar strukturiert sind und nicht geschachtelt werden müssen.

Zudem kann durch die Verwendung einer Rule Engine die Logik aus dem Quellcode ausgelagert werden. Das hat unter anderem den Vorteil, dass die Regeln dynamisch zur Laufzeit hinzugefügt, entfernt oder abgeändert werden können. Beispielsweise ist es möglich, eine Regel zu definieren, die je nach Fahrverhalten des Fahrers sich selbst oder eine andere Regel abändert. Wenn beispielsweise der Fahrer einen hohen durchschnittlichen Benzinverbrauch hat, könnte die Regel so angepasst werden, dass sie früher auf einen niedrigen Tankstand reagiert als bei einem sparsam fahrenden Fahrer. Zusätzlich hat man durch die Rule Engine die Möglichkeit das Fahrerprofil zu transportieren. Ein Geschäftsmann, der öfters Mietwagen benötigt, könnte so sein eigenes Profil, etwa in einem USB-Stick, von Auto zu Auto mitnehmen und ihm steht, obwohl er noch nie in dem Automobil saß, sofort eine personalisierte Mensch-Maschine-Interaktion zur Verfügung.

Der Einsatz von künstlichen neuronalen Netzen als Kontextmodell ist gerade im Bereich Automotive als eher kritisch zu betrachten. Ein künstliches neuronales Netz benötigt

sehr viele Schritte, um ein bestimmtes Verhalten zu erlernen. Bei einem Tankassistenten wäre das nicht möglich, da selbst ein Vielfahrer nicht öfter als zweimal pro Woche tanken wird. Zudem ist das Verhalten des künstlichen neuronalen Netzes nicht vorhersagbar und bei unerwünschten Resultaten hat man kaum Möglichkeiten das neuronale Netz zu verändern, da sich eine kleine Änderung auf das ganze System auswirken kann. Somit hat man kaum Möglichkeiten, zusätzlich erwünschtes Verhalten hinzuzufügen, zu verändern oder zu entfernen. Das bedeutet, dass man im Grunde das komplette Netz neu aufbauen müsste.

Prädikatenlogische Kontextmodelle haben einige Vorteile, das Hauptproblem dabei ist aber, geeignete Regeln zu finden, die aus dem gegebenen Kontext die richtigen Schlussfolgerungen ziehen.

5.4 Adaptive Eigenschaften des Tankassistenten

Um auf die Adaptivitätseigenschaften eingehen zu können, wird zuvor kurz die Dienstearchitektur anhand Abbildung 5.3 vorgestellt.

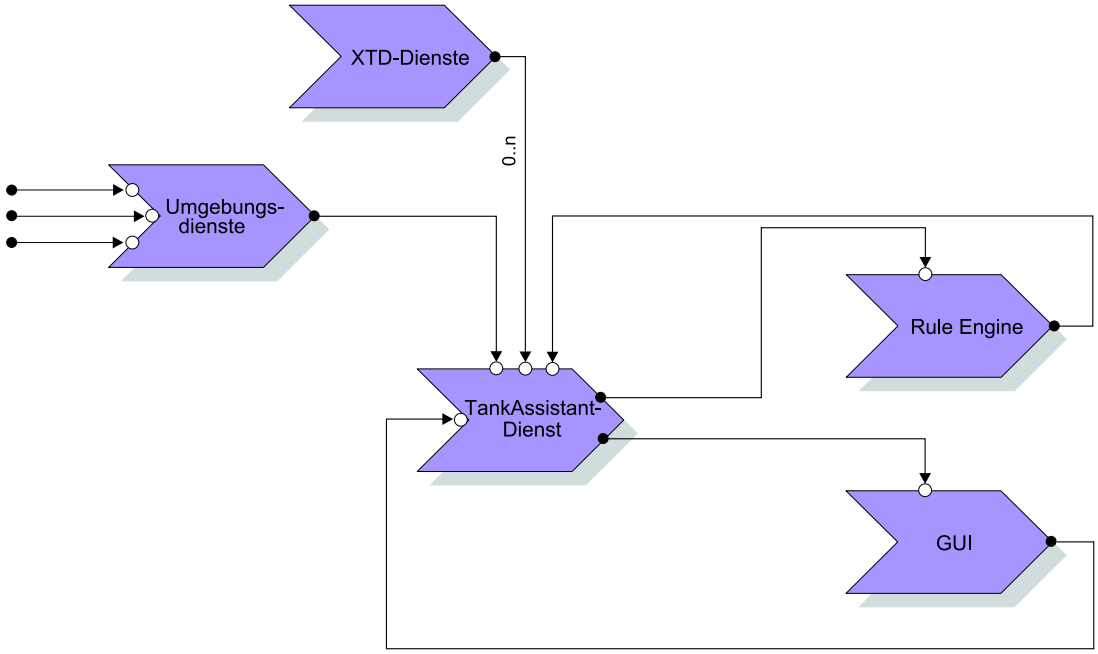


Abbildung 5.3: Ausschnitt aus der Dienstearchitektur

Der Dienst des Tankassistenten verbindet sich mit dem GUI-Dienst, um mit dem Fahrer kommunizieren zu können. Beispielsweise werden darüber die Meldungen über Tankmöglichkeiten angezeigt. Wie schon erwähnt wurde, benutzt der Dienst des Tankassistenten diverse Umgebungsdienste, um an Informationen wie beispielsweise den Tankstand oder den aktuellen Verbrauch zu gelangen. Diese Informationen werden als Fakten verpackt

und der Rule Engine übergeben. Bevor dies geschieht übergibt der Dienst des Tankassistenten diverse Regeln und Funktionen an die Rule Engine. Die einzige für den Fahrer relevante Regel, die der Dienst des Tankassistenten übergibt ist die in Abschnitt 5.2.1 vorgestellte Regel `sollTanken`. Diese Regel ist nicht adaptiv, da sie unabhängig vom Fahrer bei einem Tankstand von unter 10 % feuert. Der Tankassistent handelt also in diesem Zustand nicht adaptiv.

Nun kommen die XTD-Dienste ins Spiel. Diese können zur Laufzeit eingebunden und wieder entfernt werden. Sie beinhalten zusätzliche Regeln, wie die in Abschnitt 5.2.1 vorgestellte adaptive Regel `sollTankenXTD`. Wird einer dieser Dienste zur Laufzeit gestartet, wird das dem Fahrer mitgeteilt. Dieser hat nun die Möglichkeit den Dienst zu nutzen. Entscheidet sich der Fahrer dazu, werden die Regeln der Rule Engine zur Laufzeit übergeben. Wird nun beispielsweise der Durchschnittstankstand erreicht, feuert die adaptive Regel `sollTankenXTD` und der Dienst der Rule Engine teilt dies dem Dienst des Tankassistenten mit. Dieser wiederum fragt daraufhin den Fahrer, ob er tanken will.

Gemäß Abschnitt 3.2 hat der Tankassistent, als ein adaptiver Dienst, folgende Eigenschaften:

Konfiguration

Der Fahrer kann über die Benutzerschnittstelle (GUI) einstellen, ob adaptive Zusatzdienste (XTD) gestartet oder gestoppt werden sollen.

Dynamik

Ein Attribut, welches sich im Laufe der Zeit verändern kann, ist z. B. der bevorzugte Treibstoffkonzern des Fahrers. Diese Information ist in der Rule Engine vorhanden und wird gegebenenfalls aktualisiert.

Selektion

Dem Dienst des Tankassistenten ist unabhängig davon, wie z. B. der Umgebungsdienst Tankinfo implementiert ist, solange der Dienst die gleichen Schnittstellen wie der Dienst des Tankassistenten implementiert (z. B. `public int getTankstand()`). Dem zufolge kann der Dienst beliebig gewechselt werden, bei geeigneter Implementierung sogar dynamisch.

Personalisierung

Das Profil des Fahrers wird in einer Datei gespeichert und ständig aktualisiert. Explizit geschieht das beim Tankassistenten in Form von Fakten über die aktuelle Situation (Tankstand, Tankstelle), bei der der Fahrer zum Tanken fährt.

Lernfähigkeit

Durch das Profil, welches ständig erweitert und aktualisiert wird, ändert sich das Verhalten des Tankassistenten. Aus der Gesamtheit der Fakten über die Tankvorgänge wird

der Tankstand errechnet, bei dem der Fahrer durchschnittlich zum Tanken fährt. Anhand dieser Information fragt ihn der Tankassistent bei ähnlich niedrigem Tankstand, ob er tanken will. Ebenso erlernt der Tankassistent den bevorzugten Treibstoffkonzern des Fahrers. Je nach Häufigkeit der Tankvorgänge bei einer Tankstelle wird der Zähler erhöht und die am meisten besuchte Tankstelle ermittelt.

Fallback

Beim Wegfallen der adaptiven Zusatzdienste (XTD) fällt der Tankassistent in seinen ursprünglichen Zustand zurück und alarmiert den Fahrer nur dann, wenn der Tankstand auf unter 10 % fällt.

Optimierung

Bei Hinzufügen der adaptiven Zusatzdienste (XTD) kann der Fahrer auswählen, ob diese Dienste gestartet werden sollen. Durch starten der Dienste wird die Funktionalität des Tankassistenten erheblich erweitert und somit in Bezug auf den Fahrer optimiert.

Plug & Play

Die adaptiven Zusatzdienste können wie schon erwähnt zur Laufzeit eingebunden und wieder entfernt werden.

Kontextsensitiver Verhalten

Der Tankassistent benötigt und verarbeitet Umgebungsinformationen wie z. B. den Tankstand oder die nächste Tankstelle. Fällt z. B. der Tankstand unter einen bestimmten Schwellenwert, wird der Fahrer darauf aufmerksam gemacht. Der Tankassistent reagiert also auf Kontextveränderungen.

6 Zusammenfassung und Ausblick

In diesem technischen Bericht wurden die Aspekte der Adaptivität von Diensten vorgestellt. Das Konzept der Dienste eignet sich sehr gut zur Entwicklung verteilter und multifunktionaler Systeme. Adaptive Dienste sind eine spezielle Weiterentwicklung von Diensten, die sich in ihrem Verhalten der Umgebung anpassen. Es wurde gezeigt, wie im Requirements Engineering Prozess die einzelnen Systemfunktionen als Dienste modelliert und miteinander in Beziehung gesetzt werden.

Eng mit dem Begriff Adaptivität ist der Begriff Kontext verbunden. Dieser repräsentiert die Umgebung von adaptiven Diensten. Die Kontextwerte werden über spezielle Kanäle (implizite Eingabekanäle) in den Dienst eingegeben. Ohne das Vorhandensein von Kontextwerten zeigt ein adaptiver Dienst ein definiertes Standardverhalten, d. h. er ist auch ohne Information über die Umgebung sinnvoll lauffähig. Weiterhin wurde gezeigt, dass sich adaptive Dienste durch eine Reihe zusätzlicher Eigenschaften zu herkömmlichen Diensten unterscheiden.

Adaptive Dienste verhalten sich also angepasst an den Kontext. Die verbindende Einheit zwischen adaptiven Dienst und Kontext ist das so genannte Kontextmodell. Es wird durch eine beliebige Entscheidungslogik (z. B. Aussagenlogik, komplexes Expertensystem) repräsentiert.

In einer Fallstudie wurden die vorgestellten Konzepte praktisch umgesetzt. Es wurde ein Tankassistent für ein Fahrzeug modelliert, der dem Fahrer anhand seiner Tankgewohnheiten eine bestimmte Tankstelle vorschlägt bzw. auf die Notwendigkeit eines Tankvorgangs aufmerksam macht.

Jedoch sind weitere Arbeiten auf diesem Gebiet notwendig. Im vorgestellten Konzept ist es grundsätzlich möglich, die Entscheidungslogiken auszutauschen. Da diese unterschiedlich komplex und mächtig sind, eignen sie sich nicht zwingend für alle Einsatzmöglichkeiten gleich gut. Daher ist eine tiefgehende Untersuchung notwendig, welche Entscheidungslogik für welchen Einsatz die jeweils geeignete ist.

Um die Verwendbarkeit in der Praxis zu testen, ist es notwendig, weitere und komplexe Prototypen von adaptiven Systemen auf Basis von Diensten zu erstellen, deren Funktionsfähigkeit dann beispielsweise in einem Testfahrzeug von einer breiten Schicht von Nutzern evaluiert werden können.

Literaturverzeichnis

[Bro03] Manfred Broy. Modeling Services and Layered Architectures. In H. König, M. Heiner, and A. Wolisz, editors, *Formal Techniques for Networked and Distributed Systems*, volume 2767 of *Lecture Notes in Computer Science*, pages 48–61. Springer-Verlag, 2003.

[Bro05] Manfred Broy. Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures – The Janus-Approach. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, number 195 in Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, July 2005. Springer-Verlag.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces and Refinement.* Springer-Verlag, 2001.

[CS93] Volker Claus and Andreas Schwill, editors. *Duden Informatik – ein Sachlexikon für Studium und Praxis.* Dudenverlag, 2te edition, 1993.

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2/4):97–166, 2001.

[Deu05] Martin Deubler. *Strukturierte Nutzungssicht auf multifunktionale Systeme.* PhD thesis, Technische Universität München, 2005.

[DGP+04a] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, and Christian Salzmann. Tool Supported Development of Service Based Systems. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC) 2004, Busan (Korea), November 30 – December 3*, pages 99–108. IEEE Computer Society, 2004.

[DGP+04b] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, and Christian Salzmann. Towards a Model-Based and Incremental Development Process for Service-Based Systems. In M. H. Hamaza, editor, *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 183–188, Innsbruck, Austria, February, 17–19 2004.

[Fah05] Michael Fahrmair. *Kalibrierbare Kontextadaption für Ubiquitous Computing.* PhD thesis, Technische Universität München, 2005.

[Jam01] Anthony Jameson. Modelling both the context and the user. *Personal Ubiquitous Comput.*, 5(1):29–33, 2001.

[Jes] Jess, the Rule Engine for the Java™ Platform. `http://herzberg.ca.sandia.gov/jess/`.

[Mer04] Stephan Merk. Evaluierung des Java Sicherheitsmodells im Kontext von Softwaredownload innerhalb des OSGi-Frameworks. Master's thesis, Technische Universität München, 2004.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development. Technical Report TUM-I0204, Technische Universität München, 2002.

[SS03] Christian Salzmann and Bernhard Schätz. Service based Software Specification. In *Proceedings of Intl. International Workshop on Test and Analysis of Component Based Systems (TACOS)*, Warsaw, Poland, 2003.