Universität Koblenz–Landau
Fachbereich Informatik
Institut für Softwaretechnik

Diplomarbeit

Komponentenorientierte Systemspezifikation und -implementation

Alexander Kaczmarek
In der Weglänge 9
56072 Koblenz

betreut von Prof. Dr. Jürgen Ebert, Dr. Andreas Winter

Dokumentversion: 1.01, 29. September 2005

Abstract

This thesis deals with component orientation. The first goal of the thesis is to get a better understanding of what software components are. Different approaches towards component oriented software development are studied, in order to collect a set of criteria regarding software components.
Based on those criteria a catalogue of requirements is developed, that covers all relevant features a software component should fulfill. Today's so called component technologies are afterwards checked on the basis of the developed catalogue of requirements.

The second goal of the thesis is the design of a new component model. This model focuses on the composition of software components. After the introduction of the new component model it will be validated against the proposed requirements.

The feasibility of the introduced component model is demonstrated in a case study. This case study is based on the scenario of a corporate information system.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit befasst sich mit dem Thema **Komponentenorientierung**. Der Schwerpunkt der Arbeit liegt in der Aufstellung eines **Anforderungskataloges** an Komponenten und der Entwicklung eines neuen **Komponentenkonzeptes**, das genutzt werden kann, um Softwaresysteme komponentenorientiert spezifizieren und implementieren zu können.
Zunächst wird allgemein das Vorgehen innerhalb der komponentenorientierten Entwicklung, sowie zugehörige Überlegungen und Motivationen betrachtet, die den Einsatz von Komponenten so vielversprechend erscheinen lassen.

1.1 Komponentenorientierte Entwicklung

Die Softwaretechnik ist ein immer noch junges Fach, das mit klassischen Ingenieurswissenschaften in vielen Bereichen sehr verwandt ist. Im Gegensatz zur Softwaretechnik wird in anderen Fachbereichen schon seit vielen Jahren komponentenorientiert entwickelt und gefertigt. *Komponentenorientiert* bedeutet in diesem Fall, dass vorgefertigte Bauteile durch Zusammenstecken größere Einheiten bilden, und dass ein Bauteiltyp in unterschiedlichen Szenarien oder Produkten zum Einsatz kommt.
Klassisches Beispiel für eine solche Art der Fertigung ist die Automobilindustrie. Anhand dieses Beispiels werden im Folgenden Analogien zur Softwaretechnik untersucht.

1.1.1 Allgemeine Problemstellungen

Von großer Wichtigkeit in der heutigen Produktentwicklung ist die **Dauer** von der ersten Planung bis zur Fertigung eines neuen Produktes. Diese Dauer wird als *Time To Market* bezeichnet.

Neben der **Schnelligkeit** zählt mindestens ebenso die **Reife** des neuen Produktes. Die Produktreife wird gekennzeichnet durch eine niedrige Fehlerrate, die durch ausgiebige Erprobung, sowie praktischen Einsatz sichergestellt werden kann.
Der dritte relevante Punkt für die erfolgreiche Einführung eines neuen Produktes ist die **Kostenminimierung**. Für Kunden hat zwar die Entwicklungsdauer und Korrektheit der neuen Software höchste Priorität, die sich aus diesen beiden Faktoren potentiell höheren Kosten sollen dabei für den Kunden möglichst nicht anfallen. Eine Möglichkeit zur Vermeidung höherer Entwicklungskosten ergibt sich durch mehrfache Verwendung eines Bauteiletyps in verschiedenen Einsatzszenarien.

Die heutige Fertigung im Automobilsektor wäre ohne den Ansatz der Komponentenorientierung nicht mehr denkbar. Bei der Fertigung eines neu konzipierten Fahrzeugs kann der Hersteller auf einen großen Fundus bereits vorhandener Bauteile zurückgreifen. Oftmals setzt ein neues Fahrzeug sogar vollständig auf eine bereits vorhandene Plattform auf. Einzelne Teile der Mechanik, Elektronik oder der sonstigen Ausstattung sind ebenfalls identisch mit Bauteilen anderer Automobile.
Diese gemeinsame Nutzung unterschiedlicher Bauteile verläuft sogar über Herstellergrenzen hinweg. Verschiedene Zulieferer der Automobilbranche haben sich auf Subbereiche der Fertigung spezialisiert. So existieren z. B. Zulieferer für Bremsanlagen, die von mehreren Herstellern in der Fertigung eingesetzt werden.
Die zuvor genannten Problemstellungen Schnelligkeit, Korrektheit und Kostenminimierung werden im Automobilsektor durch die Verwendung **vorgefertigter Bauteile, auch Komponenten genannt,** erreicht. In der Softwareentwicklung existieren die gleichen Grundprobleme: Software sollte schnell entwickelt werden können, sie sollte korrekt arbeiten und die Kosten sollten möglichst gering gehalten werden. Die Verwendung von Software-Komponenten verspricht diese Probleme zu lösen.

Laut Clemens Szyperski [12] ist für Software-Komponenten **Kompositionsfähigkeit** das hauptsächliche Merkmal, einhergehend mit **Wiederverwendbarkeit**. Eine Software-Komponente kann demnach vereinfachend betrachtet werden als ein in sich **abgeschlossener, komponierbarer, wiederverwendbarer Softwarebaustein**, der über eine definierte syntaktische Schnittstelle und Semantik verfügt.

Die Komponentenorientierung innerhalb der Softwaretechnik versucht relativ analog zu verwandten Ingenieurswissenschaften komponentenorientiertes

Vorgehen wie in der Automobilbranche auf die Entwicklung von Software zu übertragen. Aus bestehenden Software-Komponenten werde komplexe Softwaresysteme **assembliert**. Diese Systeme sind einzig durch Zusammenstecken und Konfektionieren bereits vorhandener Bausteine realisierbar. Existiert für einen Spezialfall noch keine passende Komponente, so muss und kann diese spezifiziert und implementiert werden.
Der Aspekt der **Standardisierung** ist im Rahmen der Komponentenorientierung ein interessantes Thema. Die Leistungen von Komponenten könnten abstrakt spezifiziert werden, um so verbindliche Standards für den gesamten Komponentenmarkt zu schaffen. Sowohl Komponentenentwickler wie auch Systemassemblierer (vgl. 1.1.2) könnten sich auf verbindliche Standards bei der Entwicklung und der Verwendung von Komponenten beziehen.
Für die erfolgreiche Verbreitung und die Schaffung eines Marktes für Komponenten wäre eine Standardisierung von Vorteil. Um eine Standardisierung vornehmen zu können, muss allerdings zunächst die benötigte Grundlage in Form eines allgemein akzeptierten Komponentensystems geschaffen werden.

Die heute verfügbare Literatur zum Thema *Komponentenorientierung* befasst sich hauptsächlich mit **technologischen Problemen**, die beim Einsatz von Komponenten überwunden werden müssen. Nur stellenweise werden die Auswirkungen der Ausrichtung der Softwareentwicklung auf die Komponentenorientierung angesprochen.
Um einen kurzen Einblick zu bekommen, wie die Komponentenorientierung die heutige Softwareentwicklung verändern kann, werden im Folgenden potentielle Rollen und Aufgabenverteilungen im Rahmen der komponentenorientierten Entwicklung vorgestellt.

1.1.2 Stakeholder

Nachfolgend werden die Personen und Rollen, die sogenannten *Stakeholder*, betrachtet, die an der Entwicklung und dem Einsatz eines komponentenorientierten Systems beteiligt sind. Dazu passend werden mögliche Ausprägungsformen der Aufgabenverteilung in einem Komponentenszenario besprochen.
Bei der Betrachtung der verschiedenen involvierten Stakeholder wird in diesem Abschnitt davon ausgegangen, dass ein Komponentensystem etabliert ist, alle technologischen Probleme gelöst sind, und ein Markt für Software-Komponenten existiert.
In dieser Betrachtung werden vier Stakeholder identifiziert. Dazu zählen *Komponentenspezifizierer*, *Komponentenentwickler*, *Systemassemblierer* und *Anwender*.

Komponentenspezifizierer

Ein *Komponentenspezifizierer* spezifiziert abstrakt den **Leistungsumfang** einer Komponente. Diese Leistungspezifikationen enthalten syntaktische und semantische Aspekte.

Komponentenentwickler

Die Aufgabe eines *Komponentenentwicklers* besteht darin, nach einer gegebenen **Leistungsspezifikation** eine Komponente gemäß einer **Komponentensystemspezifikation** zu entwickeln. Die Komponentensystemspezifikation gibt den **technischen Rahmen** vor, der erfüllt sein muss, damit es sich um eine korrekte Komponente handelt. Die Leistungsspezifikation dagegen umfasst Anforderungen an die Schnittstelle und den Leistungsumfang der zu entwickelnden Komponente.
Komponenten werden schon heute relativ unabhängig vom tatsächlichen Verwendungszweck entwickelt. Ein Komponentenhersteller geht dabei von einer aus seiner Sicht nützlichen Funktionalität aus, die er als Komponente entwickelt und veröffentlicht. Ob es einen tatsächlichen Anwendungsfall gibt, und ob die Komponente in genau dieser Form verwendet wird, ist zum Zeitpunkt der Entwicklung oft nicht abzusehen.

Systemassemblierer

Vorgabe für einen *Systemassemblierer* ist die **Spezifikation der Funktionalität** eines gesamten Systems. Hiervon ausgehend wählt ein Systemassemblierer Komponenten aus, die möglichst alle benötigten Funktionalitäten abdecken. In einem weiteren Schritt werden diese Komponenten durch den Systemassemblierer so verschaltet, dass sich das Gesamtsystem aus den einzelnen Bausteinen ergibt.
Sollten mehrere Komponenten nicht direkt miteinander kooperieren können, so ist es ebenfalls Aufgabe des Systemassemblierers, den benötigten **Glue Code** zu schreiben. Dieser verbindende Programmcode sollte in Form von Skripten oder ähnlichem geschrieben werden können.
Zur Assemblierung von Komponenten zählt auch die Konfektionierung. In vielen Fällen wird es nötig sein, Komponenten an den Ausführungskontext per Konfektionierung anzupassen. Der Rahmen der Anpassungsfähigkeit wird durch die Komponenten selbst vorgegeben. Die Anpassung selbst muss innerhalb dieses Rahmens ohne Eingriff in die Implementierung einer Komponente möglich sein.

Anwender

Der *Anwender* eines kompontenbasierten Systems benutzt wie in herkömmlichen Softwaresystemen eine **Benutzerschnittstelle**, um Zugriff auf das Softwaresystem zu erhalten. Dabei bemerkt der Anwender im Regelfall nicht, ob es sich um ein komponentenorientiertes System handelt oder nicht.
Allerdings ist nicht ausgeschlossen, dass manche heutige Anwender in einem komponentenorientierten Szenario auch zu Systemassemblierern werden. Vor allem im Bereich der optionalen Erweiterung bzw. Anpassung an eigene Bedürfnisse könnten erfahrene Anwender ihr System mit den Mitteln der Komponentenorientierung modifizieren.

Neben den einzelnen Stakeholdern ist ebenfalls die Verteilung der mit diesen Rollen verbundenen Aufgaben von Interesse. Der nächste Abschnitt widmet sich möglichen Aufgabenverteilungen bei der Erstellung und dem Einsatz komponentenorientierter Softwaresysteme.

1.1.3 Szenarien der Aufgabenverteilung

Ausgehend von den zuvor vorgestellten Stakeholdern sind verschiedene tatsächliche Verteilungen der Aufgaben möglich. Die zwei Hauptformen *herstellerseitige* und *kundenseitige Assemblierung* werden im Folgenden kurz vorgestellt.
Charakteristisch für die herstellerseitige Assemblierung ist, dass die komponentenorientierte Entwicklung vollständig beim Softwarehersteller liegt und für die Systemanwender versteckt ist. Im Gegensatz dazu ist im Rahmen der kundenseitigen Assemblierung der Systemanwender selbst Teil der komponentenorientierten Entwicklung. Zusätzlich zu diesen Formen sind weitere Mischformen denkbar.

Herstellerseitige Assemblierung

Der Hersteller und Anbieter eines Softwaresystems liefert dieses **System als Ganzes** seinen Kunden aus. Das System arbeitet komponentenorientiert, es kommen sowohl eigene wie auch Fremdkomponenten zum Einsatz. Neben den notwendigen Komponentenentwicklern existieren auch Systemassemblierer, die alle Komponenten zu dem Gesamtsystem zusammensetzen. Der Kunde erhält ein Gesamtsystem, dem die Komponentenorientierung nicht anzusehen ist.
In diesem Szenario nutzt der Hersteller die Komponentenorientierung zur effizienten Entwicklung der eigenen Produkte. Für den Kunden ist die Art

der Realisierung irrelevant.

Kundenseitige Assemblierung

Es gibt **keinen eigentlichen Hersteller** des gesamten Softwaresystems. Vielmehr nutzt ein Endanwender die komponentenorientierte Entwicklung, um sich selbst aus verfügbaren Komponenten ein eigenes Softwaresystem zusammenzustellen. Dieses System ist speziell auf die Anforderungen dieses Konsumenten zugeschnitten und wird nicht weiter vermarktet.
Diese Art der Nutzung der Komponentenorientierung kann vor allem in größeren Firmen mit eigener IT-Abteilung durchgesetzt werden. Die hier vorhandenen Administratoren sollten in der Lage sein, die Rolle von Systemassemblieren auszufüllen.

1.2 Ziel der Arbeit

Diese Arbeit gibt einen **Überblick** über heutige Auffassungen und Umsetzungen der komponentenorientierten Entwicklung. Aufbauend auf den aus diesen Betrachtungen gewonnenen Erkenntnissen wird ein **Komponentenkonzept** entwickelt, das die Ziele der Komponentenorientierung hinsichtlich Komponierbarkeit und Wiederverwendbarkeit erfüllt.

1.3 Aufbau der Arbeit

Nach dem einleitenden Kapitel beschäftigt sich zunächst Kapitel 2 mit grundlegenden Betrachtungen des Themenbereichs *Komponentenorientierung*. Dazu werden die Arbeiten einiger **Autoren** betrachtet, die sich bereits intensiv mit der Thematik auseinandergesetzt haben.
Ausgehend von den Anforderungen an Komponentensysteme, die durch diese Autoren aufgestellt werden, schließt sich an die Literaturbetrachtung die Entwicklung eines eigenen **Anforderungskataloges** an Komponenten an. Aufgrund dieser gesammelten Kriterien soll die Einordnung einer Technologie als Komponententechnologie möglich sein.
Angewendet wird die so aufgestellte Liste von Kriterien im weiteren Verlauf des zweiten Kapitels bei der Betrachtung heute existenter sogenannter **Komponententechnologien**. Die einzelnen Technologien werden unter Berücksichtigung der Anforderungen betrachtet und der Grad der Erfüllung beurteilt. Anhand dieser Beurteilung kann festgestellt werden, inwiefern es sich tatsächlich um Komponententechnologien handelt.

Auf das Grundlagenkapitel folgt in Kapitel 3 die Entwicklung und Vorstellung eines **eigenen Konzeptes** für Komponenten und vollständige, komponentenbasierte Systeme. Nach einem allgemeinen Überblick über die Konzeptidee werden die einzelnen Bestandteile detailliert vorgestellt und erläutert.
Neben dem eigentlichen eher abstrakten Konzept werden in diesem Kapitel ebenfalls Möglichkeiten zur **Repräsentation** dieser Komponenten geliefert und auch Vorgehensweisen bei der Erstellung eines Komponentensystems vorgestellt.

Um das zuvor aufgestellte Konzept möglichst praxisnah mit einem realistischen Umfeld zu konfrontieren, folgt in Kapitel 4 ein **Fallbeispiel**. Dieses Beispiel soll möglichst viele Aspekte des Komponentenkonzepts aufgreifen und deren Funktionsweise an einer gegebenen Aufgabenstellung demonstrieren. Auf diesem Weg soll die Mächtigkeit und die Grenzen des neuen Konzeptes verdeutlicht werden.

Die Arbeit schließt mit Kapitel 5, das zunächst eine **Zusammenfassung** und ein **Fazit** der Arbeit bietet. Basierend auf den gesammelten Erkenntnissen bei der Entwicklung des Konzeptes und der Übertragung auf das Fallbeispiel wird ebenfalls ein **Ausblick** gegeben, wie das Komponentenkonzept erweitert und fortentwickelt werden könnte.

Kapitel 2

Grundlagen

F Der Begriff der **Komponente** wird in der Literatur, Forschung und Industrie häufig verwendet. Dabei divergieren die Bedeutungen und oft impliziten Anforderungen an eine Komponente beinahe ebenso oft.

Um eine Grundlage für die Einordnung eines Stückes Software als Komponente zu schaffen, beschäftigt sich dieses Kapitel mit den unterschiedlichen Auslegungen des Begriffes. Dazu werden verschiedene Quellen und Autoren herangezogen, um deren Standpunkte zusammenfassend darzulegen.
Die so gesammelten Ansichten werden nachfolgend verglichen. Vor allem aus den sich ergebenden Differenzen können hier Rückschlüsse auf die Sicht des jeweiligen Autors getroffen werden.
Nach der Literaturbetrachtung wird ein Anforderungskatalog für Software-Komponenten entwickelt. Anhand dieses Kataloges soll es möglich sein, den Grad der Komponentenorientierung einer Technologie einschätzen zu können.
Abschließend werden in diesem Kapitel Vertreter sogenannter Komponententechnologien betrachet. Anhand des zuvor aufgestellten Kataloges werden die Eigenschaften und der angesprochene Grad der Komponentenorientierung dieser Technologien untersucht.

2.1 Autorenmeinungen

Diverse Bücher und weitere Quellen existieren rund um das Thema Komponenten. Eine Auswahl wird in diesem Kapitel näher auf die darin enthaltenen Anforderungen an Komponenten betrachtet.

2.1.1 Frank Griffel, *Componentware*

Frank Griffel entwickelt innerhalb des Schlusskapitels seines 1998 erschienenen Buches *Componentware* [3] eine Anforderungsliste an ein Stück Software, das eine Komponente darstellen soll. Diese Liste umfasst folgende Punkte:

Um als Komponente betrachtet werden zu können muss ein Stück Software nach Griffel eine **nützliche Funktionalität** bereitstellen, die von anderen Softwareteilen sinnbringend eingesetzt werden kann. Dabei solle diese Funktionalität **keine Speziallösung** darstellen, sondern gängiger Natur sein. Diese Funktionalität solle so wie sie sich darstellt **akzeptiert** werden. Zwar seien dann öfter Kompromisse nötig, da eine Komponente nicht immer die tatsächlich benötige Funktionalität aufweist. Aber durch diese Kompromissfähigkeit werde die Möglichkeit eines **Plug&Play-Verhaltens** eröffnet.

Um von einer Komponente zu sprechen, müsse der Software der Gedanke der **möglichst häufigen Wiederverwendung** zugrunde liegen. Würde die Software nur für einen Spezialfall entwickelt und könnte sie somit nur schlecht wieder verwendet werden, so stelle sie auch keine Komponente dar.

Ebenso ist eine **strikte Kapselung** für Griffel charakteristisch für eine Komponente. Die Funktionalität solle nur über **verbindliche Schnittstellen** der Außenwelt verfügbar gemacht werden. Die Schnittstellenbeschreibungen sollen in **plattformneutraler Form** vorliegen, getrennt von der tatsächlichen Implementation. Eine Komponente solle auch aktiv Unterstützung zur eigenen Benutzung anbieten, indem sie eine **Selbstdokumentation** bzw. Selbstbeschreibungsfähigkeit zur Verfügung stellt.

Damit ein Stück Software *Komponente* genannt werden kann, muss sie laut Griffel **kooperativ** (komponierbar) sein. Sie sei immer Teil einer Anwendung oder einer komplexeren Komponente. In einem Verbund **lose gekoppelter** Softwareteile solle sie zu einer **übergeordneten Funktionalität** beitragen. Eine Ad-hoc-Interaktion mit anderen Softwareteilen solle möglich sein, ohne dass diese Zusammenarbeit zuvor explizit vorgesehen war.

Nach Griffel soll eine Software-Komponente außerdem weitestgehend **orts- und plattformübergreifend** einsetzbar sein. Die Ortsunabhängigkeit bezieht sich auf den Ort der Ausführung einer Komponente, der ein lokaler Rechner oder auch ein Server im Intra-/Internet sein kann. Plattformübergreifend einsetzbar ist eine Komponente, wenn sie auf verschiedenen Betriebsystemen bzw. Laufzeitumgebungen funktioniert.

Des Weiteren besitze eine Komponente ein **zustandbehaftetes Verhalten**. Der Zustand solle dabei vorkonfigurierbar sein, und auch später im laufenden Betrieb angepasst werden können. Durch diese Anpassbarkeit solle die Komponente auch in Szenarien nutzbar sein, an die während der Entwicklung nicht explizit gedacht worden ist.

Die Nutzung einer Komponente soll laut Griffel **ohne systemspezifische Programmierkenntnisse** möglich sein. Für den Fall der Benutzerinteraktion sei eine **visuelle Repräsentation** wünschenswert. Auf diesem Weg kann die Komposition und Konfektionierung von Komponenten auch von Benutzern ohne umfangreiche Programmierkenntnisse vorgenommen werden.

Griffel führt weiter an, dass eine Komponente spezifische Merkmale aufweisen solle, anhand derer sie **klassifizierbar** ist. Eine einmal veröffentlichte Komponente solle auch **langfristig** ausgelegt sein in Hinblick auf Verfügbarkeit und Stabilität der Schnittstellen. Schlussendlich bildet nach Griffel eine Komponente eine **Einheit der Auslieferung**, d.h. sie wird niemals nur teilweise ausgeliefert, sondern immer als voll funktionsfähige Einheit.

Neben diesen Kriterien unterscheidet Griffel zwei Arten von Komponenten: **aktive** und **passive**. Eine aktive Komponente ist demnach charakterisiert durch eine gewisse Selbständigkeit, denn sie läuft in einem eigenen Prozess. Dagegen ist eine passive Komponente stärker abhängig von dem aufrufenden Element, da sie keinen eigenen Prozess besitzt und nur zusammen mit einer Hostanwendung lauffähig ist. Beispiele für passive Komponenten sind Bibliotheken.

Bewertung der Quelle

Das Buch *Componentware* von Frank Griffel weist eine deutliche Schwäche auf: der Begriff der Komponente wird fast bis zum letzten Kapitel unscharf gelassen. Der Autor unternimmt diesen Schritt vollkommen bewusst, da er den Leser nicht einschränken möchte in dessen eigener Vorstellung einer Komponente. Das Ergebnis ist allerdings, dass dadurch nie ganz klar ist, was eigentlich unter einer Komponente zu verstehen ist.

Erst im Schlusskapitel geht Griffel auf die oben aufgeführten Eigenschaften ein. Für ein besseres Verständnis und eine klare Begriffsschärfung sollte auf eine solch späte Klärung der Begrifflichkeiten verzichtet werden.

Ansonsten enthält das Buch einige interessante Aspekte. Heutige Komponententechnologien, wie auch heutige Probleme im Umgang mit Komponenten werden beleuchtet und auf verständliche Art präsentiert.

2.1.2 Clemens Szyperski, *Component Software*

Clemens Szyperski bespricht in *Component Software* [12] ebenfalls Eigenschaften von Komponenten. Drei Punkte sind nach dem 1999 veröffentlichten Buch charakteristisch für Software-Komponenten:

- „A component is a unit of **independent deployment**."
  Aus dieser Eigenschaft folge, dass eine Komponente gut von ihrer Umgebung und anderen Komponenten abgeschirmt sein müsse, und ihre eigenen Eigenschaften kapselt. Außerdem werde eine Komponente niemals teilweise installiert, da sie eine Einheit bilde.

- „A component is a unit of **third-party composition**."
  Der Aspekt der *third-party* verweist darauf, dass in der Regel die Entwickler und Anwender von Komponenten verschiedene Personen sind. Damit Komponenten durch Dritte komponiert werden können, sei es Voraussetzung, dass sie in sich geschlossen sind. Zusätzlich seien klare Spezifikationen nötig für Anforderungen und Leistungen der Komponenten.

- „A component has **no persistent state**."
  Eine Komponente kann laut Szyperski nicht von einer eigenen Kopie unterschieden werden. Da eine Komponente keinen Status habe, machten mehrere Kopien einer Komponente in einem System wenig Sinn. Dabei schließt Szyperski nicht aus, dass eine Komponente zustandsabhängige Ergebnisse liefert. Nur müsse unterschieden werden zwischen der Komponente und z. B. den Daten auf denen die Komponente arbeitet. Als Beispiel wird ein Datenbankserver und zugehörige Datenbank angeführt. Die Kombination aus Datenbank und -server kann nach obiger Definition nicht als Komponente bezeichnet werden, wohl aber das statische Datenbank-Server-Programm, das das „Datenbankobjekt" unterstützt. Nach Szyperski sei diese Unterscheidung notwendig, um den unveränderbaren *Plan* von veränderlichen *Instanzen* zu trennen, um so massive Wartungsprobleme zu vermeiden.

Szyperski greift abschließend eine klare Definition auf, die auf der *European Conference on Object-Oriented Programming 1996* unter seiner Mitwirkung entstanden ist und die zuvor genannten Eigenschaften nochmals aufgreift:

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“

Bewertung der Quelle

Der Untertitel *Beyond Object-Oriented Programming* gibt bereits Aufschluss über die Ausrichtung des Buches *Component Software* von Clemens Szyperski. Szyperski sieht die Komponentenorientierung als eine alternative Softwareentwicklungsform der Zukunft an. Aus dieser Motivation heraus schafft er eine gute terminologische Grundlage, um so besser über Komponenten und z. B. Objekte sprechen zu können. Des Weiteren werden die wichtigsten Probleme bei der Entwicklung von und im Umgang mit komponentenorientierten Systemen besprochen.
Im weiteren Verlauf des Buches bespricht Szyperski wichtige und verbreitete heute verfügbare Technologien aus dem Komponentenbereich, und zeigt Ansätze für die zukünftige Entwicklungsarbeit auf. Sehr gut gefallen hat hier ein Ausblick auf mögliche zukünftige Komponentenmärkte und auch neue Berufsfelder, die nach Etablierung der komponentenorientierten Softwareentwicklung nötig sein dürften.
Das Buch ist eine wichtige Lektüre für jeden, der sich mit Komponentenorientierung und deren Auswirkungen befasst.

2.1.3 Volker Gruhn, *Komponentenmodelle*

Gruhn stellt in seinem 2000 erschienenen Buch *Komponentenmodelle* [5] Eigenschaften auf, die eine Komponente aufweisen sollte. Zusammenfassend stellen sich diese Eigenschaften wie folgt dar:

Für Gruhn muss eine Komponente einen **wohldefinierten Zweck** erfüllen. Eine Komponente sei ein Spezialist für eine bestimmte Aufgabe. Diese Aufgabe sei umfassender als der Dienst eines einzelnen Objektes, aber auch nicht so umfangreich, dass eine Komponente eine eigenständige Anwendung darstellt.

Die Schnittstelle einer Komponente solle alle Informationen liefern, die nötig sind, um über die Benutzung einer Komponente zu entscheiden. Arbeiten mehrere verteilte Komponenten zusammen, so solle dies möglichst unabhängig von den gegebenen Rahmenbedingungen möglich sein. Diese Eigenschaft nennt Gruhn **Kontextunabhängigkeit**.

Bei der Entwicklung einer Komponente solle die Wahl der Programmiersprache dem Entwickler freigestellt sein. Einzig die universelle Verwendbarkeit der fertigen Komponente spiele eine Rolle, nicht aber die Art der eingesetzten Entwicklungsumgebung. Auch der spätere Einsatz solle plattformunabhängig erfolgen können. Diese Eigenschaften werden unter **Portabilität und Programmiersprachenunabhängigkeit** zusammengefasst.

Kommt eine Komponente zum Einsatz, so solle es irrelevant sein, wo die Komponente ausgeführt wird. Mit **Ortstransparenz** betitelt Gruhn die Eigenschaft, dass es für das Verhalten einer Komponente keine Rolle spiele, ob sie auf einem lokalen Rechner oder einem entfernten Server befindet.

Eine strikte Kapselung der Implementierung wird ebenfalls gefordert. Eine Komponente stelle für den Benutzer eine Blackbox dar, ein Zugriff auf das Innenleben sei ausdrücklich nicht erwünscht. Durch die **Trennung von Schnittstelle und Implementierung** müssten die spezifizierten Schnittstellen ausreichen, um mit der Außenwelt zu interagieren.

Des Weiteren solle eine Komponente über **Selbstbeschreibungsfähigkeit** verfügen. Die von ihr bereitgestellten Dienste sollten durch die Komponente selbst beschrieben werden können. Für die Komponierbarkeit bzw. späte Bindung zur Laufzeit eines Systems sei diese Eigenschaft essentiell.

Eine Komponente solle weiterhin in der Lage sein, sich selbst zu installieren und zu registrieren. Dadurch solle erreicht werden, dass Komponenten **Plug&Play-fähig** sind.

Für Gruhn ist es wichtig, dass Komponenten zu größeren Einheiten zusammengesetzt werden können, die wiederum Komponenten darstellen. **Diese Integrations- und Kompositionsfähigkeit** solle auch nicht nur statisch zur Entwurfszeit erfolgen können, sondern dynamisch zur Laufzeit eines Systems. Dabei solle eine Komponentenarchitektur benötigte Basisdienste für z. B. den Nachrichtenaustausch zur Verfügung stellen.

Den Aspekt der **Wiederverwendbarkeit** greift Gruhn nochmals separat auf. Demnach solle eine Komponente möglichst einfach wieder verwendet werden können. Ziel sei es, die Wiederverwendung auf Anwendungsniveau zu ermöglichen. Zusammen mit der bereits erwähnten Trennung von Schnittstelle und Implementierung ergibt sich eine Blackbox-Wiederverwendung, die für Komponenten gefordert wird.

Eine weitere Eigenart von Komponenten ist laut Gruhn, dass die genaue spätere Verwendung zum Zeitpunkt der Entwicklung nicht immer absehbar sei. Daher solle eine Komponente **konfigurierbar** und **anpassbar** sein. Dadurch solle erreicht werden, dass eine Komponente nach der Auslieferung ohne erneute Kompilierung in der jeweiligen Einsatzsituation betrieben werden kann.

Abschließend führt Gruhn an, dass Komponenten besonders ausgiebig erprobt und getestet werden sollten, um so die **Bewährtheit** zu gewährleisten. Außerdem solle eine Komponente **binär verfügbar** sein, d.h. in lauffähiger Form vorliegen. Der Quelltext einer Komponente müsse nicht mit ausgeliefert werden.

Zusätzlich zu diesen Eigenschaften unterscheidet Gruhn prinzipiell zwischen **Client**- und **Server**-Komponenten. Während Client-Komponenten demnach Benutzungsoberflächen und dazugehörigen Plausibilitätsprüfungen realisieren, werden in Server-Komponenten Geschäftsobjekte mitsamt ihren Daten und Methoden abgebildet. Diese Art von Komponenten repräsentiert nach Gruhn die Anwendungslogik, während je nach Präsentationskanal bzw. Endgerät verschiedene Client-Komponenten die Darstellung übernehmen.

Bewertung der Quelle

Bereits im Grundlagenkapitel schafft Gruhn Klarheit über den Komponentenbegriff innerhalb seines Buches. Ganz allgemein geht er auf Anforderungen an Komponenten ein, ohne vorhandene Technologien zu berücksichtigen. Nachfolgend werden gängige Technologien wie COM und CORBA betrachtet, um abschließend im letzten Kapitel anhand einiger Kriterien vergleichend bewertet zu werden. Diese Gegenüberstellung ist sehr interessant. Etwas verwirrend ist hier, dass sich die Kriterien aus dem Grundlagenkapitel nicht direkt mit den Vergleichskriterien des Abschlusskapitels decken.
Zusammenfassend kann man sagen, dass das Buch von Gruhn gut strukturiert einen umfangreichen Überblick über aktuelle Komponententechnologien gibt, und dabei auch allgemeine Aspekte der Komponentenorientierung beleuchtet. Alleine vom Umfang liegt hier allerdings der Fokus auf der Vorstellung dieser Technologien, was auch durch den Untertitel des Buches *DCOM, JavaBeans, Enterprise JavaBeans, CORBA* eindeutig aufgezeigt wird.

2.1.4 Gesellschaft für Informatik, *Vereinheitlichte Spezifikation von Fachkomponenten*

Der Arbeitskreis „Komponentenorientierte betriebliche Anwendungssysteme“ (AK 5.10.3) der Gesellschaft für Informatik hat im Februar 2002 unter dem Herausgeber Klaus Turowski eine Spezifikation von so genannten *Fachkomponenten* veröffentlicht. Im Rahmen der Spezifikation werden auch hier Eigenschaften von Komponenten beschrieben, die zusammen mit allgemeinen Überlegungen der Autoren zur Gesamtthematik nachfolgend zusammengefasst werden.
Ausgehend von traditionellen Ingenieursdisziplinen stellen die Autoren initial fest, dass analoge Verfahren und Standards in der Softwaretechnik heute noch fehlen. Gelten in anderen Bereichen in der Regel verbindliche Standards für Notation, Benennung und Bemaßung, die zur Spezifikation von Konstruktionsergebnissen herangezogen werden, um so die Wiederverwendung und Benutzung durch Dritte zu vereinfachen, so gebe es solche Standards in der Softwaretechnik noch nicht.
Ergebnis der vorgestellten Spezifikation soll demnach die Erarbeitung von Vorschlägen sein, wie eine solche zukünftige Spezifikation von Fachkomponenten aussehen kann.
Als Ziel der Komponentenorientierung wird durch die Autoren die kundenindividuelle Kombination von Softwarekomponenten zu einem Anwendungssystem postuliert. Auf diesem Weg sollen sich die Vorteile der Verwendung von Standard- und Individualsoftware verbinden. Diesem Ziel liege also „die Idee einer kompositorischen, plug-and-play-artigen Wiederverwendung von Blackbox-Komponenten zu Grunde, deren Realisierung einem Verwender vorborgen bleibt und die auf einem Softwaremarkt gehandelt werden.“ [S.1]
Unterschieden wird in der Spezifikation zwischen Komponenten und Fachkomponenten. Der Begriff der *Komponente* wird wie folgt definiert:

„Eine Komponente besteht aus verschiedenartigen (Software-)Artefakten. Sie ist wiederverwendbar, abgeschlossen und vermarktbar, stellt Dienste über wohldefinierte Schnittstellen zur Verfügung, verbirgt ihre Realisierung und kann in Kombination mit anderen Komponenten eingesetzt werden, die zur Zeit der Entwicklung nicht unbedingt vorhersehbar ist.“

In dieser Definition erscheinen einige Merkmale und Eigenschaften, die durch eine Komponente erfüllt werden müssen. **(Software-)Artefakte** sind den Autoren folgend ausführbarer Code, verankerte Grafiken, Textkonstanten und ähnliche Elemente, sowie die Daten, die den initialen Zustand einer Komponente beschreiben, und schließlich die Spezifikation, Dokumentation und

auch Tests.
Der erste von einer Komponente zu erfüllende Aspekt ist der der **Wiederverwendbarkeit**. Laut den Autoren sei dieses Kriterium erfüllt, wenn ein Stück Software mit nur geringem Integrationsaufwand für verschiedene Softwaresysteme einsetzbar sei. Dabei dürften die zugehörigen Artefakte nicht modifiziert werden müssen, es sei denn durch eine vorgesehene Parametrisierung.
Die **Abgeschlossenheit** werde erreicht, wenn sich die betroffene Komponente als Ganzes klar von anderen Systemteilen abgrenze. Durch eine eindeutige Zuordnung der Bestandteile (Artefakte) einer Komponente werde diese Eigenschaft erzielt.
Das nächste Kriterium der Definition ist die **Vermarktbarkeit**. Die zuvor genannte *Abgeschlossenheit* sei Voraussetzung für die Vermarktbarkeit einer Komponente. Denn um vermarktbar zu sein, müsse sie als ein für sich selbst stehendes Gut identifizierbar sein. Nur so könne sie auf einem offenen oder auch unternehmensinternen Markt gehandelt werden.

Den Begriff der Komponente verfeinernd wird durch die Autoren der Begriff der *Fachkomponente* wie folgt eingeführt:

„Eine Fachkomponente ist eine Komponente, die eine bestimmte Menge von Diensten einer betrieblichen Anwendungsdomäne anbietet.“

Der relevante Teil dieser Definition ist der Aspekt der *betrieblichen Anwendungsdomäne*. Eine Fachkomponente ist per Definition immer bezogen auf die Lösung einer bestimmten Aufgabe in einem expliziten betrieblichen Umfeld. Im Rahmen der Spezifikation wird hier als Beispiel und Fallstudie eine Fachkomponente *Flugticketverkauf* genannt. Diese Komponente „stellt Dienste zur Verfügung, die ein Reisebüro zum Verkauf von Flugtickets benötigt. Dazu zählen Verwaltung und Auswahl von Flugverbindungen und Verwaltung und Verkauf von Flugreisen durch ein Reisebüro.“ [Ackermann, S. 2]

Neben dieser Art von einfachen Komponenten wird eine weitere Gruppe benannt, die der *Systemkomponenten*. Während Fachkomponenten anwendungsspezifische Funktionalität erfüllten, stellten Systemkomponenten generische Funktionen zur Verfügung. Diese Unterscheidung wird auch bei zusammengesetzten Komponenten vorgenommen. Ein *Komponenten-System-Framework* stellt demnach mehrere zusammengesetzte Systemkomponenten dar, ein *Komponenten-Anwendungs-Framework* zusammengesetzte Fachkomponenten.
Fertige zusammengesetzte Komponenten, die in ihrer vorliegenden Form ohne

Modifikation eingesetzt werden können, werden durch die Autoren *Baugruppe* bzw. *Anwendung* genannt, je nachdem, ob sie die Funktionalität ihrer Domäne passiv zur Verfügung stellen, oder in sich vollständige Anwendungssysteme bilden.

Bewertung der Quelle

Die *Vereinheitlichte Spezifikation von Fachkomponenten* des Arbeitskreises 5.10.3 der Gesellschaft für Informatik ist eine sehr interessante Quelle. Da es sich hierbei nicht um eine eher allgemeine Betrachtung der Komponentenorientierung, sondern um eine zielgerichtete Spezifikation handelt, sind die Ergebnisse deutlich greifbarer als die der zuvor vorgestellten Quellen. Vor allem die Klärung der Begrifflichkeiten, mitsamt der Einführung des Begriffes *Fachkomponente* gefallen hier gut.
Dem interessierten Leser ist dieses Memorandum unbedingt zu empfehlen, da explizite Vorschläge für die Realisierung von Fachkomponenten auf allen möglichen Ebenen enthalten sind.

2.2 Eigenschaften von Software-Komponenten

Nachdem in den vorangegangenen Abschnitten die Auffassung anderer Autoren gesammelt und bewertet worden sind, fasst dieser Abschnitt nun die gewonnenen Erkenntnisse zusammen. Ergebnis soll eine Liste von Kriterien sein, anhand derer entschieden werden kann, ob es sich bei einem Stück Software um eine Komponente handelt. Außerdem werden weitere Eigenschaften betrachtet, die einer Komponente zu einem möglichst breiten Einsatzspektrum verhelfen.

Folgende Charakterisierung soll der Leitfaden für die nachfolgend angeführten Eigenschaften von Komponenten sein:

Software-Komponenten kapseln eine Funktionalität, um diese wiederverwendbar zur Verfügung zu stellen. Eine Software-Komponente muss mit anderen Software-Komponenten interagieren. Die Gesamtfunktionalität eines Systems kann durch Kombination geeigneter interagierender Software-Komponenten dargestellt werden.

2.2.1 Ziel der Komponentenorientierung

Die eigentliche Motivation für einen komponentenorientierten Ansatz ist die **Wiederverwendung**. Wird eine Software für nur genau eine Spezialaufgabe entwickelt, die in anderen Einsatzszenarien nicht verwendet werden kann, so lohnt sich der Mehraufwand, der durch die Komponentenorientierung verursacht wird, wahrscheinlich nicht. Denn prinzipiell ist es deutlich aufwendiger, ein Stück Software als Komponente anzubieten, als die gleiche Funktionalität im eigenen System durch z. B. eine Menge von Objekten abzubilden.

Bei der verteilten Entwicklung eines größeren Systems werden häufig Teilbereiche durch verschiedene Teams bearbeitet. Möchte man dem gesamten Entwicklungsteam eine definierte Umgebung als Entwicklungsrahmen zur Verfügung stellen, innerhalb dessen Teilaufgaben abgegrenzt vom Rest des Systems implementiert werden können, so eignet sich der Komponentenansatz auch hierfür äußerst gut. Beispiele für diese Art der zum Teil massiven verteilten Entwicklung sind OpenSource-Projekte, die auf einer Plug-in-Architektur basieren, wie z. B. die Programme der Mozilla Foundation [1] oder der Instant Messenger Miranda [2].
In diesem Umfeld stellt die Begrenzung auf einen expliziten Anwendungskontext allerdings eine Einschränkung der Komponentenorientierung dar. Für einen allgemeinen Komponentenansatz wäre die Abwesenheit einer solchen Einschränkung wünschenswert.

Die Anforderungen an Software-Komponenten gliedern sich in drei Abschnitte: **Kernanforderungen, gefolgerte Eigenschaften und Zusatzanforderungen**. Während Kernanforderungen **zwingend erfüllt** sein müssen, um von einer Software-Komponente sprechen zu können, **ergeben** sich die gefolgerten Eigenschaften praktisch zwangsläufig aus den Kernanforderungen. Die Zusatzanforderungen hingegen sind sozusagen *nice-to-have*-Eigenschaften, die allerdings den tatsächlichen **Wert** und die **Nutzbarkeit** einer Komponententechnologie stark beeinflussen können.

2.2.2 Kernanforderungen

Die **Kernanforderungen** sind genau die geforderten Eigenschaften, die zwingend erfüllt sein müssen, um von einer Software-Komponente sprechen zu können.

[1] vgl. http://www.mozilla.org
[2] vgl. http://www.miranda-im.org

Diese Anforderungen sind:

- Kompositionsfähigkeit
- Interaktionsfähigkeit
- Auslieferungseinheit (Atomizität)
- Ausführbarkeit

Eine Komponente bildet zumeist eine klar definierte Teilaufgabe ab. Dabei sollte diese Aufgabe einen gewissen Mindestumfang besitzen. Um die gewünschte Funktionalität eines gesamten Systems zu erreichen, müssen demnach mehrere Komponenten zusammengeschaltet werden. Diese Möglichkeit wird als **Kompositionsfähigkeit** bezeichnet. Ein Stück Software ist kompositionsfähig, genau dann wenn es mit anderen Softwareeinheiten zusammengeschaltet werden kann, um dadurch eine größere Aufgabe zu bewältigen, als es alleine dazu im Stande ist. Anzustreben ist hier die möglichst automatisierte Zusammenarbeit von Komponenten, so dass jeweils zusätzlicher Aufwand zur reinen Verdrahtung von Komponenten entfallen kann.

Eng verbunden mit der Kompositionsfähigkeit ist die **Interaktionsfähigkeit**. Ein Stück Software ist interaktionsfähig, genau dann wenn es mit anderen Softwareeinheiten kommunizieren kann. Diese Kommunikation ist nötig, um Benachrichtigungen und auch Daten auszutauschen.

Als weiteres wird gefordert, dass eine Komponente eine **Auslieferungseinheit** darstellt, sozusagen **atomar** ist. Eine ausgelieferte Komponente ist demnach in sich abgeschlossen, stellt für die Zielumgebung quasi ein fertiges Produkt dar. Der Zielmarkt dieses Produktes muss allerdings nicht unbedingt ein öffentlicher Markt sein. Unternehmensinterne Komponentenmärkte sind genauso denkbar.

Wird eine Komponente ausgeliefert und installiert, so muss sie in einer **ausführbaren Form** vorliegen. Die Art der Ausführbarkeit hängt dabei vom jeweiligen Kontext ab. Sowohl binäre Ausführbarkeit ist denkbar, wie auch Ausführbarkeit durch Interpretation. Wichtig ist bei diesem Anforderungspunkt, dass möglichst keine menschliche Interaktion nötig ist, um eine Komponente auszuführen. Ausführbarkeit ist nicht gegeben, falls eine Komponente nur als Quelltext vertrieben wird, der zunächst durch einen Entwickler compiliert werden müsste, bevor die so genannte Komponente einsatzbereit wäre. Wird der Quelltext automatisch durch das Komponentensystem interpretiert, dann ist auch diese Ausliefungsvariante möglich.

2.2.3 Gefolgerte Eigenschaften

Aus den zuvor besprochenen Kernanforderungen ergeben sich zwangsläufig **weitere Eigenschaften**, die durch eine Software-Komponente verwirklicht werden sollten, um den Mindestanforderungen zu genügen. Ohne Erfüllung dieser gefolgerten Eigenschaften können auch die Kernanforderungen nicht vollständig erfüllt werden.

Diese Eigenschaften sind:

- Selbstbeschreibungsfähigkeit
- Strikte Kapselung, Blackbox-Wiederverwendung
- Konfektionierbarkeit
- Wissen über Komponentenabhängigkeiten

Möchte man mehrere Komponenten komponieren, um so eine gewünschte Funktionalität zu erreichen, so muss man zunächst entscheiden können, ob bzw. welche angebotene Komponente eine Teilaufgabe übernehmen kann. Dazu müssen die infrage kommenden Komponenten über sich selbst Auskunft geben können. Diese **Selbstbeschreibung** oder auch **Selbstdokumentation** sollte sowohl die veröffentlichten Schnittstellen, wie auch die bereitgestellte Funktionalität bzw. Semantik der Komponente umfassen. Diejenige Person, die eine komponentenbasierte Anwendung zusammenstellt, muss dadurch nicht über komponentenspezifisches Konstruktions- und Schnittstellenwissen verfügen, da die Komponente selbst darüber Auskunft geben kann. Entwicklungswerkzeuge können einen solchen Mechanismus ebenfalls verwenden, um die einfache Komposition beliebiger Komponenten (einer Art) zu unterstützen.

Die Implementierung einer Komponente sollten **strikt gekapselt** sein, die Funktionalität sollte also nur über die veröffentlichten Schnittstellen abrufbar sein. Das Innenleben einer Komponente sollte demnach vor den Anwendern vollständig verborgen werden, da die Implementierung auch von keinem Interesse für die Benutzung einer Komponente sein sollte.
Genau diesen Aspekt beschreibt der Begriff der **Blackbox-Wiederverwendung**. Einzig die angebotene Funktionalität und die Schnittstellen einer Komponente sind für deren Benutzung von Relevanz. Die Art der Realisierung muss und soll keine Rolle spielen.

Da eine Komponente eine in sich geschlossene Einheit darstellt, darf diese

Einheit auch nicht durch den Anwender aufgebrochen werden. Daraus folgt, dass eine Komponente so eingesetzt werden muss, wie sie sich darstellt. Anpassungen an bestimmte Gegebenheiten in einem expliziten Kontext dürfen demnach nicht durch Änderungen auf Code-Ebene vollzogen werden.
Damit eine Komponente in möglichst vielen Szenarien eingesetzt werden kann, ohne sie an die jeweiligen Gegebenheiten auf Code-Ebene anzupassen und neu zu übersetzen, muss sie **konfektionierbar** sein. Die Konfektionierung kann dann auch durch den Anwender einer Komponente durchgeführt werden, der Hersteller wird hierzu nicht benötigt. Auf diesem Weg sollte eine Komponente auch an Situationen angepasst werden können, die bei der Entwicklung so nicht vorhergesehen wurden. Der Aspekt der Komponente als Auslieferungseinheit wird auf diesem Weg weiter unterstützt.

Natürlich ist der Konfektionierbarkeit immer eine Grenze gesetzt, die im Fall einer Komponente durch die zu erfüllende Aufgabe vorgegeben wird. Die Komponente sollte so flexibel sein, dass angemessen viele variable Rahmenbedingungen vor bzw. während des Einsatzes so einstellbar sind, dass diese Aufgabe nicht nur auf einen Spezialfall beschränkt ist, sondern in wechselnden Kontexten durch die Komponente übernommen werden kann.

Manche Komponenten werden ihre Aufgabe unter Rückgriff auf Funktionalität anderer Komponenten erfüllen. Damit eine Komponente als Auslieferungseinheit betrachtet werden kann, müssen diese wissen, von welchen **Fremdkomponenten** sie **abhängig** ist. Ohne dieses Wissen wäre es nicht möglich festzustellen, ob eine installierte Komponente auch tatsächlich betriebsbereit ist. Eine Komponente, die erst durch einen Fehler zur Laufzeit eine fehlende Fremdkomponente bemerkt, kann nicht als in sich geschlossen betrachtet werden.
Es ist nicht nötig, dass eine Komponente immer als Ganzes ausgeliefert wird und alle benötigten Fremdkomponenten inkludiert. Wichtig ist hier das Wissen über Abhängigkeiten von anderen Komponenten.

2.2.4 Zusatzanforderungen

Zusätzlich zu den Kernanforderungen und gefolgerten Eigenschaften existieren Kriterien, die einer Software-Komponente erst die gewünschte Flexibilität und Unabhängigkeit geben, die nötig ist, um tatsächlich Nutzen aus dem Komponentenansatz zu ziehen. Zu diesen Eigenschaften zählen:

- Unabhängigkeit von Programmiersprachen
- Unabhängigkeit von Plattformen

- Komposition ohne Programmierkenntnisse
- Ortstransparenz, Verteiltheit
- Klassifizierbarkeit, Kategorisierbarkeit
- Lose Kopplung
- Existenz eines Verzeichnisdiensts
- Plug&Play-Fähigkeit

Die Etablierung eines Komponentenmarktes ist eine kritische Aufgabe. Ohne einen möglichst großen Anbieter- und Nachfragemarkt kann kein Komponentenansatz erfolgreich sein. Um diesem Problem zu begegnen, sollten Komponenten einer bestimmten Gattung möglichst **unabhängig von Programmiersprachen und Plattformen** sein.
Durch diese Unabhängigkeit wird neben einem möglichst großen Entwicklerpotential auch gewährleistet, dass in einem heterogenen Umfeld Komponenten einer Art eingesetzt werden können. Die Komponenteninfrastruktur liefert dadurch automatisch einen Integrationsansatz.
Die Entwickler einer Komponente sollen im Idealfall frei wählen können, welche Programmiersprache sie für die Implementierung einer Komponente verwenden. Analog sollten die Benutzer von Komponenten nicht eingeschränkt werden in der Wahl der eigenen Systemplattform. Daraus folgt, dass eine Komponente entweder auf möglichst vielen Plattformen lauffähig ist, oder zumindest von diesen aus aufgerufen werden kann.

Um die komponentenorientierte Anwendungsentwicklung oder besser *Anwendungskomposition* einem möglichst breiten Publikum nahe zu bringen, sollte das Zusammenstellen mehrerer Komponenten zu einem Gesamtsystem möglichst einfach sein. Dazu zählt, dass für die Komposition weitestgehend **keine Programmierkenntnisse** vorausgesetzt werden sollten. Eine Unterstützung durch Werkzeuge wäre hier von großem Vorteil.

Für den Nutzer einer Komponente sollte es unerheblich sein, wo sich diese tatsächlich befindet, d.h. auf welchem Rechner in einer verteilten Umgebung die Komponente ausgeführt wird. Dadurch ergeben sich vielfältige Möglichkeiten zur Verteilung von Last oder auch Anbindung ansonsten weit entfernter Funktionalität.
Erfüllt sich die Anforderung der **Ortstransparenz**, so sind Anwendungen auf Thin-Clients möglich, die auf die Rechen- und Datenkapazität größerer Server zurückgreifen, und sich selbst auf die Darstellung konzentrieren.

Auch müsste in einem Installationsszenario bei realisierter Ortstransparenz eine Komponente prinzipiell an nur einer Stelle installiert, aber von mehreren Clients benutzt werden können. Der Wartungsaufwand des Gesamtsystems z. B. beim Einspielen eines benötigten Updates wird somit verringert.
Bei dieser Art der Verwendung von Komponenten muss allerdings gesichert sein, dass neben der Konfektionierbarkeit der Komponente als solches, auch die Verwendung durch einen Klienten in einem gegebenen Kontext parametrisierbar ist. Ansonsten kann diese Verwendung der geforderten Konfektionierbarkeit und somit der Anpassbarkeit an bestimmte Rahmenbedingungen widersprechen.

Nach Etablierung eines Marktes für Komponenten könnten verschiedene Hersteller Komponenten anbieten, die eine identische Funktionalität bereitstellen. Der Anwendungskomponist weiß, welche Aufgabe er durch eine Komponente erfüllt haben möchte. Ist eine Komponente bzw. sind deren Dienste **klassifizierbar**, so kann über den bekannten Komponententyp nach Vertretern dieser Klasse gesucht werden, die die geforderte Funktionalität anbieten. Muss eine Software-Komponenten auf Dienste anderer Komponenten zurückgreifen, um die eigene Funktionalität bereit zu stellen oder zu erweitern, so reicht ihr dann ebenfalls das Wissen über die Klasse der benötigten Fremdkomponenten.

Nach Inbetriebnahme eines Systems kann es erforderlich oder wünschenswert sein, einen bestimmten Teil auszutauschen. Beispielsweise wurde ein neuer Server in Betrieb genommen, auf dem eine Komponente ausgeführt wird, die eine bisherige ersetzen soll. Durch **lose Kopplung** kann erreicht werden, dass zur Laufzeit des Systems eine Komponente durch eine andere ersetzt werden kann. Es würde genügen, die Kopplungsinformationen zu aktualisieren, damit die neue Komponente zum Einsatz kommt.
Dieses Vorgehen ist nicht in jedem Fall und nicht für jede Art von Komponente möglich. Dennoch sollte es nicht durch die Komponentenarchitektur ausgeschlossen werden.

In einem gegebenen Einsatzszenario für Komponenten sollte immer ein **Verzeichnisdienst** für Komponenten zur Verfügung stehen. An einen solchen Dienst müssen Anfragen nach Komponenten gerichtet werden können, die zur Bewältigung einer gegebenen Aufgabe herangezogen werden können. Eine Möglichkeit besteht dabei darin, nach den Klassifizierungsmerkmalen einer Komponenten zu suchen und genau die Komponenten geliefert zu bekommen, die diese Kriterien erfüllen.
Eine fortgeschrittene Möglichkeit wäre die Formulierung einer Anfrage, die

nach Komponenten sucht, die bestimmte Aufgaben erfüllen. Die Semantik der zu erfüllenden Aufgabe wäre hierbei Teil der Anfrage an den Verzeichnisdienst.

Der **Plug&Play-Gedanke** ist für Komponenten interessant. Nach Installation einer Komponente sollte die durch die Komponente angebotene Funktionalität der Zielumgebung direkt verfügbar sein. Plug&Play ist kein Aspekt der einzeln überprüfbar ist, vielmehr folgt es aus der Erfüllung anderer Eigenschaften, wie der Selbstbeschreibungfähigkeit einer Komponente.

2.3 Ausgesuchte Technologien im Komponentenumfeld

Bereits heute existieren Technologien, die von sich selbst oder von anderen als Komponententechnologien bezeichnet werden. In diesem Abschnitt werden ausgesuchte Vertreter dieser Technologien vorgestellt.
Schwerpunkt dieses Abschnittes ist die Untersuchung der Technologien in Hinblick auf die zuvor aufgestellten Eigenschaften von Software-Komponenten. Jeweils ausführliche Beschreibungen der Technologien sind an dieser Stelle aufgrund des Umfanges nicht möglich.
Betrachtet werden pro Technologie alle oben aufgeführten Eigenschaften, beginnend mit den gefolgerten Eigenschaften. Denn aus der Erfüllung der gefolgerten Eigenschaften lassen sich direkte Rückschlüsse auf den Grad der Erfüllung der Kernanforderungen ziehen. Die Zusatzanforderungen werden jeweils abschließend betrachtet.

2.3.1 COM

COM (*Common Object Model*, vgl. [7]) ist Microsofts Basistechnologie für die Interaktion von Komponenten auf allen Microsoft Plattformen. So oder vergleichbar wird COM von den meisten Quellen charakterisiert. Im Gegensatz zu CORBA (vgl. 2.3.2) wurde COM nicht als Standard konzipiert und definiert, sondern ist aus der kontinuierlichen Weiterentwicklung von Technologien entstanden, die Microsoft in den eigenen Betriebssystemen als Grundlage für die Kommunikation zwischen einzelnen Betriebssystem- oder z. B. Office-Elementen integriert. Mittlerweile ist COM bzw. DCOM (*Distributed COM*) Basistechnologie vieler verteilter Anwendungen, und auch auf anderen Plattformen verfügbar.

Szyperski bezeichnet COM genau wie CORBA als *wiring*-Technologie[12])].

Diese Einordnung bedeutet, dass COM Möglichkeiten zur Verschaltung und Kommunikation zwischen Softwarebausteinen bietet. Diese Bausteine werden in der COM-Terminologie Komponenten genannt. Was genau allerdings eine Komponente ist, wird durch COM bewusst nicht definiert. Die einzige Anforderung an eine COM-Komponente ist das Veröffentlichen und Bereitstellen eines COM-Interfaces. Dazu muss ein Interface bereitgestellt werden, dass von dem COM-Interface *IUnknown* abgeleitet wird. Einmal eingeführt, ist ein COM-Interface unveränderlich. Auf diese Weise begegnet Microsoft dem Problem der Versionsverwaltung. Nämlich durch Verbieten unterschiedlicher Versionen eines Interfaces. Eine COM-Komponente darf allerdings mehrere Interfaces anbieten, wodurch neue Funktionalität der Öffentlichkeit zugänglich gemacht werden kann.

Eine ausführliche Beschreibung von COM findet sich u. a. in [12]. Die folgenden Abschnitte beschäftigen sich mit der Überprüfung der zuvor aufgestellten Eigenschaften von Komponenten bezogen auf COM und COM-Komponenten.

Selbstbeschreibungsfähigkeit

Über die COM-Registrierung können zu COM-Komponenten Metainformationen abgerufen werden. Diese Metainformationen listen zunächst die angebotenen COM-Interfaces auf und die Kategoriezugehörigkeit (vgl. *Klassifizierbarkeit, Kategorisierbarkeit*). Zu jedem Interface können dann die einzelnen Methoden samt Parameteranzahl und -typ erfragt werden. Über die Semantik enthalten die Metainformationen keine Aussagen, da diese Art von Aussage durch COM auch gar nicht vorgesehen ist.

Strikte Kapselung, Blackbox-Wiederverwendung

COM-Komponenten sind strikt gekapselt. Sie sind nur über die veröffentlichten Schnittstellen benutzbar, über die Implementierung werden keine Informationen benötigt.

Konfektionierbarkeit

COM enthält keinen Mechanismus zur Konfektionierung von COM-Komponenten. Einzelne COM-Komponenten können natürlich konfektionierbar sein, eine Eigenschaft von COM ist das allerdings nicht.

Wissen über Komponentenabhängigkeiten

COM liefert keine Mittel um sicherzustellen, dass eine ausgelieferte COM-Komponente auch tatsächlich ihre Aufgabe alleine erfüllt. Abhängigkeiten von anderen Komponenten sind nicht explizit.

Kompositionsfähigkeit

COM-Komponenten sind mit Einschränkungen kompositionsfähig. Zur Zusammenschaltung von COM-Komponenten muss eine Anwendung geschrieben werden, die die benötigten COM-Komponenten kennt und diese explizit anfordert, da semantische Informationen über die Komponenten nicht vorliegen. Die Verdrahtung und der korrekte Einsatz mehrerer COM-Komponenten ist dann Aufgabe eines Entwicklers.

Auslieferungseinheit (Atomizität)

Die Installation einer COM-Komponenten umfasst das Installieren der ausführbaren Teile und das Registrieren in der Windows Registry. In der Regel sollte eine COM-Komponente danach zur Verfügung stehen. Werden durch die COM-Komponente allerdings weitere COM-Komponenten vorausgesetzt, so wird deren Vorhandensein prinzipiell erst zur Laufzeit überprüft.

Ausführbarkeit

Eine COM-Komponente liegt immer in binärer Form vor. Einmal installiert und registriert ist sie ausführbar.

Unabhängigkeit von Programmiersprachen

Die Entwicklung von COM-Komponenten ist prinzipiell unabhängig von der verwendeten Programmiersprache. Einzig das binäre Ausgabeformat ist Voraussetzung, ansonsten können beliebige Sprachen und Compiler verwendet werden.

Unabhängigkeit von Plattformen

COM existierte zunächst nur auf Betriebssystemen von Microsoft. Aufgrund des Erfolges von COM im Windows-Umfeld existieren heute aber auch Umsetzungen für andere Plattformen, die von Drittherstellern angeboten werden. Damit ist die Situation vergleichbar mit der von CORBA. In beiden Fällen muss auf der Zielplattform die benötigte Infrastruktur gegeben sein.

Komposition ohne Programmierkenntnisse

Eine Komposition ohne Programmierkenntnisse wird durch COM nicht unterstützt. Denkbar sind hier zusätzliche Tools, die den Entwickler bei dieser Aufgabe unterstützen. Teilweise sind solche Möglichkeiten in diverse Entwicklungsumgebungen wie z. B. Visual Studio von Microsoft integriert.

Ortstransparenz, Verteiltheit

Mit der Einführung von DCOM (*Distributed COM*) ist COM nicht mehr auf einen lokalen Rechner beschränkt. Vollkommen ortstransparent ist DCOM allerdings nicht, der anbietende Server muss bekannt sein. Ist eine DCOM-Komponente allerdings einmal instanziiert, so spielt es für die verwendenden Elemente keine Rolle mehr, wo die Software tatsächlich läuft.

Klassifizierbarkeit, Kategorisierbarkeit

COM-Komponenten sind wie bereits angesprochen kategorisierbar. Dabei stellt eine Kategorie im Prinzip nur Anforderungen an die zu erwartenden Interfaces, die durch die COM-Komponenten dieser Kategorie veröffentlicht werden. Klassifizierbarkeit im engeren Sinne existiert nicht.
Auf diese Weise können zumindest Anfragen nach komplexen COM-Komponenten effizienter durchgeführt werden. Da die Kategorien genau wie einzelne COM-Komponenten nicht in ihrer Semantik definiert sind, nützt auch diese Eigenschaft nicht zur vollständigen Beschreibung von COM-Komponenten.

Lose Kopplung

Da die Kopplung immer über COM-Interfaces geschieht, besteht nur eine lose Kopplung. Die binären COM-Komponenten können ausgetauscht werden, ohne aufrufende Elemente zu beeinträchtigen. Auch muss eine Verbindung zu einer COM-Komponente erst bei Bedarf hergestellt werden, statische Verknüpfungen zur Designzeit sind nicht notwendig.

Existenz eines Verzeichnisdiensts

Bereits erwähnt worden ist, dass die Windows Registry den zentralen Verzeichnisdienst für COM darstellt. Bei den Umsetzungen für andere Betriebssysteme durch andere Hersteller wird hier die Windows Registry durch eigene Konzepte ersetzt.

Plug&Play-Fähigkeit

Nach Installation und Registrierung stehen COM-Komponenten direkt zur Verfügung.

Fazit

COM wird häufig als Komponententechnologie bezeichnet, tatsächlich ist es aber weniger. Die Einschätzung Szyperski's von COM als wiring-Technologie ist eher zutreffend. COM selbst stellt eigentlich nur Mittel zur Kapselung und Kommunikation zwischen Softwarebausteinen auf einem lokalen Rechner. DCOM geht einen Schritt weiter und macht die Verfügbarkeit von COM-Komponenten für die aufrufende Software weitgehend ortstransparent.
Allerdings fehlt dem gesamten Ansatz ein allgemeines Bild der eigentlichen Komponenten. Es wird sich einzig und alleine auf die technischen Aspekte der Kommunikation konzentriert, Aussagen über Semantik oder Funktionalität der auf diesem Wege entwickelten Softwarebausteine werden nicht getroffen. COM könnte Grundlage für ein Komponentenmodell sein, falls an die COM-Komponenten weitere Anforderungen gestellt und sie mit weiteren Möglichkeiten vor allem in der Selbstbeschreibungsfähigkeit ausgestattet würden. Ohne das ist COM hauptsächlich ein (wenn auch mächtiges) Infrastrukturmittel.

2.3.2 CORBA

CORBA (*Common Object Request Broker Architecture*, vgl. [11]) ist eine offene herstellerunabhängige Architektur und Infrastruktur der OMG (*Object Management Group*), die mit dem Ziel der Standardisierung der Entwicklung von verteilten objektorientierten Anwendungen aufgestellt worden ist.

Die OMG stellt CORBA als Middleware dar, erhebt also keinen direkten Anspruch darauf, dass CORBA eine Komponententechnologie ist. Trotzdem existiert in einem Großteil der Komponenten-Literatur mindestens ein Kapitel zu CORBA, denn häufig wird hier CORBA als Kommunikationsmittel zwischen Komponenten angeführt. In diesem Sinne bezeichnet auch Szyperski CORBA als *wiring*-Technologie [12].

Im Mittelpunkt des CORBA-Ansatzes steht der ORB (Object Request Broker). Diese zentrale Vermittlungsinstanz regelt die Kommunikation zwischen anbietenden und nachfragenden Programmen. Jeder Objekttyp, der über einen ORB zur Verfügung gestellt werden soll, muss seine Schnittstelle über

die OMG IDL (Interface Definition Language) definieren. Liegt eine Interface-Beschreibung in IDL vor, so kann diese compiliert und in einem ORB registriert werden. Ab diesem Zeitpunkt ist über den ORB der neue Objekttyp bekannt, und ein Anwendungsprogramm könnte ein Objekt dieses Typs anfragen.
Um ein Anwendungs- oder ein Serverprogramm zu erzeugen, das CORBA-Objektzugriffe anfragt bzw. anbietet, benutzt man sogenannte *Stubs* bzw. *Skeletons*, die durch einen IDL-Compiler erzeugt werden können. Ein Stub kann instanziiert werden und stellt sich für die aufrufende Anwendung wie ein *normales* Objekt dar. Tatsächlich leitet es allerdings alle Aufrufe durch den ORB zu dem Zielobjekt durch. Ein Skeleton dagegen empfängt eingehende Aufrufe und ruft seinerseits die entsprechende Methode des Zielobjektes auf.

Selbstbeschreibungsfähigkeit

Die Schnittstelle eines CORBA-Objektes wird per OMG IDL (*Interface Definition Language*) beschrieben. Diese Beschreibungssprache ist unabhängig von Implementierungssprachen und umfasst alle benötigten syntaktischen Elemente der Objektschnittstelle.
Allerdings ist genau das auch die Einschränkung, denn über die Syntax geht diese Beschreibung nicht hinaus. Semantische Elemente finden sich in der Beschreibung eines Objektes und der angebotenen Funktionalität nicht wieder.

Strikte Kapselung, Blackbox-Wiederverwendung

CORBA-Objekte sind strikt gekapselt, mit der tatsächlichen Implementierung kommen Benutzer dieser Objekte nicht in Berührung.

Konfektionierbarkeit

CORBA selbst bietet keine integrierte Möglichkeit der Konfektionierung der zur Verfügung stehenden Softwarebausteine. Jeder dieser Bausteine kann durchaus mit eigenen Mitteln konfigurierbar sein, Teil von CORBA ist diese Funktionalität allerdings nicht.

Wissen über Komponentenabhängigkeiten

Ein CORBA-Objekt kapselt eine Aufgabe. Ob es diese Aufgabe selbständig erledigen kann, wird und kann durch CORBA nicht überprüft werden, da das Wissen über benötigte Fremdkomponenten nicht explizit ist. Es kann

also CORBA-Objekte geben, die ihre Aufgabe nicht erledigen können, weil benötigte Teile nicht vorhanden sind.

Kompositionsfähigkeit

CORBA-Objekte können zu größeren Einheiten zusammengesetzt werden. Dazu ist allerdings jeweils neuer Code zu entwickeln, der diese Komposition darstellt. CORBA bietet keinen direkten Mechanismus, der ein solches Verschalten unterstützt.

Interaktionsfähigkeit

Der Nachrichtenaustausch geschieht im Rahmen von CORBA seit der Version 2.0 über das General Inter-ORB Protocol (kurz *GIOP*). Dabei handelt es sich um ein abstraktes Protokoll, über das u. a. das binäre Format für IDL-Datentypen und die verwendeten Nachrichtenformate festlegt werden.
Eine konkrete Ausprägung des Protokolls ist das Internet Inter-ORB Protocol (kurz *IIOP*). Innerhalb dessen Spezifikation wird festgelegt, wie GIOP auf TCP/IP aufgesetzt wird.
Mit diesen Mitteln wird sichergestellt, dass mehrere ORBs untereinander kommunizieren können. Ebenfalls gesichert wird auf diesem Weg der Nachrichtenaustausch zwischen beliebigen CORBA-Objekten. Allerdings ist hiermit keine automatische Interaktion gegeben, die durch eine Komponente bei Bedarf angestoßen und durchgeführt wird. Vielmehr besitzen Entwickler die Möglichkeit, ihre CORBA-Objekte so zu gestalten, dass sie mit anderen CORBA-Objekten kommunizieren können, ohne sich um die technische Realisierung der Kommunikation zu sorgen.

Auslieferungseinheit (Atomizität)

Technisch betrachtet sind CORBA-Objekte in sich geschlossen. Es kann aber implizite Abhängigkeiten zwischen CORBA-Objekten geben, die durch das System nicht auf Erfüllung überprüft werden können. Dadurch kann es zur Laufzeit eines CORBA-basierten Systems dazu kommen, dass einige CORBA-Objekte nicht vollständig einsatzbereit sind, da benötigte Fremdkomponenten nicht verfügbar sind.

Ausführbarkeit

CORBA-Objekte sind per Definition ausführbar.

Unabhängigkeit von Programmiersprachen

Die OMG IDL ist sprachunabhängig, es existieren zahlreiche Umsetzungen auf die meisten verbreiteten Sprachen, so z. B. C, C++, Java, COBOL, Smalltalk oder Ada. Somit ist die IDL das hauptsächliche Mittel, um ganz allgemein eine Sprachunabhängigkeit für CORBA zu erreichen.

Unabhängigkeit von Plattformen

ORB-Implementationen (*Object Request Broker*) existieren für die unterschiedlichsten Plattformen. Prinzipiell ist die Entwicklung für beliebige Plattformen möglich. CORBA ist demnach potentiell plattformunabhängig, solange ein ORB für die jeweilige Plattform existiert.

Komposition ohne Programmierkenntnisse

CORBA-Objekte können mit direkten CORBA-Möglichkeiten ohne Programmierkenntnisse nicht verschaltet werden. Das Fehlen einer solchen Möglichkeit ist durchaus verständlich, da die Motivation für CORBA in der Verteiltheit und nicht der Komponierbarkeit liegt.

Ortstransparenz, Verteiltheit

Durch Verwendung des ORB (*Object Request Broker*) ist Ortstransparenz gewährleistet. Ein Client muss nicht wissen, wo sich ein angefordertes Objekt befindet, dieses Wissen muss nur der vermittelnde ORB aufweisen.
Verteiltheit ist ein Kernziel von CORBA und wird dementsprechend auch erreicht.

Klassifizierbarkeit, Kategorisierbarkeit

Weder Klassifizierungs- noch Kategorisierungsmittel sind in CORBA vorgesehen.

Lose Kopplung

Durch die Verwendung eines Brokers ist die Kopplung von Server und Client immer lose.

Existenz eines Verzeichnisdiensts

Der Verzeichnisdienst wird durch den ORB bereitgestellt.

Plug&Play-Fähigkeit

Ist ein CORBA-Objekt erst einmal einem ORB bekannt, so kann es direkt von Klienten benutzt werden. Diese Klienten müssen allerdings von der Existenz des Objektes wissen, und auch die Funktionalität dieses Objektes explizit anfordern. Plug&Play im Sinne von automatischer Zusammenarbeit ohne vorherige Kenntnis ist mit CORBA so nicht zu verwirklichen.

Fazit

Ziel des CORBA-Ansatzes ist die Realisierung verteilter Anwendungen, die unabhängig von Programmiersprachen und Plattformen lauffähig sind (vlg. [12], S. 178). Die OMG sieht CORBA also selbst nicht direkt als Komponententechnologie, sondern vielmehr als Middleware. Trotzdem wird CORBA in gängigen Publikationen fortlaufend als Komponententechnologie bezeichnet. Diese Maßnahme ist allerdings nicht wirklich korrekt, da nur mit CORBA alleine kein Komponentensystem darzustellen ist.
Allerdings kann CORBA als Grundlage eines Komponentensystems herangezogen werden. Es bietet viele Funktionen, die im Rahmen eines solchen Systems benötigt werden. Näher betrachtet werden kann in diesem Kontext das CORBA Komponentenmodell, das durch die OMG verabschiedet worden ist.

2.3.3 Eclipse PlugIns

Ursprünglich Ziel der Entwicklung von Eclipse war die Schaffung einer Entwicklungsumgebung für Java. Im Laufe der Zeit ist daraus aber mehr geworden, denn neben der Java-IDE-Funktionalität, die Eclipse dem Anwender bietet, ist es außerdem aufgrund seiner Plugin-Architektur eine Ablaufplattform für Java-Applikationen.
Die eigentliche Motivation bei der Entwicklung war wie bereits erwähnt die Erstellung einer IDE für Java-Entwickler. Das Grundkonzept von Eclipse baut dabei auf Plugins auf, um so die Entwicklungsplattform möglichst erweiterunsfähig zu halten. Der Kern des ganzen Systems ist daher nur zuständig für die Ausführung von Plugins, alle weitere Funktionalität der Eclipse-IDE wird bereits über spezielle Plugins zur Verfügung gestellt.
Benutzt man diesen Kern ohne die Entwicklungsplugins, so hat man eine Ausführungsplattform für beliebige Plugins vorliegen. Diese Plattform und die Plugin-Architektur können problemlos als Basis für eigene Anwendungen dienen.
Im Folgenden soll daher betrachtet werden, inwiefern es sich bei den Eclipse-

Plugins um Komponenten und dem Eclipse-Kern um ein Komponenten Framework im Sinne dieser Arbeit handelt.

Selbstbeschreibungsfähigkeit

Eclipse-Plug-ins werden durch Java-Klassen realisiert. Basis für alle Eclipse-Plug-ins ist die abstrakte Klasse `Plugin`, von der ein neues Plug-in spezialisiert werden muss. Dadurch kann bereits per Java-Reflection auf alle öffentlichen Informationen der Klasse zugegriffen werden.
Zu jedem Eclipse-Plug-in gehört eine Manifest-Datei, in der Angaben über benötigte Plug-ins und angebotene Erweiterungspunkte (*extension points*) festgehalten sind. Über Erweiterungspunkte können sich Plug-ins in die Funktionalität eines anderen Plug-ins einklinken. In der Manifest-Datei wird zu jedem Erweiterungspunkt ein *Extension Point Schema* definiert. Diese Schemata enthalten Informationen über die Attribute der zugehörigen Erweiterungspunkte, beschreiben also die syntaktische Verwendung eines Erweiterungspunktes.
Beschreibungen zur Wirkungsweise der Erweiterungspunkte sind in den Manifest-Dateien oder Erweiterungsschemata nicht enthalten. Zu bereits existenten Eclipse-Erweiterungsschemata liegen natürlichsprachliche Beschreibungen vor, diese bestehen allerdings losgelöst von den eigentlichen Plug-ins. Demzufolge bietet Eclipse keine Unterstützung zur semantischen Beschreibung von Plug-ins.

Strikte Kapselung, Blackbox-Wiederverwendung

Eclipse-Plug-ins sind prinzipiell strikt gekapselt. Zugriff auf ein Plug-in sollte nur über die Erweiterungspunkte gegeben sein. Da es sich bei einem Plug-in aber um eine Java-Klasse handelt, liegt die Kapselung letztendlich in der Hand des Plug-in-Entwicklers.
Durch das Konzept der Erweiterungspunkte kann man aber von einer Blackbox-Wiederverwendung sprechen.

Konfektionierbarkeit

Eclipse bietet über die Manifest-Dateien ein integriertes Konzept für die Konfektionierung von Plug-ins an. Zunächst wird dieses Konzept im Rahmen von Eclipse selbst verwendet, da über die hier getroffenen Einstellungen Eclipse-Plug-ins in die Eclipse-Arbeitsumgebung integriert werden. Zusätzlich kann dieser Mechanismus aber auch für Plug-in-spezifische Konfektionierungen verwendet werden.

Wissen über Komponentenabhängigkeiten

Jedes Eclipse-Plug-in beschreibt in der Datei `plugin.xml` welche Erweiterungspunkte es bereitstellt, und in welche es sich einklinkt. Somit ist das Wissen über Abhängigkeiten zwischen Eclipse-Plug-ins vollkommen explizit. Neben diesen expliziten Abhängigkeiten von anderen Plug-ins kann es auch noch implizite Voraussetzungen durch Plug-ins geben, die nicht durch Eclipse überprüft werden können. Hier liegt es im Verantwortungsbereich des Plug-in-Entwicklers, diese Elemente bei Auslieferung ebenfalls mitzuliefern, um so die Einheit des Plug-ins zu gewährleisten.

Kompositionsfähigkeit

Eclipse-Plug-ins sind einfach komponierbar durch Verwendung des Konzeptes der Erweiterungspunkte.

Interaktionsfähigkeit

Der Informationsaustausch verläuft unter Eclipse-Plug-ins eventbasiert, abgesehen von der expliziten Verdrahtung über Erweiterungspunkte. Dabei kann sich ein Plug-in entweder direkt als *Listener* bei einem anderen Plug-in anmelden oder aber auf die allgemeine Ereignisverwaltung von Eclipse zurückgreifen. Auf diesem Wege können Plug-ins auch auf Ereignisse reagieren, die durch andere Plug-ins ausgelöst werden, über deren Existenz das reagierende Plug-in nicht informiert ist.

Auslieferungseinheit (Atomizität)

Eclipse-Plug-ins werden in Form von binären Java-Dateien (Java-Archiven, *.jar*) mitsamt der Manifest-Datei und weiteren Ressourcen (Bildern, Hilfeseiten) ausgeliefert. Eclipse-Plug-ins müssen nicht für sich alleine stehend eine Aufgabe erfüllen können. In der Manifest-Datei wird explizit auf vorausgesetzte weitere Plug-ins verwiesen, die in der Eclipse-Umgebung existieren müssen, damit das Plug-in lauffähig ist. Diese Abhängigkeiten sind durch Eclipse überprüfbar.

Ausführbarkeit

Einmal installiert ist ein Eclipse-Plug-in durch den Eclipse-Kern ausführbar.

Unabhängigkeit von Programmiersprachen

Eclipse ist ein reines Java-Projekt. Plug-ins können nur in Java implementiert werden und müssen sogar von einer fixen abstrakten Klasse spezialisiert werden. Unabhängigkeit von Programmiersprachen ist also nicht gegeben.

Unabhängigkeit von Plattformen

Durch den Einsatz von Java ist allerdings eine weitgehende Plattformunabhängigkeit erreicht, da Java selbst plattformunabhängig ist. Voraussetzung ist die Existenz einer Java-Virtual Machine auf der gewünschten Zielplattform.

Komposition ohne Programmierkenntnisse

Neue Plug-ins können einem Eclipse-System einfach hinzugefügt werden, ohne dass hierfür Programmierkenntnisse von Nöten wären. Durch das Konzept der Erweiterungspunkte binden sich neu installierte Plug-ins in das System ein. Durch die Auswahl aller benötigten Plug-ins lässt sich so ein Gesamtsystem ohne zusätzlichen Programmieraufwand flexibel zusammenstellen.

Ortstransparenz, Verteiltheit

Eclipse ist konzipiert für den Einsatz auf einem lokalen Rechner. Verteiltheit spielt bis heute keine Rolle in den Entwicklungsperspektiven von Eclipse. Die Frage nach Ortstransparenz stellt sich somit für Eclipse nicht.

Klassifizierbarkeit, Kategorisierbarkeit

Als Klassifizierungsmerkmal können im Eclipse-Umfeld die Erweiterungspunkte der Plug-ins betrachtet werden. Zu beachten gilt hierbei allerdings, dass es sich bei den Erweiterungspunkten prinzipiell nur um die Definition von Schnittstellen handelt.

Lose Kopplung

Durch den ereignisbasierten Informationsaustausch wird eine relativ lose Kopplung der einzelnen Plug-ins erreicht. Auch das Konzept der Erweiterungspunkte trägt hierzu noch bei, da die Kopplung nur über die Schemainformationen der einzelnen Plug-ins vorgenommen wird. Auch wird die Kopplung nur bei Existenz der Plug-ins vorgenommen, und diese Existenz wird erst zur Laufzeit überprüft.

Existenz eines Verzeichnisdiensts

Der Eclipse-Kern ist selbst eine Art Verzeichnisdienst. Alle Plug-ins müssen sich hier registrieren und können vorhandene Plug-ins abfragen.

Plug&Play-Fähigkeit

Wie aus den vorherigen Beschreibungen hervor geht, sind Eclipse-Plug-ins einfach zu installieren und sofort einsatzbereit. Plug&Play-Fähigkeit ist also gegeben.

Fazit

Der Plug-in-Ansatz von Eclipse ist äußerst interessant. Aus der früher reinen Java-Entwicklungsumgebung ist im Laufe der Zeit eine plug-in-basierte Ablaufumgebung für beliebige Eigenentwicklungen entstanden.
Die Eclipse-Plug-ins weisen viele Eigenschaften von Komponenten auf. Jedoch fehlen semantische Aspekte der Funktionsbeschreibung von Plug-ins. Da Eclipse für lokale Applikationen gedacht ist, bleibt ebenfalls der Aspekt der Verteiltheit unberücksichtigt.

2.3.4 Enterprise JavaBeans

Enterprise JavaBeans (kurz: EJB) wurden wie die später vorgestellten JavaBeans von Sun Microsystems vorgestellt. Außer dem Namen haben die beiden Ansätze allerdings wenig gemeinsam. EJB ist ein Komponentenmodell für die Entwicklung serverseitiger Java-Komponenten, die demnach über keinerlei GUI-Funktionalität verfügen (vgl. [5]). Mit EJB-Komponenten soll die Geschäftslogik abgebildet werden, für die Visualisierung muss man auf andere Mittel zurückgreifen.

Selbstbeschreibungsfähigkeit

EJB-Komponenten besitzen öffentliche Schnittstellenbeschreibungen, die die Eigenschaften und Methoden auflisten. Über diese syntaktischen Informationen hinaus gehend existieren keine integrierten Beschreibungsmerkmale.

Strikte Kapselung, Blackbox-Wiederverwendung

Die Implementierung von EJB-Komponenten ist vor dem Anwender verborgen. Ausgeliefert werden die EJB-Komponenten im Bytecode-Format, Zugriff erhält man über definierte Interfaces.

Konfektionierbarkeit

EJB-basierte Anwendungen lassen sich durch deklarative Anweisungen steuern und konfigurieren, ohne dass direkte Eingriffe in die EJB-Komponenten erforderlich sind (vgl. [5]).

Wissen über Komponentenabhängigkeiten

EJB-Komponenten besitzen kein explizites Wissen über Abhängigkeiten zu weiteren EJB-Komponenten.

Kompositionsfähigkeit

Die Kompositionsfähigkeit von EJB entspricht der von COM oder CORBA. Es wird ein Entwickler benötigt, der vorhandene EJB-Komponenten durch zusätzlichen Code miteinander verbindet, um so eine größere Funktionalität zu erreichen.

Interaktionsfähigkeit

Zusätzlich zur expliziten Kommunikation zwischen zwei EJB-Komponenten, die durch den Entwickler durch Aufrufe fest vorgesehen ist, existiert seit der Version 2.0 noch die Möglichkeit der asynchronen Kommunikation per JMS (*Java Message Service*). Dabei steht sowohl eine *publish-subscribe* wie auch eine *point-to-point* Variante dieser Kommunikationsform zur Verfügung (vgl. [4]).
Jede EJB-Komponente kann zum Sender oder Empfänger einer asynchronen Nachricht werden, wodurch sich die Kommunikationsmächtigkeit im Gegensatz zu EJB 1.0 erheblich vergrößert hat.

Auslieferungseinheit (Atomizität)

EJB enthält keinen Mechanismus, der sicherstellt, dass eine Komponente als Einheit ausgeliefert wird. Es kann nicht festgestellt werden, ob alle benötigten Fremdkomponenten vorhanden sind, auch kann nicht getestet werden, ob alle Hilfsklassen im Archiv enthalten sind. Nicht erfüllte Abhängigkeiten resultieren in Laufzeitfehlern.

Ausführbarkeit

Ausgeliefert werden EJB-Komponenten als JAR-Dateien, die in diesem Zusammenhang EJB-JAR-Dateien genannt werden. Diese Archivdateien enthalten alle Bytecode-Dateien der EJBs, alle Interfaces, wie auch zusätzliche

Hilfsklassen. Ein EJB-Applikationsserver kann diese Archiv-Dateien lesen und die enthaltenen EJB-Komponenten installieren. Nach dieser Installation sind die neuen Komponenten ausführbar.

Unabhängigkeit von Programmiersprachen

EJB-Komponenten werden in Java entwickelt, es existiert also eine vollkommene Abhängigkeit von der Programmiersprache bezogen auf die Entwicklung von EJB-Komponenten.
Allerdings können Klienten, die EJB-Komponenten benutzen, in einer beliebigen Programmiersprache entwickelt werden. Hier muss nur das Kommunikationsprotokoll von EJB eingehalten werden.

Unabhängigkeit von Plattformen

Für die Ausführung von EJB-Komponenten wird ein EJB-Laufzeitsystem vorausgesetzt, häufig *Applikationsserver* genannt. Prinzipiell sollte eine EJB-Komponente in einem EJB-Applikationsserver eines beliebigen Herstellers lauffähig sein. In der Praxis haben viele Hersteller allerdings proprietäre Erweiterungen ihrer Systeme, so dass diese Art der Plattformunabhängigkeit nicht immer gegeben ist (vgl. [5]).

Komposition ohne Programmierkenntnisse

EJB enthält kein Konzept zur Komposition von EJB-Komponenten ohne Anfertigung zusätzlichen Programmcodes.

Ortstransparenz, Verteiltheit

EJB-Komponenten werden durch einen EJB-Applikationsserver ausgeführt. Diese Server können an beliebigen Stellen im Inter-/Intranet platziert sein. Auch kann eine Anwendung prinzipiell mehrere EJB-Applikationsserver nutzen. Verteiltheit ist also zentrales Element der EJB-Architektur.
EJB-Applikationsserver verfügen zusätzlich über einen Naming-Service, um so sicherzustellen, dass aufrufende Komponenten keine Informationen über den Ort von aufzurufenden Komponenten besitzen müssen. Allerdings muss sich seit der Version 2.0 der Entwickler bei Verwendung einer EJB-Komponente entscheiden, ob es sich um eine lokale Nutzung, oder aber einen Remote-Zugriff handelt. Davon hängt ab, welches Interface (Locale oder Remote) der EJB-Komponente genutzt wird.
Ortstransparenz ist also eigentlich gegeben, wird durch explizite Bindung an ein lokales oder entferntes Interface aber wieder relativiert.

Klassifizierbarkeit, Kategorisierbarkeit

EJB bietet kein integriertes Konzept zur Klassifizierung von Komponenten an.

Lose Kopplung

Durch den Einsatz des EJB-Naming-Service sind EJB-Komponenten lose gekoppelt. Erst bei Bedarf wird die Verbindung zu einer EJB-Komponente hergestellt, einzig der Name der Komponente muss bekannt sein.

Verzeichnisdienst

Die EJB-Applikationsserver nehmen auch die Rolle eines Verzeichnisdienstes wahr. Allerdings bieten sie keine Möglichkeit, EJB-Komponenten mit bestimmten Eigenschaften zu suchen. Da EJB-Komponenten außer den syntaktischen Schnittstellen aber auch keine Informationen über sich selbst veröffentlichen, sind weitergehende Anfragen auch überhaupt nicht möglich.

Plug&Play-Fähigkeit

Nach Installation stehen EJB-Komponenten zur Verfügung. Echtes Plug& Play ist allerdings nicht gegeben, da neue Komponenten erst durch einen Anwendungsentwickler und neuen Quellcode in einer Anwendung oder komplexeren Komponente zum Einsatz kommen.

Fazit

Enterprise JavaBeans bietet im Gegensatz zu COM oder CORBA neben der reinen Kommunikation vor allem zusätzliche Dienste an. So kann sich der Anwendungsentwickler auf die Entwicklung der Geschäftslogik konzentrieren, während Standardaufgaben wie Transaktionsverwaltung oder sicherheitsrelevante Aufgaben durch den Applikationsserver übernommen werden.
Trotz dieser Vorteile ist auch EJB kein echtes Komponentensystem, wenn *Komponierbarkeit* das Hauptmerkmal eines solchen Systems darstellen soll. Die Kompositionsfähigkeit von EJB-Komponenten ist nämlich nicht Bestandteil des EJB-Konzeptes. Auch in diesem Ansatz ist die Arbeit eines Programmierers notwendig, falls mehrere EJB-Komponenten zusammenarbeiten sollen. Wie bei Einsatz von COM oder CORBA muss ein Entwickler den nötigen Code schreiben, um das gewünschte Verhalten zu erreichen.

2.3.5 JavaBeans

JavaBeans-Komponenten (auch *Beans* genannt) sind wieder verwendbare Software-Komponenten, die in einer graphischen Entwicklungsumgebung visuell manipuliert werden können. Beans können kombiniert werden, um so traditionelle Anwendungen oder auch Applets zu erstellen[3].
Hauptaugenmerk bei der Erstellung und Verwendung von JavaBeans liegt auf der Bereitstellung häufig verwendeter Konstrukte, die es dem Anwendungsentwickler erlauben sollen, sich auf die Erstellung der eigentlichen Anwendung zu konzentrieren. Durch JavaBeans werden ihm dazu die Mittel an die Hand gegeben, die er zu Erschaffung der graphischen Oberfläche, aber auch der intern verwendeten Datenstrukturen benötigt.

Selbstbeschreibungsfähigkeit

Ein JavaBean kann über seine öffentlichen Schnittstellen Auskunft über die eigenen Methoden, Eigenschaften und Ereignisse geben. Diese Informationen sind rein syntaktischer Natur, eine Beschreibung der Funktionalität bzw. Semantik kann nicht geliefert werden.
Einem JavaBean-Entwickler steht es offen, zusätzlich zur eigentlichen JavaBean eine BeanInfo-Klasse anzulegen, die weitere Information zur zugehörigen JavaBean enthält. Verwendet man dieses Konzept, so kann neben den bereits erwähnten Schnittstelleninformationen eine textuelle Beschreibung, ein Icon und einige weitere für die IDE interessante Eigenschaften festgelegt werden.

Strikte Kapselung, Blackbox-Wiederverwendung

Da es sich bei einem JavaBean um eine Java-Klasse handelt, kann man von einer strikten Kapselung ausgehen.

Konfektionierbarkeit

Die Konfektionierbarkeit einer JavaBean ist wichtiger Gegenstand des Gesamtkonzeptes. Ein JavaBean soll über eine graphische IDE visuell manipulierbar sein. Um diese Art der Konfektionierung zu unterstützen, muss sich der JavaBean-Entwickler an Konventionen bei der Namensgebung von Methoden halten. Werden diese Konventionen eingehalten, so sind die öffentlichen Eigenschaften einer JavaBean im Folgenden innerhalb der IDE manipulierbar.

[3]vgl. http://java.sun.com/products/javabeans/

Wissen über Komponentenabhängigkeiten

Eine JavaBean kann eigentlich keine direkte Auskunft darüber geben, welche Fremdkomponenten von ihr benötigt werden, um die eigene Funktionalität zu sichern. Sind die fremden JavaBeans Teil der visuell manipulierbaren Eigenschaften, so kann aber durch einen Entwickler innerhalb der IDE zumindest festgestellt werden, welche dieser JavaBeans bereits zugeordnet sind und welche nicht.

Kompositionsfähigkeit

Werden alle Möglichkeiten von Java und der visuellen Manipulation ausgeschöpft, so ist es möglich, einfach komponierbare JavaBeans zu entwickeln. Da die meisten Techniken allerdings nicht direkt zu JavaBeans-Mitteln zählen, sondern ihre Auswahl und Umsetzung im Verantwortungsbereich des jeweiligen Entwicklers liegen, kann nicht davon gesprochen werden, dass JavaBeans selbst eine integrierte Kompositionsfähigkeit besitzen. Hierzu ist jeweils zusätzlicher Quellcode oder eine gut umgesetzte JavaBean notwendig.

Interaktionsfähigkeit

Neben der Möglichkeit, Methoden einer JavaBean direkt aufzurufen, kann die Kommunikation zwischen JavaBeans auch ereignisorientiert verlaufen. Bei dieser Art der Kommunikation handelt es sich um eine Umsetzung des Publish-Subscriber-Entwurfsmusters.

Auslieferungseinheit (Atomizität)

Ausgeliefert werden JavaBeans als JAR-Datei. Dabei handelt es sich um eine Archiv-Datei, die alle benötigten Java-Klassen, Icons, und sonstigen Elemente beinhaltet. Optional kann eine Manifest-Datei Teil des Archivs sein. Diese Manifest-Datei listet alle Elemente der JAR-Datei auf und kennzeichnet explizit die JavaBean-Klasse.
Da das Wissen über weitere benötigte Fremdkomponenten nicht vollständig explizit ist, kann es vorkommen, dass eine JavaBean in einer Umgebung installiert wird, in der sie nicht lauffähig ist, da benötigte Teile fehlen.

Ausführbarkeit

Die ausgelieferte JAR-Datei kann von einer Entwicklungsumgebung eingelesen werden, so dass die JavaBean automatisch dem Anwendungsentwickler

zur Verfügung gestellt wird. Ausführbar ist eine JavaBean allerdings niemals alleine, immer wird eine umgebende Applikation benötigt.

Unabhängigkeit von Programmiersprachen

JavaBeans werden ausschließlich in Java implementiert, somit ist eine vollkommene Abhängigkeit von dieser Programmiersprache gegeben.

Unabhängigkeit von Plattformen

Da Java plattformunabhängig ist, solange eine Java-Virtual-Machine auf der Zielplattform existiert, sind auch JavaBeans im gleichen Maße plattformunabhängig.

Komposition ohne Programmierkenntnisse

JavaBeans müssen per Definition in einer graphischen Entwicklungsumgebung visuell manipulierbar sein. Die Konfektionierung einer JavaBean kann also ohne Programmierkenntnisse geschehen. Soweit durch die Entwickler einer Bean vorgesehen, gehört zu dieser Konfektionierung auch das Verschalten mit weiteren JavaBeans. In diesem Fall ist auch die Komposition von JavaBeans ohne Programmierkenntnisse möglich, allerdings ist das nicht fester Bestandteil der JavaBean-Architektur.

Ortstransparenz, Verteiltheit

JavaBeans sind konzipiert für den Einsatz auf einem lokalen Rechner. Ziel ist der einfache Einsatz gut gekapselter Funktionalität. Verteiltheit spielt hierbei keine Rolle, demzufolge ist auch eine Betrachtung der Ortstransparenz hinfällig.

Klassifizierbarkeit, Kategorisierbarkeit

Als Klassifizierungsmerkmal lassen sich für JavaBeans die Schnittstellen-Informationen heranziehen. Allerdings beschränkt sich dieses Merkmal rein auf syntaktische Eigenschaften, über Funktionalität kann dabei keine Aussage getroffen werden.

Lose Kopplung

Lose Kopplung kann im Rahmen von JavaBeans durch die Verwendung der ereignisorientierten Kommunikation erreicht werden. Ansonsten muss man von einer eher fixen Kopplung ausgehen.

Existenz eines Verzeichnisdiensts

Einen tatsächlichen Verzeichnisdienst gibt es für JavaBeans nicht. Eine vergleichbare Aufgabe übernimmt hier die graphische Entwicklungsumgebung, die dem Anwendungsentwickler alle verfügbaren JavaBeans meistens in Form von sogenannten Komponenten-Paletten anzeigt. Allerdings handelt es sich hierbei nur um eine einfache Auflistung der verfügbaren JavaBeans, weitergehende Anfragen sind nicht möglich.

Plug&Play-Fähigkeit

Ist ein JavaBean einmal installiert, so kann es umgehend durch den Anwendungsentwickler eingesetzt werden. Mit Plug&Play-Fähigkeit ist im engeren Sinne aber eigentlich nicht diese Art von Verfügbarkeit zur *Design-Zeit* gemeint, sondern vielmehr die direkte Einsatzbereitschaft in einem laufenden System, die durch die Art der JavaBeans schon gar nicht gegeben sein kann.

Fazit

JavaBeans sind ein sehr hilfreiches Mittel zur schnellen Entwicklung von Applikationen. Allerdings handelt es sich bei ihnen nicht um Komponenten im Sinne dieser Arbeit. Ziel der JavaBeans ist die einfache und vor allem schnelle Entwicklung von zumeist graphischen Anwendungen durch Wiederverwendung gut gekapselter Funktionseinheiten. Das Zusammenspiel, also die Komposition, dieser Einheiten tritt dabei nicht in den Vordergrund.
Auch sind JavaBeans niemals alleine lauffähig. Sie benötigen immer eine Applikation, die sie aufnehmen und überhaupt erst in einen ausführbaren Zustand versetzen.
Vergleichbar mit JavaBeans sind weitere visuelle Komponenten in anderen Entwicklungsumgebungen, z.B. Borland Delphi oder MS Visual Studio.

2.3.6 JKogge

JKogge ist der in Java implementierte Nachfolger von KOGGE (Koblenzer Generator für Graphische Entwurfsumgebungen), eines Meta-CASE-System, d.h. eines Generator für Werkzeuge, die graphische Software-Entwurfsansätze unterstützen.
Interessant an JKogge ist die Architektur des Gesamtsystems, denn laut den Entwicklern ist JKogge komponentenbasiert (vgl. [2] S.1). Eine ausführliche Beschreibung des Systemkonzeptes findet sich in [2], hier beschränken wir uns auf die für die Einordnung relevanten Aspekte.

JKogge besteht aus drei logischen Elementen. Dabei handelt es sich um *Dokumente*, das *Basissystem* und *JKogge-Plug-ins*. Dokumente sind klar abgegrenzte Informationseinheiten mit einer bestimmten Bedeutung.
Vereinfachend gesprochen ist die einzige Aufgabe des Basissystem das Laden und Verwalten von Dokumenten und Plug-ins. Dabei ist jeweils eine URL (*Unified Resource Locator*) nötig, um ein zu ladendes Dokument oder Plug-in zu identifizieren. Das Basissystem arbeitet nach dem On-Demand-Prinzip, d.h. Dokumente werden nur dann geladen, wenn sie explizit angefordert werden.
Außerdem ist das Basissystem auch Kommunikationsgrundlage für die Interaktion der geladenen Plug-ins. Plug-ins können über sogenannte *JKogge-Messages* Nachrichten und Informationen austauschen.

Die Ziele der JKogge-Plug-ins sind

- auf jeder mit dem Internet verbundenen Plattform lauffähig zu sein
- über URLs erreichbar und zugreifbar zu sein
- bei Bedarf geladen werden zu können
- unabhängig von anderen Plug-ins laufen zu können
- über festgelegte Schnittstellen miteinander kommunizieren zu können

Konzipiert sind die JKogge-Plug-ins als dokumentenverarbeitende Einheiten. Jedes Plug-in verfügt über sogenannte *slots*, die jeweils Dokumente eines bestimmten Typs aufnehmen können. Dokumente können zur Laufzeit einem Slot hinzugefügt oder wieder entfernt werden.
Des Weiteren können Plug-ins Verbindungen zu weiteren Plug-ins besitzen. Eine solche Verbindung wird bei Bedarf durch ein Plug-in beim Basissystem angefragt und durch das Basissystem hergestellt.
JKogge-Plug-ins erscheinen in Form von gepackten Java-Archivdateien. Genau eine Klasse des Plug-ins ist Spezialisierung der abstrakten Klasse `PlugIn`. Diese Klasse ist zusätzlich auch als *JavaBean* gekennzeichnet. Das Basissystem ist in der Lage, solche gepackten Plug-ins zu entpacken und anschließend zu laden.
Die Kommunikation zwischen Plug-ins verläuft asynchron, es werden die bereits erwähnten JKogge-Messages ausgetauscht. Diese Nachrichten können Anfragen, Daten oder Benachrichtigungen darstellen. Eine JKogge-Nachricht ist ein Objekt, das von einem zu einem anderen Plug-in verschickt wird. Verarbeitet werden sie in der Reihenfolge des Eintreffens beim Empfänger, wobei sie in einer Warteschlange zwischen gespeichert werden.

Selbstbeschreibungsfähigkeit

Die Beschreibungsfähigkeit eines JKogge-Plug-ins beschränkt sich hauptsächlich auf die angebotenen Dokumenten-Slots. Die Anzahl, Namen und Typen der einzelnen Slots können abgefragt und somit festgestellt werden, welche Dokumente durch ein JKogge-Plug-in verarbeitet werden können. Über die Art der Verarbeitung kann das Plug-in allerdings keine Aussage treffen.

Strikte Kapselung, Blackbox-Wiederverwendung

Da es sich bei JKogge-Plug-ins um Java-Klassen handelt, ist ein JKogge-Plug-in hier nur soweit gekapselt, wie es der Entwickler durchgeführt hat. Da die Kommunikation zwischen den Plug-ins auf JKogge-Messages beschränkt und ansonsten nur Dokumente ausgetauscht werden, kann man aber prinzipiell von einer strikten Kapselung und Blackbox-Wiederverwendung ausgehen.

Konfektionierbarkeit

JKogge-Plug-ins besitzen keinen integrierten Mechanismus zur Konfektionierung.

Wissen über Komponentenabhängigkeiten

Zur Laufzeit kennt ein JKogge-Plug-in alle bereits durch es angefragten Plug-ins, da diese über Referenzen vorgemerkt werden. Allerdings besteht keine Möglichkeit, ein JKogge-Plug-in zu befragen, welche anderen Plug-ins insgesamt benötigt werden, um die gewünschte Funktionalität zu gewährleisten.

Kompositionsfähigkeit

Ein JKogge-System besteht aus einer Sammlung von JKogge-Plug-ins. Wird ein Plug-in mit einem ihm unbekannten Dokumenttyp konfrontiert, so kann die JKogge-Registry ein Plug-in liefern, das für die Verarbeitung eines solchen Dokumentes geeignet ist.
Auf diese Weise können bei diszipliniertem Einsatz von Plug-ins Systeme geschaffen werden, die ohne zusätzliche Verlinkung der Plug-ins eine komplexe Funktionalität bereitstellen. Allerdings kann dieses Vorgehen auch zu unkontrollierbaren Abläufen innerhalb des Systems führen.
Die Kompositionsfähigkeit der JKogge-Plug-ins ist definitiv gegeben. Im Vergleich zu den vorgestellten *wiring*-Technologien ist man hier einen bedeutenden Schritt weiter.

Interaktionsfähigkeit

JKogge-Plug-ins sind interaktionsfähig. Bei Bedarf fordern sie andere Plug-ins durch das Basissystem an, für den Nachrichten- und Datenaustausch werden anschließend JKogge-Messages benutzt. Ohne zuvor explizites Wissen über die Existenz eines bestimmten anderen Plug-ins kann also zur Laufzeit eine Kommunikation hergestellt und durchgeführt werden.

Auslieferungseinheit (Atomizität)

Die JKogge-Plug-ins werden als Java-Archiv ausgeliefert. So gesehen bilden sie also eine in sich geschlossene Einheit. Auf funktionaler Ebene allerdings hängt es von der Implementation des jeweiligen Plug-ins ab, ob die Atomizität tatsächlich gewährleistet wird. Es ist vorstellbar, dass ein ausgeliefertes JKogge-Plug-in die eigene Funktionalität nicht alleine bewältigen kann, da es von anderen Softwareelementen abhängig ist, die separat installiert werden müssten.

Ausführbarkeit

Das JKogge-Basissystem kann Plug-ins bei Bedarf laden und ausführen. JKogge-Plug-ins erfüllen also das Kriterium der Ausführbarkeit.

Unabhängigkeit von Programmiersprachen

JKogge-Plug-ins können nur in Java implementiert werden, zumindest die Plug-in-Startklasse.

Unabhängigkeit von Plattformen

Durch Java ist allerdings eine relativ große Plattformunabhängigkeit gegeben, da Java selbst plattformunabhängig ist und nur das Vorhandensein einer Java Virtual Machine voraussetzt.

Komposition ohne Programmierkenntnisse

Existierende JKogge-Plug-ins können weitgehend ohne Programmierkenntnisse komponiert werden. Wie zuvor beschrieben, ist die Registry in der Lage, Plug-ins zu liefern, die mit bestimmten Dokumenttypen umgehen können. So kann ein JKogge-System zusammengestellt werden, ohne die einzelnen Plug-ins über zusätzliche Programmierarbeit miteinander zu verbinden.

Ortstransparenz, Verteiltheit

JKogge ist konzipiert für den Einsatz auf einem lokalen Rechner. Allerdings können die zu ladenden Dokumente über URLs (*Uniform Resource Locator*) spezifiziert werden, so dass Plug-ins und Daten auf beliebigen Rechnern im Inter-/Intranet abgelegt werden können.
Ausgeführt werden die Plug-ins allerdings immer lokal und die genaue Adresse der zu ladenden Dokumente muss dem Basissystem auch bekannt gemacht werden. Echte Verteiltheit ist nicht gegeben, Ortstransparenz ebenfalls nicht.

Klassifizierbarkeit, Kategorisierbarkeit

JKogge-Plug-ins sind realisiert über Java-Klassen. Eine Klassifizierung anhand dieses Merkmals ist also möglich, ebenso die Spezialisierung von Plug-ins. Eine eigentliche Klassifizierbarkeit von JKogge-Plug-ins in Hinblick auf den Komponentenaspekt ist allerdings nicht vorgesehen.

Lose Kopplung

Durch die Verwendung des Basissystems als Vermittler und den Nachrichtenaustausch per JKogge-Messages sind die einzelnen Plug-ins in JKogge lose gekoppelt.

Existenz eines Verzeichnisdiensts

Das JKogge-Basissystem ist vergleichbar mit einem Verzeichnisdienst. Auf Anfrage kann es Plug-ins liefern, die mit bestimmten Dokumententypen umgehen kann.

Plug&Play-Fähigkeit

Nach Auslieferung und Registrierung eines JKogge-Plug-ins steht dieses dem gesamten System zur Verfügung. Zusammen mit den zuvor beschriebenen Möglichkeiten der Interaktion ist die Plug&Play-Fähigkeit also gegeben.

Fazit

JKogge kommt einem Komponentensystem sehr nahe. Das Konzept des Basissystems, die Verwendung eines gemeinsamen Kommunikationsmediums und die aus diesen Punkten resultierende einfache Komponierbarkeit sind positiv hervorzuheben.
Einschränkend ist die Beschränkung auf Java als Implementationssprache, sowie die Verwendung einer abstrakten Oberklasse für alle JKogge-Plug-ins.

Ein solches Vorgehen schränkt die Verbreitung dieses Ansatzes stark ein, da jeder Plug-in-Entwickler Zugriff auf diese Basisklasse haben müsste. Bei Änderungen an der Basisklasse treten auch direkt Versionsprobleme auf, da bereits ausgelieferte Plug-ins unter Umständen nicht dem aktuellen Stand entsprechen.
Nicht vollständig erfüllt ist der Aspekt der Verteiltheit von Komponenten. Für das JKogge-Szenario ist dieser Aspekt irrelevant, für einen allgemeinen Komponentenansatz ist er je nach Anwendungsszenario wünschenswert.

2.3.7 Web Services

Web Service-Technologie wird mit zunehmender Häufigkeit eingesetzt. Vor allem begünstigt wird diese Entwicklung durch integrierte Unterstützung z. B. in Microsoft .NET oder auch J2EE von Sun Microsystems.
Im engeren Sinne stellen Web Services keine Komponententechnologie dar, sie weisen aber z. B. im Vergleich zu CORBA (vgl. 2.3.2) einige Gemeinsamkeiten auf. Aufgrund dieser Gemeinsamkeiten widmet sich dieser Abschnitt der Betrachtung von Web Services in Hinblick auf den Einsatz als Komponententechnologie.

Da der Begriff *Web Service* für viele verschiedene Technologien verwendet wird, muss zunächst festgelegt werden, welche Kombination aus WS-Technologien in diesem Abschnitt betrachtet werden soll. Ausgehend von den bereits zuvor erwähnten Plattformen Microsoft .NET und J2EE von Sun Microsystems und anhaltenden Standardisierungsbemühungen bilden *WSDL* (Web Service Description Language), *SOAP* (Simple Object Access Protocol) und *UDDI* (Universal Description, Discovery and Integration) das Grundgerüst aller Web Services.

WSDL dient der Beschreibung eines Web Service. Dabei enthält eine WSDL-Beschreibung Angaben zum Nachrichtenformat und den Protokollen des Dienstes, sowie Informationen zur Bindung an ein Transportprotokoll, um dem Sender zu vermitteln, wie Nachrichten zu verschicken sind (vgl. [10]). In den meisten Fällen wird WSDL zusammen mit SOAP genutzt, die Bindung an SOAP ist sogar Bestandteil der WSDL-Spezifikation.

SOAP ist ein Transportprotokoll für XML-basierten Nachrichtenaustausch. Während WSDL also die Schnittstellen des Dienstes beschreibt, ist SOAP für die Kommunikation mit einem bestehenden Web Service zuständig.

UDDI ist ein Protokoll für einen Web Service-Verzeichnisdienst. Bestehen-

de Web Services, die per WSDL beschrieben sind, können in eine UDDI-Registrierung aufgenommen werden. Die Interaktion mit einer UDDI-Registrierung verläuft per SOAP-Nachrichtenaustausch.

Die Suche nach registrierten Web Services kann innerhalb einer UDDI-Registrierung anhand verschiedener Kriterien erfolgen. Als Suchergebnis erhält die Anfragestelle die WSDL-Beschreibung des Web Service und einige zusätzliche Informationen.

Die Kombination aus WSDL, SOAP und UDDI wird anhand der bekannten Liste von Kriterien auf ihre Tauglichkeit als Komponententechnologie überprüft. Im Folgenden werden Dienste im Inter- oder Intranet, die auf diesen drei Technologien beruhen, als Web Service bezeichnet.

Selbstbeschreibungsfähigkeit

Zu jedem Web Service gehört eine WSDL-Beschreibung. Diese Beschreibung enthält Informationen zur syntaktischen Schnittstelle des Dienstes. Zwar enthält eine WSDL-Beschreibung auch Informationen zu Operationen auf den ausgetauschten Nachrichten, allerdings erweitern diese zusätzlichen Informationen nur den syntaktischen Aspekt der Beschreibung. Semantische Elemente enthält die Beschreibung per WSDL nicht.
Innerhalb einer UDDI-Registrierung können darin enthaltene Web Services nicht nur ihre WSDL-Beschreibung veröffentlichen. Zusätzlich können hier textuelle Beschreibungen angegeben werden. Außerdem ist es möglich, einen Web Service einer Kategorie zuzuordnen, um so das Anwendungsspektrum eines Dienstes genauer zu erfassen. Da diese Kategorien aber nicht mit einer explizit festgelegten Semantik verknüpft sind, ist der Wert dieser Kategorisierung nur von geringer Bedeutung und auch nur für einen menschlichen Nutzer der Registrierung sinnvoll auswertbar.

Strikte Kapselung, Blackbox-Wiederverwendung

Web Services sind strikt gekapselt. Per WSDL werden die Schnittstellen veröffentlicht, die Implementation ist dem Nutzer verborgen.

Konfektionierbarkeit

Web Services bieten in dieser Form keine Unterstützung zur Konfektionierung.

Wissen über Komponentenabhängigkeiten

Ein Web Service kann zwar intern von anderen Web Services abhängig sein. Diese Abhängigkeit liegt allerdings nur in Form von Programmcode vor. Sie ist also nicht explizit und auch nicht abfragbar.

Kompositionsfähigkeit

Web Services dieser Art bieten keine integrierte Unterstützung zur Komposition mehrerer Dienste. Web Services können auf anderen Web Services aufbauen, dazu ist allerdings jeweils neuer Programmcode nötig, der die Verschaltung vornimmt.

Interaktionsfähigkeit

Untereinander kommunizieren Web Services mit den gleichen Mitteln, wie auch andere Nutzer der Dienste mit ihnen in Kontakt treten. Der Nachrichtenaustausch zwischen Web Services geschieht per SOAP, die Nachrichtenübermittlung muss explizit angestoßen werden.

Auslieferungseinheit (Atomizität)

Web Services werden prinzipiell als Einheit ausgeliefert. Sie stellen selbstständige Webanwendungen dar.

Ausführbarkeit

Ein ausgelieferter Web Service ist per Definition immer ausführbar.

Unabhängigkeit von Programmiersprachen

Die Web Service-Technologien sind unabhängig von Programmiersprachen. Erreicht wird dies über XML, das Basisformat für WSDL und SOAP.

Unabhängigkeit von Plattformen

Durch die Verwendung von standardisierten und weit verbreiteten Internettechnologien wie HTTP für den Nachrichtentransport und die Nutzung der plattformunabhängigen Sprachen WSDL und SOAP sind Web Services selbst auch unabhängig von der eingesetzten Plattform.
Web Services können auf beinahe beliebigen Plattformen angeboten werden, die Nutzung erfolgt vollkommen unabhängig von der Plattform, auf der der Dienst ausgeführt wird.

Komposition ohne Programmierkenntnisse

Web Services können in dieser Form nicht ohne Programmierkenntnisse zu größeren Einheiten zusammengesetzt werden.

Ortstransparenz, Verteiltheit

Verteiltheit ist eines der Hauptmerkmale von Web Services. Für den Nutzer eines Web Service erfüllt sich die Ortstransparenz vor allem in Kombination mit einer UDDI-Registrierung, die die benötigte WSDL-Beschreibung mitsamt den Lokationsinformationen veröffentlicht.

Klassifizierbarkeit, Kategorisierbarkeit

Kategorisierbarkeit ist Bestandteil von UDDI. Allerdings besitzen die Kategorien keine definierte Semantik, die Einteilung ist sozusagen willkürlich. Klassifizierbarkeit im engeren Sinne ist durch Web Services nicht gegeben. Durch die Verwendung von Ontologien könnte die Kategorisierbarkeit aufgewertet werden und einen semantischen Mehrwert liefern.

Lose Kopplung

Die Kopplung an einen Web Service ist lose. Erst zum Zeitpunkt des Leistungskonsums werden an den entsprechenden Web Service Nachrichten geschickt. Nach erfolgter Quittierung des Nachrichteneingangs kann der Konsument dann davon ausgehen, dass die Nachricht eingegangen ist.

Verzeichnisdienst

UDDI-Registrierungen sind die Verzeichnisdienste von Web Services.

Plug&Play-Fähigkeit

Ein installierter und in einer UDDI-Registrierung veröffentlichter Web Service kann von potentiellen Nutzern gefunden und direkt genutzt werden. Die Nutzung geschieht allerdings nicht automatisiert, hier muss in der Regel zunächst der entsprechende Programmcode geschrieben werden. Echtes Plug&Play ist somit nicht realisiert.

Fazit

Die Entwicklung von Web Service-Technologie geht stetig voran. Neben den hier erwähnten Sprachen und Protokollen existieren viele weitere. Geeignete

Kombinationen dieser Technologien dürften in der Lage sein, weitere Kriterien erfüllen zu können. Diese sind allerdings in den meisten Fällen nicht verbreitet und zumeist auch weit von einer Standardisierung entfernt.
Das in diesem Abschnitt vorgestellte Grundgerüst für Web Services stellt eine *wiring*-Technologie vergleichbar mit CORBA (vgl. 2.3.2) dar. In einem Paper der Sankhya Technologies Private Limited [4] werden CORBA IDL und WSDL verglichen, mit dem Ergebnis, dass beide Sprachen in weiten Teilen ähnliche Konzepte aufweisen, und prinzipiell die gleichen Sachverhalte mit beiden Sprachen beschrieben werden können.
Werden heutige Bestrebungen zur automatischen Kopplung und Einbindung von Web Services in Zukunft weiter verfolgt, so sollten diese Web Services erneut unter dem Aspekt der Komponentenorientierung betrachtet werden. Heute verbreitete oder standardisierte Mittel reichen aber noch nicht aus, um von einer echten Komponententechnologie sprechen zu können.

2.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Literaturquellen zum Thema *Komponentenorientierung* in Hinblick auf die unterschiedlichen Anforderungen an Software-Komponenten betrachtet. Die auf diesem Weg gewonnenen Erkenntnisse wurden zusammengefasst in der Entwicklung eines Anforderungskataloges an Software-Komponenten.
Nachfolgend wurden heutige sogenannte Komponententechnologien auf den Grad der Erfüllung gegenüber dem aufgestellten Anforderungskataloges untersucht. Dabei konnte festgestellt werden, dass der größte Teil dieser Technologien nur Kommunikationsmittel für verteilte sogenannte Komponenten liefert. Die Aspekte **Komponierbarkeit** und **Wiederverwendbarkeit** werden von den meisten Konzepten **nicht berücksichtigt**.

Im folgenden Kapitel wird ein Komponentenkonzept entwickelt und vorgestellt, dass die Aspekte des Anforderungskataloges erfüllen soll. Im Mittelpunkt dieses Konzeptes steht die **Komponierbarkeit** von Software-Komponenten.

[4] http://www.products.nasscom.org/downloads/idl-wsdl.pdf, 20.05.2005

Kapitel 3

Konzept eines Komponentensystems

Das vorangegangene Kapitel hat gezeigt, dass die Versprechungen und Erwartungen an ein echtes Komponentensystem durch heute vorhandene Ansätze nur unvollständig erfüllt werden. Um den Zielen der Komponentenorientierung näher zu kommen, bedarf es einer umfassenderen Betrachtung und Erarbeitung neuer Lösungsansätze.
Aufbauend auf den Grundlagen des letzten Kapitels wird in diesem Kapitel ein abstraktes Komponentenkonzept erarbeitet und vorgestellt. Ziel dieses Konzepts ist die Unterstützung komponentenorientierter Spezifikation und Entwicklung. Die hier zum Einsatz kommenden Komponenten sollen meist größere Funktionseinheiten kapseln. Wichtigstes Ziel ist die einfache Komponierbarkeit einzelner Elemente zu einem komplexeren System.

Zunächst werden die Ziele des zu entwickelnden Konzeptes und die Grundidee besprochen. Nach diesen einführenden Abschnitten folgen Detailbeschreibungen des Konzeptes.
Die Eigenschaften von Software-Komponenten, die im Rahmen des in Kapitel 2 entwickelten Anforderungskataloges aufgestellt worden sind, sollen durch das im folgenden vorgestellte Konzept möglichst weitgehend erfüllt werden. Am Ende dieses Kapitels erfolgt dazu die Überprüfung des Erfüllungsgrades. Das Kapitel schließt mit einer Zusammenfassung und einem Ausblick auf potentielle Erweiterungen des Konzeptes.

3.1 Einführung

Die meisten in Kapitel 2 betrachteten Komponententechnologien stellen reine Verbindungstechnologien dar, die prinzipiell nur Objekten einer bestimmten Art Infrastrukturmittel zur verteilten Kommunikation liefern.
Das allgemeine Konzept der Komponentenorientierung führt als wichtigsten Punkt die Komponierbarkeit von Komponenten an. Dieser Aspekt wird von reinen *wiring*-Technologien wie COM (vgl. 2.3.1) nicht ausdrücklich unterstützt.
Die wenigen Komponententechnologien aus Kapitel 2, die sich der Komponierbarkeit widmen, haben hingegen andere Schwächen, die die Entwicklung eines neuen Ansatzes motivieren. Bei diesen Schwächen handelt es sich z. B. um Abhängigkeiten von bestimmten Programmiersprachen, oder sogar expliziten Basisklassen innerhalb einer gegebenen Sprache.

Ziel muss die Entwicklung eines neuen Komponentenkonzeptes sein, das die Komponierbarkeit von Komponenten in den Mittelpunkt stellt. Dabei sollten Aspekte wie Verteiltheit, Unabhängigkeit von Programmiersprachen und Plattformen so berücksichtigt werden, dass eine spätere Umsetzung des Konzeptes auch diese Eigenschaften aufweisen kann. Die Entwicklung eines solchen neuen Konzeptes bedingt zunächst eine Klärung des Begriffes *Komponente*.

3.1.1 Der Komponentenbegriff

Der Begriff *Komponente* wird im Umfeld der Komponentenorientierung sehr häufig auf unterschiedliche Weise verwendet. Je nach Autor oder System beschreibt *Komponente* vollkommen unterschiedliche Artefakte.
Um Komponenten innerhalb des Konzeptes dieser Arbeit von anderen Komponentenbegriffen unterscheiden zu können, wird zunächst ein neuer Begriff eingeführt. Eine Komponente im Kontext dieser Arbeit heißt demnach **K-Komponente**.
K-Komponenten implementieren und repräsentieren Funktionalität. Die Syntax und Semantik von K-Komponenten wird in sogenannten *Diensten* definiert (vgl. 3.2). Eine K-Komponente implementiert mindestens einen Dienst. Zugriff auf eine K-Komponente und deren Leistung erhält man über sogenannte *Eigenschaften*, *Methoden* und *Events*. K-Komponenten sind komponierbar und können relativ simpel mit anderen K-Komponenten zu größeren Netzen verschaltet werden. Abschnitt 3.2 enthält eine detaillierte Beschrei-

bung von Diensten und deren Merkmalen, Abschnitt 3.3 widmet sich den K-Komponenten und den zugehörigen Aspekten.

Sinnbild für ein verschaltetes Netz von K-Komponenten ist der *Komponentenschaltplan*, der im nächsten Abschnitt überblicksartig vorgestellt wird.

3.1.2 Der Komponentenschaltplan

Im Fokus des Konzeptes steht die **Komponierbarkeit** von K-Komponenten. Metapher für diese Eigenschaft ist die eines **Schaltplans** bestehend aus miteinander gekoppelten funktionalen Einheiten. Diese Einheiten sind zumeist Blackboxes, deren Ein- und Ausgänge in einem auch möglichst visuell erfassbaren Plan untereinander so verschaltet werden, dass sich die gewünschte Gesamtfunktionalität aus den Einzelfunktionalitäten der gekoppelten Elemente ergibt.
Die funktionalen Einheiten der Metapher stellen die K-Komponenten dar, die in einem Komponentenschaltplan über Kopplungen miteinander verbunden werden.
Eine *Kopplung* ist ein Bindeglied zwischen einer anbietenden und einer nutzenden K-Komponente. Dabei können K-Komponenten über eine Kopplung miteinander verbunden werden, falls die nutzende K-Komponente die anbietende als sogenannten *Import* benötigt, um die eigene Leistung zu erbringen. Eine andere Art der Kopplung ist die zwischen den sogenannten *Events* und *Eventhandlern*. Eine Ereignisquelle kann so mit einem Ereignisverarbeiter verschaltet werden.
Detaillierte Informationen zum Komponentenschaltplan und Kopplungen arbeitet Abschnitt 3.4 auf.

Im Folgenden wird ein detaillierter Überblick über K-Komponenten und alle zugehörigen Aspekte gegeben.

3.2 Dienste

Ein *Dienst* ist die Definition eines minimal zu erfüllenden Leistungsumfanges für K-Komponenten. Über diese Definition wird sowohl die Syntax wie auch die Semantik der implementierenden K-Komponenten definiert. Eine K-Komponente implementiert mindestens einen Dienst, potentiell beliebig viele. Dienste bilden das hauptsächliche Klassifizierungsmerkmal für K-Komponenten.
Ein Dienst muss einen systemweit eindeutigen Bezeichner besitzen, um so

zweifelsfrei identifiziert werden zu können. Des Weiteren umfasst ein Dienst eine natürlichsprachliche Beschreibung der Semantik, also der Aufgabe und des Leistungsumfanges des Dienstes.
Zusätzlich zu diesen Angaben werden innerhalb eines Dienstes Definitionen für folgende Elemente erfasst:

- Datentypen
- Eigenschaften
- Methoden
- Exceptions
- Events
- Eventhandler
- Protokolle
- Imports
- Constraints

Die nun folgenden Abschnitte stellen diese einzelnen Elemente eines Dienstes detailliert vor und erläutern deren Verwendung.

3.2.1 Datentypdefinitionen

Einem Dienst liegen häufig spezielle Datentypen zugrunde. Um diese Datentypen potentiellen Nutzern zugänglich zu machen, werden in den *Datentypdefinitionen* alle öffentlich benötigten Datentypen definiert. Durch diese Definitionen lassen sich komplexe Datentypen erzeugen, die als Typisierung von Parametern verwendet werden können.
Eine Datentypdefinition besteht aus einem eindeutigen Bezeichner und der eigentlichen Datenstruktur. Jedes Strukturelement setzt sich zusammen aus einem Namen und der Angabe eines Datentyps. Der Datentyp kann ein einfacher Typ sein (z. B. String, Boolean, Integer, DateTime), oder ein bereits definierter komplexer Typ.

3.2.2 Eigenschaftendefinitionen

Eine *Eigenschaft* ist ein benannter Wert. Genutzt werden Eigenschaften, um Aussagen über Eigenarten oder das Verhalten einer K-Komponente treffen zu können, oder um eine K-Komponente an ein gegebenes Einsatzszenario anzupassen.
Eine *Eigenschaftendefinition* umfasst den pro Dienst eindeutigen Bezeichner, sowie den Wertetyp. Zusätzlich existiert zu jeder Eigenschaftendefinition eine natürlichsprachliche Beschreibung der Semantik der Eigenschaft.

Eine Eigenschaft kann entweder gesetzt werden, um so die K-Komponente zu konfektionieren, oder sie steht für eine K-Komponente unveränderlich fest, um so potentiellen Nutzern Auskunft über die so beschriebenen Aspekte der K-Komponente zu liefern. Die Eigenschaftendefinition umfasst dazu die Angabe, ob es sich um eine *Write-* oder *Read-Eigenschaft* handelt, wobei Write-Eigenschaften gesetzt und Read-Eigenschaften nur gelesen werden können.
Beispiel für eine Write-Eigenschaft ist das Festlegen einer Speicherstelle, an der Berechnungsergebnisse abgelegt werden sollen. Eine Read-Eigenschaft ist beispielsweise eine Aussage über Antwortzeiten bestimmter Funktionen.

Das Setzen einer Write-Eigenschaft kann unter manchen Umständen zu einem Fehler innerhalb der betroffenen K-Komponente führen. In einem solchen Fehlerfall wird eine sogenannte *Exception* geworfen, die Auskunft über den Fehler gibt. Um anzuzeigen, welche Fehler beim Setzen einer Write-Eigenschaft auftreten können, umfasst die Definition eine Auflistung aller Exceptiondefinitionen (vgl. 3.2.4) der Exceptions, die potentiell geworfen werden können.

3.2.3 Methodendefinitionen

Methoden bieten öffentlichen Zugriff auf die funktionalen Operationen von K-Komponenten. Eine *Methodendefinition* umfasst den pro Dienst eindeutigen Bezeichner, die benannten und typisierten Parameter, sowie die benannten Rückgabewerte der Operation. Neben dieser syntaktischen Beschreibung existiert zu jeder Methodendefinition eine natürlichsprachliche Beschreibung der Semantik.
In den meisten Fällen repräsentieren die Methoden einer K-Komponente deren eigentliche Funktionalität. Dem folgend sind die Methodendefinitionen vor allem bezogen auf die semantischen Aspekte von essentieller Bedeutung für die gesamte Dienstbeschreibung.
Analog zum Setzen von Write-Eigenschaften kann beim Aufruf einer Methode

ein Fehler auftreten. Auch hier greift das Konzept der Exceptions. Ebenfalls analog zur Definition von Eigenschaften umfasst die Methodendefinition eine Auflistung aller Exceptiondefinitionen (vgl. 3.2.4) der Exceptions, die potentiell geworfen werden können.

3.2.4 Exceptiondefinitionen

Eine *Exception* ist eine Benachrichtigung über einen eingetretenen unerwarteten Fehlerfall. In der *Exceptiondefinition* wird festgelegt, um welchen Fehler es sich handelt und welche Informationen über diesen weitergegeben werden. Dazu besteht die Definition aus einer natürlichsprachlichen Beschreibung des aufgetretenen Fehlers, sowie einer Menge von benannten und typisierten Parametern, die Zusatzinformationen über den Kontext des Fehlerfalls liefern können.
Eine Exception kann auftreten beim Aufruf einer Methode und ebenso beim Setzen einer Eigenschaft.

3.2.5 Eventdefinitionen

Ein *Event* ist eine veröffentlichte Benachrichtigung über ein durch eine K-Komponente wahrgenommenes Ereignis. Eine *Eventdefinition* besitzt einen pro Dienst eindeutigen Bezeichner. Neben der Benamung existiert zusätzlich sowohl eine syntaktische wie auch eine semantische Beschreibung des Events. Die Syntax umfasst die Parameter, die bei Auslösung des Ereignisses mitgeteilt werden. Die natürlichsprachliche semantische Beschreibung umfasst Erläuterungen zum Auslösegrund und zu den zur Verfügung gestellten Parametern.
Beispiel für einen Dienst mit Events ist ein Timer. Über eine Eigenschaft wird das Zeitintervall festgelegt, eine Methode oder ein empfangenes Event startet den Timer, ebenso kann er wieder gestoppt werden. Ist die über das Zeitintervall festgelegte Zeitspanne abgelaufen, so wird das zugehörige Event ausgelöst.

3.2.6 Eventhandlerdefinitionen

Ein *Eventhandler* ist das Gegenstück zu einem Event und beschreibt somit die Fähigkeit, auf ein bestimmtes Ereignis reagieren bzw. es verarbeiten zu können. Eine *Eventhandlerdefinitionen* eines Dienstes bestimmt, welche Events ein Eventhandler dieser Art verarbeiten kann und welche Semantik die Verarbeitung besitzt. Die Vergabe eines eindeutigen Bezeichners schließt die Eventhandlerdefinition ab.

Das zuvor angesprochene Beispiel des Timers kann hier nochmals aufgegriffen werden. Will eine K-Komponente auf das Timersignal, das durch ein Event dargestellt wird, reagieren, so muss sie einen Eventhandler bereitstellen, der zu dem Event des Timerdienstes passt.
Ebenso wie bei einer Methode kann bei der Verarbeitung eines Events ein Fehler auftreten. Dieser Fehler wird im Gegensatz zu den Methoden allerdings nicht über eine Exception an die aufrufende Einheit zurückgegeben. Denn im Fall eines Eventhandlers weiß die Eventquelle grundsätzlich nichts von der Eventverarbeitung und ist auch nicht an dabei auftretenden Fehlern interessiert. Aus diesem Grund muss die Definition eines Eventhandlers auch keine Aussagen über potentiell zu werfende Exceptions aufweisen.

3.2.7 Protokolldefinitionen

Dienste umfassen in der Regel mehrere Methoden und Eventhandler, die untereinander Abhängigkeiten aufweisen können. Beispielsweise muss zunächst die Initialisierung einer K-Komponente durchgeführt werden, bevor die eigentlichen Leistungen zugegriffen werden können. Die Methode zur Initialisierung muss in diesem Fall vor allen anderen aufgerufen werden.
Analog verhält es sich potentiell mit bestimmten Eventhandlern. Nicht jedes Ereignis kann zu einem beliebigen Zeitpunkt verarbeitet werden, sondern bestimmte Rahmenbedingungen können Voraussetzung für einen Eventhandler sein.
Diese Zusammenhänge und Abhängigkeiten unter den einzelnen Methoden und Eventhandlern eines Dienstes werden über die definierten *Protokolle* aufgezeigt. Ein Protokoll bestimmt die möglichen Aufrufreihenfolgen der Methoden und Eventhandler eines Dienstes, wobei ein Protokoll Methoden und Eventhandler gleichberechtigt berücksichtigt. Es wird nicht zwischen Protokollen für Methoden und Eventhandler unterschieden. Durch Angabe eines Protokolls wissen potentielle Nutzer eines Dienstes genau, wie eine korrekte Kommunikation zu erfolgen hat.
Wird die definierte Aufrufreihenfolge nicht eingehalten, so dürfen außerhalb dieser Ordnung erfolgte Methoden- oder Eventhandleraufrufe nicht ausgeführt werden. Hierfür müssen die implementierenden K-Komponenten eines Dienstes Sorge tragen.
Unterschieden wird dabei zwischen zwei Arten von Protokollen: *Kontroll-* und *Sessionprotokolle*. Im Fall eines Kontrollprotokolls gilt der aktuelle Kommunikationszustand global für alle nutzenden K-Komponenten, wohingegen das Sessionprotokoll pro Nutzer einer K-Komponente einzuhalten ist. Das Sessionprotokoll ist also ein Einbenutzerprotokoll, das für jede nutzende K-Komponente einen separaten Kommunikationszustand vermerkt.

3.2.8 Dienst-Importdefinitionen

Es kann für einen Dienst charakteristisch sein, auf fremde Dienste angewiesen zu sein. Das ist der Fall, wenn explizit Leistungen eines anderen Dienstes benutzt werden sollen, z. B. um mehrere Dienste zusammenzuführen und zusätzlich die Ergebnisse zu veredeln. Für dieses Szenario muss ein Dienst die zu importierenden Fremddienste kennen.
Eine *Dienst-Importdefinition* umfasst den eindeutigen Dienstbezeichner, sowie den pro importierten Dienst eindeutigen Rollennamen des Imports. Die Verwendung von Rollen ist dadurch begründet, dass auf diesem Weg mehrere Imports des gleichen Dienstes unterscheidbar erfolgen können. Besitzt ein Dienst nur genau einen Import pro fremden Dienst, dann ist die Angabe eines Rollennamens optional.
Beispiel für den mehrfachen Import eines Dienstes in verschiedenen Rollen ist das *Quelle-Ziel-Szenario*. Quelle und Ziel sind Entitäten des gleichen Typs, durch ihre Rolle werden sie aber unterschieden. Beispielsweise sollen Daten von einer Datenbank in eine andere transportiert werden. Der Transportdienst greift dafür zweimal auf den gleichen Datenbankdienst zurück, der in seiner späteren Ausprägung zwei verschiedene Datenbanken zugreifbar macht. Über die Rollenzuweisung können die beiden Datenbanken unterschieden werden.
Zu jedem importierten Dienst in einer Rolle gehört die Angabe der Multiplizität. Die Multiplizität eines Imports ist die Festlegung der erlaubten Grenzwerte der Kardinalitäten in einem gegebenen Schaltplan. In den meisten Fällen beträgt diese genau eins, höhere Multiplizitäten sind jedoch möglich. Als Beispiel kann ein Dienst angeführt werden, der die Ergebnisse von unbestimmt vielen K-Komponenten eines bestimmten Dienstes aufnimmt und aggregiert.
Die Angabe einer natürlichsprachlichen Beschreibung des Zweckes des Dienst-Imports schließt eine Dienst-Importdefinition ab.

3.2.9 Constraints

Ein Dienst kann an sich selbst und zu importierende fremden Dienste weitergehende Bedingungen aufstellen. Diese Bedingungen heißen *Constraints*. Es gibt zwei Arten von Constraints, die zu unterscheiden sind. Dabei handelt es sich um *Eigenschaften-* und *Import-Constraints*.

Eigenschaften-Constraints

Ein *Eigenschaften-Constraint* ist eine auswertbare boolsche Bedingung über den Eigenschaften. Grundlage für diese Constraints sind die Eigenschaften des Dienstes selbst und die weiteren Eigenschaften der importierten Dienste. Innerhalb von Eigenschaften-Constraints können sowohl Read- wie auch Write-Eigenschaften genutzt werden, um die gewünschten Bedingungen aufzustellen.
Durch diese Constraints können einschränkende Aussagen über eigene und fremde Eigenschaften eines Dienstes getroffen werden, die implementierende K-Komponenten einhalten müssen. Vor allem können über Eigenschaften-Constraints die geforderten Besonderheiten der Imports eines Dienstes näher spezifiziert werden.
Angenommen es gibt einen importierten Datenbankdienst, der per Eigenschaft darüber Auskunft gibt, welche Datenbanken unterstützt werden. Beispiel für ein Eigenschaften-Constraint kann dann die Forderung nach Unterstützung einer bestimmten Datenbank, z. B. Firebird, sein.

Import-Constraints

Neben der Formulierung von Constraints auf Eigenschaften sind auch Bedingungen struktureller Art möglich. Diese Anforderungen heißen *Import-Constraints* und sind Bedingungen über die eigenen Dienst-Importdefinitionen und die der zu importierenden Dienste. Sie sind auswertbare boolsche Ausdrücke, die die Gleichheit oder Ungleichheit bestimmter Imports ausdrücken können.
Vorstellbar ist ein Dienst, der direkt auf einen Datenbankdienst angewiesen ist und gleichzeitig auf einen weiteren Fremddienst, der wiederum auf einer Datenbank arbeitet. Über einen Import-Constraint kann formuliert werden, dass der Datenbankdienst des Ausgangsdienstes und der des Fremddienstes identisch sein müssen.

3.3 K-Komponenten

Eine *K-Komponente* ist die Implementierung eines oder mehrerer Dienste. Die durch die Dienste definierten syntaktischen und semantischen Vorgaben müssen durch die K-Komponente erfüllt werden. Die Erfüllung der einzelnen implementierten Dienste ist getrennt voneinander zu betrachten. Die veröffentlichten Dienste einer K-Komponente weisen untereinander keine Abhängigkeiten auf.
Aufgrund der einzelnen definierten Dienstsignaturen ergeben sich implizit die

Hauptmerkmale von K-Komponenten. Zusätzlich können K-Komponenten weitere Aspekte aufweisen, die zusammen mit den Hauptmerkmalen in den folgenden Abschnitten vorgestellt werden. In Abbildung 3.1 wird der Zusammenhang zwischen K-Komponenten und Diensten aufgezeigt.
Ein Dienst umfasst wie im letzten Abschnitt erläutert die Definition von Eigenschaften, Methoden, Events, Eventhandlern und Dienst-Imports, sowie die aufgestellten Protokolle und Constraints.
Eine K-Komponente implementiert mindestens einen Dienst, potentiell beliebig viele. Gemäß der Definitionen der Dienste muss die K-Komponente Eigenschaften, Methoden, Events, Eventhandler und Dienst-Imports umsetzen, sowie die Einhaltung der Protokolle und Constraints gewährleisten.

Alle diese Punkte werden im Folgenden einzeln näher betrachtet.

3.3.1 Eigenschaften

K-Komponenten besitzen mindestens die je Dienst definierten *Eigenschaften*. Diese Eigenschaften müssen der **Definition entsprechen** und **vollständig** sein. D. h. zu jeder pro Dienst vorkommenden Eigenschaftendefinition muss genau eine Eigenschaft in der K-Komponente existieren, die zunächst den eindeutigen Bezeichner aufweist. Eindeutigkeit bezogen auf eine K-Komponente ist gegeben über den Dienstbezeichner und den dann eindeutigen Bezeichner der Eigenschaft. Darüber hinaus müssen der Wertetyp und die Eigenschaftenart (Write/Read-Eigenschaft) übereinstimmen.

Zusätzlich zu den im Dienst definierten Eigenschaften dürfen K-Komponenten weitere Write-Eigenschaften besitzen. Analog zur Eigenschaftendefinition im Dienst muss neben der Festlegung des Wertetyps natürlichsprachlich die Semantik der Zusatzeigenschaft beschrieben werden. Solche Zusatzeigenschaften dürfen das im Dienst definierte Verhalten nicht verändern. Sie bieten die Möglichkeit, die speziellen Charakteristiken der Implementation auszudrücken und diese konfektionierbar zu machen.

Read-Eigenschaften sind als Zusatzeigenschaften nicht sinnvoll, da sie im Dienst bekannt sein müssen, um aus ihnen Nutzen zu ziehen. Wird ein Constraint über Eigenschaften eines Imports aufgestellt, so sind nur die im Dienst definierten Eigenschaften bekannt. Die Zusatzeigenschaften können hier keine Verwendung finden. Da Read-Eigenschaften aber nur in Zusammenhang mit Constraints zu nutzen sind, benötigt man sie nicht als Zusatzeigenschaft.

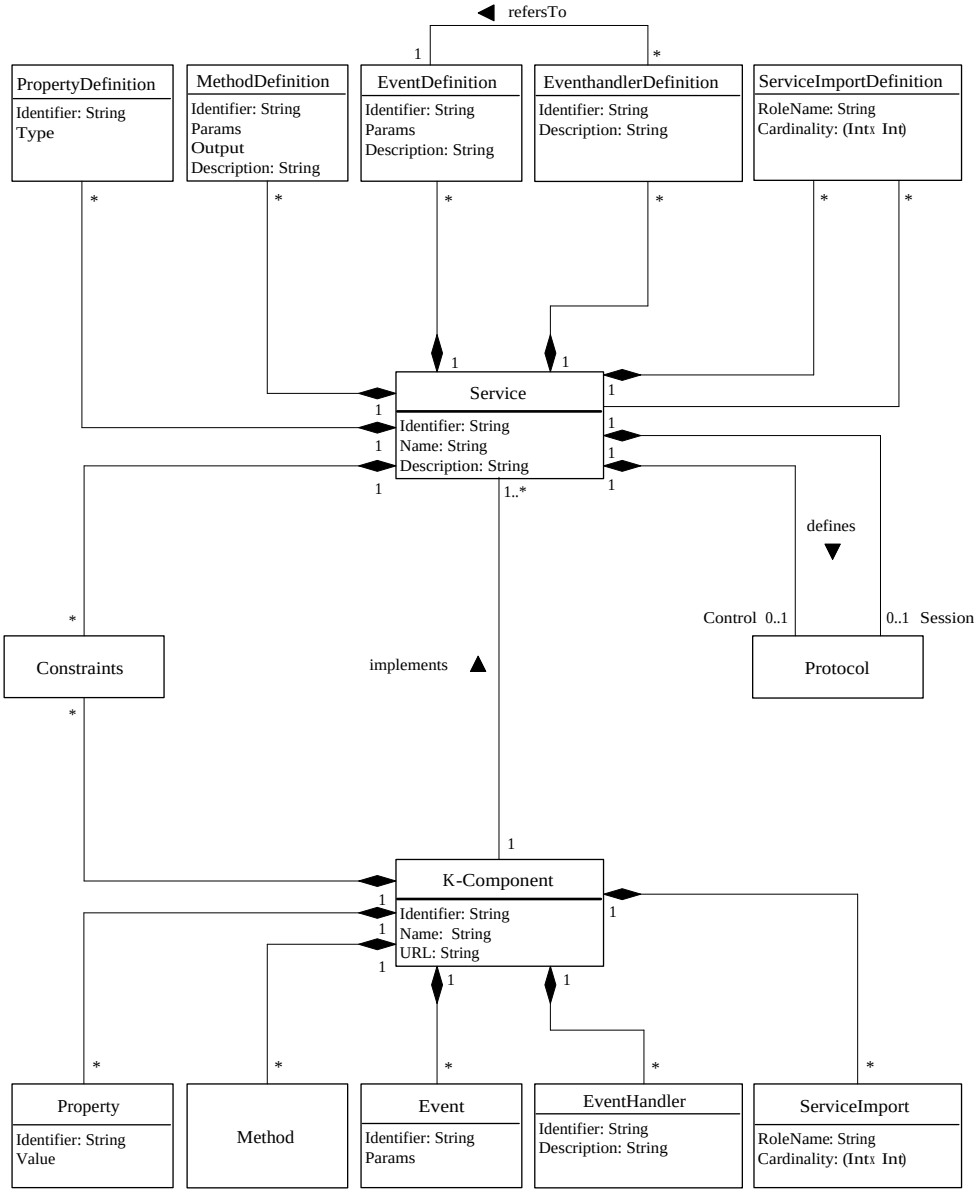

Abbildung 3.1: Dienst und K-Komponente

3.3.2 Methoden

Ebenso wie bei den Eigenschaften muss eine K-Komponente alle im Dienst definierten *Methoden* **korrekt** und **vollständig** implementieren. Korrekt ist eine Methodenimplementation genau dann, wenn Sie der im Dienst definierten Syntax und Semantik entspricht. Vollständig sind die Methoden genau dann, wenn jede im Dienst definierte Methode genau einmal in der K-Komponente implementiert worden ist. Die Überprüfung auf Vollständigkeit kann über den vollständig qualifizierten Methodenbezeichner erfolgen, da diese eindeutig sein müssen. Der vollständig qualifizierte Methodenbezeichner umfasst den Bezeichner des Dienstes, innerhalb dessen die Methodendefinition erfolgt ist.

3.3.3 Events

Ein *Event* ist die Veröffentlichung eines durch eine K-Komponente wahrgenommenen Ereignisses. Zu jedem Event einer K-Komponente können beliebig viele passende Eventhandler (vgl. 3.3.4) zugeordnet werden. Tritt das durch ein Event repräsentierte Ereignis ein, so wird das Event ausgelöst, wodurch an jeden registrierten Eventhandler eine Nachricht mit den aus der Eventdefinition bekannten Parametern und zusätzlich Angaben zur Eventquelle verschickt werden.
Eine K-Komponente muss die Eventdefinitionen des Dienstes **korrekt** und **vollständig** umsetzen. Eine Eventdefinition ist genau dann korrekt umgesetzt, wenn die Syntax und die Semantik der Definition durch den Event der K-Komponente erfüllt werden. Alle Eventdefinitionen sind genau dann vollständig umgesetzt, wenn zu jeder Eventdefinition des Dienstes genau eine passende Eventimplementation in der K-Komponente existiert. Die Überprüfung der Vollständigkeit der Events kann über den vollständig qualifizierten Eventbezeichner vorgenommen werden, da dieser eindeutig sein muss.

3.3.4 Eventhandler

Ein *Eventhandler* ist die Möglichkeit einer K-Komponente, auf bestimmte Events reagieren und diese verarbeiten zu können. Zu jedem Eventhandler können beliebig viele Eventquellen zugeordnet werden. Über die in den empfangenen Nachrichten enthaltenen Absenderinformationen sind diese eindeutig einer Quelle zuzuordnen.
Die Eventhandlerdefinitionen des Dienstes müssen durch eine K-Komponente **korrekt** und **vollständig** erfüllt werden. Korrekt ist die Umsetzung einer Eventhandlerdefinition genau dann, wenn sie der definierten Syntax und Se-

mantik entspricht. Vollständig sind die Eventhandler genau dann, wenn jeder pro Dienst definierten Eventhandler implementiert ist. Die Überprüfung auf Vollständigkeit kann über die eindeutige Kombination aus Dienst- und Eventhandlerbezeichnung vorgenommen werden.

3.3.5 Protokolleinhaltung

Die Einhaltung der im Dienst definierten Protokolle ist Aufgabe der implementierenden K-Komponente. Die K-Komponente muss sicherstellen, dass die durch die Protokolle definierte erlaubte Aufrufreihenfolge der Methoden und Eventhandler eingehalten wird. Sollte die Aufrufreihenfolge verletzt werden, so darf die angeforderte Methode oder der Eventhandler nicht ausgeführt werden. Ein Mechanismus zur Fehlerbehandlung ist hier wünschenswert.

Die Unterscheidung in Session- und Kontrollprotokoll muss ebenfalls durch die K-Komponente vorgenommen werden. In Sessionprotokollen gilt die erlaubte Aufrufreihenfolge lokal für *einen*, bei Kontrollprotokollen dagegen global für *alle* Nutzer.

3.3.6 Dienst-Imports

Eine K-Komponente kann auf fremde K-Komponenten angewiesen sein. Diese Abhängigkeit kann einen von zwei Gründen besitzen.

Der erste Grund ergibt sich aus den Dienst-Importdefinitionen der Dienste einer K-Komponente. Diese Importdefinitionen müssen durch die K-Komponente erfüllt werden. Zu jeder Dienst-Importdefinition muss es in der K-Komponente genau einen entsprechenden *Dienst-Import* geben. Der zu importierende Dienst, die Rollenbezeichnung und die Multiplizität müssen übereinstimmen.

Der zweite mögliche Grund liegt in der Implementation der K-Komponente. Um die eigenen Dienste zu erfüllen, kann eine K-Komponente auf Leistungen anderer K-Komponenten zurückgreifen. Dabei wird aber nicht angegeben, welche Fremdkomponten benötigt werden, sondern welche Dienste. Denn über Dienste ist die Leistung jeder benötigten Fremdkomponente eindeutig spezifiziert. Diese Imports sind nicht im Dienst definiert, da es von der K-Komponente abhängt, wie sie die eigenen Dienste erfüllt. Die Imports sind in diesem Fall nicht charakteristisch für den Dienst, sondern für die spezielle K-Komponente.
Analog zur Importdefinition gehört zu diesen zusätzlichen Dienst-Importen

die Angabe des Dienstes und die Vergabe eines eindeutigen Rollenbezeichners. Die Kardinalität ist jedoch nicht frei wählbar, sie beträgt immer genau eins.

3.3.7 Constraints

Die Constraints einer K-Komponente verlaufen analog zu denen eines Dienstes. Aussagen über Eigenschaften und auch die Importstruktur können mit ihrer Hilfe getroffen werden.
Eine K-Komponente übernimmt zunächst alle Constraints des Dienstes. Neben diesen können weitere verfasst werden, um so die Besonderheiten der Dienst-Implementierung durch die K-Komponente zu berücksichtigen. Diese Zusatzconstraints ergänzen die im Dienst erfassten Bedingungen und sollten diesen nicht widersprechen. Im Fall eines Widerspruchs könnte die K-Komponente niemals in Betrieb genommen werden, da zur Laufzeit immer alle Constraints erfüllt sein müssen.

K-Komponenten stellen einzelne Funktionseinheiten dar, deren Leistung über Dienste spezifiziert ist. Der folgende Abschnitt beschäftigt sich mit der Assemblierung eines vollständigen Systems bestehend aus einzelnen K-Komponenten.

3.4 Systemassemblierung

In den letzten beiden Abschnitten wurden K-Komponenten und die zugehörigen Dienste im Detail vorgestellt. Der nächste Schritt ist die **Assemblierung** eines gesamten Softwaresystems. Für den Zweck der Beschreibung eines ganzen Systems anhand von K-Komponenten, wird ein weiteres Mittel benötigt, der *Komponentenschaltplan*.

3.4.1 Übersicht über Komponentenschaltpläne

In einen Schaltplan werden beliebig viele K-Komponenten aufgenommen, die untereinander so verschaltet werden, dass sie die gewünschte Gesamtfunktionalität des Systems verwirklichen.
Die Elemente eines Schaltplans sind K-Komponenten mitsamt ihren enthaltenen Eigenschaften (Properties), Eventhandlern, Events und Dienst-Imports (Service-Imports). Des Weiteren sind die Kopplungen zwischen diesen Elementen Teil eines Schaltplans. Die globalen Konfektionierungsparameter (Global Parameters) und Glassbox-Komponenten, die dem Schaltplan Skripting-

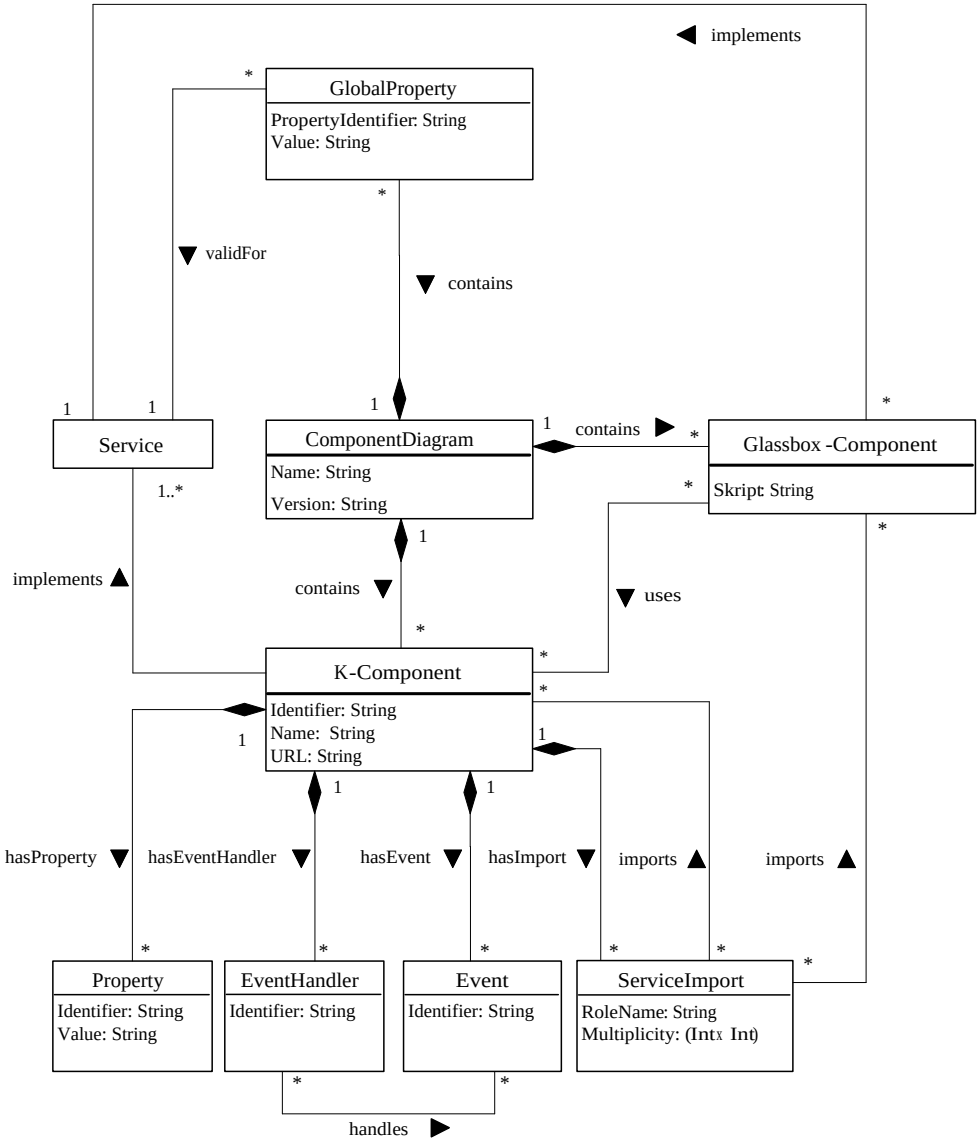

Abbildung 3.2: Schema Komponentenschaltplan

fähigkeiten verleihen (vgl. 3.4.4), komplettieren das Schema. Abbildung 3.2 enthält das Schema für Komponentenschaltpläne als UML-Klassendiagramm.

Nach dieser Übersicht über Komponentenschaltpläne werden in den folgenden Abschnitten alle Merkmale eines Komponentenschaltplans detailliert beschrieben. Zu diesen Merkmalen zählen *Kopplungen*, *Konfektionierung*, *Glassboxkomponenten* und *leere Schaltplankomponenten*.
Zunächst werden die verschiedenen Kopplungsformen eines Komponentenschaltplans betrachtet.

3.4.2 Kopplungen

Die Hauptaufgabe eines Schaltplans ist die Assemblierung von K-Komponenten. Die notwendigen Verschaltungen werden *Kopplungen* genannt. Es existieren zwei Arten von Kopplungen, **Dienst-Import-Kopplungen**und **Event-Handler-Kopplungen**. Beide Arten werden nachfolgend näher beschrieben.

Dienst-Import-Kopplungen

K-Komponenten können beliebig viele Dienst-Imports besitzen, über die sie mit fremden K-Komponenten verbunden werden können. Diese Verbindungen heißen *Dienst-Import-Kopplungen*.
Eine Dienst-Import-Kopplung dient dem Zweck, dass die importierende K-Komponente auf die Methoden der importierten K-Komponente zugreifen kann.

Eine Dienst-Import-Kopplung kann genau dann vorgenommen werden, wenn

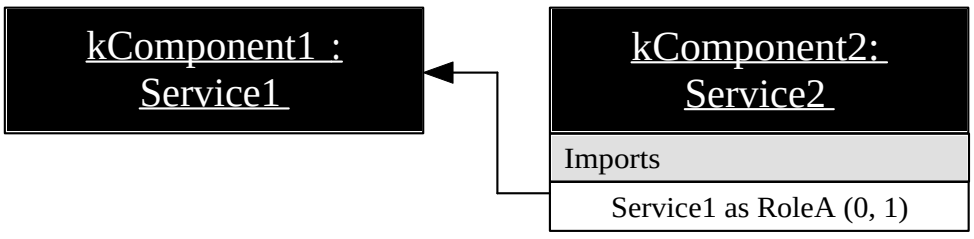

Abbildung 3.3: Skizze Dienst-Import-Kopplung

die zu importierende K-Komponente mindestens den Dienst implementiert, den der Dienst-Import vorschreibt (vgl. 3.2.8). Dabei muss auf die Einhaltung der Importmultiplizitäten des Dienst-Imports geachtet werden. Durch diese

Multiplizität werden die erlaubten Grenzwerte für Dienst-Import-Kopplungen vorgegeben.

In Abbildung 3.3 wird die Kopplung des zu importierenden Dienstes `Service1` der K-Komponente `kComponent2` mit der K-Komponente `kComponent1` skizziert. Die Darstellung sagt aus, dass `kComponent2` einen Dienst-Import des Dienstes `Service1` in der Rolle `RoleA` besitzt. Die Multiplizität des Imports beträgt null bis eins. Der Import wird erfüllt durch die Kopplung an die K-Komponente `kComponent1`, die den Dienst `Service1` implementiert. Aus der Kopplung resultiert, dass `kComponent1` den entsprechenden Dienst-Import von `kComponent2` erfüllt, wodurch diese auf die Methoden der importierten K-Komponente zugreifen kann.
Eine ausführliche Beschreibung der verwendeten Notation der Skizze folgt in Abschnitt 3.5.3.

Event-Handler-Kopplungen

K-Komponenten können beliebig viele Events und Eventhandler besitzen. Die Verbindung eines Eventhandlers mit einem Event heißt *Event-Handler-Kopplung*.
Eine Event-Handler-Kopplung dient dem Zweck, dass der Eventhandler davon in Kenntnis gesetzt wird, sobald das zugeordnete Event ausgelöst wird.

Ein **Eventhandler** verarbeitet nur eine bestimmte Sorte von Events, die

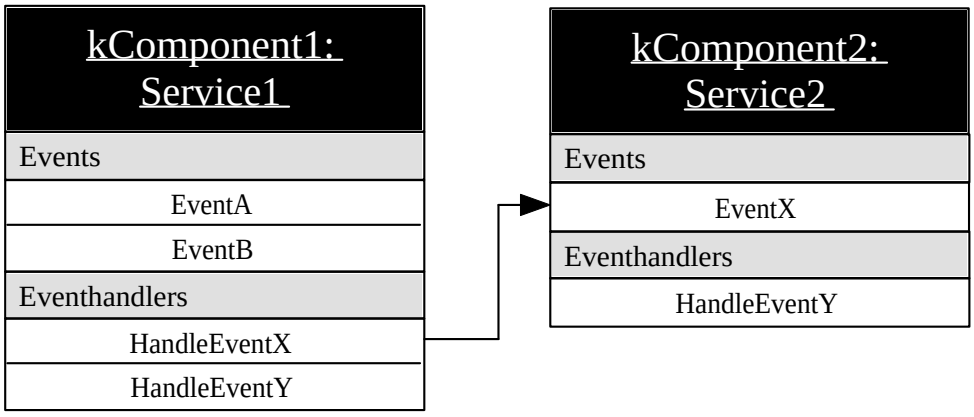

Abbildung 3.4: Skizze Event-Handler-Kopplung

über eine angegebene Eventdefinition festgelegt ist (vgl. 3.2.6). Ein **Event** einer K-Komponente basiert auf einer Eventdefinition des implementierten

Dienstes (vgl. 3.2.5). Eine **Event-Handler-Kopplung** kann genau dann vorgenommen werden, wenn die Eventdefinition des Eventhandlers mit der des Events übereinstimmt.
Bei Auslösen eines Events werden alle aufgeschalteten Eventhandler durch Übermittlung der Eventnachricht gemäß der Eventdefinition über das Eintreten des repräsentierten Ereignisses informiert.
In einem Schaltplan werden Eventkopplungen so vorgenommen, dass das gewünschte Verhalten erzielt wird. Der Ersteller des Schaltplans muss demnach wissen, welche K-Komponenten von welchen Ereignissen benachrichtigt werden soll.

In Abbildung 3.4 wird eine Event-Handler-Kopplung dargestellt. Der Eventhandler `HandleEventX` der K-Komponente `kComponent1` wird mit dem Event `EventX` der K-Komponente `kComponent2` gekoppelt. D. h. bei Auslösen des Events `EventX` wird der Eventhandler `HandleEventX` benachrichtigt die Verarbeitung des Events durch den zugeordneten Eventhandler kann erfolgen.

3.4.3 Konfektionierung

Ein Schaltplan beschreibt ein konkretes System. K-Komponenten sind allerdings meistens nicht auf ein konkretes Einsatzszenario beschränkt, sondern können an einen gegebenen Kontext angepasst werden. Diese Anpassung geschieht über die Write-Eigenschaften der K-Komponenten und heißt *Konfektionierung*.

Lokale Konfektionierung

Von *lokaler* Konfektionierung spricht man, wenn die Write-Eigenschaften einer K-Komponente im Schaltplan direkt gesetzt werden. Die so gesetzten Eigenschaften gelten nur für die betreffende K-Komponente.

Globale Konfektionierung

Die Konfektionierung einer Eigenschaft heißt genau dann *global*, wenn sie für den gesamten Schaltplan vorgenommen wird. Hierzu wird nicht der Eigenschaftswert einer bestimmten K-Komponente gesetzt, sondern global für den gesamten Schaltplan. Eindeutig identifiziert wird eine zu setzende Write-Eigenschaft über den Dienst und ihren Bezeichner. Ein so gesetzter Wert gilt für alle K-Komponenten, die den angegeben Dienst implementieren. Übersteuert werden kann eine globale Konfektionierung durch Angabe eines lokalen Wertes.

3.4.4 Glassbox-Komponenten

In den meisten Fällen liefern K-Komponenten die Funktionalität des Systems. K-Komponenten stellen **Blackboxes** dar, denn sie verstecken ihre Implementation und Verfahrensweise gegenüber allen anderen. In manchen Fällen möchte man aber nicht auf eine Blackbox angewiesen sein, wenn es um die Erfüllung einer eher **trivialen Aufgabe** geht. Hier bietet sich ein Glassbox-Ansatz an, um möglichst einfach benötigte Funktionalität einem Schaltplan hinzufügen zu können. Die Erfassung der Glassbox-Logik erfolgt per Angabe eines **Skripts**.

Prinzipiell besitzt die Verwendung von Glassbox-Komponenten zwei Aspekte. Der erste Aspekt ist die Implementierung **einfacher Logik**. Für eine einfache und übersichtliche K-Komponente muss keine eher komplexe Blackbox-K-Komponente erstellt werden, sondern eine per einfachem Skripting implementierte und sofort einsatzbereite Glassbox-Komponente ist oft die bessere Alternative.
Der zweite Aspekt bezieht sich auf das **Zusammenführen der Leistungen** von mehreren Fremdkomponenten. Mithilfe des Skriptings können diese Leistungen gebündelt und mit zusätzlicher Logik versehen werden.

Eine Glassbox-Komponente implementiert genau **einen Dienst**. Die Implementation eines Dienstes ist notwendig, damit K-Komponenten auf die Leistungen (Methoden) der Glassbox-Komponente zugreifen können. Die Beschränkung auf einen Dienst entspringt der eher einfachen Komplexität der Skripting-Komponenten.

Glassbox-Komponenten können als **Adapter** oder auch als **einfache logische Bausteine** eingesetzt werden.
Die Aufgabe eines Adapters besteht darin, K-Komponenten, die miteinander verbunden werden sollen, deren Import-Schnittstellen aber nicht zueinander passen, dennoch indirekt miteinander zu koppeln. Dazu wird eine Glassbox-Komponente geschaffen, die die zu importierende K-Komponente aufnimmt und deren Schnittstelle in die gewünschte Form überträgt. Der implementierte Dienst der Glassbox-Komponente entspricht genau dem durch die Import-Schnittstelle geforderten Dienst. Dieses Vorgehen ist nur möglich, falls die Leistungen der importierten K-Komponente in den entsprechenden Dienst konvertiert werden können.
Neben den Adaptionsaufgaben sind Glassbox-Komponenten auch das einzusetzende Mittel, um einem Komponentenschaltplan einfache Logik und Berechnungsfähigkeiten hinzuzufügen. Die Berechnungsmächtigkeit der Glass-

box-Komponenten hängt dabei von der eingesetzten Skriptingsprache und dem -interpreter ab.

In einem Komponentenschaltplan soll eine Glassbox-Komponente praktisch **gleichberechtigt** zu K-Komponenten genutzt werden können. Für eine konsumierende K-Komponente sollten sich bei der Nutzung keine Unterschiede ergeben.

3.4.5 Leere Schaltplankomponenten

Ein Komponentenschaltplan besteht aus K-Komponenten und Kopplungen. Bei der Aufstellung eines Schaltplans kann aber der Zustand auftreten, dass der Systemassemblierer bereits einen Dienst identifiziert und selektiert hat, der in den Schaltplan aufgenommen werden soll. Da aber nur K-Komponenten und **nicht Dienste** Teil eines Schaltplans sind, wird an dieser Stelle ein **Hilfskonstrukt** benötigt.
Eine *leere Schaltplankomponente* ist ein **Platzhalter** für eine K-Komponente innerhalb eines Schaltplans. Sie wird bestimmt durch mindestens einen ausgewählten Dienst. Die leere Schaltplankomponente weist alle Eigenschaften auf, die in den zugeordneten Diensten spezifiziert sind und kann wie eine K-Komponente genutzt und gekoppelt werden.

Auf diesem Weg können in einen Schaltplan praktisch Dienste aufgenommen werden, ohne ihnen direkt eine erfüllende K-Komponente zuzuweisen. Dieses Vorgehen ist sinnvoll, wenn in ersten Schritten bei der Erstellung eines Schaltplans zunächst die **zentralen Dienste identifiziert** und in den Schaltplan übernommen werden sollen. Zu einem späteren Zeitpunkt müssen dann diese Platzhalter durch K-Komponenten ausgefüllt werden.
Ein Schaltplan, der leere Schaltplankomponenten enthält, kann nicht ausgeführt werden, da hierfür Vollständigkeit gefordert ist. Ein Komponentenschaltplan ist genau dann *vollständig*, wenn er keine leeren Schaltplankomponenten enthält.

Nachdem dieser Abschnitt einen detaillierten Überblick über den Komponentenschaltplan und dessen Merkmale gegeben hat, widmet sich der folgende Abschnitt dem Vorgang des Aufstellens und der Ausführung eines fertigen Schaltplans.

3.5 Repräsentation von Diensten, K-Komponenten und Schaltplänen

Dienste, K-Komponenten und Komponentenschaltpläne müssen in einer festgelegten Form beschrieben werden können. In diesem Abschnitt werden sowohl textuelle Repräsentationen für Dienste und K-Komponenten vorgestellt, wie auch eine grafische Notation für Komponentenschaltpläne.

3.5.1 Textuelle Repräsentation eines Dienstes

Ein Dienst ist prinzipiell die Spezifikation eines Softwarebausteins. Ausgehend von dieser Feststellung wird in diesem Abschnitt eine Vorlage für die textuelle Repräsentation vorgestellt, die alle Elemente eines Dienstes aufgreift.

Bezeichner
Der systemweit eindeutige Bezeichner des Dienstes.

Beschreibung
Die natürlichsprachliche Beschreibung der Aufgaben, die der Dienst kapselt.

Imports
Die Auflistung aller Dienst-Imports des Dienstes mitsamt allen benötigten Informationen.

Dienst	*Der Dienstbezeichner des importierten Dienstes*
Rolle	*Die bezogen auf die Imports des Dienstes eindeutige Rolle*
Multiplizität	*Die Angabe der Multiplizität in Form eines zweistelligen Integer-Tupels bestehend aus dem minimalen und maximalen Grenzwert der Kardinalitäten im Schaltplan*
Beschreibung	*Die natürlichsprachliche Beschreibung des Zweckes des Imports*

Datentypdefinitionen
Die Auflistung aller Datentypdefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Datentypen des Dienstes eindeutige Bezeichner*
Struktur	*Die eigentliche definierte Datenstruktur. Die Definition einer Datenstruktur ist eine Menge von Bezeichner-Typ-Tupeln, wobei der Typ ein einfacher Typ sein kann (Boolean, Integer, Real, String, etc.) oder ein in einem Dienst definierter Datentyp. In letzterem Fall muss der vollqualifizierte Datentypbezeichner angegeben werden.*

Exceptions
Die Auflistung aller Exceptiondefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Exceptions des Dienstes eindeutige Bezeichner*
Parameter	*Die Auflistung der Exceptionparameter in der Form* `Parametertyp Parameterbezeichner`
Beschreibung	*Die natürlichsprachliche Beschreibung des Fehlerfalles, den die Exception repräsentiert.*

Eigenschaften
Die Auflistung aller Eigenschaftendefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Eigenschaften des Dienstes eindeutige Bezeichner*
Typ	*Der Typ der Eigenschaft*
Zugriff	*Die Angabe der Zugriffsart der Eigenschaft:* `Read` *oder* `Write`
Exceptions	*Die Auflistung aller Exceptions, die durch diese (Write-)Eigenschaft geworfen werden können.*
Beschreibung	*Die natürlichsprachliche Beschreibung der Eigenschaft*

Methoden
Die Auflistung aller Methodendefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Methoden des Dienstes eindeutige Bezeichner*
Parameter	*Die Auflistung der Methodenparameter in der Form* `Parametertyp Parameterbezeichner`
Rückgabewerte	*Die Auflistung der benannten Rückgabewerte der Methode in der Form* `Werttyp Wertbezeichner`. *Wird kein Rückgabewert geliefert, so erfolgt an dieser Stelle die Angabe* `void`
Exceptions	*Die Auflistung aller Exceptions, die durch diese Methode geworfen werden können.*
Beschreibung	*Die natürlichsprachliche Beschreibung der Semantik der Methode*

Events
Die Auflistung aller Eventdefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Events des Dienstes eindeutige Bezeichner*
Parameter	*Eine Auflistung der Eventparameter in der Form* `Parametertyp Parameterbezeichner`
Beschreibung	*Die natürlichsprachliche Beschreibung des auslösenden Ereignisses und der übergebenen Parameter*

Eventhandler
Die Auflistung aller Eventhandlerdefinitionen des Dienstes mitsamt allen benötigten Informationen.

Bezeichner	*Der bezogen auf die Eventhandler des Dienstes eindeutige Bezeichner*
Event	*Die Angabe des Events, auf den der Eventhandler reagieren kann. Die Angabe erfolgt in der Form* `Dienstbezeichner.Eventbezeichner`
Beschreibung	*Die natürlichsprachliche Beschreibung der Semantik des Eventhandlers*

Kontrollprotokoll
Die Definition des Kontrollprotokolls. Sie erfolgt durch die Angabe eines regulären Ausdruckes über den Methoden- und Eventhandlerbezeichnern. Ist die Aufrufreihenfolge bezogen auf das Kontrollprotokoll beliebig, so muss das Protokoll nicht angegeben werden.

Sessionprotokoll
Die Definition des Sessionprotokolls. Sie erfolgt analog zum Kontrollprotokoll.

Constraints
Eine Auflistung aller Eigenschaften- und Import-Constraints.

Allgemeine Anmerkungen

Besitzt ein Dienst einen bestimmten Aspekt überhaupt nicht, so entfällt dieser in der Beschreibung vollständig. Umfasst ein Dienst z. B. keine Eigenschaftendefinitionen, so fehlt der Abschnitt *Eigenschaften* in der Dienstbeschreibung.

Bei Angabe von Datentypen wird zurückgegriffen auf einfache Datentypen wie String, Integer und Boolean, sowie auf die in den Diensten definierten komplexen Datentypen, wobei der vollständig qualifizierte Bezeichner des Datentyps verwendet wird. Der vollständig qualifizierte Datentypbezeichner umfasst den Dienstbezeichner und den eigentlichen Bezeichner des Datentyps.

Auf Basis der einfachen und komplexen Datentypen können zusätzlich Tupel gebildet werden, um zusammengesetzte Strukturen abzubilden. Ein Tupel wird konstruiert durch umschließende einfache Klammern `( )` und `x` als Separator der Tupelelemente. Beispiel für einen Tupeldatentyp ist `(ServiceXYZ.DataType1 x Integer)`, wobei `ServiceX.DataType1` ein in einem Dienst definierter komplexer Datentyp ist.

Des Weiteren ist die Bildung von Mengentypen möglich. Ein Mengentyp wird durch die Mengenklammern `{ }` und einen innerhalb der Klammern angegebenen Datentypen konstruiert. Beispiel für einen Mengentyp ist `{Integer}`. Dieser Typ entspricht einer Menge von Integers. Mengentypen werden wie alle anderen Typen behandelt und können in komplexen und zusammengesetzten Typen verwendet werden.

3.5.2 Textuelle Repräsentation einer K-Komponente

Eine K-Komponente wird hauptsächlich bestimmt durch die Dienste, die sie implementiert. Ergänzend dazu kann sie nur wenige Zusatzeigenschaften aufweisen. Daher fällt die textuelle Beschreibung im Vergleich zu der eines

Dienstes auch entsprechend knapp aus.

In diesem Abschnitt wird folgend eine Vorlage für die textuelle Beschreibung einer K-Komponente vorgestellt.

Bezeichner
Der systemweit eindeutige Bezeichner der K-Komponente.

Dienste
Die Auflistung der durch die K-Komponente implementierten Dienste.

Beschreibung
Eine optionale zu den implementierten Diensten ergänzende natürlichsprachliche Beschreibung der Besonderheiten der K-Komponente.

Eigenschaften
Die Auflistung aller zu den Diensten zusätzlichen Write-Eigenschaften der K-Komponente mitsamt allen benötigten Informationen. Eine K-Komponente besitzt keine zusätzlichen Read-Eigenschaften.

Bezeichner	*Der bezogen auf die Eigenschaften der K-Komponente eindeutige Bezeichner*
Typ	*Der Typ der Eigenschaft*
Beschreibung	*Die natürlichsprachliche Beschreibung der Eigenschaft*

Imports
Die Auflistung aller zu den Diensten zusätzlichen Imports der K-Komponente mitsamt allen benötigten Informationen.

Dienst	*Der Dienstbezeichner des importierten Dienstes*
Rolle	*Die bezogen auf die Imports der K-Komponente eindeutige Rolle*
Multiplizität	*Die Angabe der Multiplizität in Form eines zweistelligen Integer-Tupels bestehend aus dem minimalen und maximalen Grenzwert der Kardinalitäten im Schaltplan.*
Beschreibung	*Die natürlichsprachliche Beschreibung des Zweckes des Imports*

Constraints
Eine Auflistung aller zu den Diensten zusätzlichen Eigenschaften- und Struk-

turconstraints.

Nachdem sowohl die textuelle Repräsentationen für Dienste wie auch K-Komponenten vorgestellt worden sind, widmet sich der folgende Abschnitt der visuellen Darstellung ganzer Komponentenschaltpläne.

3.5.3 Grafische Repräsentation eines Schaltplans

Komponentenschaltpläne sind im Rahmen dieses Konzeptes das zentrale Mittel der Systemmodellierung. Oftmals wird das Modell eines Softwaresystems grafisch notiert, da diese Form der Darstellung viele Vorteile vor allem in Hinblick auf die Übersichtlichkeit bietet. In diesem Abschnitt wird eine Möglichkeit der visuellen Notation von Komponentenschaltplänen vorgestellt.

Benötigt wird zunächst eine grafische Repräsentation für K-Komponenten.

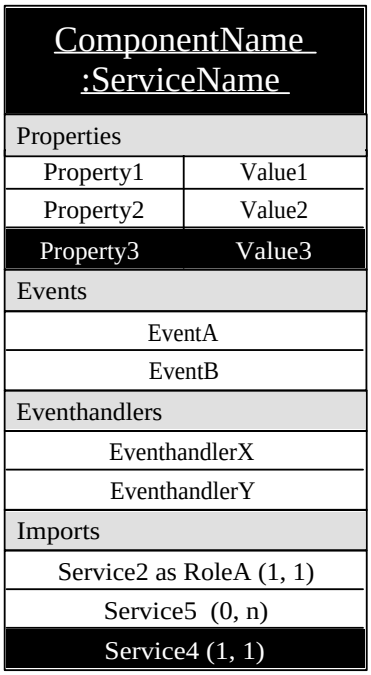

Abbildung 3.5: K-Komponente als Schaltplanelement

Abbildung 3.5 zeigt eine für einen Schaltplan **vollständig beschriebene** K-Komponente. Die einzelnen Elemente der Darstellung sind der Bezeichner der K-Komponente *ComponentName*, der Bezeichner des implementierten Dienstes *ServiceName*, sowie Auflistungen der Write-Eigenschaften, Events,

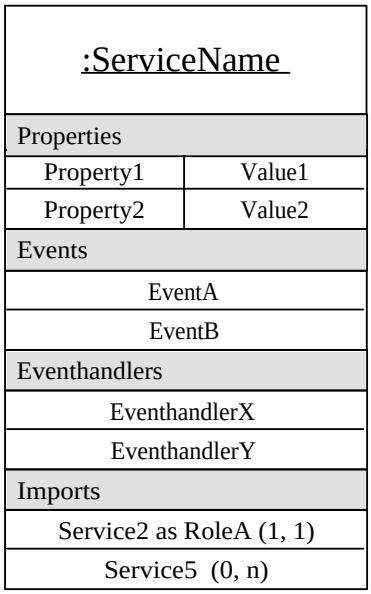

Abbildung 3.6: Leere Schaltplankomponente als Schaltplanelement

Eventhandler und Imports. Die Eigenschaften werden als *Name-Wert*-Paare angegeben, die Eventauflistung besteht nur aus den Eventbezeichnern. Ein Eventhandler wird repräsentiert durch den pro Dienst eindeutigen Eventhandler-Bezeichner, und ein Import setzt sich zusammen aus der Angabe des zu importierenden Dienstes, dem Rollennamen, der bei Diensteindeutigkeit optional ist, und der Multiplizität.
Die Dienste, die eine K-Komponente implementiert, bestimmen deren **minimale Eigenschaften**. Allerdings kann eine K-Komponente zusätzliche Eigenschaften aufweisen. In einem Schaltplan sollte visuell unterschieden werden können, welche Teile einer K-Komponente durch die implementierten Dienste vorgegeben und welche Zusatzelemente sind. Erreicht wird diese Markierung durch das Einfärben der betreffenden Elemente. Alle durch Dienste vorgegebenen Teile einer K-Komponente werden dargestellt durch **schwarze Schrift** auf weißem Grund. Handelt es sich um einen Zusatz der K-Komponente, so verhält es sich genau umgekehrt, es wird **weiße Schrift** auf schwarzem Grund verwendet.
In dem Beispiel aus Abbildung 3.5 ist dieser Umstand gut zu erkennen. Die Eigenschaft *Property3*, sowie der Import des Dienstes *Service4* sind nicht durch den Dienst definiert, sondern Besonderheit der K-Komponenten-Implementierung.
Die dargestellte K-Komponente enthält alle für einen Schaltplan relevanten Informationen. Da der Schaltplan eines ganzen Systems sehr umfangreich

werden kann, besteht die Möglichkeit, verschiedene Sichten darauf einzunehmen. Zu diesem Zweck können einzelne Segmente einer K-Komponenten-Repräsentation sozusagen ein- oder ausgeblendet werden (*Auslassung / Elision*). Durch die Angabe mehrerer Schaltplanvisualisierungen kann so das gesamte komplexe System dargestellt werden, ohne auf Übersichtlichkeit verzichten zu müssen.

Der Vorgang des Erstellens eines Schaltplans verläuft meist in mehreren Schritten. Zu Beginn der Modellierung kann zwar bereits feststehen, welche Dienste im Rahmen des Systems benötigt werden. Die Ausfüllung durch eine K-Komponente muss zu diesem Zeitpunkt allerdings noch nicht bekannt sein.
Aus diesem Grund muss es eine Möglichkeit geben, solche Umstände zu notieren. Abbildung 3.6 zeigt eine sozusagen *leere* Schaltplankomponente, die einen bestimmten Dienstes implementieren soll. Die Notation verläuft analog zu der einer tatsächlichen K-Komponente, bis auf den Unterschied, dass **nur dienstspezifische** Elemente dargestellt werden können. Eine so eingefügte K-Komponente ist im Prinzip ein **Platzhalter** für eine tatsächliche K-Komponente.
Damit ein Schaltplan **vollständig** und somit **ausführbar** ist, dürfen keine leeren K-Komponenten existieren. Dass es sich um eine leere K-Komponente handelt, wird durch das Weglassen des K-Komponenten-Bezeichners und der Verwendung von schwarzer Schrift auf weißem Grund gekennzeichnet. Analog dazu wird eine nicht leere K-Komponenten dargestellt durch die Verwendung von weißer Schrift auf schwarzem Grund.

Hauptaufgabe eines Komponentenschaltplans ist die **Verschaltung von K-Komponenten**. Dabei existieren zwei Arten von Kopplungen, die dargestellt werden müssen. Die erste Kopplungsart sind Verbindungen zwischen Events und Eventhandlern, also zwischen Ereignisquellen und -verarbeitern. Abbildung 3.7 enthält eine exemplarische Darstellung. Repräsentiert werden die Event-Aufschaltungen durch Kanten mit Pfeilspitzen, die von einem Eventhandler zu einem passenden Event verlaufen. In Abschnitt 3.4.2 wird besprochen, wann eine Event-Handler-Kopplung vorgenommen werden darf. Kopplungsflächen bieten die Seiten der Eventhandler und Events, wobei es keine Bedeutung hat, ob eine Kante mit der linken oder rechten Seite verbunden wird.
Die zweite Kopplungsart stellen die Imports dar. Dabei werden konsumierende K-Komponenten mit Fremdkomponenten verbunden, die die zu importierenden Dienste implementieren. In Abbildung 3.8 ist ein Beispiel für diese Verschaltungen enthalten. Analog zu den Eventkopplungen werden diese Ver-

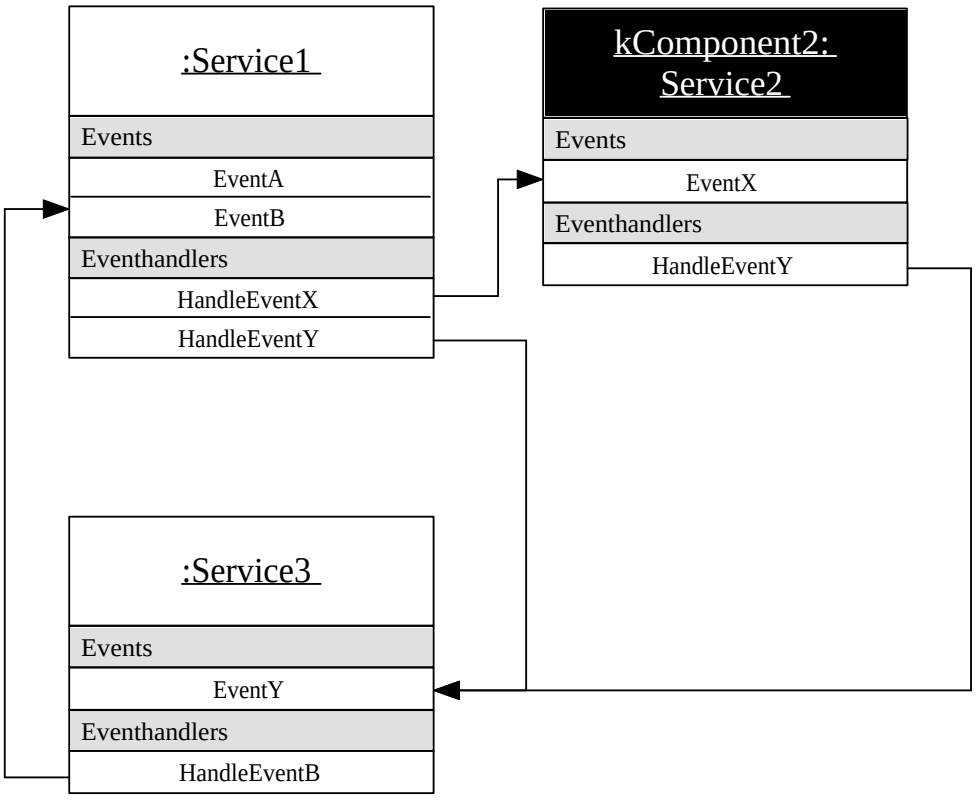

Abbildung 3.7: Event-Handler-Kopplungen im Schaltplan

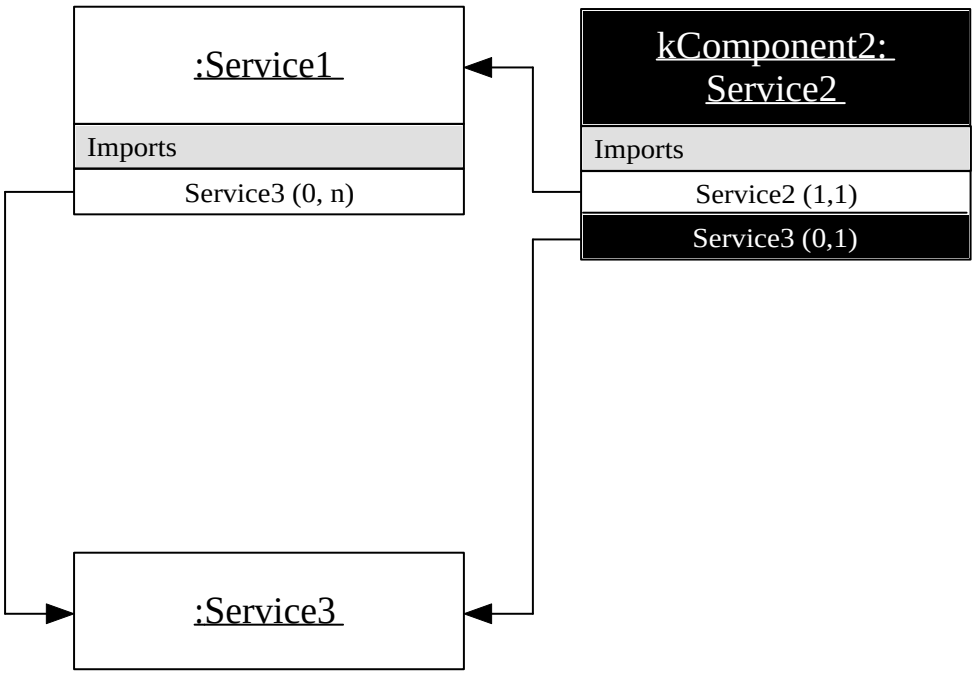

Abbildung 3.8: Dienst-Import-Kopplungen im Schaltplan

bindungen ebenfalls als Kanten mit Pfeilspitzen dargestellt, die jeweils von einer konsumierenden zu einer anbietenden K-Komponente verlaufen. Ebenso wie bei den Events kann die Startseite einer Importkante mit den Seiten eines Imports verbunden werden, die andere Seite muss an das Kopfelement einer passenden K-Komponente angeschlossen werden. In Abschnitt 3.4.2 wird besprochen, wann eine Dienst-Import-Kopplung vorgenommen werden darf.

Mit den vorgestellten Mitteln ist es bereits möglich, Komponentenschaltpläne für komplexe Systeme zu modellieren und übersichtlich zu gestalten. Für ein tatsächliches Diagramm sind **zusätzliche Angaben** allerdings wünschenswert. Interessant sind hier vor allem Angaben zu dem Titel des Schaltplans, den Verfassern, dem Datum der Erstellung, der Version des Plans, sowie dem relevanten Aspekt des Ausschnitts. Der Aspekt eines Schaltplans kann sich beziehen auf die eingeblendeten Informationen (Eigenschaften, Events, Imports) oder auch auf Ausschnitte des gesamten Systems. Wie solche Zusatzangaben aussehen können, zeigt Abbildung 3.9. In einem Informationsabschnitt des Diagramms sind alle allgemeinen Angaben zum Schaltplan enthalten.

Auf den ersten Blick kann ein Komponentenschaltplan an **UML-Klassendiagramme** erinnern. Die dargestellten Verhältnisse weisen allerdings eini-

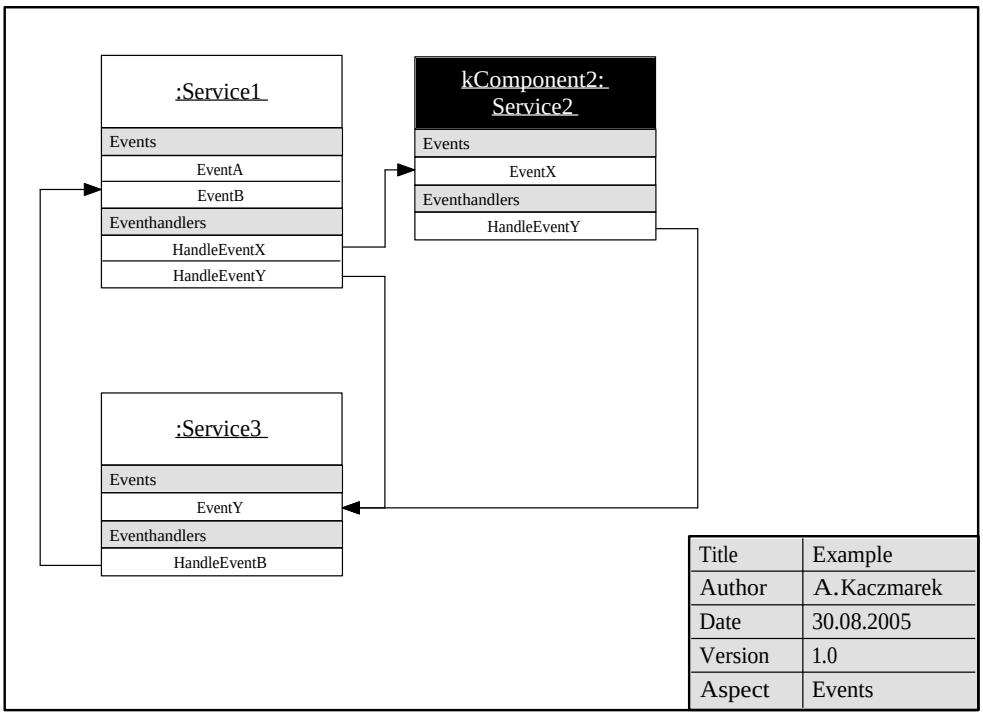

Abbildung 3.9: Zusatzangaben eines Schaltplans

ge **Unterschiede** auf. Ein UML-Klassendiagramm zeigt die statischen Beziehungen zwischen Klassen, ein Komponentenschaltplan die statischen Beziehungen zwischen Komponenten eines Systems. Soweit ergeben sich keine größeren Unterschiede zwischen den beiden Diagrammarten. Die Differenzen liegen in den dargestellten Beziehungen.
In einem UML-Klassendiagramm können beliebige Assoziationen, Aggregationen und Kompositionen aufgenommen werden. Die jeweilige Assoziation besitzt allerdings keine feste Bedeutung, abgesehen von der Vergabe von Rollen für die verbundenen Klassen und einem Bezeichner für die Assoziation selbst. Diese textuellen Zusatzinformationen erhalten allerdings nur durch Interpretation durch einen Leser Sinn.
Im Gegensatz dazu existieren in Komponentenschaltplänen nur **fest definierte Arten** von Beziehungen. In 3.4 wurden diese Typen bereits vorgestellt. Ein Komponentenschaltplan besitzt durch diese Einschränkung zwar nicht die Flexibilität eines UML-Klassendiagrammes, allerdings können nur durch diese feste Vorgabe von Kopplungsmöglichkeiten Komponentenschaltpläne anhand entsprechender Mittel gelesen und ausgeführt werden. Ein ausführbares UML-Klassendiagramm kann hingegen nicht existieren.

Zusammenfassend kann festgehalten werden, dass ein UML-Klassendiagramm die abstrakte Darstellung der statischen Aspekte eines Systems ist. Ein Komponentenschaltplan hingegen beschreibt durch die Darstellung von K-Komponenten und deren Kopplungen untereinander ein bestehendes, ausführbares Komponentensystem.

Nachdem in diesem Abschnitt zuletzt eine Notation für Komponentenschaltpläne vorgestellt worden ist, beschäftigt sich der folgende Abschnitt mit der Thematik des Aufstellens und Ausführens von Komponentenschaltplänen. Dabei wird betrachtet, welche zusätzlichen Elemente benötigt werden, damit ein Systemassemblierer einen Komponentenschaltplan erfassen und ein Anwender ein beschriebenes System ausführen kann.

3.6 Aufstellen und Ausführen von Schaltplänen

Die vorangegangenen Abschnitte haben sich mit den Konzepten von Diensten, K-Komponenten und Schaltplänen befasst. Diese Sektion befasst sich mit dem Vorgang des Aufstellens eines Komponentenschaltplans und dem Ausführen eines so beschriebenen Systems.

3.6.1 Erfassen eines Komponentenschaltplans

Soll ein vollständiges System anhand von K-Komponenten in einem Komponentenschaltplan beschrieben werden, so sollten neben den bisher vorgestellten Konzepten weitere das Bild vervollständigen.
Benötigt wird eine möglichst grafische Oberfläche, die Komponentenschaltpläne darstellen und editieren kann. Ein solcher Editor muss dabei möglichst auf eine Registratur zurückgreifen können, die Informationen über ihr bekannte K-Komponenten liefern kann.

Dienst- & Komponentenregistratur

Eine *Dienst- & Komponentenregistratur* ist eine Art Verzeichnis für Dienste und K-Komponenten. Innerhalb einer Dienst- & Komponentenregistratur kann nach bereits bekannten Diensten und K-Komponenten gesucht werden. Über eine Volltextsuche können die natürlichsprachlichen Semantikbeschreibungen, oder im Fall von K-Komponenten die implementierten Dienste Suchkriterien darstellen.

Schaltplaneditor

Der *Schaltplaneditor* dient dem Erstellen und Bearbeiten von Komponentenschaltplänen. Ein solcher Editor sollte über eine grafische Benutzeroberfläche verfügen, die den editierten Schaltplan darstellen und editieren kann. Sowohl die Vernetzungsstruktur wie auch die Konfektionierungsdaten sollten über den Editor erfasst werden können.
Der Schaltplaneditor greift auf eine Dienst- & Komponentenregistratur zu, damit der Ersteller eines Schaltplans für die Erfassung des Systems auf vorhandene Dienste und K-Komponenten zurückgreifen kann. Über die Oberfläche wird die Suchfunktionalität der Registratur angesprochen, die Suchergebnisse werden visualisiert und können dem Schaltplan hinzugefügt werden. Der Schaltplaneditor muss ebenfalls überprüfen können, ob der editierte Schaltplan vollständig ist und gegebenenfalls Hinweise auf fehlende Teile liefern können.

3.6.2 Interpretation von K-Komponenten

Ein Komponentenschaltplan beschreibt ein vollständiges Softwaresystem. Diese Beschreibung muss zunächst interpretiert werden, bevor sie ausgeführt werden kann. Die Interpretation bezieht sich darauf, wie genau ein Komponentenschaltplan zu lesen und umzusetzen ist, um ein ausführbares System aufzubauen. Prinzipiell sind mehrere Interpretationen eines Komponentenschaltplans möglich.

Nachfolgend werden zwei Interpretationsansätze vorgestellt, K-Komponenten als dauerhafte Service-Objekte und K-Komponenten-Instanzen.
In ersten Fall werden K-Komponenten als dauerhaft beständige Service-Objekte betrachtet, die unabhängig von Schaltplänen existieren und in verschiedenen Systemen zeitgleich zum Einsatz kommen können.
Im Gegensatz dazu betrachtet der zweite Ansatz K-Komponenten als instanziierbare Entitäten. Pro ausgeführtem Schaltplan und darin enthaltener K-Komponente wird eine K-Komponenten-Instanz erzeugt, die nur für diesen Systemkontext gültig ist.
Die beiden Ansätze werden im Folgenden genauer gegenübergestellt.

K-Komponenten als dauerhafte Service-Objekte

Eine K-Komponente kann betrachtet werden als eine dauerhaft verfügbare Entität. Ist eine K-Komponente in einem Schaltplan enthalten, dann gehört zu dieser K-Komponente die Adressinformation, unter der sie zu erreichen ist.

Genau wie ein Web Service (vgl. [10]) ist eine solche K-Komponente praktisch immer verfügbar und kann unter ihrer Adresse erreicht werden. Da eine solche K-Komponente gleichzeitig in mehreren Komponentenschaltplänen enthalten sein kann, muss in der Kommunikation mit ihr jedesmal die Kontextinformation zum ausgeführten Schaltplan enthalten sein. Diese Kontextinformationen müssten beispielsweise die Belegung der Write-Eigenschaften beinhalten.

Im Gegensatz zu dieser Sicht auf eine K-Komponente steht der nun folgende Ansatz.

K-Komponenten-Instanzen

In einem Komponentenschaltplan werden K-Komponenten assembliert. Die Verschaltungen einer solchen Assemblierung haben den Charakter eines zu instanziierenden Plans: Zu jeder K-Komponente wird bei Ausführung des Schaltplans eine K-Komponenten-Instanz erzeugt, die die relevanten Daten des Schaltplans mitgeteilt bekommt. Diese Instanzen werden ausschließlich für den Kontext des auszuführenden Schaltplans erzeugt. Die benötigten Kontextinformationen über den Schaltplan für die Kommunikation mit anderen K-Komponenten sind daher nur bei der Instanziierung von Interesse, denn sie bleiben über die gesamte Lebensdauer der K-Komponenten-Instanzen unverändert.

Der Ansatz der K-Komponenten-Instanzen ist dem der K-Komponenten als dauerhaften Service-Objekten vorzuziehen. Vor allem wird die Komplexität der Umsetzung von K-Komponenten automatisch reduziert, da eine K-Komponenten-Instanz immer nur innerhalb genau eines Schaltplankontextes arbeitet. Seiteneffekte, die bei dauerhaften Service-Objekten auftreten können, werden ebenfalls vermieden.
Die Wahl der letztendlichen Interpretation ist eng verknüpft mit der späteren Realisierung des K-Komponentenkonzeptes. Daher kann und soll die Entscheidung für eine Interpretation an dieser Stelle nicht abschließend getroffen werden.

Unabhängig davon, auf welche Interpretationssicht man sich festlegt, bestehen zusätzlich verschiedene Möglichkeiten, wie die Verschaltungen eines Komponentenschaltplans umgesetzt werden. Nachfolgend werden zwei unterschiedliche Möglichkeiten vorgestellt.

3.6.3 Das Komponentensystem zur Laufzeit

Ein über einen Komponentenschaltplan beschriebenes Softwaresystem muss ausgeführt werden. Dabei existieren verschiedene Möglichkeiten, wie sich ein Komponentensystem zur Laufzeit darstellt. Zwei Alternativen werden nachfolgend betrachtet, die der *zentralen Kontrolleinheit* und die der *autonomen Komponenten*.

Zentrale Kontrolleinheit

Die *zentrale Kontrolleinheit* stellt für die K-Komponenten eines Schaltplans eine ausführende Umgebung dar. Die Kontrolleinheit kann einen vollständigen Schaltplan lesen und übernimmt die Aufgabe einer Vermittlungszentrale. Der Nachrichtenaustausch verläuft über diesen Kommunikationsvermittler, der die korrekte Zustellung der Daten gewährleistet. Die einzelnen K-Komponenten müssen in diesem Fall keine Informationen über ihre Verschaltung besitzen, sie kommunizieren nur indirekt mit anderen K-Komponenten über die Kontrolleinheit. Auch die Eventbenachrichtigung wird über diese Vermittlungszentrale abgewickelt.
Vorteil des Ansatzes ist die einfache Art der Kommunikation. Eine K-Komponente tauscht Nachrichten nur mit der zentralen Kontrolleinheit aus, die den Weitertransport übernimmt. Die K-Komponenten müssen keine Angaben zu Bindungen an Fremdkomponenten besitzen
Daraus ergibt sich allerdings auch der Hauptnachteil dieses Ansatzes. Die Kontrolleinheit bildet automatisch einen Flaschenhals für die Kommunikation, da alle Daten des Systems über sie ausgetauscht werden.

Autonome Komponenten

Möchte man einen Kommunikationsflaschenhals vermeiden, so müssen die K-Komponenten direkt untereinander Nachrichten austauschen. Diesen Ansatz verfolgt die Idee der *autonomen Komponenten*. Es wird keine zentrale Kontrolleinheit vorausgesetzt, die während der gesamten Laufzeit des Komponentensystems verfügbar ist. Die autonomen Komponenten müssen durch eine startende Einheit initial verschaltet werden, d. h. diese Einheit liest einen Schaltplan ein und teilt jeder beteiligten K-Komponente alle benötigten Bindungsinformationen mit. Analog zum Start des Komponentensystems kann es beendet werden. Über die gleiche Einheit wird allen vernetzten K-Komponenten das Beenden des Systems mitgeteilt, die Verschaltungen untereinander können daraufhin gelöst werden.
Da die zentrale Vermittlungsstelle entfällt, gibt es auch keinen Kommunikationsflaschenhals. Allerdings müssen die einzelnen K-Komponenten mehr

leisten, da sie in diesem Szenario selbst für den Nachrichtenaustausch mit Fremdkomponenten zuständig sind.

Einem systembedingten Flaschenhals versucht man prinzipiell zu vermeiden. Aus diesem Grund sind autonome Komponenten einer zentralen Kontrolleinheit vorzuziehen. Im Rahmen des Ausblicks wird allerdings eine Möglichkeit besprochen, die Vorteile einer zentralen Kontrolleinheit mit denen von autonomen Komponenten zu verbinden (vgl. 3.7.5).

3.7 Zusammenfassung und Ausblick

Zum Abschluss dieses Kapitels soll zunächst rückblickend überprüft werden, welche Ziele mit dem vorgestellten Konzept erreicht werden konnten. Dazu werden nochmals die in Kapitel 2 aufgestellten Kriterien betrachtet und auf das vorgestellte Komponentenkonzept angewendet.
Im Fokus des Ausblicks befinden sich weiterführende Themen und Überlegungen, die den Weg in das Komponentenkonzept dieser Arbeit nicht mehr gefunden haben. Zu diesen Themen zählen eine mögliche Erweiterung der Notation für Komponentenschaltpläne, die Spezialisierung von Diensten, der Einsatz eines Komponentenservers, sowie Überlegungen zur Versionierung von Diensten und K-Komponenten.

3.7.1 Überprüfung der Anforderungen an Komponenten

Die in Kapitel 2 entwickelten Kriterien an Komponenten werden in diesem Abschnitt auf das zuvor vorgestellte Komponentenkonzept angewandt. Manche Punkte werden dabei jedoch ausgelassen, da diese nur im Zusammenspiel mit einer tatsächlichen Implementierung des Konzeptes überprüfbar sind.

Selbstbeschreibungsfähigkeit

Jede K-Komponente kann Auskunft geben über die implementierten Dienste, sowie obligatorische und optionale Dienst-Import-Abhängigkeiten. Diese Informationen sind zuerst syntaktischer Natur, da ein Dienst wiederum Auskunft über die Signaturen der enthaltenen Eigenschaften, Methoden, Events und Eventhandler geben kann.
Jedes dieser Merkmale und die K-Komponente selbst besitzt zusätzlich auch eine natürlichsprachliche Beschreibung, die sich mit dem jeweiligen Sinn und

der Verwendung befasst, wodurch die Selbstbeschreibungsfähigkeit der K-Komponenten ebenfalls einen semantischen Aspekt aufweist.

Strikte Kapselung, Black-Box-Wiederverwendung

K-Komponenten sind strikt gekapselt, die Implementierung bleibt verborgen. Die öffentliche Schnittstelle wird durch den implementierten Dienst definiert. Offengelegt werden die Import-Abhängigkeiten von fremden Diensten.

Konfektionierbarkeit

Die Konfektionierung ist Bestandteil des Komponentenschaltplans. Es können sowohl globale wie auch lokale Parameter erfasst werden. Globale Parameter gelten für alle enthaltenen K-Komponenten, lokale Parameter entsprechen den Write-Eigenschaften der K-Komponenten.

Wissen über Komponentenabhängigkeiten

Jede K-Komponente kennt die obligatorischen und optionalen Dienste, von denen sie abhängig ist. In einem Komponentenschaltplan werden diese abstrakten Abhängigkeiten mit tatsächlichen K-Komponenten erfüllt. Über den Schaltplan sind somit auch die Abhängigkeiten zwischen K-Komponenten explizit.

Kompositionsfähigkeit

Die Kompositionsfähigkeit steht im Mittelpunkt des vorgestellten Konzeptes. Über das Mittel der Komponentenschaltpläne lassen sich vollständige Systeme durch Komposition einzelner K-Komponenten beschreiben.
Die Komposition ist dabei verteilt auf verschiedene Ebenen bzw. Sichten. Zur Komposition zählen die die Dienst-Import-, sowie die Event-Handler-Kopplungen.

Interaktionsfähigkeit

K-Komponenten können miteinander interagieren. Interaktionsmittel sind Methodenaufrufe, sowie Eventbenachrichtigungen. Geregelt wird die Interaktion durch das Kontroll- und Sessionprotokoll.

Auslieferungseinheit (Atomizität)

Die Form der Auslieferung hängt von einer späteren Implementation des Konzeptes ab. Grundsätzlich sind K-Komponenten atomar, Abhängigkeiten sind explizit bekannt.

Ausführbarkeit

Eine K-Komponente ist per Definition ausführbar, falls alle obligatorischen Abhängigkeiten erfüllt sind.

Unabhängigkeit von Programmiersprachen

Die *Unabhängigkeit von Programmiersprachen* kann nicht losgelöst von einer Implementierung des K-Komponentenkonzeptes betrachtet werden. Aus diesem Grund kann an dieser Stelle keine Aussage über den Grad der Erfüllung getroffen werden.

Unabhängigkeit von Plattformen

Siehe *Unabhängigkeit von Programmiersprachen*

Komposition ohne Programmierkenntnisse

Die Komposition von K-Komponenten erfolgt ohne Programmierkenntnisse (im engeren Sinne). Sind Glassbox-Komponenten Teil eines Schaltplans, so müssen die entsprechenden Skripte erstellt werden. Umfassende Programmierkenntnisse werden hierfür allerdings nicht benötigt.
Das Aufstellen eines Komponentenschaltplans kann bei Verwendung eines Graphformats prinzipiell anhand eines passenden Editors geschehen. Sehr vereinfachen würde diesen Vorgang die Unterstützung durch einen visuellen Editor, der das zugrunde liegende Format erstellen kann. Solch ein Editor muss die Mechanismen, die durch K-Komponenten und den Verzeichnisdienst bereitgestellt werden, weitgehend ausschöpfen.

Ortstransparenz, Verteiltheit

Siehe *Unabhängigkeit von Programmiersprachen*

Klassifizierbarkeit, Kategorisierbarkeit

Die implementierten Dienste bilden die Klassifizierungsmerkmale für K-Komponenten.

Lose Kopplung

Das vorgestellte Komponentenkonzept legt nicht endgültig fest, ob die Kopplungen zwischen K-Komponenten lose oder fest sind. Wie in 3.6.3 aber ersichtlich geworden ist, liegt es auch an der Laufzeitumgebung, wie mit den Kopplungsinformationen eines Schaltplans umgegangen wird.
Im engeren Sinn sind die Kopplungen zwischen K-Komponenten allerdings fest. Auch wenn K-Komponenten nicht unbedingt direkt miteinander gekoppelt, sondern auf vermittelnde Einheiten angewiesen sind, so sind bezogen auf das Gesamtsystem die Kopplungen allerdings fest, d. h. nicht zur Laufzeit änderbar.

Verzeichnisdienst

Das Dienst- & Komponentenregistratur liefert den gewünschten Verzeichnisdienst.

Plug&Play-Fähigkeit

Nach erfolgreicher Registrierung in der Dienst- & Komponentenregistratur ist eine K-Komponente sofort bereit für den Einsatz.

Fazit

Die meisten Eigenschaften an Komponenten werden bereits durch das hier vorgestellte Konzept erfüllt. Die noch verbleibenden Kriterien können erst betrachtet werden, wenn eine Umsetzung des Konzeptes durchgeführt worden ist.

Nach der Untersuchung des vorgestellte Komponentenkonzept anhand des Anforderungskataloges folgt abschließend eine Sammlung ausblickender Überlegungen.

3.7.2 Globale Definitionen

In dem vorgestellten Konzept werden alle Definitionen im Rahmen von Diensten vorgenommen. Dieses Vorgehen ist nicht unbedingt nötig und unter Umständen sogar nicht die beste Lösung. Denkbar sind alternative Ansätze, um Teildefinitionen eines Dienstes global zu erfassen.
Zu den Kandidaten für eine globale Definition zählen in erster Linie die **Datentypen** (vgl. 3.2.1) und **Exceptions** (vgl. 3.2.4). Aber auch **Events** (vgl. 3.2.5) könnten theoretisch losgelöst von Diensten definiert werden.

Im Fall von Datentypen ist eine globale Definition beispielsweise sinnvoll. Ein Datentyp erhält einen systemweit eindeutigen Bezeichner, die restliche Definition erfolgt analog zur bekannten Definition im Dienst. Vorteil dieses Ansatzes ist, dass ein Dienst auf allgemein bekannte Datentypen zurückgreifen kann, ohne auf die Dienste angewiesen zu sein, die den Datentyp definieren. Die doppelte Erfassung eigentlich identischer Datentypen ist somit auch unnötig.

Es bleibt festzuhalten, dass nochmals überprüft werden sollte, welche Definitionen, die zurzeit in einem Dienst erfolgen, möglicherweise besser unabhängig von Diensten vorgenommen werden sollten.

3.7.3 Erweiterung der Schaltplannotation

Die in Abschnitt 3.5.3 vorgestellte grafische Notation für Komponentenschaltpläne umfasst nur die Elemente, die für die Erfassung eines ausführbaren Schaltplans zwingend benötigt werden. Ein Komponentenschaltplan sollte neben Kopplungsbeschreibung auch einen informativen Charakter aufweisen und alle interessanten Informationen über enthaltene Dienste und K-Komponenten darstellen können.
Daher sollte die grafische Notation für Komponentenschaltpläne dahingehend erweitert werden, dass möglichst alle bekannten Informationen über Dienste, K-Komponenten und Kopplungen notiert werden können. In der vorgestellten Notation fehlt u. a. die Aufnahme und Darstellung der Methoden, Protokolle und Constraints.

3.7.4 Spezialisierung von Diensten

Das vorliegende Konzept berücksichtigt aus Komplexitätsgründen keine Spezialisierung von Diensten. Prinzipiell wäre eine solche Fähigkeit allerdings sinnvoll und wünschenswert.
Wenn man Dienste spezialisieren möchte, so müssen verschiedene Probleme gelöst werden. Zunächst muss man festlegen, welche Teile eines Dienstes von der Spezialisierung betroffen sein sollten. Handelt es sich bei einer Spezialisierung nur um eine Erweiterung oder sogar um eine Verfeinerung?
Auch Teilaspekte eines Dienstes und der Umgang mit diesen bei einer Spezialisierung werfen weitere Fragen auf. Wie kann man z. B. die Spezialisierung eines Protokolls beschreiben? Wann ist ein Protokoll die Spezialisierung eines anderen? Ein erster Ansatz ist hier, einen Kompatibilitätsbegriff auf Protokollen einzuführen und diesen bei der Spezialisierung zu verwenden. Dann

muss allerdings noch eindeutig geklärt sein, wann genau ein Protokoll kompatibel zu einem anderen ist.

3.7.5 Komponentenserver

K-Komponenten implementieren die durch die Dienste vorgegebene **Logik**, müssen daneben aber auch einem **technischen Rahmen** genügen. Damit K-Komponenten untereinander kommunizieren können, muss ein **Transportkanal** für den Informationsaustausch zwischen K-Komponenten definiert sein. Jede K-Komponente muss die technischen Vorgaben eines solchen Transportkanals erfüllen.
Zusätzlich besitzen K-Komponenten die Fähigkeit zur **Selbstbeschreibung**. Auch hierfür wird vorgegeben, wie die Umsetzung dieser Fähigkeit realisiert wird und jede K-Komponente muss dafür Sorge tragen, diese Vorgabe zu erfüllen.

Diese Aufgaben sind **allgemein** für alle K-Komponenten **gültig** und auf abstrakter Ebene **identisch**. Bei der Entwicklung einer K-Komponente ist es wünschenswert, dass man sich auf die Implementation der eigentlichen Leistung konzentrieren könnte, ohne bei jeder Neuentwicklung Zeit und andere Ressourcen für die Erfüllung der technischen Rahmenbedingungen aufwenden zu müssen.

In diesem Abschnitt wird eine Möglichkeit zur Erreichung dieses Ziels besprochen, der Einsatz von sogenannten **Komponentenservern**. Die potentielle Plattformunabhängigkeit und Unabhängigkeit von Programmiersprachen der K-Komponenten soll im Rahmen dieses Ansatzes gewahrt bleiben, um keine Einschränkung des ursprünglichen Konzeptes zu verursachen.

Ein *Komponentenserver* ist eine Softwareeinheit, die für K-Komponenten die technischen Rahmenbedingungen des Komponentensystems erfüllt. Dabei ist ein Komponentenserver **nicht** mit der zentralen Kontrolleinheit aus Abschnitt 3.6.2 zu verwechseln, vielmehr handelt es sich um eine Umsetzung des Komponenten-Instanz-Ansatzes.

Zu den Aufgaben eines Komponentenservers zählt die **Instanziierung** von K-Komponenten, die Bereitstellung eines **Infrastrukturrahmens** für K-Komponenten, sowie der **Destruktion** von K-Komponenten bei Beenden eines laufenden Systems.
Innerhalb eines durch einen Komponentenschaltplan beschriebenen Systems können **mehrere Komponentenserver** zum Einsatz kommen, die für un-

terschiedliche K-Komponenten des Schaltplans zuständig sind. Diese Komponentenserver können auf unterschiedlichen Rechnern ausgeführt werden, wodurch **Verteiltheit** für K-Komponenten direkt unterstützt wird.
Festgelegt werden muss die **Kommunikation** zwischen Komponentenservern. Allerdings dürfte sich diese nicht sonderlich unterscheiden von der Kommunikation zwischen K-Komponenten in dem Szenario ohne Komponentenserver.

Ein Komponentenserver wird für eine gegebene **Plattform** entwickelt. Nahe liegend ist auch die Festlegung einer **Programmiersprache**, in der K-Komponenten für den jeweiligen Komponentenserver implementiert werden müssen. Auch wenn diese Festlegung nicht erfolgt, muss auf jeden Fall die **Schnittstelle** definiert werden, über die K-Komponenten auf die Leistungen eines Komponentenservers zugreifen können. Im Fall der Festlegung einer Programmiersprache kann das erreicht werden, indem ein zu implementierendes Interface oder eine zu spezialisierende Basisklasse vorgegeben wird.

Durch den Ansatz der Verwendung von Komponentenservern wird erreicht, dass bei der Entwicklung einer K-Komponente die Umsetzung der durch die Dienste vorgegebenen **Logik im Mittelpunkt** steht. Die allgemein gültigen Aufgaben werden durch den Komponentenserver wahrgenommen. Es lassen sich verteilte Systeme realisieren, die vollständig unabhängig von Betriebssystemen und Programmiersprachen sein können. Der Grad dieser Unabhängigkeit ist gebunden an die Anzahl der implementierten Komponentenserver für die zu unterstützenden Plattformen.

Die bisher besprochenen Aufgaben eines Komponentenservers sind nicht unbedingt vollständig. Z. B. ist es vorstellbar, dass ein Komponentenserver auch die Aufgaben einer **Dienst-&Komponentenregistratur** (vgl. 3.6.1) übernimmt. Diese Idee liegt nahe, da ein Komponentenserver sowieso ein Verzeichnis aller ihm bekannten K-Komponenten führen muss. Die Veröffentlichung dieser Informationen ist ein kleiner Schritt, er erspart allerdings eine separate Registratur. Ein Schaltplaneditor (vgl. 3.6.1) könnte auf einen oder mehrere Komponentenserver zugreifen und die Registraturfunktionalität nutzen, um einem Systemassemblierer bekannte Dienste und K-Komponenten anzubieten.

Vergleichbar ist die Idee der Komponentenserver mit dem Konzept des *Application Servers* im Umfeld von EJB (vgl. 2.3.4). Ein Application Server bietet EJB-Komponenten ebenfalls eine Laufzeitumgebung, die allgemeine Aufgaben übernimmt und Standardfunktionalität bietet. Zu diesen Funktionsberei-

chen zählen u. a. Sicherheit, Transaktionsmanagement, sowie Namens- und Verzeichnisdienste. Der Unterschied der beiden Ideen liegt in der Komponierbarkeit der Komponenten. Während weder eine EJB-Komponente selbst, noch ein Application Server eine explizite Kopplung mit anderen EJB-Komponenten vorsieht, ist genau das zentraler Bestandteil von K-Komponenten und einem zugehörigen Komponentenserver.
Analog zur Vorgehensweise bei der Entwicklung einer EJB-Komponente, kann sich der Entwickler einer K-Komponente bei Verwendung eines Komponentenservers ebenfalls auf die Umsetzung der Geschäftslogik konzentrieren. Zusätzlich wird ihm aber auch die Verschaltung mit benötigten Fremdkomponenten abgenommen, da diese erst durch einen Schaltplan festgelegt und durch den Komponentenserver zur Laufzeit realisiert wird.

3.7.6 Versionierung

Vollständig außerhalb der Überlegungen zu dem Komponentenkonzept stand bisher die Frage nach der **Versionierung** von Diensten, K-Komponenten und auch Komponentenschaltplänen. Eine Versionierung von K-Komponenten und Schaltplänen kann problemlos vorgenommen werden, da jede Version eine neue Entität darstellt und für sich selbst stehen kann. Anders stellt sich die Situation bei Diensten dar.

Möchte man Versionierung für Dienste zulassen und gleichsam Schwierigkeiten und Konflikte vermeiden, so kann man das analog zu der beschrieben Versionierung von K-Komponenten und Schaltplänen erreichen und für jede neue Version eines Dienstes einen neuen Dienst einführen. Die Versionierung wäre dann sozusagen nur Bestandteil des Dienstbezeichners.
Allerdings gewinnt man durch diese Art der Versionierung nichts hinzu, da jede versionierte Entität vollkommen für sich steht und nichts mit den eigenen Vorgängern oder Nachfahren zu tun hat.

Möchte man erreichen, dass Versionierung von Diensten einen zusätzlichen Effekt neben der reinen Nummerierung darstellt, muss definiert werden, wie sich unterschiedliche Versionen eines Dienstes zueinander verhalten. Welche Auswirkungen hat die Version eines Dienstes auf einen Schaltplan? Kann eine K-Komponente, die eine bestimmte Version eines Dienstes implementiert, in einem Schaltplan auch den gleichen Dienst in einer früheren oder späteren Version erfüllen? Was sind die Bedingungen dieser Erfüllung?

Die Komplexität dieser und weiterer Fragestellungen ist nicht gering und passt nicht in den Rahmen dieser Arbeit. Aus diesem Grund wird festgelegt,

dass es für Dienste **keine Versionierung** gibt. Ein einmal veröffentlichter Dienst bleibt **unveränderbar**.

In diesem Kapitel wurde ein Komponentenkonzept erarbeitet und vorgestellt, das die **Komponierbarkeit** und **Wiederverwendbarkeit** von Software-Komponenten in den Mittelpunkt stellt. Alle bisherigen Überlegungen und Feststellungen waren rein theoretischer Art. Um K-Komponenten mit einer möglichst realistischen Aufgabe zu konfrontieren, folgt im nächsten Kapitel die Darstellung eines Fallbeispiels.

Kapitel 4

Fallbeispiel

Nachdem im letzten Kapitel das **eher theoretische** Konzept von K-Komponenten vorgestellt worden ist, soll dieses Kapitel den K-Komponentenansatz mit einem realitätsnahen Beispiel konfrontieren und anhand dessen validieren. Das Fallbeispiel stammt aus dem Bereich der **betrieblichen Informationssysteme**. Möglichst viele der in Kapitel 3 vorgestellten Konzepte werden aufgegriffen, um so anhand des Fallbeispiels verdeutlicht zu werden und um Möglichkeiten der Umsetzung vorzustellen.

Im ersten Abschnitt dieses Kapitels erfolgt eine ausführliche **inhaltliche Beschreibung** des Fallbeispiels mitsamt allen beteiligten Einzelbereichen und dem Gesamtsystem. Anschließend an diese inhaltliche Beschreibung folgt die Identifikation und Spezifikation der benötigten **Dienste und K-Komponenten**, sowie die Vorstellung des **Komponentenschaltplans**, der das System des Fallbeispiels repräsentiert. Das Kapitel schließt mit einer Zusammenfassung, die ein Fazit für den Einsatz von K-Komponenten im Szenario des Fallbeispiels umfasst.

Das Fallbeispiel soll **exemplarisch** das K-Komponenten-Konzept auf ein realistisches Szenario übertragen. Dabei sollen möglichst viele Aspekte des Konzeptes durch das Fallbeispiel abgedeckt werden. Aufgrund der nicht geringen Komplexität der Thematik und zugunsten eines einheitlichen Beispiels im Gegensatz zu mehreren abgegrenzten Beispielen, die sich jeweils auf einzelne Konzeptaspekte beschränken, kann das Fallbeispiel nicht vollständig alle Bestandteile des K-Komponentenkonzepts umfassen. Außerdem nimmt das Fallbeispiel eine vereinfachende Sicht auf das dargestellte Szenario ein, um den Umfang des Beispiels zu beschränken.

4.1 Das Fallbeispiel

Szenario für das Fallbeispiel ist ein Abrechnungssystem für Telekommunikationsunternehmen. Abgebildet werden verschiedene Bereiche des Systems. Dazu zählen der Bereich der *Stammdatenverwaltung*, die *Leistungserfassung* sowie die *Leistungsabrechnung*.

In der Stammdatenverwaltung werden die **Kundendaten erfasst und verwaltet**. Die Kundendaten sind die grundlegende Basis für jede nachgelagerte Abrechnung.
Die zweite Grundlage für eine Leistungsabrechnung ist die Leistungserfassung. Konsumierte **Leistungen werden erfasst** und dem verursachenden Kunden zugeordnet.
Kundendaten und Leistungsdaten werden in der Leistungsabrechnung zusammengeführt, um darauf basierend **Rechnungen zu erstellen**. Kundenrechnungen müssen zugestellt und archiviert werden.

Nach diesem kurzen Überblick über das Szenario des Fallbeispiels, werden nachfolgend die einzelnen Funktionsbereiche näher betrachtet.

4.1.1 Funktionsbereiche des Szenarios

Das Szenario des Fallbeispiels umfasst für den Kontext spezifische Bereiche, sowie Aufgaben, die prinzipiell allgemein in Informationssystemen vorkommen. Zu den kontextspezifischen Aufgaben gehören beispielsweise die Stammdatenverwaltung und die Leistungsabrechnung. Allgemeiner Natur sind Aufgaben der Benutzerverwaltung oder der Dokumentenarchivierung.
Abbildung 4.1 zeigt einen Überblick über die hauptsächlichen Funktionsbereiche des Fallbeispiels. Alle diese Aspekte werden im Folgenden einzeln vorgestellt.

Stammdatenverwaltung

Die Stammdaten sind Basis eines jeden Abrechnungssystems. In diesem Bereich erfolgt durch Mitarbeiter der CustomerService-Abteilung die Erfassung und Pflege der Kundendaten, die neben den persönlichen Angaben zu einer Person auch Informationen zu Anschriften und sonstigen relevanten Daten umfassen.

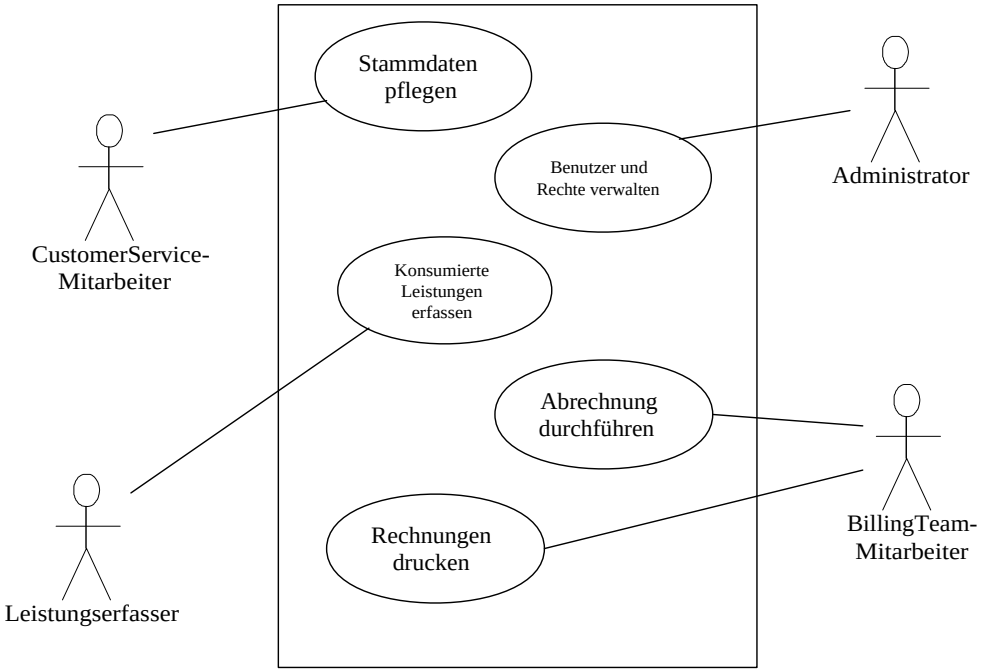

Abbildung 4.1: Funktionsbereiche des Fallbeispiels

Leistungserfassung

Eine Abrechnung umfasst immer Leistungen, die durch die in den Stammdaten enthaltenen Kunden konsumiert worden sind. Diese Leistungen müssen erfasst und dem verursachenden Kunden zugeordnet werden. Zusammen mit den Stammdaten bilden sie die Grundlage für die Abrechnung. Der Leistungserfasser kann eine Person oder eine automatisierte Softwareeinheit sein.

Leistungsabrechung

Ausgehend von den Stammdaten und den je Kunde konsumierten Leistungen können Rechnungen produziert werden. Generierte Rechnungsdokumente müssen archiviert und zugestellt werden. Die Zustellung kann digital per E-Mail erfolgen, oder durch Ausdruck und Zusendung der Rechnungsdokumente. Mitarbeiter des Billingteams sind zuständig für die Durchführung dieser Tätigkeiten.

Benutzer- und Rechteverwaltung

Das Abrechnungssystem kann von mehreren Benutzern bedient werden. Über die Benutzer- und Rechteverwaltung werden durch einen Administrator die

Benutzer des Systems erfasst und Berechtigungen für verschiedene Teilbereiche des Systems zugeordnet.
Die bereits zugeordneten Berechtigungen eines Benutzers können ebenfalls abgefragt werden, um so die Autorisierung einer Benutzers zu überprüfen.

Diese vier Aufgabenbereiche bilden die Eckpfeiler des Abrechnungssystems. Im folgenden Abschnitt wird das Zusammenspiel dieser Bereiche beschrieben, inklusive charakteristischer Arbeitsabläufe innerhalb des Szenarios.

4.1.2 Beschreibung des Gesamtsystems

Nach der groben Übersicht über die einzelnen vorhandenen Teilbereiche und -aufgaben des darzustellenden Systems, folgt in diesem Abschnitt die Darstellung des gesamten Systems, ausgehend von der gemeinsamen Datenbasis bis hin zu den charakteristischen Abläufen innerhalb des Systems.

Datenbasis

Die Bereiche Stammdaten, Leistungserfassung und Leistungsabrechnung arbeiten alle zusammen auf der gleichen Datenbasis, in diesem speziellen Fall auf der gleichen Datenbank. Die allgemeineren Bereiche Benutzer- und Rechteverwaltung können davon separiert arbeiten, eine gemeinsame Datenbank ist allerdings möglich.

Charakteristische Arbeitsabläufe

Innerhalb des Fallbeispielszenarios existieren mehrere charakteristische Arbeitsabläufe, die in einzelnen Funktionsbereichen stattfinden und die Bereiche aber auch miteinander verbinden. Anhand von überblicksartigen Diagrammen und zugehörigen Beschreibungen werden diese Abläufe vorgestellt.

Abbildung 4.2 zeigt typische Arbeitsabläufe für Mitarbeiter der Abteilung **Customer-Service**. Neukunden mitsamt allen relevanten Daten werden erfasst. Ebenso fällt die Pflege der Kundendaten in das Aufgabengebiet dieser Abteilung. Zur Stammdatenpflege zählt u. a. die Änderung der persönlichen Kundendaten oder der hinterlegten Anschrift.

Abbildung 4.3 stellt Aufgaben dar, die für die **Leistungserfassung** charakteristisch sind. Die Erfassung eines Leistungskonsums kann durch einen Mitarbeiter manuell erfolgen oder durch einen automatisierten Import von Verbrauchsdaten aus einer definierten Datenquelle. Unabhängig von der Art

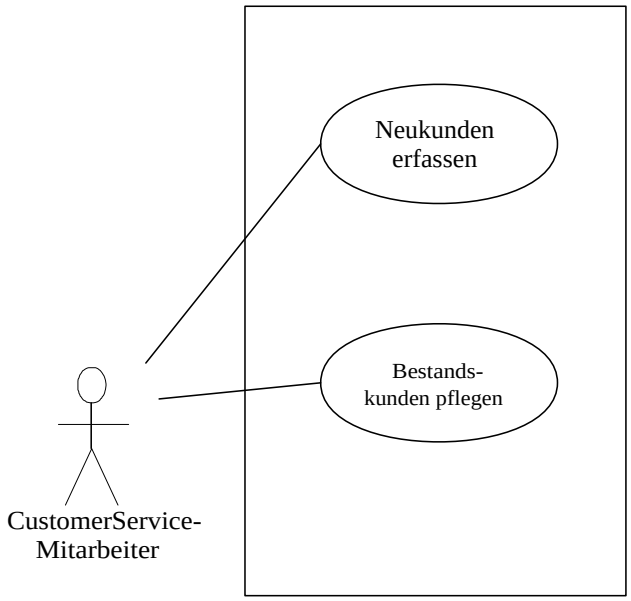

Abbildung 4.2: Fallbeispiel: CustomerService-Tätigkeiten

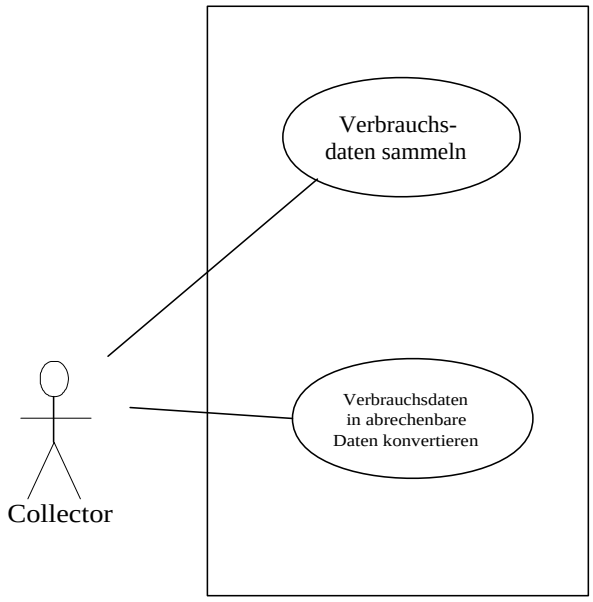

Abbildung 4.3: Fallbeispiel: Collector-Tätigkeiten

der Leistungserfassung werden die gesammelten Verbrauchsdaten der Abrechnungseinheit zur Verfügung gestellt, wozu unter Umständen die Konvertierung in ein bestimmtes Datenformat nötig sein kann.

Abbildung 4.4 beinhaltet Vorgänge, die durch Mitarbeiter der Abteilung

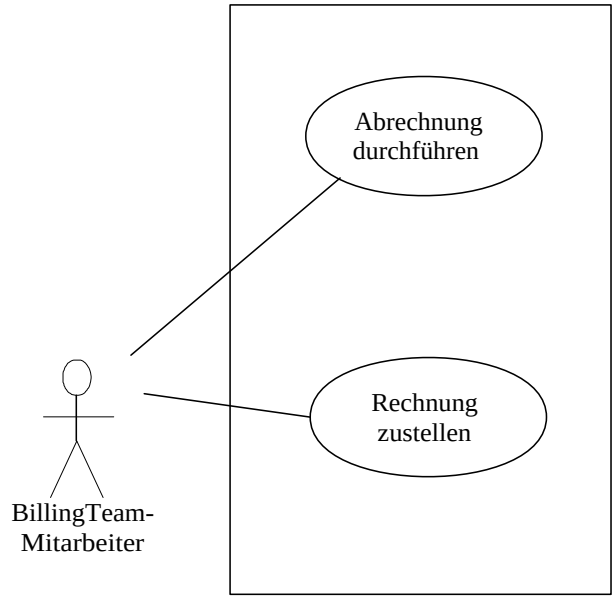

Abbildung 4.4: Fallbeispiel: Billing-Tätigkeiten

Billing auszuführen sind. Bei Durchführung einer Abrechnung durch Mitarbeiter des Billing-Teams werden alle aktiven Kunden mitsamt den erfassten Leistungen betrachtet und entsprechende Rechnungen generiert. Nach einem erfolreichem Rechnungslauf werden die erstellten Rechnungsdokumente archiviert und dem Kunden zugestellt. Die Zustellung erfolgt dabei automatisch in elektronischer Form per E-Mail, oder durch Ausdruck und Versenden der Rechnungsdokumente.

Abbildung 4.5 zeigt die charakteristischen Aufgaben eines **Administrators** im Rahmen des Abrechnungssystems. Der Administrator ist zuständig für die Erfassung und Verwaltung der Benutzerdaten. Dazu zählt die Vergabe von Benutzernamen und -passwörtern. Außerdem vergibt der Administrator Berechtigungen an die verschiedenen Benutzern, um diese für bestimmte mit den Berechtigungen verknüpften Aufgaben zu autorisieren.

Nach dieser etwas ausführlicheren Besprechung der charakteristischen Arbeitsabläufe innerhalb des Fallbeispiels, widmet sich der folgende Abschnitt

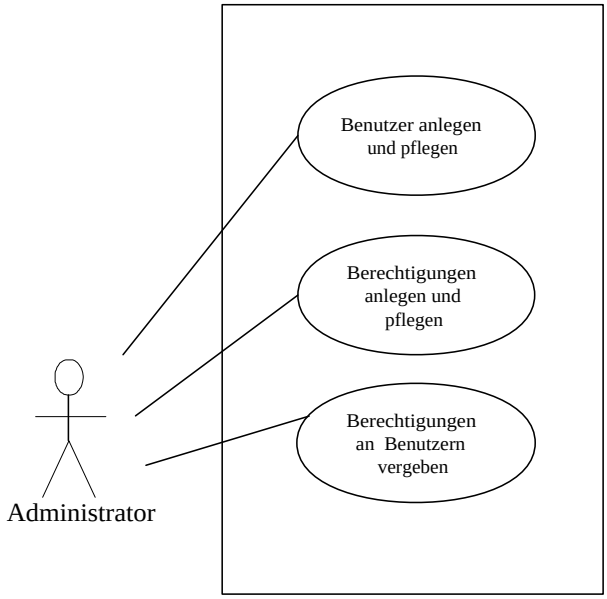

Abbildung 4.5: Fallbeispiel: Administrationstätigkeiten

der Realisierung des beschriebenen Abrechnungssystems unter Anwendung des K-Komponenten-Konzeptes.

4.2 Realisierung durch K-Komponenten

Das zuvor vorgestellte Fallbeispiel soll durch Verwendung von K-Komponenten realisiert werden. Das in Abschnitt 4.1 skizzierte Abrechnungssystem muss dazu durch einen **Komponentenschaltplan** beschrieben werden. Diese Aufgabe übernimmt der **Systemassemblierer**, der aus einer Auswahl vorhandener **K-Komponenten** benötigte Bausteine selektiert und in einem Schaltplan assembliert.
Für dieses Beispiel müssen jedoch **alle** Dienste und K-Komponenten zunächst aufgestellt werden, da noch keine Auswahl an Diensten und K-Komponenten existiert. Dazu werden die benötigten Dienste **identifiziert und spezifiziert**. Anschließend werden passend zu den definierten Diensten **K-Komponenten implementiert**, die die definierten Leistungen bereitstellen. Abschließend kann aus den dann existenten Diensten und K-Komponenten der **Komponentenschaltplan aufgestellt** werden.

Anhand dieses einfachen Beispiels ist bereits erkennbar, dass bei der Neuentwicklung eines komponentenorientierten Systems Kernstücke des neuen

Systems teilweise noch nicht als fertige Komponente vorliegen, sondern erst entwickelt werden müssen.
Wenn aber nur wenige spezielle Teile neu entwickelt werden müssen und für den Großteil der Systemfunktionalität auf vorhandene Komponenten zurückgegriffen werden kann, dann werden die Vorteile komponentenorientierte Entwicklung schnell deutlich.

Wie in 3.6.2 besprochen existieren verschiedene Ansätze, einen Komponentenschaltplan zu interpretieren. Innerhalb des Fallbeispiels wird der Ansatz der **K-Komponenten-Instanzen** verwendet. D. h. bei Ausführung des Komponentenschaltplans des Fallbeispiels wird pro enthaltener K-Komponente jeweils eine Instanz dieser K-Komponente erzeugt, die mit den Konfektionierungs- und Verschaltungsdaten des Schaltplans initialisiert wird.

4.2.1 Charakteristische Dienste

Um einen ersten Überblick über das Gesamtsystem zu erhalten, betrachtet dieser Abschnitt die charakteristischen Dienste des Fallbeispiels und deren Import-Beziehungen untereinander.
In Abbildung 4.6 ist ein Schaltplanauszug dargestellt, der sich zunächst nur auf Dienste und Verschaltungen bezüglich Imports konzentriert. Die grafische Notation folgt den Vorgaben aus 3.5.3.

In diesem Abschnitt werden die Dienstbezeichner mit den zugehörigen textuellen Beschreibungen aufgelistet. Die ausführlichen Dienstspezifikationen befinden sich im Anhang 5.2.
Die Bezeichner der Dienste, wie im Folgenden alle Elemente der Dienst- und K-Komponentenbeschreibungen, sind durchgängig in englischer Sprache angegeben.

Folgende Dienste sind in Abbildung 4.6 miteinander verschaltet:

UserRightsManagement

Der Dienst *UserRightsManagement* dient der Verwaltung von Benutzerdaten und der Zuordnung von Berechtigungen zu Benutzern oder Benutzergruppen. Der gesamten Verwaltung liegen folgende Annahmen zugrunde: Es gibt Benutzer, die durch den Datentyp `User` dargestellt werden. Es gibt Berechtigungen, die durch den Datentyp `Right` dargestellt werden. Es gibt Rollen, die durch den Datentyp `Role` dargestellt werden. Einer Rolle können Berechtigungen zugeordnet werden. Einem Benutzer können Rollen zugeordnet

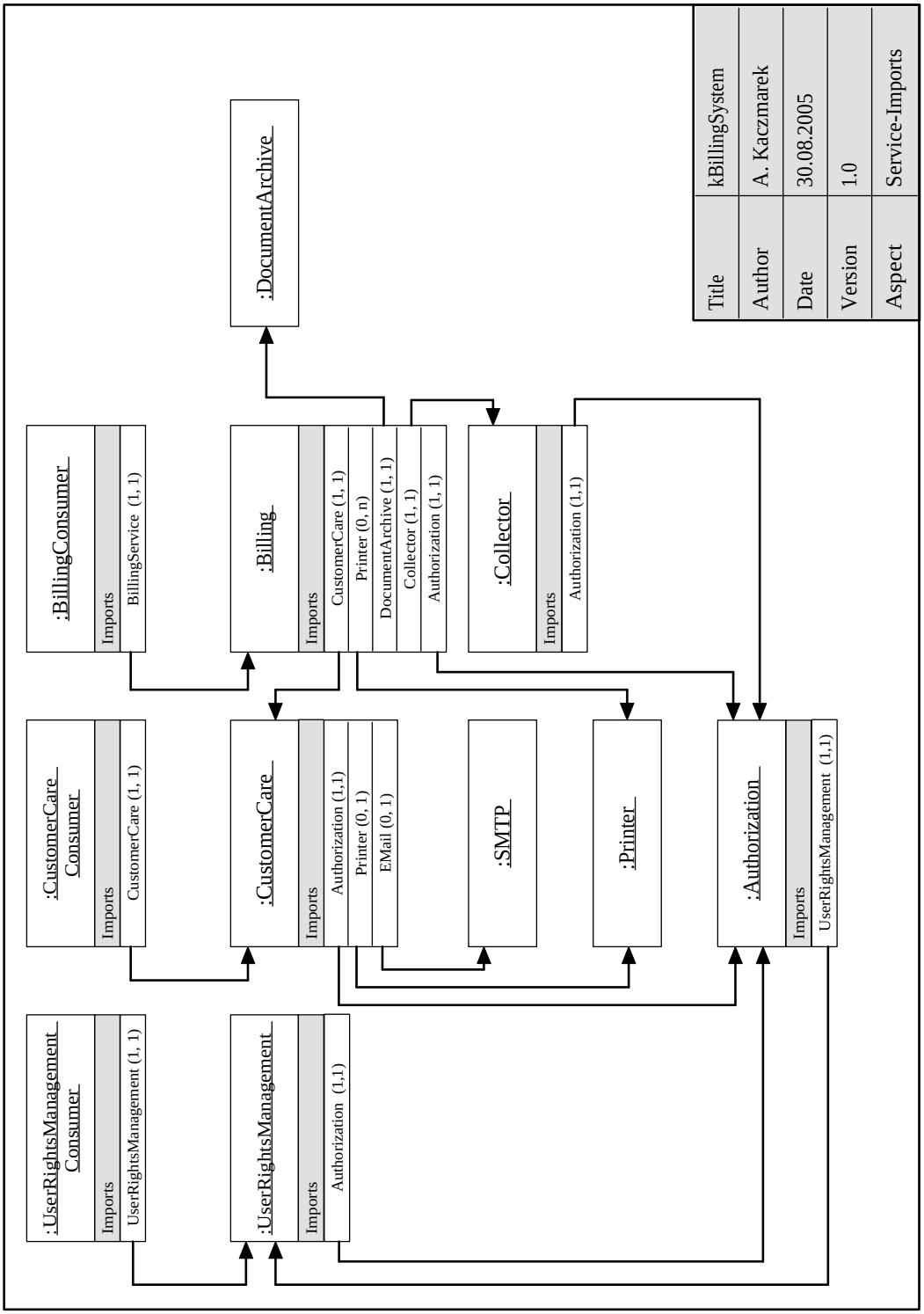

Abbildung 4.6: Schaltplan, Dienst-Import-Kopplungen

werden. Ein Benutzer besitzt genau dann eine Berechtigung, wenn er eine Rolle besitzt, der die betreffende Berechtigung zugeordnet ist.

Die logischen Hauptaufgaben des Funktionsbereichs *Benutzer- und Rechteverwaltung* werden durch diesen Dienst abgebildet.

UserRightsManagementConsumer

Der Dienst *UserRightsManagementConsumer* importiert den Dienst *UserRightsManagement*, um dessen Leistungen nutzen zu können.

Dieser Dienst ist praktisch nur ein Platzhalter für beliebige Konsumenten des Dienstes *UserRightsManagement* und wird später für eine GUI-Komponente verwendet.

Authorization

Der Dienst *Authorization* kapselt Funktionalität zur Anmeldung eines Systembenutzers, sowie Überprüfung der Berechtigungen des angemeldeten Benutzers. Der Dienst kennt ein Administratorkonzept. Über entsprechende Eigenschaften lässt sich der Administratorzugang konfigurieren. Der Administrator hat per Definition jede mögliche Berechtigungen.

SMTP

Der Dienst *SMTP* kapselt die Funktionalitäten eines einfachen E-Mail-Versenders nach dem SMTP-Protokoll (*Simple Mail Transport Protocol*). E-Mails samt Anhang können über die Schnittstelle des Dienstes versendet werden.

Printer

Der Dienst *Printer* kapselt die Funktionalität eines Druckers. Dokumente können an den Drucker übergeben und in beliebiger Anzahl ausgedruckt werden.

CustomerCare

Der Dienst *CustomerCare* kapselt die Funktionalität einer Verwaltungseinheit für Kundenstammdaten. Zu diesen Daten gehören persönliche Informationen über einen Kunden, die durch den Datentyp `Customer` repräsentiert werden. Ergänzend dazu können pro Kunde Adressinformationen hinterlegt

werden, die durch den Datentyp `Address` repräsentiert werden.

Dieser Dienst bildet die logischen Hauptaufgaben des Funktionsbereichs *Stammdatenverwaltung* ab.

CustomerCareConsumer

Der Dienst *CustomerCareConsumer* importiert den Dienst *CustomerCare*, um dessen Leistungen nutzen zu können.

Dieser Dienst ist praktisch nur ein Platzhalter für beliebige Konsumenten des Dienstes *CustomerCare* und wird später für eine GUI-Komponente verwendet.

Collector

Der Dienst *Collector* kapselt die abstrakte Funktionalität eines Sammlers von Verbrauchsdaten, die relevant sind für ein Abrechnungssystem, repräsentiert durch den Dienst `Billing`.

Dieser Dienst bildet die logischen Hauptaufgaben des Funktionsbereichs *Leistungserfassung* ab.

DocumentArchive

Der Dienst *DocumentArchive* kapselt die Funktionalität eines Dokumentenarchivs. Dokumente können an das Archiv übergeben werden, sie erhalten dabei eine eindeutige Dokumentennummer, über die sie zu einem späteren Zeitpunkt wieder zugreifbar sind. Dokumente können auch aus dem Archiv dauerhaft entfernt werden. Ein Eintrag im Dokumentenarchiv besteht aus einer eindeutigen Dokumentennummer, einem Titel, einem beschreibenden Text und dem Dokument selbst. Ein Eintrag wird repräsentiert durch den Datentyp `Document`.

Billing

Der Dienst *Billing* kapselt die Funktionalität einer verbrauchsorientierten Abrechnungseinheit. Die Basisdaten der Abrechnung erhält das Billing durch einen importierten `CustomerCare`-Dienst für die Kundendaten, sowie ein oder mehrere `Collector`-Dienste für die Verbrauchsdaten. Daraus werden zeitraumbezogene Rechnungen generiert, die gedruckt, archiviert und auch per E-Mail verschickt werden können.

BillingConsumer

Der Dienst *BillingConsumer* importiert den Dienst *Billing*, um dessen Leistungen nutzen zu können.

Dieser Dienst ist praktisch nur ein Platzhalter für beliebige Konsumenten des Dienstes *Billing* und wird später für eine GUI-Komponente verwendet.

Neben den Import-Beziehungen werden in Abbildung 4.7 die Eventverschaltungen der charakteristischen Dienste, sowie in Abbildung 4.8 die Konfektionierung der Eigenschaften dargestellt. Die Bedeutung der Events und Eventhandler, wie auch die der Eigenschaften kann der Dienstspezifikation im Anhang 5.2 entnommen werden.

Nachdem nun die für das Fallbeispiel charakteristischen Dienste bekannt sind, widmet sich der folgende Abschnitt den K-Komponenten, die die Dienstspezifikationen erfüllen und das System implementieren.

4.2.2 K-Komponenten

Die aus dem vorherigen Abschnitt bekannten Dienste müssen durch K-Komponenten implementiert werden, um zu einem funktionsfähigen System zu gelangen. In diesem Abschnitt werden die Dienst-Umsetzungen und der fehlende Teil des Komponentenschaltplans vorgestellt.
Abbildung 4.9 zeigt den Komponentenschaltplan des Fallbeispiels, reduziert auf die Importbeziehungen zwischen den einzelnen K-Komponenten. Nachfolgend werden die einzelnen K-Komponenten besprochen, wobei die Vorgaben zur textuellen Repräsentation aus 3.5.2 verwendet werden.
Zunächst muss allerdings ein weiterer Dienst spezifiziert werden, der nicht zu den charakteristischen Diensten zählt. Hierbei handelt es sich um den Dienst `SQLDatabase`. Dieser Dienst-Import wird durch mehrere K-Komponenten zusätzlich zu den in den jeweiligen Diensten spezifizierten Imports vorausgesetzt.

SQLDatabase

Der Dienst *SQLDatabase* kapselt die Funktionalität einer SQL-Datenbank. Über die Schnittstellen des Dienstes kann das Datenbanksystem angegeben werden, auf das zugegriffen werden soll. Außerdem können an die so spezifizierte Datenbank beliebige SQL-Anweisungen übertragen werden. Die vollständige Spezifikation dieses Dienstes ist ebenfalls im Anhang 5.2 zu fin-

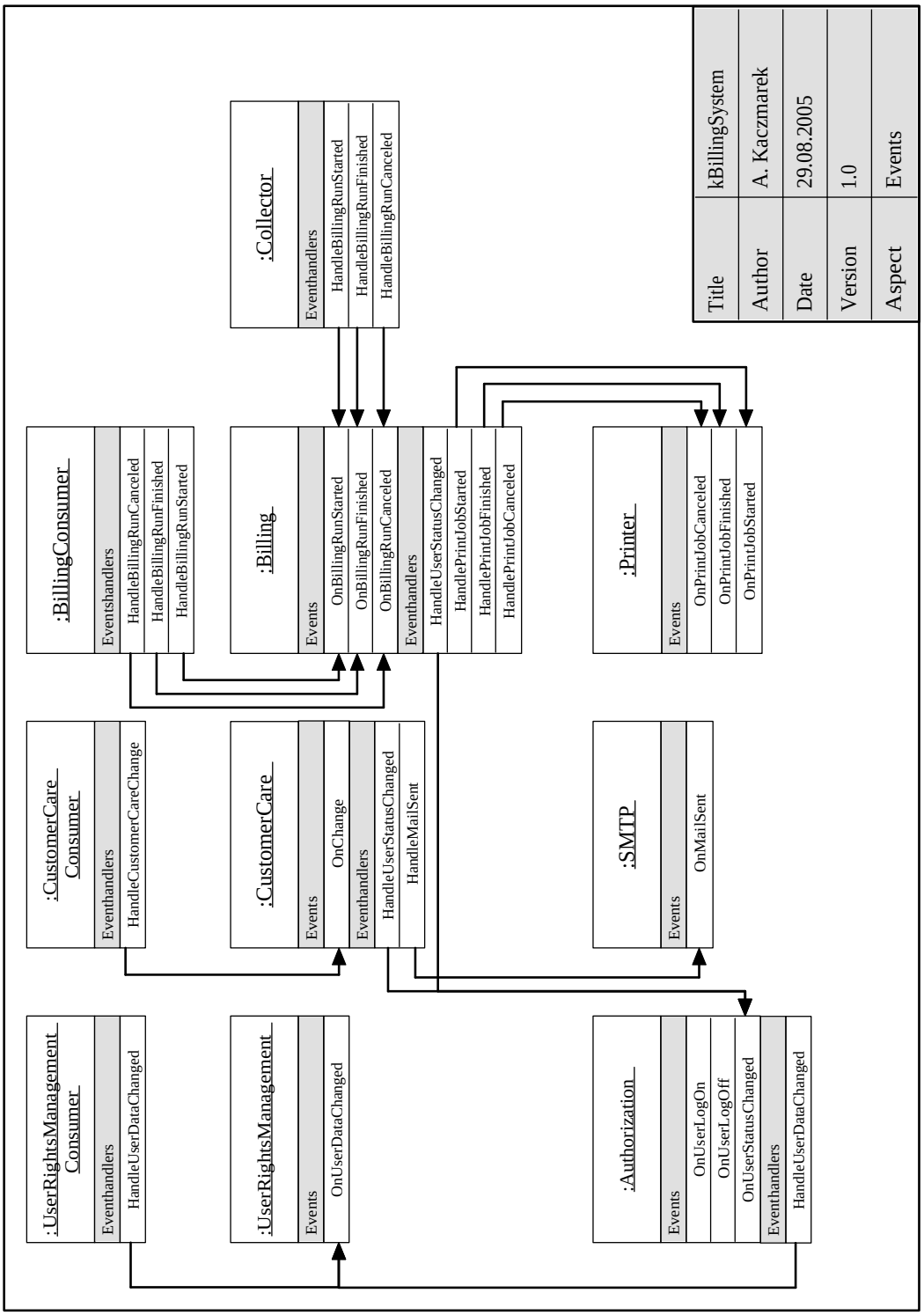

Abbildung 4.7: Schaltplan, Event-Handler-Kopplungen

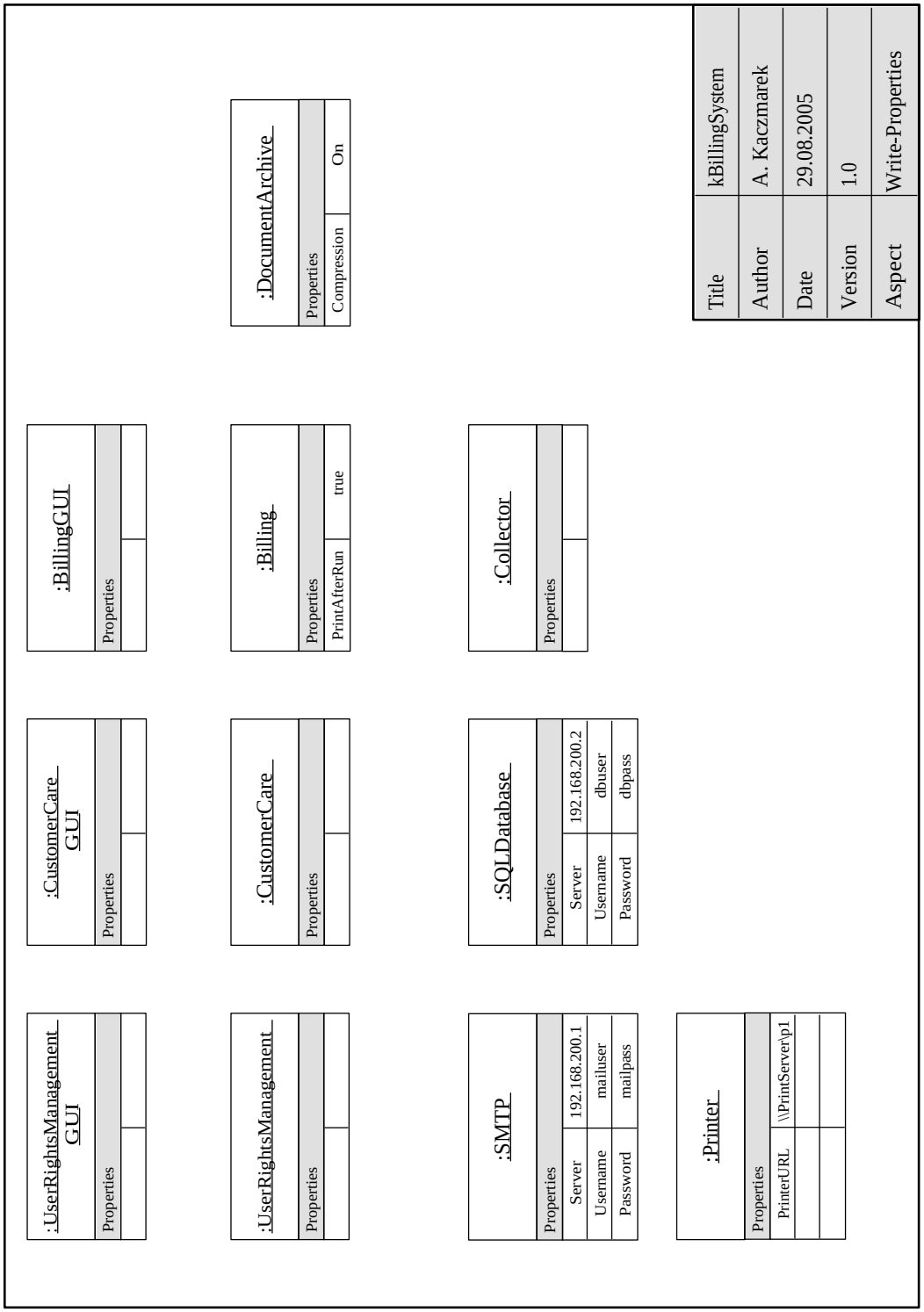

Abbildung 4.8: Schaltplan, Eigenschaften

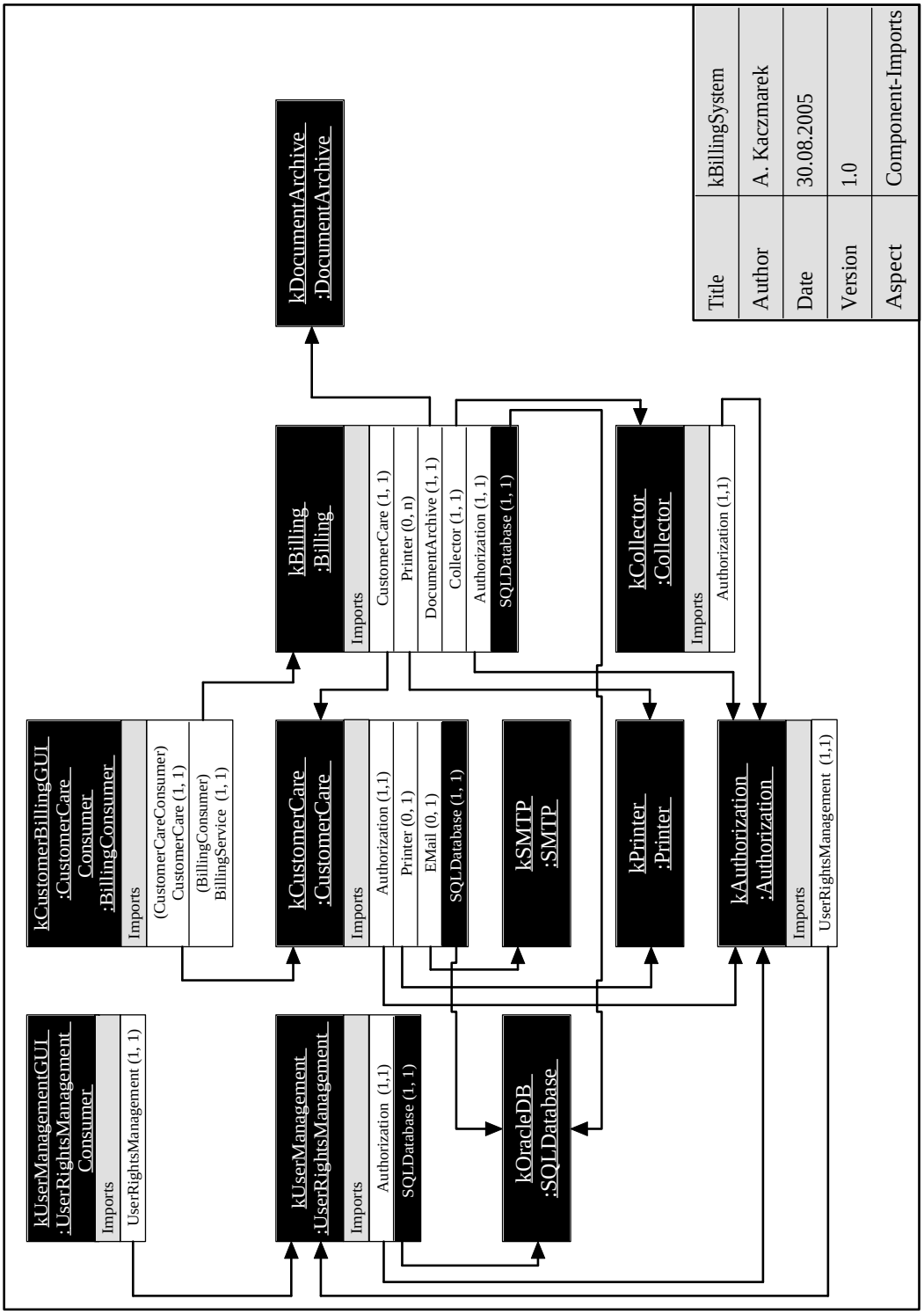

Abbildung 4.9: Schaltplan, Finale Dienst-Import-Kopplungen

den.

Nachdem jetzt alle benötigten Dienste beschrieben sind, widmet sich dieser Abschnitt den verwendeten K-Komponenten. Folgende K-Komponenten kommen zum Einsatz:

kOracleDB

Die K-Komponente *kOracleDB* implementiert den Dienst SQLDatabase. Sie kapselt die Schnittstelle zu einer relationale Oracle-Datenbank gemäß der Dienstspezifikation.

Dienste `SQLDatabase`

kUserManagement

Die K-Komponente *kUserManagement* implementiert den Dienst `UserRights-Management`. Die zu verwaltenden Daten werden dabei in einer SQL-Datenbank abgelegt. Um Zugriff auf die SQL-Datenbank zu erhalten, wird der Dienst `SQLDatabase` importiert.

Dienste `UserRightsManagement`

Imports

Dienst	`SQLDatabase`
Multiplizität	`(1, 1)`
Beschreibung	*Die SQL-Datenbank, in der die Benutzer-, Rollen- und Rechtedaten abgelegt werden.*

kUserManagementGUI

Die K-Komponente *kUserManagementGUI* implementiert den Dienst User-RightsManagementConsumer. Diser Import wird verwendet, um die Leistungen des importierten Dienstes `UserRightsManagement` in einer grafischen Benutzeroberfläche zusammenzufassen. Die Daten der Benutzer, Rechte, Rollen und der Zuordnungen untereinander werden visualisiert und sind über die Oberfläche editierbar.

Dienste `UserRightsManagementConsumer`

kAuthorization

Die K-Komponente *kAuthorization* implementiert den Dienst `Authorization`. Sie stellt auch eine grafische Oberfläche zur Verfügung, über die ein Benutzer die Zugangsdaten eingeben und authentifizieren lassen kann. Ein angemeldeter Benutzer kann sich über die Oberfläche auch wieder abmelden. K-Komponenten, die diese K-Komponente nutzen, müssen demnach keine eigenen Eingabemasken für den Anmeldevorgang bereitstellen.

Dienste `Authorization`

kSMTP

Die K-Komponente *kSMTP* implementiert den Dienst `SMTP`. Gemäßt der Dienstspezifikation können E-Mails via SMTP verschickt werden.

Dienste `SMTP`

kPrinter

Die K-Komponente *kPrinter* implementiert den Dienst `Printer`. Gemäß der Dienstspezifikation werden Druckfunktionalitäten zur Verfügung gestellt.

Dienste `Printer`

kCustomerCare

Die K-Komponente *kCustomerCare* implementiert den Dienst CustomerCare. Die zu verwaltenden Daten werden dabei in einer SQL-Datenbank abgelegt. Um Zugriff auf die SQL-Datenbank zu erhalten, wird der Dienst `SQLDatabase` importiert.

Dienste `CustomerCare`

Imports

Dienst	`SQLDatabase`
Multiplizität	`(1, 1)`
Beschreibung	*Die SQL-Datenbank, in der die Kunden- und Adressdaten abgelegt werden.*

kCollector

Die K-Komponente *kCollector* implementiert den Dienst `Collector`. Gesammelt werden als Verbrauchsdaten sogenannte CDRs (*Call Detail Record*, die

Angaben zu geführten Telefongesprächen beinhalten. Dazu werden die bereitgestellten Daten von Telefonie-Switches abgegriffen, die per HTTP erreichbar sind.

Dienste `Collector`

Eigenschaften

Bezeichner	`Switches`
Typ	`String IPs`
Beschreibung	*Eine kommaseparierte Auflistung von IP-Adressen, unter denen abzurufende Telefonie-Switches erreichbar sind.*

kDocumentArchive

Die K-Komponente *kDocumentArchive* implementiert den Dienst `DocumentArchive`. Gemäß der Dienstspezifikation wird Funktionalität zur Archivierung von digitalen Dokumenten zur Verfügung gestellt.

Dienste `DocumentArchive`

kBilling

Die K-Komponente *kBilling* implementiert den Dienst Billing. Abgerechnet werden Telefonieverbrauchsdaten, die durch den importierten Dienst `Collector` bereitgestellt werden. Der Import des Dienstes `Collector` muss durch die K-Komponente `kCollector` erfüllt werden. Die generierten Rechnungsdaten werden dabei in einer SQL-Datenbank abgelegt. Um Zugriff auf die SQL-Datenbank zu erhalten, wird der Dienst `SQLDatabase` importiert.

Dienste `Billing`

Imports

Dienst	`Collector`
Multiplizität	`(1, 1)`
Beschreibung	*Der Sammler von Telefonieverbrauchsdaten, die Basis für die Abrechnung sind.*
Dienst	`SQLDatabase`
Multiplizität	`(1, 1)`
Beschreibung	*Die SQL-Datenbank, in der die Kunden- und Adressdaten abgelegt werden.*

Constraints

ImplementorOf(Collector)==`kCollector`

kCustomerBillingGUI

Die K-Komponente *kCustomerBillingGUI* implementiert die Dienste *CustomerCareConsumer* und *BillingConsumer*. Sie bietet eine grafische Benutzeroberfläche, die die Leistungen der importierten Dienste `CustomerCare` und `Billing` zusammenführt. Sowohl die vollständigen Kundendaten, wie auch die Rechnungsdaten werden visualisiert. Zusätzlich bietet die Oberfläche Zugriff auf die Methoden der beiden importierten Dienste, um so die Kundendaten zu editieren und Rechnungsläufe zu initiieren.

Dienste `CustomerCare, Billing`

Nach der vollständigen Beschreibung aller zum Einsatz kommenden K-Komponenten wird im folgenden Abschnitt der Komponentenschaltplan des Abrechnungssystems genauer betrachtet.

4.2.3 Komponentenschaltplan

In den letzten beiden Abschnitten wurden die benötigten Dienste und implementierenden K-Komponenten des Fallbeispiels vorgestellt. Dabei ist in den Abbildungen 4.6, 4.7, 4.8 und 4.9 bereits der Komponentenschaltplan dargestellt worden. In diesem Abschnitt wird nochmals einzeln auf die verschiedenen Aspekte des Schaltplans eingegangen.

Abbildung 4.6 beinhaltet nur leere Schaltplankomponenten und deren Import-Kopplungen untereinander. Es sind noch keine erfüllenden K-Komponenten ausgewählt, nur die charakteristischen Dienste und deren direkte Kopplungen stehen in diesem Stadium des Schaltplans bereits fest. Dabei ergeben sich die Import-Kopplungen direkt aus den Abhängigkeiten, die in den Dienstspezifikationen vorgegeben sind.

In Abbildung 4.7 wird der Aspekt der Event-Handler-Kopplungen des Schaltplans dargestellt, ebenso nur bestehend aus leeren Schaltplankomponenten. Auch diese Kopplungen ergeben sich direkt aus den Dienstspezifikationen.
Teil der Darstellung ist beispielsweise die Event-Handler-Kopplungen des Eventhandlers `CustomerCare.HandlerMailSent` auf die Eventquelle `SMTP.On MailSent`.
Diese Aufschaltung bedeutet, dass bei Eintritt des Events `SMTP.OnMailSent`

der Eventhandler `CustomerCare.HandlerMailSent` darüber durch Übermittlung einer Eventnachricht informiert wird.

In 4.8 werden die Belegungen der Write-Eigenschaften dargestellt. Auch in diesem Fall sind nur leere Schaltplankomponenten im Diagramm enthalten. Die gemäß der Dienstspezifikationen zu setzenden Eigenschaften der einzelnen K-Komponenten werden aufgelistet. Beispielsweise wird für die leere Schaltplankomponenten `:SMTP` die Eigenschaft `Server` mit der IP des SMTP-Servers gesetzt.

Im Gegensatz zu den vorherigen Diagrammen ist das Diagramm in Abbildung 4.9 vollständig, denn es enthält keine leeren Schaltplankomponenten. Die vorherigen Platzhalter sind mit erfüllenden K-Komponenten besetzt, die zusätzlich benötigte K-Komponente `kOracleDB` ist in den Plan aufgenommen worden. Diese K-Komponente wurde nicht durch die Dienste gefordert, sondern ist ein zusätzlich angeforderter Import der verwendeten K-Komponenten.
Diese durch die K-Komponenten zusätzlich definierten Imports werden dadurch gekennzeichnet, dass ein Import mit weißer Schrift auf schwarzem Grund dargestellt wird, anstatt mit schwarzer Schrift auf weißem Grund.

@@ak Das Fallbeispiel ist mit der Besprechung des Komponentenschaltplans abgeschlossen. Die Ergebnisse und Erkenntnisse werden in der abschließenden Zusammenfassung betrachtet, bevor im Schlusskapitel der Arbeit das allgemeine Fazit gezogen und der Ausblick besprochen wird.

4.3 Zusammenfassung

Das in diesem Kapitel besprochene Fallbeispiel hat gezeigt, dass ein realitätsnahes Szenario eines Softwaresystems anhand des im vorherigen Kapitel vorgestellten Komponentenkonzeptes realisiert werden kann. Deutlich geworden ist der Spezifikationscharakter der Dienstbeschreibungen, der das gesamte Beispiel dominiert. Überraschend ist diese Beobachtung allerdings nicht, da bei der komponentenorientierten Umsetzung eines Softwaresystems die Schnittstellen zwischen den Komponenten im Mittelpunkt stehen. Die Spezifikation dieser Schnittstellen in diesem Fall durch Dienste ist essentieller Bestandteil der komponentenorientierten Systementwicklung.

Das nun folgende letzte Kapitel fasst die Ergebnisse dieser Arbeit zusammen und bietet abschließend einen Ausblick auf zukünftige Aufgaben.

Kapitel 5

Zusammenfassung und Ausblick

Das definierte Ziel dieser Arbeit war es, einen Überblick über heutige Auffassungen und Umsetzungen der komponentenorientierten Entwicklung zu geben. Aufbauend auf den aus diesen Betrachtungen gewonnenen Erkenntnissen sollte ein Komponentenkonzept entwickelt werden, dass die Ziele der Komponentenorientierung hinsichtlich Komponierbarkeit und Wiederverwendbarkeit erfüllt. Dabei wurde gezeigt, dass die genaue Spezifikation der Komponentenleistungen integraler Bestandteil eines solchen Konzeptes sein muss, um die zuvor genannten Ziele zu erreichen.

In diesem Kapitel werden die Ergebnisse und Erkenntnisse der Arbeit zusammengefasst und besprochen. Außerdem umfasst dieses Kapitel den Ausblick auf mögliche zukünftige Arbeiten, die auf den Resultaten dieser Arbeit aufsetzen.

5.1 Ergebnisse der Arbeit

Die definierten Ziele der Arbeit sind erreicht worden. Es wurden Erkenntnisse über vorherrschende Meinungen und Ansichten zum Thema *Komponentenorientierung* gewonnen, eine Sammlung von Anforderungen an Komponententechnologien wurde aufgestellt, ein ganzes Komponentenkonzept inklusive ausführlichem Fallbeispiel wurde erstellt. Diese einzelnen Punkte werden im Folgenden in Hinblick auf die erreichten Ziele betrachtet.

Durch die **Literaturbetrachtung in Kapitel 2** wurden grundlegende Merkmale gesammelt, die es ermöglichen, den Begriff der *Komponentenorientierung* besser zu fassen. Relativ schnell wurde allerdings auch ersichtlich, dass verschiedene Autoren zwar teilweise Gemeinsamkeiten bei der Sicht auf soge-

nannte Komponenten aufweisen, aber mindestens ebenso viele Unterschiede. Durch das Aufstellen des **Anforderungskataloges in 2.2** wurden die Gemeinsamkeiten und Unterschiede zusammengeführt und es entstand dabei ein Mittel zur besseren Bewertung von Komponententechnologien und -ansätzen.

Durch die **Betrachtung und Bewertung** heute verfügbarer selbsternannter **Komponententechnologien** anhand des zuvor aufgestellten Anforderungskataloges wurde nachfolgend festgestellt, dass keine der betrachteten Technologien der Vision der Komponentenorientierung in Hinblick auf Komponierbarkeit und Wiederverwendbarkeit gerecht wird. Vor allem technologische Probleme werden durch die besprochenen Ansätze in den Mittelpunkt gestellt und auch gelöst. Die Beschränkung auf technologische Probleme meist syntaktischer Natur ist allerdings unzureichend, um die Komponentenorientierung vollständig und umfassend zu realisieren.

Aus diesen Unzulänglichkeiten heraus motiviert wurde in **Kapitel 3 ein eigenes Komponentenkonzept** entwickelt und vorgestellt, dass sich weniger um technologische Aspekte kümmert, als um grundlegende Überlegungen und Umsetzungsvorschläge für die Vision der Komponentenorientierung. Im Fokus des Konzeptes steht die **Komponierbarkeit von Komponenten**.
Durch die Erkenntnisse und Ergebnisse der Konzeptentwicklung ist äußerst deutlich geworden, dass Grundlage für die Komponierbarkeit das **Aufstellen klarer Spezifikationen** ist, vor allem syntaktischer aber auch semantischer Art. Ohne diese Spezifikationen, die sich im vorgestellten Konzept als Dienste wiederfinden, wäre die angestrebte Komponierbarkeit nicht erreichbar gewesen.

Durch das **abschließende Fallbeispiel** wurde an einem möglichst realistischen Beispiel demonstriert, dass das vorgestellte Komponentenkonzept tragfähig und einsatzfähig ist. Die meisten Merkmale des Konzeptes wurden im Fallbeispiel aufgegriffen und deren Bedeutung sowie Verwendung demonstriert.

Eine weitere indirekte Erkenntnis dieser Arbeit ist, dass die Komponentenorientierung, wie sie im Rahmen dieser Arbeit verstanden wird, **nicht in Konkurrenz** zur Objektorientierung steht. Vielmehr setzt die Komponentenorientierung auf der Objektorientierung auf. Beide Ansätze zusammengenommen können so das wohl größte Potential entwickeln, da sie auf **zwei verschiedenen Ebenen** einsetzt werden sollten.
Objektorientierte Entwicklung wird und sollte verwendet werden, um Komponenten zu erstellen. Hierzu werden ausgeprägte Programmierfähigkeiten

und technisches Wissen vorausgesetzt, die heutige typische Softwareentwickler aufweisen.
Die komponentenorientierte Entwicklung wird in der nächst abstrakteren Entwicklungsebene eingesetzt, wenn es darum geht, aus vorgefertigten Komponenten ganze Softwaresysteme zusammen zu setzen. Das Profil eines sogenannten Systemassemblierers, der diese Aufgabe übernehmen kann, weist zwar auch technisches Wissen auf, vor allem aber Kenntnisse über die Anwendungsdomäne des zu entwickelnden Systems.
Durch diese Art der Trennung sollte ein noch höheres Maß an Wiederverwendung und auch Standardisierung erreicht werden können, als es schon heute der Fall ist.

5.2 Ausblick

Die Entwicklung eines neuen Komponentenkonzeptes ist ein umfangreiches und aufwändiges Unterfangen. Im Rahmen dieser Arbeit wurden bereits viele Problemstellungen betrachtet und Vorschläge für deren Lösung erarbeitet, die sich im Komponentenkonzept in Kapitel 3 wiederfinden.
Allerdings blieben dabei einige Fragen unbeantwortet bzw. musste teilweise vereinfachend abstrahiert werden, um die Komplexität der Thematik in Grenzen zu halten.

In diesem abschließenden Abschnitt soll betrachtet werden, welche Probleme und Aufgaben vorrangig anzugehen sind, um das vorgestellte Komponentenkonzept aus der theoretischen Konzeption in die praktische Anwendung zu führen.

Das **Kernstück** des vorgestellten Komponentenkonzeptes ist der **Komponentenschaltplan**. In Kapitel 3 wurden die strukturellen Merkmale dieser Schaltpläne besprochen, eine praktische Umsetzung anhand einer geeigneten Sprache fehlt allerdings. Da es sich bei Komponentenschaltplänen um typisierte, attributierte, gerichtete Graphen handelt, kann eine dazu passende Beschreibungssprache für den Zweck der textuellen Schaltplanrepräsentation herangezogen werden. Eine Sprache, die diese Anforderungen erfüllt, ist GXL (*Graph Exchange Language*, vgl. [14]).
Das in Abbildung 3.2 dargestellte Schema für Komponentenschaltpläne kann im Fall der Verwendung von GXL als Vorlage für das GXL-Schema dienen. Komponentenschaltpläne wären somit ohne bedeutenden Mehraufwand direkt in einer Graphsprache erfassbar. Ein weiterer Vorteil, der sich durch die

Verwendung von GXL ergäbe, sind die in dessen Umfeld bereits vorhandenen Werkzeuge. So existiert bereits eine GXL-API (*GXL Application Programming Interface*, vgl. [6]), über die GXL-Graphen aufgebaut, sowie strukturell durchlaufen werden können. Des Weiteren besteht mit dem GXL-Validator (vgl. [9]) ein Validierungswerkzeug, dass sowohl GXL-Schemata wie auch GXL-Instanzen auf Korrektheit gemäß der zugrundeliegenden Schemainformationen untersuchen kann. Angewandt auf GXL-Komponentenschaltpläne kann ein gegebener Schaltplan gegenüber dem Komponentenschaltplanschema validiert werden, um die syntaktische Korrektheit der Verschaltungen zu überprüfen.

Ein Aspekt des vorgestellten Komponentenkonzeptes ist der Einsatz der sogenannten **Glassbox-Komponenten**. Bisher wurde vereinfachend nur davon ausgegangen, dass die Logik per Skript angegeben wird, nicht aber welche Skriptsprache zum Einsatz kommt und wie diese angewandt wird. Demzufolge muss für eine praktische Umsetzung der Glassbox-Komponenten-Idee eine Skriptsprache ausgewählt und deren Interpretation festgelegt werden. Hier bietet es sich an, auf bestehende und erprobte Sprachen wie z. B. LUA[1] zurückzugreifen.

Eine weitere **Sprache** die benötigt wird, ist die für die Erfassung der **Constraints** in Diensten und K-Komponenten. Auch hier wurde im Rahmen dieser Arbeit von der späteren Umsetzung abstrahiert. Die Constraint-Sprache muss eine logische Sprache sein, mit der komplexe Ausdrücke über Eigenschaften und strukturelle Kopplungen unter Diensten und K-Komponenten ausgedrückt werden können.

Nach Klärung und Lösung dieser offenen Probleme kann mit der Realisation eines **Prototypen** begonnen werden. Für einen solchen Prototyp müssen zunächst die Rahmenbedingungen festgelegt werden. Das bedeutet, dass zu bestimmen ist, ob es sich bei K-Komponenten um dauerhafte Service-Objekte oder um instanziierbare Entitäten handelt (vgl. 3.6.2). Auch bleibt festzulegen, ob ein Komponentenserveransatz (vgl. 3.7.5) verfolgt werden soll, oder nicht.

Obwohl im Rahmen des Fallbeispiels enthalten, wurde ein wichtiger Aspekt eines fertigen Komponentensystems bisher fast vollständig ausgespart: die **grafische Benutzeroberfläche**. Praktisch jedes Anwendungssystem benötigt eine solche Benutzerschnittstelle, trivial ist die Umsetzung im Kontext

[1] vgl. http://www.lua.org

eines Komponentensystems wie dem hier vorgestellt allerdings nicht. Geht man davon aus, dass auf dem durch einen Anwender eingesetzten Rechner eine Laufzeitumgebung für K-Komponenten existiert, so könnte diese Laufzeitumgebung auch für die Darstellung grafischer Oberflächen zuständig sein. Vergleichbar ist dieser Ansatz mit der Lösung des gleichen Problems durch Eclipse (vgl. [1]). Auch hier existiert eine ausführende Umgebung, die einen Rahmen für die Anzeige der grafischen Benutzerschnittstelle liefert.

Innerhalb einer prototypischen Umsetzung, spätestens aber bei Verlassen des prototypischen Status, sollte der Aspekt der Dienst- & Komponentenregistratur nochmals genauer betrachtet werden. Vorbild für einen solchen Verzeichnisdienst könnte das UDDI-Protokoll (*Universal Description, Discovery and Integration*[2]) darstellen, das aus dem Web-Service-Bereich stammt.

Dienste und K-Komponenten sind auf syntaktischer Ebene klar spezifiziert. Die Semantikbeschreibungen beschränken sich zumeist auf **natürlichsprachliche Texte**. Diese Beschränkung war im Rahmen der vorliegenden Arbeit notwendig, um die breiten Grundlagen zu erarbeiten, ohne zu sehr in einen Detailaspekt einzusteigen.
Im Gegensatz zu den meisten in Kapitel 2 besprochenen Komponententechnologien ist die natürlichsprachliche Semantikbeschreibung von Diensten und K-Komponenten bereits ein Schritt in die richtige Richtung. Die Erfassung einer detaillierteren und automatisch auswertbaren Semantik für Dienste und K-Komponenten ist aber ein sehr umfangreiches Thema, das genügend Stoff für weitere Arbeiten liefert.
Vorstellbar sind Semantikbeschreibungen, die auf **Ontologien** [3] basieren. Verfolgt man solche Ansätze zur Erfassung maschinell auswertbarer Dienst- und K-Komponentensemantik, dann ergeben sich vor allem für das Aufstellen von Komponentenschaltplänen neue Möglichkeiten zur Suche nach benötigten Diensten und zugehörigen K-Komponenten.

Diese Arbeit hat gezeigt, dass die komponentenorientierte Systemspezifikation und -entwicklung ein äußerst interessantes und vielversprechendes Forschungsgebiet darstellt. Durch die Betrachtung des heutigen Entwicklungsstandes, dem Aufstellen eines Anforderungskataloges an Komponenten und der Entwicklung eines neuen Komponentenkonzeptes konnten vorhandene Erkenntnisse zusammengefasst und neue gewonnen werden.

[2] vgl. http://www.uddi.org
[3] vgl. http://de.wikipedia.org/wiki/Ontologie_(Informatik), 20.09.2005

Um die letzendlichen Ziele der Komponentenorientierung zu erreichen, können die in dieser Arbeit unternommenen Schritte eine erste Annäherung darstellen. Wie aber zuletzt im Ausblick zu erkennen war, müssen weitere Schritte folgen.

Dienste des Fallbeispiels

UserRightsManagement

Der Dienst *UserRightsManagement* dient der Verwaltung von Benutzerdaten und der Zuordnung von Berechtigungen zu Benutzern oder Benutzergruppen. Der gesamten Verwaltung liegen folgende Annahmen zugrunde: Es gibt Benutzer, die durch den Datentyp `User` dargestellt werden. Es gibt Berechtigungen, die durch den Datentyp `Right` dargestellt werden. Es gibt Rollen, die durch den Datentyp `Role` dargestellt werden. Einer Rolle können Berechtigungen zugeordnet werden. Einem Benutzer können Rollen zugeordnet werden. Ein Benutzer besitzt genau dann eine Berechtigung, wenn er eine Rolle besitzt, der die betreffende Berechtigung zugeordnet ist.

Imports

Dienst	`Authorization`
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um festzustellen, ob der Benutzer des Systems berechtigt ist, bestimmte Aktionen durchzuführen.*

Datentypdefinitionen

Bezeichner	`User`

Struktur	`String Username`
	`String Password`
	`String Lastname`
	`String Firstname`

Bezeichner	`Right`
Struktur	`String Rightname`

Bezeichner	`Role`
Struktur	`String Rolename`

Exceptions

Bezeichner	`UsernameAlreadyExists`
Parameter	`User u`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, der Verwaltung einen neuen Benutzer* `u` *hinzuzufügen, dessen Benutzername bereits in der Verwaltung existiert. Ein Benutzername darf maximal einmal vergeben sein. Mit der Exception wird der bereits in der Verwaltung existente Benutzer* `u` *übergeben, der den identischen Benutzernamen besitzt.*

Bezeichner	`UserDoesNotExists`
Parameter	`User u`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, in der Verwaltung auf den Benutzer* `u` *zuzugreifen, der aber nicht in der Verwaltung existiert.*

Bezeichner	`RolenameAlreadyExists`
Parameter	`Role r`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, der Verwaltung eine neue Berechtigung* `r` *hinzuzufügen, deren Bezeichner bereits als Berechtigung in der Verwaltung existiert.*

Bezeichner	`RoleDoesNotExists`
Parameter	`Role r`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, in der Verwaltung auf die Rolle* `r` *zuzugreifen, die aber nicht in der Verwaltung existiert.*

Bezeichner	`RightnameAlreadyExists`
Parameter	`Right r`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, der Verwaltung eine neue Rolle* `r` *hinzuzufügen, deren Bezeichner bereits als Rolle in der Verwaltung existiert.*

Bezeichner	`RightDoesNotExists`
Parameter	`Right r`
Beschreibung	*Diese Exception wird ausgelöst, falls versucht wird, in der Verwaltung auf die Berechtigung* `r` *zuzugreifen, die aber nicht in der Verwaltung existiert.*

Bezeichner	`RightInUse`
Parameter	`Right r`
Beschreibung	*Diese Exception wird ausgelöst, falls explizit versucht wird, der Verwaltung die Berechtigung* `r` *zu entfernen, die mindestens einer Rolle zugeordnet ist.*

Bezeichner	`RoleInUse`
Parameter	`Role r`
Beschreibung	*Diese Exception wird ausgelöst, falls explizit versucht wird, der Verwaltung die Rolle* `r` *zu entfernen, die mindestens einem Benutzer zugewiesen ist.*

Methoden

Bezeichner	`AddUser`
Parameter	`User u`
Rückgabewerte	`void`
Exceptions	`UsernameAlreadyExists`
Beschreibung	*Der in* `u` *angegebene Benutzer wird der Verwaltung ohne Rollenzuordnung hinzugefügt.*

Bezeichner	`RemoveUser`
Parameter	`User u`
Rückgabewerte	`void`
Exceptions	`UserDoesNotExists`
Beschreibung	*Der in* `u` *angegebene Benutzer wird aus der Verwaltung mitsamt allen Rollenzuordnungen ausgetragen. Falls kein Benutzer* `u` *in der Verwaltung existiert, wird eine* `UserDoesNotExist`-*Exception geworfen.*

Bezeichner	`AddRight`
Parameter	`Right r`
Rückgabewerte	`void`
Exceptions	`RightnameAlreadyExists`
Beschreibung	*Die in* `r` *angegebene Berechtigung wird der Verwaltung ohne Zuordnung zu einer Rolle hinzugefügt.*

Bezeichner	`RemoveRight`
Parameter	`Right r` `Boolean RemoveUnusedOnly`
Rückgabewerte	`void`
Exceptions	`RightDoesNotExist` `RightInUse`
Beschreibung	*Die in* `r` *angegebene Berechtigung wird genau dann aus der Verwaltung entfernt, wenn sie keiner Rolle zugeordnet ist, oder wenn der Parameter* `RemoveUnusedOnly` *den Wert* `false` *besitzt. Existiert eine Rollenzuordnung und ist der Wert des Parameters* `RemoveUnusedOnly` *gleich* `true`*, so wird die Berechtigung nicht entfernt und die entsprechende Exception wird geworfen. Falls keine Berechtigung* `r` *in der Verwaltung existiert, wird eine* `RightDoesNotExist`*-Exception geworfen.*

Bezeichner	`AddRole`
Parameter	`Role r`
Rückgabewerte	`void`
Exceptions	`RolenameAlreadyExists`
Beschreibung	*Die in* `r` *angegebene Rolle wird der Verwaltung ohne Zuordnung einer Berechtigung oder Zuordnung an einen Benutzer hinzugefügt.*

Bezeichner	`RemoveRole`
Parameter	`Role r` `Boolean RemoveUnusedOnly`
Rückgabewerte	`void`
Exceptions	`RoleDoesNotExist` `RoleInUse`
Beschreibung	*Die in* `r` *angegebene Rolle wird genau dann aus der Verwaltung entfernt, wenn sie keinem Benutzer zugewiesen ist, oder wenn der Parameter* `RemoveUnusedOnly` *den Wert* `false` *besitzt. Existiert eine Zuweisung zu einem Benutzer und ist der Wert des Parameters* `RemoveUnusedOnly` *gleich* `tru`*, so wird die Rolle nicht entfernt und die entsprechende Exception wird geworfen. Falls keine Rolle* `r` *in der Verwaltung existiert, wird eine* `RoleDoesNotExist`*-Exception geworfen.*

Bezeichner	`GrantRightToRole`
Parameter	`Right right` `Role role`
Rückgabewerte	`void`
Beschreibung	*Der Rolle* `role` *wird die Berechtigung* `right` *zugeordnet. Besitzt die Rolle bereits diese Berechtigung, so findet keine Veränderung statt.*

Bezeichner	`RemoveRightFromRole`
Parameter	`Right right` `Role role`
Rückgabewerte	`void`
Beschreibung	*Der Rolle* `role` *wird die Berechtigung* `right` *entzogen. Besitzt die Rolle diese Berechtigung nicht, so findet keine Veränderung statt. Falls keine Berechtigung* `r` *in der Verwaltung existiert, wird eine* `RightDoesNotExist`*-Exception geworfen. Falls keine Rolle* `r` *in der Verwaltung existiert, wird eine* `RoleDoesNotExist`*-Exception geworfen.*

Bezeichner	`UserOwnsRight`
Parameter	`User u` `Right r`
Rückgabewerte	`Boolean Result`
Exceptions	`UserDoesNotExist` `RightDoesNotExist`
Beschreibung	*Es wird im Rückgabewert* `Result` *der boolsche Wert* `true` *genau dann zurückgegeben, wenn der Benutzer* `u` *mindestens eine Rolle besitzt, der die Berechtigung* `r` *zugeordnet ist. Ansonsten lautet er Rückgabewert* `false`*. Falls kein Benutzer* `u` *in der Verwaltung existiert, wird eine* `UserDoesNotExist`*-Exception geworfen. Falls keine Berechtigung* `r` *in der Verwaltung existiert, wird eine* `RightDoesNotExist`*-Exception geworfen.*

Bezeichner	`GetUsers`
Rückgabewerte	`{User} Users`
Beschreibung	*Es wird im Rückgabewert* `Users` *die Menge aller Benutzer geliefert, die in der Benutzerverwaltung eingetragen sind.*

Bezeichner	`GetRights`
Parameter	`Role r`
Rückgabewerte	`{Right} Rights`
Beschreibung	*Es wird im Rückgabewert* `Rights` *die Menge aller Berechtigungen geliefert, die in der Benutzerverwaltung eingetragen sind und der angegebenen Rolle* `r` *zugeordnet sind. Ist der Parameter* `r` *nicht gesetzt, so werden alle bekannten Berechtigungen zurückgegeben. Falls keine Rolle* `r` *in der Verwaltung existiert, wird eine* `RoleDoesNotExist`*-Exception geworfen.*

Bezeichner	`GetRights`
Parameter	`User u`
Rückgabetyp	`{Right} Rights`
Beschreibung	*Es wird im Rückgabewert* `Rights` *die Menge aller Berechtigungen geliefert, die in der Benutzerverwaltung eingetragen und die einer Rolle zugeordnet sind, die wiederum dem angegebenen Benutzer* `u` *zugwiesen ist. Ist der Parameter* `u` *nicht gesetzt, so werden alle bekannten Berechtigungen zurückgegeben. Falls kein Benutzer* `u` *in der Verwaltung existiert, wird eine* `UserDoesNotExist`*-Exception geworfen.*

Bezeichner	`GetRoles`
Parameter	`User u`
Rückgabewerte	`{Role} Roles`
Beschreibung	*Es wird im Rückgabewert* `Roles` *die Menge aller Rollen geliefert, die in der Benutzerverwaltung eingetragen sind und dem angegebenen Benutzer* `u` *zugewiesen sind. Ist der Parameter* `u` *nicht gesetzt, so werden alle bekannten Rollen zurückgegeben. Falls kein Benutzer* `u` *in der Verwaltung existiert, wird eine* `UserDoesNotExist`*-Exception geworfen.*

Events

Bezeichner	`OnUserDataChanged`
Parameter	`User u`
Beschreibung	*Die direkten oder zugeordneten Daten des Benutzers* `u` *haben sich geändert. Dabei können Änderungen der Namensangaben, des Passworts, der Rollenzuordnung oder der Berechtigungszuordnung der zugewiesenen Rollen handeln.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

UserRightsManagementConsumer

Der Dienst *UserRightsManagementConsumer* importiert den Dienst *UserRightsManagement*, um dessen Leistungen nutzen zu können.

Imports

Dienst	`UserRightsManagement`
Rolle	
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um Zugriff auf die durch den importierten Dienst gekapselte Benutzerverwaltung zu erhalten.*

Authorization

Der Dienst *Authorization* kapselt Funktionalität zur Anmeldung eines Systembenutzers, sowie Überprüfung der Berechtigungen des angemeldeten Benutzers. Der Dienst kennt ein Administratorkonzept. Über entsprechende Eigenschaften lässt sich der Administratorzugang konfigurieren. Der Administrator hat per Definition jede mögliche Berechtigungen.

Imports

Dienst	`UserRightsManagement`
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um die Berechtigungen des angemeldeten Benutzers überprüfen zu können.*

Exceptions

Bezeichner	`NoAuthentication`
Beschreibung	*Aktuell ist kein Benutzer als angemeldet registriert.*

Eigenschaften

Bezeichner	`AdminUsername`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Der Benutzername des Administrators.*

Bezeichner	`AdminPassword`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Das Passwort des Administrators.*

Methoden

Bezeichner	`Authenticate`
Parameter	`String Username` `String Password`
Rückgabewerte	`Boolean AuthenticationSuccessful` `String ResultText`
Beschreibung	*Der Benutzer mit dem Benutzernamen* `Username` *und dem Passwort* `Password` *wird gegenüber dem importierten Dienst* `UserRightsManagement` *authentifiziert und im Erfolgsfall bleibt er als angemeldeter Benutzer registriert. Der Rückgabewert* `AuthenticationSuccessful` *ist genau dann gleich* `true`*, wenn die Kombination aus Benutzernamen und Passwort in der importierten Benutzerverwaltung existieren, oder wenn sie mit der Kombination aus den Werten der Eigenschaften* `AdminUsername` *und* `AdminPassword` *identisch sind. Der Rückgabewert* `ResultText` *enthält eine natürlichsprachliche Umschreibung des Methodenergebnisses.*

Bezeichner	`LogOff`
Rückgabewerte	`void`
Beschreibung	*Ist ein Benutzer aktuell als angemeldet registriert, wird diese Registrierung aufgehoben.*

Bezeichner	`IsAuthorized`
Parameter	`String Rightname`
Rückgabewerte	`Boolean Result`
Exceptions	`NoAuthentication`
Beschreibung	*Ist ein Benutzer aktuell als angemeldet registriert, wird unter Zuhilfenahme des importierten Dienstes* `UserRightsManagement` *überprüft, ob der Benutzer die Berechtigung* `Rightname` *besitzt. Falls aktuell kein Benutzer als angemeldet registriert ist, wird eine* `NoAuthentication`*-Exception geworfen.*

Events

Bezeichner	`OnUserLogOn`
Parameter	`String Username`
Beschreibung	*Der Benutzer mit dem Benutzernamen* `Username` *wurde erfolgreich authentifiziert.*

Bezeichner	`OnUserLogOff`
Parameter	`String Username`
Beschreibung	*Der Benutzer mit dem Benutzernamen* `Username` *hat sich abgemeldet.*

Bezeichner	`OnUserStatusChanged`

Parameter	`String Username`
	`(Authenticated,`
	` LoggedOff,`
	` RightChange) Status`

Beschreibung	*Der Status des Benutzers mit dem Benutzernamen* `Username` *hat sich geändert. Die Belegung des Parameters* `Status` *zeigt an, welche Statusänderung aufgetreten ist. Entspricht die Belegung* `Authenticated`*, dann wurde eine erfolgreiche Authentifizierung des Benutzers durchgeführt. Entspricht sie* `LoggedOff`*, dann wurde der Benutzer abgemeldet. Entspricht sie* `RightChange`*, dann haben sich die Berechtigungen des Benutzers geändert.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

SMTP

Der Dienst *SMTP* kapselt die Funktionalitäten eines einfachen E-Mail-Versenders nach dem SMTP-Protokoll (*Simple Mail Transport Protocol*). E-Mails samt Anhang können über die Schnittstelle des Dienstes versendet werden.

Datentypdefinitionen

Bezeichner	`EMail`

Struktur	`String To`
	`String Cc`
	`String Bcc`
	`String Subject`
	`String Body`
	`{String AttachmentURL}`

Exceptions

Bezeichner	`InvalidIP`
Parameter	`String ServerIP`
Beschreibung	*Die in den Eigenschaften angegebene IP des SMTP-Servers* `ServerIP` *entspricht nicht dem IPv4-Format.*

Bezeichner	`ServerNotFound`
Parameter	`String ServerIP`
Beschreibung	*Der in den Eigenschaften angegebene SMTP-Server mit der IP* `ServerIP` *konnte nicht gefunden werden.*

Bezeichner	`ServerDoesNotRespond`
Parameter	`ServerIP`
Beschreibung	*Der in den Eigenschaften angegebene SMTP-Server mit der IP* `ServerIP` *antwortet nicht.*

Bezeichner	`InvalidUserData`
Parameter	`String Username` `Password Password`
Beschreibung	*Die in den Eigenschaften angegebenen Zugangsdaten des Benutzers* `Username` *mit dem Passwort* `Password` *sind nicht korrekt.*

Bezeichner	`MailServerError`
Parameter	`String ErrorText`
Beschreibung	*Der Mailserver hat einen Fehler gemeldet, zusammen mit dem Fehlertext* `ErrorText`*.*

Bezeichner	`AttachmentNotFound`
Parameter	`String attachmentURL`
Beschreibung	*Der Anhang* `attachmentURL` *konnte nicht gefunden werden.*

Eigenschaften

Bezeichner	`ServerIP`
Typ	`String`
Zugriff	`Write`
Exceptions	`InvalidIP`
Beschreibung	*Die IP des SMTP-Servers nach IPv4-Format.*

Bezeichner	`Username`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Der benötigte Benutzername für die Autorisierung am SMTP-Server.*

Bezeichner	`Password`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Das benötigte Password für die Autorisierung am SMTP-Server.*

Methoden

Bezeichner	`SendMail`
Parameter	`EMail email`
Rückgabewerte	`void`

Exceptions	`ServerNotFound`
	`ServerDoesNotRespond`
	`InvalidUserData`
	`MailServerError`

Beschreibung	*Versendet die angegebene E-Mail* `email` *über den in den Eigenschaften angegebenen SMTP-Server mit den angegebenen Zugangsdaten. Falls der SMTP-Server nicht gefunden wird, wird eine* `ServerNotFound`*-Exception geworfen. Falls der SMTP-Server erreichbar ist, aber nicht auf SMTP reagiert, wird eine* `ServerDoesNotRespond`*-Exception geworfen. Falls die Benutzerdaten bezogen auf den SMTP-Server nicht korrekt sind, wird eine* `InvalidUserData`*-Exception geworfen. Falls der SMTP-Server eine sonstige Fehlermeldung zurückliefert, wird eine* `MailServerError`*-Exception geworfen.*

Events

Bezeichner	`OnMailSent`
Parameter	`EMail email`
Beschreibung	*Die durch den Parameter* `email` *repräsentierte E-Mail ist erfolgreich versendet worden.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Printer

Der Dienst *Printer* kapselt die Funktionalität eines Druckers. Dokumente können an den Drucker übergeben und in beliebiger Anzahl ausgedruckt werden.

Datentypdefinitionen

Bezeichner	`PrintJob`

Struktur	`String DocumentURL`
	`Integer NumberOfCopies`
	`Boolean Sorted`

Exceptions

Bezeichner	`PrinterDoesNotRespond`
Parameter	`String PrinterURL`
Beschreibung	*Der Drucker, der an der Stelle* `PrinterURL` *zu finden sein soll, antwortet nicht.*

Bezeichner	`PrinterError`
Parameter	`String ErrorText`
Beschreibung	*Der Drucker hat einen Fehler veruracht und liefert dazu die Fehlermeldung* `ErrorText`*.*

Eigenschaften

Bezeichner	`PrinterURL`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Die Adresse des Druckers, der zum Drucken der Druckaufträge benutzt werden soll.*

Methoden

Bezeichner	`ExecutePrintJob`
Parameter	`Printjob pj`
Rückgabewerte	`String PrintJobNo`
Exceptions	`PrinterDoesNotRespond` `PrinterError`

Beschreibung	*Der Druckauftrag* `pj` *wird über den in den Eigenschaften (*`PrinterURL`*) definierten Drucker ausgeführt. Dabei wird das im Druckauftrag angegebene Dokument (*`DocumentURL`*) sooft ausgedruckt, wie es im Druckauftrag über den Wert von* `NumberOfCopies` *angegeben ist. Ist im Druckauftrag der Wert von* `Sorted` *gleich* `true`*, dann werden die einzelnen Kopien des Dokuments sortiert zurückgegeben, ansonsten ist dies nicht zwingend erforderlich. Die Methode liefert im Rückgabewert* `PrintJobNo` *über eine Zeichenkette einen eindeutigen Identifier, der den gestarteten Druckauftrag repräsentiert. Falls der Drucker nicht antwortet, wird eine* `PrinterDoesNotRespond`*-Exception geworfen. Falls der Drucker einen sonstigen Fehler bei Ausführung des Druckauftrages meldet, wird eine* `PrinterDoesNotRespond`*-Exception geworfen.*

Bezeichner	`CancelPrintJob`
Parameter	`String Id`
Rückgabewerte	`void`
Beschreibung	*Der Druckauftrag, der durch die Zeichenkette* `Id` *repräsentiert wird, wird abgebrochen. Existiert kein Druckauftrag, der durch die übergebene Zeichenkette repräsentiert wird, so wird keine Aktion ausgeführt.*

Events

Bezeichner	`OnPrintJobStarted`
Parameter	`PrintJob pj`
Beschreibung	*Der Druckauftrag* `pj` *ist gestartet worden.*

Bezeichner	`OnPrintJobFinished`
Parameter	`PrintJob pj`
Beschreibung	*Der Druckauftrag* `pj` *ist ohne Auftreten eines Fehlers beendet worden.*

Bezeichner	`OnPrintJobCanceld`
Parameter	`PrintJob pj`
Beschreibung	*Der Druckauftrag* `pj` *ist abgebrochen worden.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

SQLDatabase

Der Dienst *SQLDatabase* kapselt die Funktionalität einer SQL-Datenbank. Über die Schnittstellen des Dienstes kann das Datenbanksystem angegeben werden, auf das zugegriffen werden soll. Außerdem können an die so spezifizierte Datenbank beliebige SQL-Anweisungen übertragen werden.

Datentypdefinitionen

Bezeichner	`DataItem`
Struktur	`ColumnIndex`
	`ColumnName`
	`ColumnValue`

Exceptions

Bezeichner	`InvalidIP`
Parameter	`String ServerIP`
Beschreibung	*Die in den Eigenschaften angegebene IP des Datenbankservers* `ServerIP` *entspricht nicht dem IPv4-Format.*
Bezeichner	`ServerNotFound`
Parameter	`String IP`
Beschreibung	*Der Datenbankserver mit der IP* `IP` *konnte nicht gefunden werden.*
Bezeichner	`UserUnknown`
Parameter	`String Username`
Beschreibung	*Dem Datenbanksystem ist der angegebene Benutzer mit dem Benutzernamen* `Username` *nicht bekannt.*

Bezeichner	`InvalidPassword`
Parameter	`String Username` `String Password`
Beschreibung	*Das angegebene Passwort* `Password` *ist nicht das Passwort des Benutzers mit dem Benutzernamen* `Username`*.*

Bezeichner	`DatabaseError`
Parameter	`String ErrorText`
Beschreibung	*Der Datenbankserver hat einen Fehler mit dem Fehlertext* `ErrorTextd` *gemeldet.*

Eigenschaften

Bezeichner	`ServerIP`
Typ	`String`
Zugriff	`Write`
Exceptions	`ServerNotFound`
Beschreibung	*Die IP des Datenbankservers im IPv4-Format.*

Bezeichner	`Username`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Der Benutzername des Datenbankbenutzers, der für die Datenbankinteraktion herangezogen wird.*

Bezeichner	`Password`
Typ	`String`
Zugriff	`Write`
Beschreibung	*Das Passwort des Datenbankbenutzers, der für die Datenbankinteraktion herangezogen wird.*

Methoden

Bezeichner	`Connect`
Rückgabewerte	`void`

Exceptions	`InvalidIP`
	`ServerNotFound`
	`UnknownUser`
	`InvalidPassword`
	`DatabaseError`

Beschreibung	*Stellt eine Verbindung zu dem in den Eigenschaften definierten Datenbankserver (*`ServerIP`*) her unter Benutzung der in den Eigenschaften definierten Benutzerdaten (*`Username, Password`*). Falls die IP des Datenbankservers kein gültiges Format aufweist, wird eine* `InvalidIP`*-Exception geworfen. Falls der Datenbankserver nicht erreicht werden kann, wird eine* `ServerNotFound`*-Exception geworfen. Falls der Benutzer dem Datenbankserver unbekannt ist, wird eine* `UnknownUser`*-Exception geworfen. Falls das übergebene Passwort nicht dem Passwort des Benutzers am Datenbankserver entspricht, wird eine* `InvalidPassword`*-Exception geworfen. Falls der Datenbankserver eine sonstige Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Bezeichner	`Disconnect`
Rückgabewerte	`void`
Exceptions	`DatabaseError`
Beschreibung	*Trennt die Verbindung zum Datenbankserver, falls eine existiert. Ansonsten findet keine Aktion statt. Falls der Datenbankserver eine Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Bezeichner	`ExecuteSQLStatement`
Parameter	`String SQLStatement`
Rückgabewerte	`void`
Exceptions	`DatabaseError`
Beschreibung	*Die SQL-Anweisung* `SQLStatement` *wird auf dem Datenbankserver ausgeführt. Falls der Datenbankserver eine Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Bezeichner	`FetchNextRow`
Rückgabewerte	`{DataItem} DataItems`
Exceptions	`NoDataFound` `DatabaseError`
Beschreibung	*Wurde per* `ExecuteSQLStatement` *eine DQL-Anweisung (Data Query Language) ausgeführt, so kann über diese Methode auf die Ergebnismenge des abgefragten Ausdruckes zugegriffen werden. Zurückgeliefert wird je ein Tupel dieser Ergebnismenge im Rückgabewert* `DataItems`*, der in der relationalen Anschauung eine Datenzeile repräsentiert. Falls der Datenbankserver eine Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Bezeichner	`Commit`
Rückgabewerte	`void`
Exceptions	`DatabaseError`
Beschreibung	*Die Datenbanksession wird mit einem Commit an den Datenbankserver abgeschlossen und eine neue Session wird eröffnet. Falls der Datenbankserver eine Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Bezeichner	`Rollback`
Rückgabewerte	`void`
Exceptions	`DatabaseError`
Beschreibung	*Die Datenbanksession wird mit einem Rollback an den Datenbankserver abgeschlossen und eine neue Session wird eröffnet. Falls der Datenbankserver eine Fehlermeldung liefert, wird eine* `DatabaseError`*-Exception geworfen.*

Kontrollprotokoll

```
( Connect
  [ ExecuteSQLStatement || FetchNextRow || Commit || Rollback]
  Disconnect )*
```

Sessionprotokoll

```
[Connect]
(( ExecuteSQLStatement FetchNextRow* ) || Commit || Rollback )*
[Disconnect]
```

CustomerCare

Der Dienst *CustomerCare* kapselt die Funktionalität einer Verwaltungseinheit für Kundenstammdaten. Zu diesen Daten gehören persönliche Informationen über einen Kunden, die durch den Datentyp `Customer` repräsentiert werden. Ergänzend dazu können pro Kunde Adressinformationen hinterlegt werden, die durch den Datentyp `Address` repräsentiert werden.

Imports

Dienst	`Authorisation`
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um festzustellen, ob der Benutzer des Systems berechtigt ist, bestimmte Aktionen durchzuführen.*
Dienst	`Printer`
Multiplizität	`(0, 1)`
Beschreibung	*Dieser Import wird verwendet, um Kundendaten auszudrucken.*
Dienst	`SMTP`
Multiplizität	`(0, 1)`
Beschreibung	*Dieser Import wird verwendet, um an Kunden E-Mails zu versenden.*

Datentypdefinitionen

Bezeichner	`Customer`

Struktur	`String CustomerNo`
	`String Lastname`
	`String Firstname`
	`String EMail`

Bezeichner	`Address`

Struktur	`String Street`
	`String City`
	`String ZIP`
	`String Phone`
	`String Fax`

Exceptions

Bezeichner	`NoSMTPImport`
Beschreibung	*Eine E-Mail sollte versendet werden, aber der Import des Dienstes* `SMTP` *ist nicht erfüllt.*
Bezeichner	`NoPrinterImport`
Beschreibung	*Ein Druckauftrag sollte gestartet werden, aber der Import des Dienstes* `Printer` *ist nicht erfüllt.*
Bezeichner	`MailError`
Parameter	`SMTP.EMail EMail` `String ErrorText`
Beschreibung	*Beim Versenden der E-Mail* `EMail` *über den importierten Dienst* `SMTP` *ist ein Fehler aufgetreten, der durch den Fehlertext* `ErrorText` *näher beschrieben wird.*

Bezeichner	`PrinterError`
Parameter	`Printer.PrintJob PrintJob` `String ErrorText`
Beschreibung	*Beim Ausführen des Druckauftrages* `PrintJob` *über den importierten Dienst* `Printer` *ist ein Fehler aufgetreten, der durch den Fehlertext* `ErrorText` *näher beschrieben wird.*
Bezeichner	`CustomerDoesNotExist`
Parameter	`String CustomerNo`
Beschreibung	*Es existiert in der Kundenverwaltung kein Kunde mit der Kundennummer* `CustomerNo`*.*
Bezeichner	`AddressDoesNotExist`
Parameter	`Address a` `String CustomerNo`
Beschreibung	*Es existiert in der Kundenverwaltung keine Adressinformation* `a`*, die dem Kunden mit der Kundennummer* `CustomerNo` *zugeordnet ist.*
Bezeichner	`AddressAlreadyExists`
Parameter	`Address a` `String CustomerNo`
Beschreibung	*Es existiert in der Kundenverwaltung bereits eine mit* `a` *identische Adressinformation, die dem Kunden mit der Kundennummer* `CustomerNo` *zugeordnet ist.*

Methoden

Bezeichner	`AddCustomer`
Parameter	`Customer c`
Rückgabewerte	`String`
Beschreibung	*Der Kundenverwaltung wird der neue Kunde* `c` *hinzugefügt. Falls für* `c` *bereits das Feld* `CustomerNo` *gesetzt ist, so wird dieser Wert ignoriert. Bei Anlage des neuen Kunden wird eine neue eindeutige Kundennummer als Zeichenkette generiert, die durch die Methode im Rückgabewert zurückgeliefert wird.*

Bezeichner	`RemoveCustomer`
Parameter	`String CustomerNo`
Rückgabewerte	`void`
Exceptions	`CustomerDoesNotExist`
Beschreibung	*Aus der Kundenverwaltung wird der Kunde mit der eindeutigen Kundennummer* `CustomerNo` *entfernt mitsamt allen Adressinformationen. Falls dieser nicht existiert, wird eine* `CustomerDoesNotExist`*-Exception geworfen.*

Bezeichner	`AddAddress`
Parameter	`Address a` `String CustomerNo`
Rückgabewerte	`void`
Exceptions	`CustomerDoesNotExist` `AddressAlreadyExists`
Beschreibung	*Der Kundenverwaltung wird die neue Adressinformation* `a` *hinzugefügt. Zugeordnet wird sie dem Kunden mit der Kundennummer* `CustomerNo`*. Falls dieser nicht existiert, wird eine* `CustomerDoesNotExist`*-Exception geworfen. Die Neueintragung wird ebenfalls nur vorgenommen, falls dem Kunden mit der Kundennummer* `CustomerNo` *nicht bereits eine identische Adressinformation zugeordnet ist. Falls dieser Fall eintritt, wird eine* `AddressAlreadyExists`*-Exception geworfen.*

Bezeichner	`RemoveAddress`
Parameter	`Address a` `String CustomerNo`
Rückgabewerte	`void`
Exceptions	`CustomerDoesNotExist` `AddressDoesNotExist`

Beschreibung	*Aus der Kundenverwaltung wird die Adressinformation* `a`*, die dem Kunden mit der Kundennummer* `CustomerNo` *zugeordnet ist, entfernt. Falls der Kunde mit der Kundennummer* `CustomerNo` *nicht existiert, wird eine* `CustomerDoesNotExist`*-Exception geworfen. Falls dem Kunden mit der Kundennummer* `CustomerNo` *keine Adressinformation* `a` *zugeordnet ist, wird eine* `AddressDoesNotExist`*-Exception geworfen.*

Bezeichner	`GetCustomers`
Rückgabewerte	`{Customer} Customers`
Beschreibung	*Es wird im Rückgabewert* `Customers` *die Menge aller Kunden geliefert, die in der Kundenverwaltung eingetragen sind.*

Bezeichner	`GetAddresses`
Parameter	`String CustomerNo`
Rückgabewerte	`{Address} Addresses`
Exceptions	`CustomerDoesNotExist`
Beschreibung	*Es wird im Rückgabewert* `Addresses` *die Menge aller Addressinformationen geliefert, die in der Kundenverwaltung eingetragen sind und dem Kunden mit der Kundennummer* `CustomerNo` *zugeordnet sind. Falls der Kunden mit der Kundennummer* `CustomerNo` *nicht existiert, wird eine* `CustomerDoesNotExist`*-Exception geworfen.*

Bezeichner	`SendEMail`
Parameter	`String CustomerNo` `SMTP.EMail EMail`
Rückgabewerte	`void`

Exceptions	`NoSMTPImport`
	`CustomerDoesNotExist`
	`MissingEMailAddress`
	`MailError`

Beschreibung	*An den Kunden mit der Kundennummer* `CustomerNo` *wird die in* `EMail` *definierte E-Mail verschickt. Über die Kundennummer lässt sich eindeutig der Kunde mitsamt der hinterlegten E-Mail-Adresse bestimmen. Das Feld* `To` *der übergebenen E-Mail wird ignoriert und mit der E-Mail-Adresse des Kunden (Feld* `EMail`*) überschrieben. Falls der Import des SMTP-Dienstes nicht erfüllt ist, wird eine* `NoSMTPImport`*-Exception geworfen. Falls der Kunden mit der Kundennummer* `CustomerNo` *nicht existiert, wird eine* `CustomerDoesNotExist`*-Exception geworfen. Falls an diesem Kunden keine E-Mail-Adresse hinterlegt ist, wird eine* `MissingEMailAddress`*-Exception geworfen. Falls während des Versendes ein sonstiger Fehler auftritt, wird eine* `MailError`*-Exception geworfen.*

Bezeichner	`PrintCustomerData`
Parameter	`String CustomerNo`
Rückgabewerte	`void`
Exceptions	`NoPrinterImport` `PrinterError`
Beschreibung	*Die Kundendaten des Kunden mit der Kundennummer* `CustomerNo` *werden über den importierten Printer-Dienst ausgedruckt. Falls der Import des Printer-Dienstes nicht erfüllt ist, wird eine* `NoPrinterImport`*-Exception geworfen. Falls während des Druckens ein sonstiger Fehler auftritt, wird eine* `PrinterError`*-Exception geworfen.*

Events

Bezeichner	`OnChange`
Beschreibung	*Daten der Kundenverwaltung haben sich geändert. Ein potentieller Observer sollte benötigte Daten erneut abfragen.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

CustomerCareConsumer

Der Dienst *CustomerCareConsumer* importiert den Dienst *CustomerCare*, um dessen Leistungen nutzen zu können.

Imports

Dienst	`CustomerCare`
Rolle	
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um Zugriff auf die durch den importierten Dienst gekapselte Kundenverwaltung zu erhalten.*

Eventhandler

Bezeichner	`HandleCustomerCareChange`
Event	`CustomerCare.OnChange`
Beschreibung	*Die Daten der Kundenverwaltung haben sich geändert und werden soweit benötigt erneut abgefragt.*

Collector

Der Dienst *Collector* kapselt die abstrakte Funktionalität eines Sammlers von Verbrauchsdaten, die relevant sind für ein Abrechnungssystem, repräsentiert durch den Dienst `Billing`.

Datentypdefinitionen

Bezeichner	`CollectorDataItem`
Struktur	`DateTime TimeStamp` `String Data`

Exceptions

Bezeichner	`CollectorError`
Parameter	`String ErrorText`
Beschreibung	*Allgemeiner Fehler des Collectors, der durch den Fehlertext* `ErrorText` *näher beschrieben wird.*

Methoden

Bezeichner	`StartCollector`
Rückgabewerte	`void`
Exceptions	`CollectorError`
Beschreibung	*Es wird der Sammelvorgang für Verbrauchsdaten gestartet. Falls beim Starten ein Fehler auftritt, wird eine* `CollectorError`*-Exception geworfen.*
Bezeichner	`StopCollector`
Parameter	
Rückgabewerte	`void`
Exceptions	
Beschreibung	*Es wird det Sammelvorgang für Verbrauchsdaten beendet. Falls beim Beenden ein Fehler auftritt, wird eine* `CollectorError`*-Exception geworfen.*
Bezeichner	`GetCollectorData`
Parameter	`DateTime PeriodStart` `DateTime PeriodEnd`
Rückgabewerte	`{CollectorDataItem} CollectorDataItems`
Beschreibung	*Es werden im Rückgabewert* `CollectorDataItems` *alle gesammelten Verbrauchsdaten des Zeitraums, der über* `PeriodStart` *als Startzeitpunkt und* `PeriodEnd` *als Endzeitpunkt festgelegt ist, geliefert.*

Eventhandler

Bezeichner	`HandleBillingRunStarted`
Event	
Beschreibung	
Bezeichner	`HandleBillingRunFinished`
Event	
Beschreibung	
Bezeichner	`HandleBillingRunCanceled`
Event	
Beschreibung	

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

DocumentArchive

Der Dienst *DocumentArchive* kapselt die Funktionalität eines Dokumentenarchivs. Dokumente können an das Archiv übergeben werden, sie erhalten dabei eine eindeutige Dokumentennummer, über die sie zu einem späteren Zeitpunkt wieder zugreifbar sind. Dokumente können auch aus dem Archiv dauerhaft entfernt werden. Ein Eintrag im Dokumentenarchiv besteht aus einer eindeutigen Dokumentennummer, einem Titel, einem beschreibenden Text und dem Dokument selbst. Ein Eintrag wird repräsentiert durch den Datentyp `Document`.

Datentypdefinitionen

Bezeichner	`Document`
Struktur	`String DocumentNo`
	`String Title`
	`String Description`
	`Binary Document`

Exceptions

Bezeichner	`ArchiveError`
Parameter	`String ErrorText`
Beschreibung	*Ein allgemeiner Fehler des Archivs, der über den Fehlertext* `ErrorText` *näher beschrieben wird.*
Bezeichner	`DocumentDoesNotExist`
Parameter	`String DocumentNo`
Beschreibung	*Es existiert kein Dokument im Archiv mit der Dokumentennummer* `DocumentNo`*.*

Methoden

Bezeichner	`AddDocument`
Parameter	`Document d`
Rückgabewerte	`String DocumentNo`
Exceptions	`ArchiveError`
Beschreibung	*Es wird dem Dokumentenarchiv der neue Eintrag* `d` *hinzugefügt, wobei ein potentiell gesetzer Wert für das Feld* `DocumentNo` *ignoriert wird. Rückgabewert der Methode ist die eindeutige Dokumentennummer* `DocumentNo` *des neuen Eintrags. Falls beim Hinzufügen ein Fehler auftritt, wird eine* `ArchiveError`*-Exception geworfen.*
Bezeichner	`RemoveDocument`
Parameter	`String DocumentNo`
Rückgabewerte	`void`
Exceptions	`DocumentDoesNotExist` `ArchiveError`
Beschreibung	*Es wird aus dem Dokumentenarchiv der Eintrag mit der Dokumentennummer* `DocumentNo` *entfernt. Falls das angegebene Dokument nicht im Archiv existiert, wird eine* `DocumentDoesNotExist`*-Exception geworfen. Falls beim Entfernen ein Fehler auftritt, wird eine* `ArchiveError`*-Exception geworfen.*

Bezeichner	`GetDocuments`
Parameter	`Boolean AttachBinaries`
Rückgabewerte	`{Document} Documents`
Exceptions	`ArchiveError`
Beschreibung	*Es werden im Rückgabewert* `Documents` *alle Dokumenteneinträge des Archivs geliefert. Ist der Wert des Parameters* `AttachBinaries` *gleich* `true`*, dann sind die binären Dokumentdaten Teil der gelieferten Einträge, ansonsten bleiben diese Felder leer. Falls beim Abrufen der Einträge ein Fehler auftritt, wird eine* `ArchiveError`*-Exception geworfen.*

Bezeichner	`GetDocument`
Parameter	`String DocumentNo` `Boolean AttachBinary`
Rückgabewerte	`Document Document`
Exceptions	`DocumentDoesNotExist` `ArchiveError`
Beschreibung	*Es wird im Rückgabewert* `Document` *genau der Dokumenteneintrag des Archivs geliefert, der die Dokumentennummer* `DocumentNo` *besitzt. Ist der Wert des Parameters* `AttachBinary` *gleich* `true`*, dann werden die binären Dokumentdaten Teil des gelieferten Eintrags, ansonsten bleibt dieses Feld leer. Falls kein Dokumenteneintrag mit der Dokumentennummer* `DocumentNo` *im Archiv existiert, wird eine* `DocumentDoesNotExist`*-Exception geworfen. Falls beim Abrufen der Einträge ein Fehler auftritt, wird eine* `ArchiveError`*-Exception geworfen.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Billing

Der Dienst *Billing* kapselt die Funktionalität einer verbrauchsorientierten Abrechnungseinheit. Die Basisdaten der Abrechnung erhält das Billing durch einen importierten `CustomerCare`-Dienst für die Kundendaten, sowie ein oder mehrere `Collector`-Dienste für die Verbrauchsdaten. Daraus werden zeitraumbezogene Rechnungen generiert, die gedruckt, archiviert und auch per E-Mail verschickt werden können.

Imports

Dienst	`CustomerCare`
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um die Kundendaten für die Abrechnung zu bestimmen.*
Dienst	`Collector`
Multiplizität	`(1, n)`
Beschreibung	*Dieser Import wird verwendet, um die Verbrauchsdaten für die Abrechnung zu bestimmen.*
Dienst	`Printer`
Multiplizität	`(0, n)`
Beschreibung	*Dieser Import wird verwendet, um die erzeugten Rechnungsdokumente auszudrucken.*
Dienst	`DocumentArchive`
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um die erzeugten Rechnungsdokumente zu archivieren.*

Datentypdefinitionen

Bezeichner	`Invoice`

Struktur	`String InvoiceNo`
	`CustomerCare.Customer Customer`
	`String DocumentNo`
	`String BillingRunNo`

Exceptions

Bezeichner	`BillingError`
Parameter	`String ErrorText`
Beschreibung	*Ein allgemeiner Fehler des Abrechnungssystems, der über den Fehlertext* `ErrorText` *näher beschrieben wird.*

Bezeichner	`BillingRunDoesNotExist`
Parameter	`String BillingRunNo`
Beschreibung	*Es existiert kein aktiver Abrechnungslauf mit der Identifikationsnummer* `BillingRunNo`*.*

Eigenschaften

Bezeichner	`PrintAfterRun`
Typ	`Boolean`
Zugriff	`Write`
Beschreibung	*Ist dieser Wert auf* `true` *gesetzt, dann werden nach einem fehlerfrei durchgeführten Abrechnungslauf automatisch alle innerhalb des Rechnungslaufs generierten Rechnungsdokumente gedruckt.*

Methoden

Bezeichner	`StartBillingRun`
Parameter	`DateTime PeriodStart` `DateTime PeriodEnd`
Rückgabewerte	`String BillingRunNo`
Exceptions	`BillingError`

Beschreibung	*Es wird ein neuer Abrechnungslauf über den angegebenen Zeitraum gestartet, wobei* `PeriodStart` *den Beginn und* `PeriodEnd` *das Ende des Zeitraums bestimmt. Der Rückgabewert* `BillingRunNo` *enthält eine eindeutige alphanumerische Identifikationsnummer, die den gestarteten Abrechnungslauf eindeutig identifiziert. Jedem im Abrechnungslauf generierten Abrechnungsdokument wird diese Identifikationsnummer zugeordnet, um eine Rechung eindeutig einem Abrechnungslauf zuordnen zu können. Falls während der Abrechnung ein Fehler auftritt, wird eine* `BillingError`*-Exception geworfen.*

Bezeichner	`CancelBillingRun`
Parameter	`String BillingRunNo`
Rückgabewerte	`void`
Exceptions	`BillingRunDoesNotExist` `BillingRunError`
Beschreibung	*Es wird der aktive Abrechnungslauf abgebrochen, der über die Identifikationsnummer* `BillingRunNo` *bestimmt ist. Falls es keinen aktiven Abrechnungslauf mit dieser Nummer gibt, wird eine* `BillingRunDoesNotExist`*-Exception geworfen. Falls während der Abrechnung ein Fehler auftritt, wird eine* `BillingError`*-Exception geworfen.*

Bezeichner	`GetInvoices`
Parameter	`String BillingRunNo`
Rückgabewerte	`{Invoice} Invoices`
Exceptions	`BillingRunDoesNotExist`
Beschreibung	*Es wird im Rückgabewert* `Invoices` *die Menge der Rechnungen geliefert, die während des Abrechnungslaufes mit der Identifikationsnummer* `String DocumentNo` *generiert worden sind. Falls es keinen aktiven Abrechnungslauf mit dieser Nummer gibt, wird eine* `BillingRunDoesNotExist`*-Exception geworfen.*

Events

Bezeichner	`OnBillingRunStarted`
Parameter	`String BillingRunNo`
Beschreibung	*Der Billinglauf mit der Identifikationsnummer* `String DocumentNo` *ist gestartet worden.*
Bezeichner	`OnBillingRunFinished`
Parameter	`String BillingRunNo`
Beschreibung	*Der Billinglauf mit der Identifikationsnummer* `String DocumentNo` *ist fehlerfrei beendet worden.*
Bezeichner	`OnBillingRunCanceled`
Parameter	`String BillingRunNo`
Beschreibung	*Der Billinglauf mit der Identifikationsnummer* `String DocumentNo` *ist explizit durch den Anwender oder durch einen aufgetretenen Fehler abgebrochen worden.*

Kontrollprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

Sessionprotokoll

Die Aufrufreihenfolge der Methoden / Eventhandler ist beliebig.

BillingConsumer

Der Dienst *BillingConsumer* importiert den Dienst *Billing*, um dessen Leistungen nutzen zu können.

Imports

Dienst	`Billing`
Rolle	
Multiplizität	`(1, 1)`
Beschreibung	*Dieser Import wird verwendet, um Zugriff auf die durch den importierten Dienst gekapselte Abrechnungseinheit zu erhalten.*

Literaturverzeichnis

[1] Berthold Daum: *Java-Entwicklung mit Eclipse 3, Anwendungen, Plugins und Rich Clients*; Dpunkt Verlag; Heidelberg; 2004; 2., überarbeitete und erweiterte Auflage

[2] Jürgen Ebert, Roger Süttenbach, Ingar Uhe: *JKogge: a Component-Based Approach for Tools in the Internet*; In STJA 99, 5. Fachkonferenz Smalltalk und Java in Industrie und Ausbildung. 1999.

[3] Frank Griffel: *Componentware, Konzepte und Techniken eines Softwareparadigmas*; Dpunkt Verlag; Heidelberg; 1998

[4] Volker Gruhn, Manfred Schneider: *EJB 2.0 Anwendungen, Entwurf leistungsfähiger Java-Komponenten*; Addison-Wesley; München; 2002

[5] Volker Gruhn, Andreas Thiel: *Komponentenmodelle, DCOM, JavaBeans, Enterprise JavaBeans, CORBA*; Addison-Wesley; München; 2000

[6] Kevin Hirschmann, Volker Riediger, Andreas Winter: *GXL Schema API*; Projektbericht 1/04, Universität Koblenz-Landau, Institut für Softwaretechnik; Koblenz; 2004.

[7] Johann Hofmann, Fritz Jobst, Roland Schabenberger: *Programmieren mit COM und CORBA, Einführung in die Architekturen für verteilte Anwendungen*; Hanser Verlag; München Wien; 2001

[8] Cay S. Horstmann, Gary Cornell: *core JAVA Band 2, Expertenwissen*; Markt+Technik; München; 2000

[9] Alexander Kaczmarek: *GXL-Validator, Validierung von GXL-Dokumenten auf Instanz-, Schema- und Metaschemaebene*; Universität Koblenz-Landau, Institut für Softwaretechnik; Koblenz; September 2003

[10] Eric Newcomer: *Understanding Web Services: XML, WSDL, SOAP, and UDDI*; Addision-Wesley; Boston; 2002

[11] Object Management Group: *CORBA BASICS*; http://www.omg.org/gettingstarted/corbafaq.htm; 05.04.2005

[12] Clemens Szyperski: *Component Software, Beyond Object-Oriented Programming*; Addison-Wesley; London; 1999

[13] Klaus Turowski: *Vereinheitlichte Spezifikation von Fachkomponenten Memorandum des Arbeitskreises 5.10.3, Komponentenorientierte betriebliche´Anwendungssysteme*; Gesellschaft für Informatik; 2002

[14] Andreas Winter, B. Kullbach, V. Riediger: *An Overview of the GXL Graph Exchange Language*; Springer Verlag: S. Diehl (ed.) Software Visualization · International Seminar Dagstuhl Castle, Germany, May 20-25, 2001 Revised Lectures

Index