AUSARBEITUNG

GENERIERUNG VON ADAPTOREN

GENERIERUNG VON ADAPTOREN FÜR SOFTWARE KOMPONENTEN

UNTER BERÜCKSICHTIGUNG VON SIGNATURINFORMATIONEN

Individuelles Projekt

im SS 2004, Department für Informatik an der

Carl-Von-Ossietzky-Universität Oldenburg

Betreuer

Jun.-Prof. Dr. Ralf H. Reussner

Dipl.-Wirtsch.-Inform. Steffen Becker

Klaus Krogmann, 04.09.2004

Verfasser der Arbeit

Klaus Krogmann

Reinekeweg 2

26676 Harkebrügge

E-Mail: kelsaka@gmx.de

Homepage: http://www.kelsaka.de

Inhaltsverzeichnis

1. Abstract

Die vorliegende Ausarbeitung zum Individuellen Projekt bietet einen Einblick in die Erzeugung von Adaptoren auf Signaturniveau. Dabei wird eine differenzierte Abgrenzung zu anderen Gebieten der Adaptererzeugung vorgenommen, die sich unter anderem im einleitenden Abschnitt der Arbeit wiederfindet.

Im Mittelpunkt der Betrachtung, die aus der Sicht der Komponentenbasierten Softwareentwicklung vorgenommen wird, steht vor allem der Erzeugungsvorgang von Adaptoren für Komponenten.

Die Erzeugung von Adaptoren hängt unter anderem maßgeblich davon ab, wie viele Informationen über die zu adaptierenden Komponenten vorliegen. Betrachtet werden verschiedene Niveaus des Informationsreichtums, die aus unterschiedlicher wissenschaftlicher Sicht zu verschiedenen Ergebnissen führen können.

Für den Kontext des erstellten Adapter-Generators wird vor allem das Konzept der Konverter näher beleuchtet. Durch diesen vielseitigen Mechanismus läßt sich die Mächtigkeit der erzeugbaren Adapter drastisch erhöhen. Aufgezeigt wird neben der konkreten Implementierung auch ein Vergleich mit anderen Adaptionsansätzen.

2. Einleitung

Die vorliegende Arbeit beschreibt die Entwicklung eines Software-Werkzeugs, das auf Basis vorliegender Schnittstellenbeschreibungen zweier Komponenten, in Interaktion mit dem Benutzer des Tools, bei Notwendigkeit, einen Adapter für die gegebenen Komponenten erzeugt. Der so genannte Adapter-Generator bedient entsprechend Anforderungen aus der komponentenbasierten Softwareentwicklung[1].

Dabei werden Adapter erzeugt, die Konflikte auf Signaturebene und überdies auf semantischer Ebene mit Hilfe des Benutzers lösen können. Problemfelder die hiermit behandelt werden können, liegen z. B. im Mapping von Signaturen aufeinander. Durch die Einführung des Konzepts zusätzlicher Konverter, die frei ergänzt werden können, lassen sich zudem Typkonvertierungen unterschiedlicher Komplexität auf dem Rückgabetyp oder den Parametern einer Methode vornehmen.

Weitere Erläuterungen zu den Anforderungen an die entwickelte Software, das genaue Vorgehen sowie tiefer gehende Beschreibungen der oben angerissenen Konzepte und Ideen finden Sie in den Kapiteln 2.2 und 6.

Die folgenden Teile der Einleitung und Einführung in das Thema zeigen bereits die exakte Einordnung des Adapter-Generators in das Problemgefüge im Adaptionsbereich auf und gehen differenziert darauf ein, welche Problembereiche die Signaturadaption betreffen, aber auch, welche Überschneidungsbereiche sich zu angrenzenden Feldern ergeben.

2.1 Motivation

2.1.1 Status Quo: Komponentenbasierte Softwareentwicklung

Zeitgleich mit der „Einführung“ des Terminus der *Softwarekrise* wurde der Begriff des *Softwareengineerings* in den 70er Jahren geprägt[2]. Man hatte erkannt, dass die zunehmende Komplexität von Software eine ernst zu nehmende Herausforderung für die Branche darstellte und nur mit der Strukturierung des „Fertigungsprozesses“ auf längere Sicht nicht zu einem GAU bei der Entwicklung führen würde.

Eine der Antworten des Softwareengineerings auf die neuen Herausforderungen war die Komponente als Bezeichnung für eine logische Softwareeinheit, die sich im Idealfall

1 Auch CBSE: Component Based Software Engineering

2 NATO-Konferenz 1968 in Garmisch-Patenkirchen

annähernd wie bspws. in der Automobilbranche als ein Einzelteil benutzen läßt. Der Komponentenbegriff wurde bereits 1968 von McIlroy auf der NATO-Konferenz in Garmisch-Patenkirchen eingeführt.

Dabei ist die Definition einer Komponente keineswegs eindeutig. Auf der einen Seite wird eine Komponente als alles wiederverwendbare definiert, auf der anderen Seite als ein natürliches Konzept, dass schlicht undefinierbar ist[3]. Engere Definitionen, wie die von Clemens Szyperski, betrachten Komponenten eher aus der Sicht eines Software-Engineers als unabhängig deploy-fähige Einheit ohne persistenten Zustand und zusammengesetzt von dritten. Damit ist die Palette gängiger Definitionen noch nicht erschöpft, zeigt aber auch die in Teilen zweckorientierte Definition des Komponentenbegriff in Richtung Softwareentwicklung.

Nach dem Komponentenkonzept sollten die genauen Eigenschaften einer Komponente nach außen hin angegeben werden, wie im Vergleichsbeispiel des Automobilbaus etwa die Dämpfungseigenschaft eines Stoßdämpfer mit seinen Einbaumaßen, Gewicht und sonstigen Attributen. Größere Komponenten wie die Karosserie ergäben sich dann wiederum aus diversen Einzelteilen. Durch die exakte Spezifikation der Einzelteile und dem Wissen über die mechanischen Gesetze und Eigenschaften, sollten sich die Eigenschaft der Karosserie exakt vorhersagen lassen.

Tatsächlich existieren sowohl einfache wie auch zusammengesetzte Komponenten ebenfalls in der Softwareentwicklung. Obgleich die komponentenbasierte Softwareentwicklung für eine Strömung aus dem Informatikbereich bereits relativ alt ist, stellt das derzeitige Entwicklungsniveau immer noch keine Ebenbürtigkeit mit dem Automobilbereich dar.

Zum einen werden gerne Entwicklungsprozess und Fertigungsvorgänge vertauscht. Als markantester Prozess in der Automobilindustrie stellt sich die Fertigung heraus. Hier werden alle Komponenten zusammengefügt und es entsteht mehr als die Summe der Einzelteile. Zumeist gibt es in der Produktion dank umfassendem Qualitätsmanagement keine großen Probleme, dass einzelne Bauteile nicht den geforderten Eigenschaften genügen. Somit entstehen hier lediglich sehr wenige Probleme. Das Komponentenkonzept scheint perfekt zu funktionieren.

Auf der anderen Seite wird die Softwareentwicklung betrachtet. Das „Zusammenfügen“ von Softwarekomponenten wird als Fertigungsprozess interpretiert. Die bis heute zuweilen nischenhafte Verbreitung von Komponentenansätzen zeigt deutlich auf, dass man offenbar noch meilenweit von der Perfektion der Automobilbranche entfernt ist.

Betrachtet man die Vergleichbarkeit der beiden genannten Bereiche aus Softwareentwicklung und Automobilbau genauer, so dürfte auffallen, dass es sich bei der komponentenbasierten Softwareentwicklung jedoch nicht um einen Fertigungsprozess handelt, sondern um den

3 Siehe auch Folien zur Vorlesung CBSE, Ralf Reussner, SS2004, CvO Universität, Oldenburg

Design- und Entwicklungsprozess aus dem Automobilbereich. Der Produktion hingegen entspräche die Fertigung der Glasmaster für CDs und DVDs sowie dem abschließenden Pressvorgang, ergänzt um den Druck eines Benutzerhandbuchs. Aus diesem Bereich sind jedoch ebenso wenig Probleme zu vernehmen, wie von den Förderbändern der Automobilbranche.

Führt man den Vergleich weiter, gilt es die Entwicklung eines neuen Fahrzeugtypen mit der Entwicklung einer neuen Software zu vergleichen. In diesem Bereich ergeben sich zuweilen erstaunliche Parallelen. So sind umfangreiche Tests bei den Automobilbauern ebenso üblich wie in der Softwareentwicklung. Und die häufiger werdenden Rückrufaktionen[4] sind vergleichbar mit nachträglichen Updates für Software.

Warum also hat die komponentenbasierte Softwareentwicklung noch keine ähnlich hohe Akzeptanz wie die Ansätze im Automobilbereich gefunden? Zum einen sind die Aufgabenstellungen häufig unterschiedlich. Autos werden seit Jahrzehnten kontinuierlich weiterentwickelt. Die beschäftigten Ingenieure kennen exakt ihre Anforderungen, und haben genaue Vorstellungen von den Lösungen. Bei der Softwareentwicklung sind häufig jedoch keine Vorgehensmuster vorhanden, die einen generischen Lösungsweg vermitteln könnten. Aus dem Softwareengineering sind einzig Methodiken entstanden, die ein Vorgehen empfehlen. Eine konkrete Empfehlung aus der Erfahrung heraus zur Lösung eines solchen Problems gibt es jedoch (meistens) nicht.

Weitaus wichtiger sind jedoch immer noch bestehende Probleme, bei der Spezifikation der Eigenschaften von Komponenten. Angaben von vertraglichen Vor- und Nachbedingungen, QoS[5]-Eigenschaften und Service Effect-Automaten sind hier ein möglicher Weg aus den Problemen heraus. Was den Ingenieuren also schon lange wie selbstverständlich vorliegt, findet konzeptionell ebenfalls Einführung in der Komponentenentwicklung.

2.1.2 Teil-Problemlösung

Daneben existieren aber auch vergleichsweise trivial erscheinende Probleme, die der Entwicklung von Software mit Hilfe von Komponenten einen Riegel vorschieben. Liegen einem Entwickler zwei Software-Komponenten vor, die von ihrem Verhalten her genau den Erfordernissen entsprechen, so bedeutet dies nicht implizit, dass beide Komponenten auch wirklich zusammenpassen.

Die „Anforderungen“ der einen Komponente entsprechen nicht dem, was die andere anbietet bzw. anbieten kann. Die Verwendung der dargebotenen Softwarekomponenten ist in diesen

4 Die zugegebener Maßen zu Teilen auch von der Software im Auto her rühren. Siehe hierzu auch [TechRew7-04] S. 22

5 QoS: Quality of Service

Fällen nicht ohne weiteres möglich. (Präzisere Erläuterungen finden Sie in den folgenden Kapiteln.)

Genau mit diesen Problemen auf Signaturebene befasst sich das vorliegende Individuelle Projekt. Durch die Verwendung eines Adapters zwischen der anbietenden und der anfordernden Komponente wird eine Teilmenge der existierenden Komponenten „kompatibel“ gemacht.

Der Generator von Adaptoren unterstützt die Erstellung solcher wrapperähnlichen Softwareteile, die in Benutzerinteraktion erzeugt werden, durch die Bereitstellung von Automatismen.

Da auch Adaptoren wiederum als Komponenten aufgefasst werden können, sind auch hier die Eigenschaften der Adapter, wie QoS-Merkmale, von Interesse, denn das Verhalten zweier Komponenten, die über einen Adapter miteinander kommunizieren, sollte aus den in Kapitel 2.1.1 genannten Gründen, ebenfalls systematisch ergründbar bleiben, um nicht die Vorhersagbarkeit der Eigenschaften zusammengesetzter Komponenten zu verlieren.

Lassen sich verschiedene Komponenten komfortabel benutzen und über einen Adapter ohne großen Aufwand miteinander verknüpfen, dürfte dies in einem gewissen Maß zu einer höheren Akzeptanz der komponentenbasierten Softwareentwicklung durch die Entwickler beitragen.

2.1.3 Motivation für den Adaptereinsatz

Adapter finden vor allem in Umgebungen Einsatz, bei denen es um die Integration von Altsystemen und Alt-Komponenten in neue Umgebungen geht. Da bestehende Systeme im Allgemeinen eine große Investition darstellen, die nicht vollständig aufgegeben werden soll, entscheidet man sich häufig dafür, nur Teile durch neue Software zu ersetzen. Bei großen Systemen ist der Umstellungsaufwand zudem so groß, dass die Umstellung nicht zu einem Zeitpunkt durchgeführt werden könnte[6]. Daher müssen hier zwingend unterschiedliche Komponenten integriert werden.

Daneben lässt sich bei zwei klassischen Vorgehensweisen zum Einsatz von Komponenten der Sinn von Adaptern aufzeigen:

- **Bottom-Up:** Die Konstruktion einer Gesamtsoftware erfolgt bei diesem Prinzip aus vielen kleinen „Bausteinen“, die zusammengefügt werden.
  Beispielsweise können viele Firmen auf einen großen Fundus bestehender

6 Enthält ein Altsystem mehrere tausend Klassen, sind diese nicht innerhalb kurzer Zeit modernisierbar, ohne, dass zeitweise notwendige Aktualisierungen an mehreren Zweigen der Software parallel ausgeführt werden, was wiederum Kosten verursacht.

Softwarekomponenten zurückgreifen, die jedoch nicht explizit für den kooperierenden Einsatz im angestrebten Kontext konzipiert wurden. Gleichermaßen sind COTS[7]-Komponenten verfügbar, die, zumal sie häufig von unterschiedlichen Herstellern stammen, keine einheitlichen Schnittstellen implementieren. In diesen Fällen werden Adaptoren dazu eingesetzt, eine Integration der Komponenten herbeizuführen. Adaption kann daher auch als Customization-Prozess vorliegender Komponenten an die eigenen Bedürfnisse verstanden werden.

- **Top-Down:** Die zweite sehr weit verbreitete Methode, die sich im Komponentenbereich breit durchgesetzt hat, ist der Top-Down-Entwurf von Software. Sie stellt eine gegenläufige Entwicklung zum Bottom-Up-Verfahren dar.
  Wie beispielsweise bei Plug-Ins für Web-Browser wird zunächst eine exakte Schnittstelle spezifiziert. Die weitere Entwicklung der Software erfolgt auf die Einhaltung dieser Schnittstelle hin ausgerichtet. Alle Komponenten werden von Beginn an so entwickelt, dass sie nachher zusammenpassen. Im Allgemeinen kommt es daher hier auch nicht zum Einsatz von Adaptern.
  Im Laufe der Zeit wird jedoch auch eine Schnittstellenspezifikation einer Modernisierung unterliegen[8], etwa weil sich neue Anforderungen ergeben haben, denen das Interface Rechnung tragen muss. Damit auch für den Zeitraum nach der Umstellung alte (noch nicht angepasste) Plug-Ins mit dem Browser kommunizieren können, ist der Einsatz von Adaptoren sinnvoll, da sich diese häufig schneller erzeugen lassen, als die Anpassung der Plug-In-Software möglich ist.

2.2 Ziel des Individuellen Projekts[9]

Ähnlich wie mit der PacoSuite[10] bietet der Adapter-Generator die grafisch unterstützte Generierung von Komponenten. Im konkreten Fall stellt der Adapter die Komponente dar, die über typische Mechanismen wie DragNDrop erzeugt wird. Der Benutzer wird durch den Erstellungsprozess des Adapters begleitet, an dessen Ende Quellcode in C#, sowie die entsprechende Komponente im Palladio ComponentModel (siehe Kapitel 3.3) steht.

7 COTS: Commercial Off The Shelf; hier: kommerzielle verfügbare Komponenten; siehe hierzu auch [MCK04].

8 Als Beispiel sei die Schnittstellenspezifikation des Mozilla-Browsers genannt, die zwar über einen gewissen Zeitraum stabil bleibt, nach und nach aber auch einem Versionssprung unterliegt (wie jüngst auf Version 1.7). Siehe auch http://www.mozilla.org.

9 Zu den Zielen des Individuellen Projekts siehe auch [Proposal] zum Thema.

10 Unterstützt die komplette Generierung von komplexen Komponenten über eine einfach zu bedienende GUI. Als Resultat lässt sich der Quelltext der Anwendung ausgeben und im Weiteren verarbeiten. Homepage des Projekts: http://ssel.vub.ac.be/pacosuite/

Je nach Klasse des Adapters stehen zudem Abschätzungen über die Performance (und andere QoS-Eigenschaften) zur Verfügung, damit zumindest Teile der in Kapitel 2.1.1 angesprochenen Probleme im Einsatz von Komponenten vermieden werden.

2.3 Gliederung der Ausarbeitung

Nach der überblicksartigen Einleitung zum Thema werden im Weiteren die relevanten technischen Aspekte zur konkreten Umsetzung des Adapter-Generators vorgestellt. Zunächst wird das von der DFG Palladio-Gruppe entwickelte ComponentModel (Kapitel 3.3, Seite 15) beleuchtet, dass für den Adapter-Generator als Input und auch Output dient, jedoch auch bei der internen Datenrepräsentation verwendet wird, um die Komponentenbestandteile zu modellieren. Die angesprochenen Themen sollen eine Einordnung dieser Arbeit in den Gesamtkontext ermöglichen und einen theoretischen Hintergrund zum Aufgabenbereich des Adapter-Generators bieten.

Daneben werden die Entwicklungskonzepte (Kapitel 5, Seite 25), also das Vorgehen bei der Entwicklung, und die Details und Hintergründe zur Implementierung (Kapitel 6, Seite 36) vorgestellt. Im abschließenden Fazit (Kapitel 8, Seite 48) und Ausblick (Kapitel 9, Seite 51) werden besondere Probleme bei der Entwicklung und Anregungen für eine Weiterentwicklung aufgezeigt.

3. Einführung in die verwendeten Techniken

3.1 Komponentenbeschreibung

Komponenten können als „einfache“ Komponenten auftreten und bieten eine Requires- und eine Provides-Schnittstelle. In UML2[11]-Notation werden die Schnittstellentypen wie folgt dargestellt[12]:

11 UML: Unified Modelling Language der OMG (Object Management Group)

12 Auch „Lollipop-Notation“ auf Grund der Ähnlichkeit der Darstellungsform.

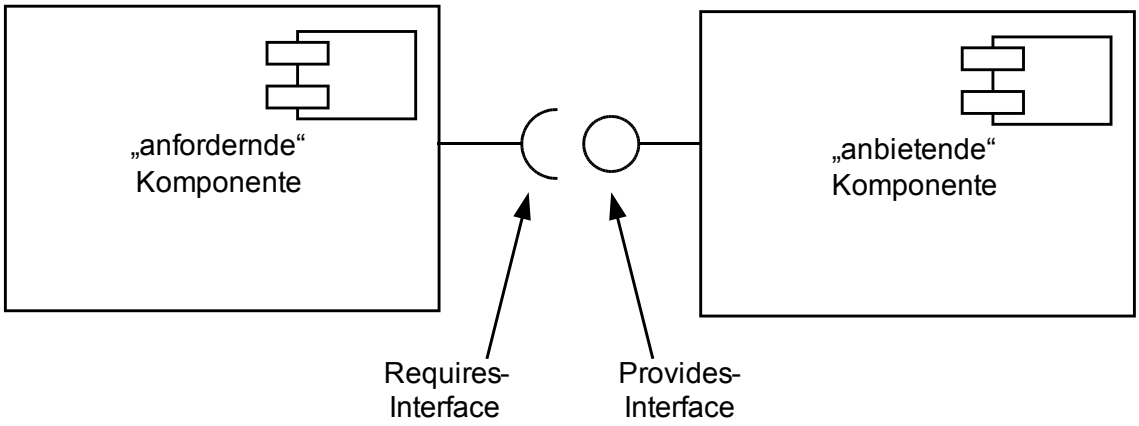


Abbildung 1 UML2-Notation einer Komponente mit Requires- und Provides-Interface

In Abbildung 1[13] werden zwei einzelne Komponenten dargestellt. Die linke Komponente besitzt nur eine Requires-Schnittstelle, die rechte Komponente besitzt nur eine Provides-Schnittstelle. Diese Formen von Komponenten können allerdings auch zugleich eine Provides- und eine Requires-Schnittstelle enthalten. Dies bedeutet, dass die offerierten Dienste nur bei Erfüllung der eigenen Requires-Schnittstelle einer Komponente verwendet werden können. Die Abhängigkeit einer Komponente verläuft jeweils vom eigenen Requires-Interface über die verwendete Provides-Schnittstelle der „nachfolgenden“ Komponente weiter über mögliche nachfolgende Abhängigkeiten, wie Abbildung 2 für den einfachen Fall linearer Abhängigkeit zeigt. Daneben sind netzartige Abhängigkeiten der Komponenten untereinander denkbar[14], so dass eine Komponente letztlich nicht von jeweils einer Komponente, sondern auch von beliebig vielen weiteren abhängig sein kann.

Während die Abhängigkeiten sich in Abbildung 2 von links („betrachtete Komponente“) nach rechts ergeben, hängen die Eigenschaften (vgl. hierzu auch Kapitel 2.1.1, Seite 7) der grau eingefärbten Komponente von allen weiter rechts stehenden ab.

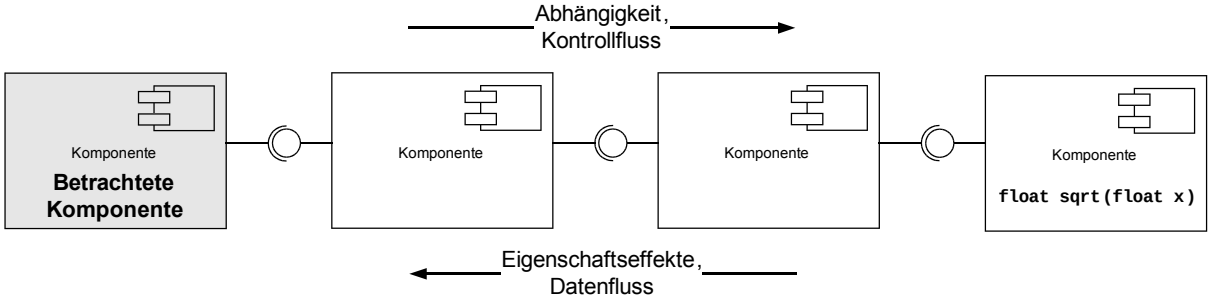


Abbildung 2 Abhängigkeiten und Eigenschaftseffekte bei Komponenten

13 Eine alternative Darstellung finden Sie in Abbildung 7, Kapitel 5.3.1, Seite 29

14 Untersuchungen und Software hierzu bietet Marko Hoyer, IP zu „Entwicklung eines Frameworks zur Simulation von Komponentennetzwerken“

Im Beispiel lässt sich dies leicht erkennen. Möchte die grau eingefärbte Komponente bspw. die Wurzel einer Zahl berechnen (`sqrt(x)`), bezieht sie die eigentliche Funktionalität aus der ganz rechts stehenden Komponente. Die beiden mittleren Komponenten würden den Aufruf der grau eingefärbten Komponente bis zur rechten weiterleiten. Jeder dieser Aufrufe wird ein gewisses Maß an Zeit benötigen. Daher wird die betrachtete Komponente das Ergebnis erst nach der sich ergebenen Verzögerung erhalten, die sich als Summe der Verzögerungen der Einzelkomponenten ergibt.

Würde die graue Komponente eine von `sqrt(x)` abhängige Funktionalität (diese Funktion muss auf das Ergebnis der Berechnung zurückgreifen) in ihrem Provides-Interface offerieren, so hinge z. B. die Antwortzeit nur zu einem gewissen Teil (nämlich dem Effekt, den die markierte Komponente selber ausübt) statisch von der grauen Komponente ab, der Rest würde sich durch die Aufrufreihenfolge ergeben.

Neben der Antwortzeit lassen sich die Abhängigkeiten auch für alle anderen QoS-Merkmale feststellen[15], da sie sich aus den gleichen Gründen aus den tieferliegenden Aufrufen ergeben. Die Eigenschaften einer Komponente werden also maßgeblich durch ihre Abhängigkeiten bestimmt.

Insbesondere für weiche Realzeitsysteme ist es erstrebenswert, eine kritische Obergrenze bestimmter QoS-Parameter angeben zu können. Wichtig ist es also die Eigenschaften aller eingesetzten Komponenten möglichst exakt zu kennen (siehe hierzu auch Kapitel 2.1.1, Seite 7), um Aussagen für eine einzelne Komponente oder gar das Gesamtsystem machen zu können.

3.2 Erweiterung der Komponentenbetrachtung

Wie bereits im Kapitel 3.1 beschrieben wurde, können sich Komponentenabhängigkeiten nicht nur linear, sondern auch netzartig ergeben. Dazu ist es erforderlich, dass eine Komponente nicht nur ein Requires-Interface besitzen kann, sondern mehrere. Damit die Zuordnung von zugeordneten Provides- und Requires-Interfaces eindeutig erfolgen kann, spricht man in diesem Zusammenhang von einer Rolle[16]. Sie stellt, bezogen auf eine konkrete Umwelt, eine eindeutige Identifikationsmöglichkeit für ein Interface dar. Da zwei Interfaces einer Komponente (namens-) gleiche Services anbieten können, müssen diese unterscheidbar bleiben, da Services trotz gleicher Identifikation innerhalb des Interfaces, nicht zwangläufig die gleiche Semantik haben müssen.

Als Beispiel (aus: [CM04]: „Role“) sei ein Service „aService“ gegeben, der von zwei

15 Die Betrachtung der QoS-Merkmale der erzeugten Adaptoren erfolgt in Kapitel 7, „QoS-Merkmale der Adaptoren“, Seite 46.

16 Vgl. hierzu auch Kapitel 3.3, „IRole“ / „Role“

Interfaces einer Komponente gleichzeitig angeboten wird. Würde der Service ohne das Rollenkonzept aufgerufen, wäre nicht entscheidbar, welcher Service gemeint ist. Durch die Einführung der Beispielrollen „A“ und „B“ wird es jedoch möglich die „aService“ zu unterscheiden, indem man beispielsweise „A.aService“ oder „B.aService“ zur Adressierung verwendet. Die Role ergibt sich in diesem Zusammenhang aus der Bedeutung eines Interfaces im Zusammenspiel mit ihrer Umwelt.

Entsprechend ist auch die Erweiterung auf der Angebotsseite sinnvoll, da eine Komponente zugleich mehrere Requires-Schnittstellen verschiedener Komponenten mit unterschiedlichen Provides-Angeboten versorgen kann. Damit lässt sich die oben angesprochene Netzstruktur vollständig beschreiben.

3.3 Palladio ComponentModel

Von zentraler Bedeutung bei der Entwicklung des Adapter-Generators war das Palladio[17] ComponentModel (Technical Report in [CM04]), ein Komponentenmodell, dass in C# mit Microsofts .NET entwickelt wurde. Nähere Informationen zur Verwendung des ComponentModel finden sich in Kapitel 6, Entwurf und Implementierung.

Das Komponentenmodell selber greift wiederum auf Service-Effect-Automaten zurück, um Abhängigkeiten von Methoden untereinander modellieren zu können. Damit bietet das ComponentModel unter anderem den folgenden Funktionsumfang:

- **Service Effekt Spezifikationen** (vgl. [CM04]) enthalten eine Beschreibung der Abhängigkeiten der Services untereinander. Ruft ein Service A intern die externen Services B und C auf, so sind B und C Teil der Service Effekt Spezifikation von A.
- **Interfaces**[18] enthalten eine Liste von Signaturen, die entweder ein Requires- oder ein Provides-Interface bilden. Neben diesen Minimuminformationen, die der Container enthält, können weitere Informationen in Form von *Auxilary Information* ergänzt werden. Diese können dann beispielsweise das Protokoll (zu den Protokollspezifikationen siehe auch Kapitel 4.2, Seite 18) in Form von z. B. endlichen Automaten (FSM[19]) oder QoS-Informationen der Schnittstelle enthalten. Der Bestand an zusätzlichen Informationen richtet sich nach dem spezifischen Wissen über ein Interface. Entsprechend des Maßes an Zusatzinformationen lassen sich später verschiedene Model-Checks durchführen (siehe hierzu auch Kapitel 5.3, Seite 28).

17 DFG Palladio-Gruppe, Leitung: Jun.-Prof. Dr. Ralf H. Reussner

18 Die Begriffe Interface und Schnittstelle werden in dieser Arbeit synonym verwendet.

19 FSM: Finite State Machine – Endlicher Automat

- Eine **Role**[20] definiert sich im ComponentModel analog zu den Forderungen aus Kapitel 3.2 und beinhaltet entsprechend eine eindeutige Identifizierung wie auch ein Requires- oder Provides-Interface.

- Die **Signature** (auch Signatur) ist Teil des Interfaces einer Komponente und gliedert sich in einen Rückgabetyp, eine Identifikation, eine geordnete Liste von Parametern und eine ungeordnete Liste von Exceptions. Damit entspricht sie exakt dem Konstrukt, das für den Adapter-Generator benötigt wird.

- **Basic Components** stellen einfache Komponenten dar, die über eine Requires- und eine Provides-Schnittstelle verfügen. Sie werden als Black-Box angesehen, das heißt also als Komponenten, deren innere Struktur und Aufbau nicht bekannt sind. Daher können Sie unter Umständen auch aus weiteren Komponenten bestehen.

- **Composite Components** setzen sich in ihrem Inneren aus anderen Komponenten zusammen, die Basic Components oder wiederum Composite Components sein können. Sie stellen einen Container dar, der üblicherweise vollständige domänenspezifische Funktionen bündelt und nach außen zur Verfügung stellt.

- **Connections** verbinden innerhalb einer Composite Component Schnittstellen untereinander. Sie unterteilen sich in *Binding* und *Mapping*, wobei das *Binding* eine Requires- und eine Provides-Schnittstelle untereinander verbindet und alle Aufrufe von der Requires-Komponente auf die Provides-Komponenten weiterleitet. Das *Mapping* hingegen tritt wiederum in zwei Varianten auf. Zum einen kann es zwei Provides-Schnittstellen aufeinander abbilden oder zum anderen zwei Requires-Schnittstellen verbinden. Es verbindet die Schnittstellen einer Basic Component und einer Composite Component miteinander, so dass z. B. das Provides-Interface einer Basic Component aus der Composite Component heraus „sichtbar“ werden kann.

Von Bedeutung für den Adapter-Generator sind vor allem Basic Components, Signaturen und Interfaces.

Das ComponentModel verfügt zudem über eine Factory, um alle verfügbaren Komponenten erzeugen und zusammenfügen zu können.

20 Die Begriffe „Role“ und „Rolle“ werden in der Ausarbeitung synonym verwendet.

4. Theoretische Grundlagen

4.1 Einordnung in den Adaptionskontext

Abbildung 3 (siehe unten) zeigt eine Klassifikation von Adaptionsmechanismen auf. Die Thematik dieses Individuellen Projekts ist die Generierung von Adaptoren, die in der Abbildung grau unterlegt wurde.

Bei der Adapter-Generierung handelt es sich um eine Black-Box-Technik. Das bedeutet, dass neben der Schnittstellenspezifikation (ergänzt evtl. um QoS-Merkmale) der Provides- und Requires-Komponente keine weiteren Informationen vorhanden sind. Insbesondere der Quellcode der Komponenten, aus dem sich das spezifische Verhalten für Eingaben oder die Art der Implementierung einer Funktionalität ermitteln ließe, sind nicht verfügbar. Daher erfolgt die weitere Einordnung unterhalb der Black-Box-Verfahren auch bei den Interface-Basierten Mechanismen.

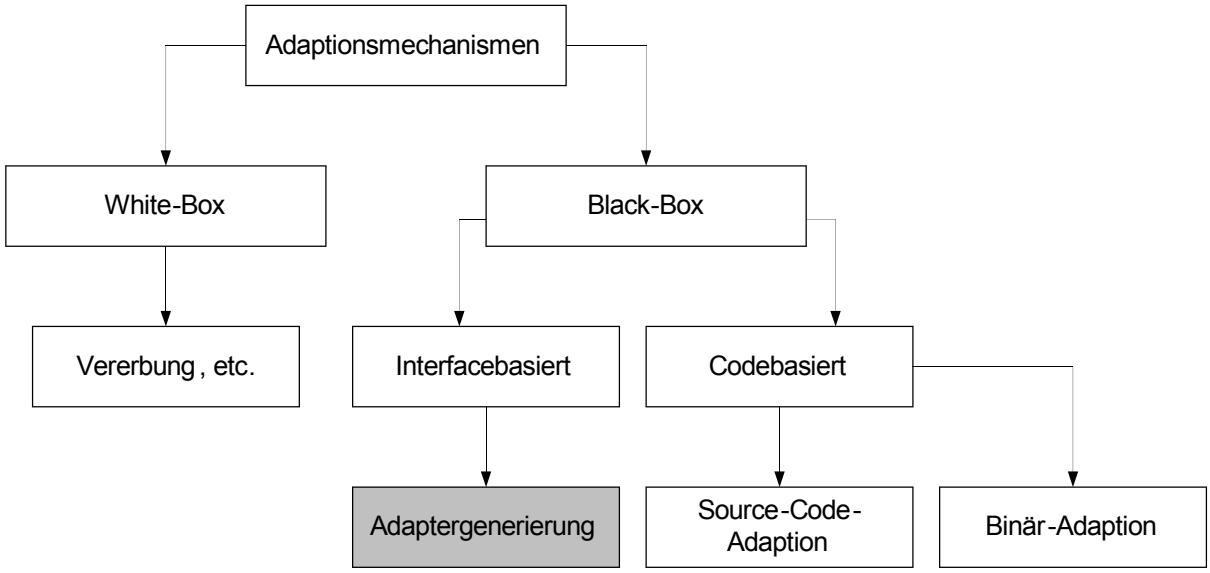


Abbildung 3 Einordnung der Adapter-Generierung in den Adaptionskontext (Veränderte Version aus: Vorlesung Komponentenbasierte Softwareentwicklung, Jun.-Prof. Dr. Ralf H. Reussner, SS04, CvO-Universität, Oldenburg)

Wie in Kapitel 6 beschrieben wird, lassen sich die Adaptionsansätze auch untereinander ergänzen. So ist denkbar, dass der Programmcode (also der Zweig „codebasiert“) sowohl als Quellcode als auch als Binär-Code analysiert wird. Daraus lässt sich eine Schnittstellenbeschreibung, beispielsweise mit Hilfe des ComponentModels, erzeugen. Die Interfacebeschreibung dient dann wieder als Input für die Adaptergenerierung. In diesem

Zusammenhang dient die Beschreibung der Schnittstelle als Zwischenschicht und abstrahiert von der Eingabe.

Diese Variante ist für den Adapter-Generator ebenfalls vorgesehen, wie in Kapitel 6.3 beschrieben wird. Die Code-Analyse wird dabei von „MBEL“ vorgenommen.

Neben den behandelten Black-Box-Mechanismen existieren erwartungsgemäß White-Box-Mechanismen, wozu beispielsweise die Vererbung zählt. Damit hier eine sinnvolle Adaption vorgenommen werden kann, muss der Quellcode bekannt sein. Da über die Vererbung auch Mechanismen der vererbenden Klassen verändert werden können[21], muss eine erbende Klasse sicherstellen können, dass durch ihre Modifikationen die Oberklasse immer noch in der erwarteten Weise funktioniert. Diese Investigation ist im Allgemeinen nur über einen Einblick in den Code möglich.

4.2 Interface-Hierarchie

Ein gebräuchliches Modell zur Beschreibung der Abhängigkeiten unter den Ebenen der Schnittstellen ist in Abbildung 4 aufgezeigt.

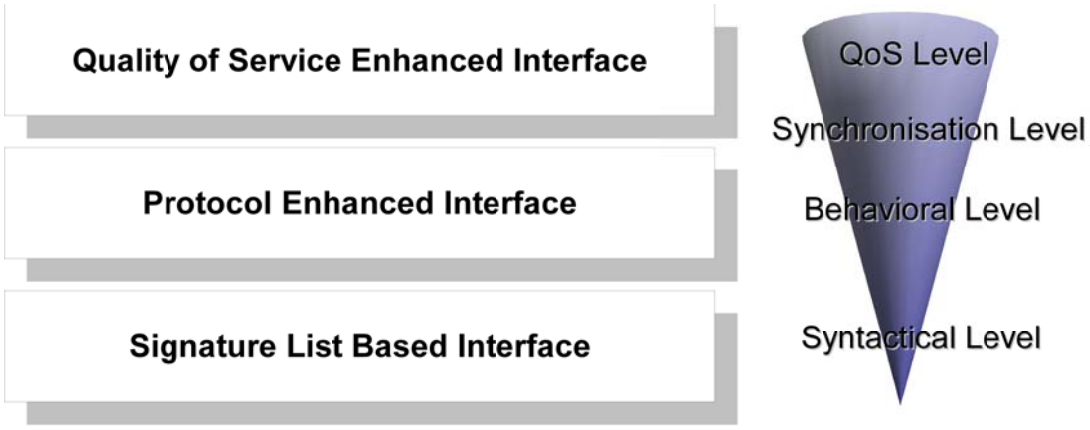


Abbildung 4 Interfacehierarchieebenen (nach [Reussner04], Fig. 1. Hierarchies of Interface Models)

Im linken Teil der Abbildung werden Interfaces in drei Ebenen unterteilt. Die niederste Ebene stellt die signaturlistenbasierten Interfaces dar, auf die um eine Protokollebene erweiterte Schnittstellen aufsetzen. Die höchste Anforderungsebene in diesem Modell bilden die QoS-Merkmale eines Interfaces.

Beschreibung der Schnittstellenebenen:

- **Signaturlistenbasierte Interfaces**: Diese Form der Interfaces steht im Zentrum der

21 etwa durch Überschreiben von Methoden

Betrachtung beim Adapter-Generator. Ein solches Interface besteht aus einer Menge von Methodensignaturen[22], die zusammen das Requires- respektive Provides-Interface bilden. Zu den Merkmalen dieser Schnittstellen zählen der Rückgabetyp der einzelnen Methoden, der Methodenname, die Typen der Parameter und ihre Reihenfolge, sowie die ungeordnete Menge von Exceptions, die von einer Methode geworfen werden können.

- **Protokollerweiterte Interfaces:** Üblicherweise wird diese Verhaltens-Ebene über Automaten beschrieben[23]. Da die Methoden eines Interfaces unter Umständen nicht in freier Reihenfolge ausgeführt werden können, gibt die Protokollebene eine Spezifikation dafür an, welche Ausführungsfolgen valide sind. Beispielsweise erfordert die Verbindung zu einer Datenbank zunächst ein `connect()`. Nur wenn eine Verbindung existiert, können Operationen ausgeführt werden. Schließlich wird eine Datenbankanfrage nur korrekt beendet, wenn abschließend ein `close()` ausgeführt wird.
  Es ist anzumerken, dass nicht alle Protokolle gleichfalls mit einem endlichen Automaten beschrieben werden können. Wird die Protokollebene beispielsweise mit Push-Down-Automaten beschrieben, so ist jedoch das Inklusions- oder Äquivalenzproblem für die spezifizierten Protokolle nicht entscheidbar. Requires- und Provides-Schnittstelle sind also nicht in jedem Fall auf Kompatiblität zueinander prüfbar (vgl. [Reussner04]).

- **Quality of Service-erweiterte Interfaces:** Wie bereits in Kapitel 3.1 angesprochen, gibt es auch auf QoS-Niveau Interdependenzen zwischen den Schnittstellen von Komponenten, also auf der Ebene der nicht-funktionalen Anforderungen.
  Zur Notation von QoS-Eigenschaften bietet sich beispielsweise die QML an, die auf dem Prinzip von Verträgen ([RFB]) zwischen Requires- und Provides-Seite basiert. Jedes Interface legt hierbei genau fest, unter welchen Bedingungen Leistungen mit welchen QoS-Merkmalen erbracht werden können, so dass sich daraus ein QML-Profil zur Beschreibung von Schnittstellen ergibt. Die QML umfasst dabei Metriken wie Mittelwert-Angaben, Varianz und die prozentuale Verteilung der definierten Attribute z. B. einer Methode.

Für das auf der linken Seite in Abbildung 4 dargestellte Interface-Hierarchie-Modell wird angenommen (vgl. [Reussner04]), das Inkompatiblitäten auf niederen Ebenen die erwartete Funktionalität auf allen höheren Ebenen unterbindet.

Der Ansatz auf der rechten Seite in Abbildung 4 von [Beugnard] ist prinzipiell mit dem linken Teil vergleichbar. Auf der Protokollebene werden lediglich Erweiterungen um Synchronisationsaspekte vorgenommen, womit sich z. B. Mutex-Mechanismen explizit in einer Schnittstellenebene wiederfinden.

22 In [Becker04] wird im Gegensatz zur hier verwendeten Sprachregelung nur eine einzelne Methodensignatur dieser Ebene zugeordnet. Entsprechend der von Adapter-Generator erfüllten Funktionalität, wurde hier eine Erweiterung vorgenommen.

23 Etwa beim Palladio ComponentModel in Form von Finit-State-Machines (FSM); siehe auch [Becker04]

4.3 Interface-Missmatches

Zu den im vorangegangenen Kapitel 4.2 vorgestellten Level im Schnittstellenmodell ergeben sich typische Missmatches[24], also Fehler beim Zusammenfügen von Komponenten. Ein Adapter des entsprechenden Levels würde die Konflikte beheben, sofern dies möglich ist.

An dieser Stelle seien einige Bespiele zu den entsprechenden Problemen betrachtet[25]:

- **Signaturebene**
  Requires-Interface:

```
ReservationID ReserveRoom(DBID database, RoomID room, DateTime
     startTime, DateTime endTime) throws RoomBlockedException;
void CancelReservation(DBID database, ReservationID id);
```

  Provides-Interface:

```
ReservationID Reserve(ResourceID res, DateTime startTime, TimeSpan
     duration) throws ResourceBlockedException;
void Cancel(ResourceID res, DateTime startTime) throws
     NoReservationFound;
```

  Bei diesem Beispiel wird deutlich, dass die Methodennamen der beiden Interfaces nicht übereinstimmen. Zudem passen die Anzahl der Parameter und die Parametertypen nicht zueinander. Die Provides-Methode `Cancel` wirft darüber hinaus eine Exception, die von der Requires-Seite nicht gefangen wird. Setzt man die semantische Gleichheit der Beispielmethoden voraus, ist die Erzeugung eines Adapters denkbar.
  Eine ausführliche Beschreibung der für den Adapter-Generator relevanten Probleme auf Signaturebene finden Sie in Kapitel 5.3, „Signatur-Missmatches“.

  Konkrete Lösungsansätze für diese Missmatches werden in Kapitel 5.4, „Adapter-Lösungsansätze“ aufgezeigt.

- **Protokollebene**
  Zur Betrachtung von Interoperabilitätsproblemen sei zunächst das Beispiel in Abbildung 5 gegeben.

  In diesem Beispiel erwartet die Requires-Seite einen einfachen Automaten, der permanent im „Ready“-Zustand ist, nach dem Ausführen oder Abbruch einer Reservierung ebenso wie nach der Feststellung, ob eine Reservierung vorliegt. Daher lässt sich jede Methode zu jeder Zeit ausführen.

24 Vgl. hierzu [Reussner04], insbesondere die Beispiele sind hieran angelehnt.

25 Die Beispiele sind gegenüber den Originalbeispielen auf die jeweils benötigte Komplexität reduziert worden.

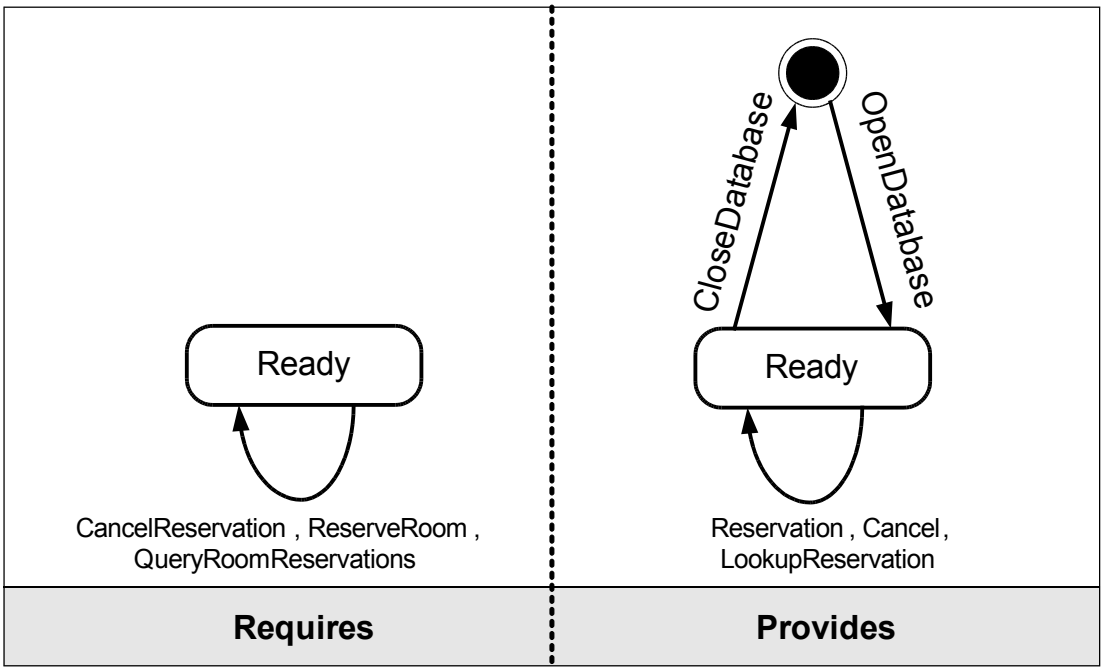


Abbildung 5 Mismatch auf Protokollebene, aus [Becker04] „Fig 4. Mismatching component protocols“

Auf der Provides-Seite hingegen wird erwartet, dass zunächst `OpenDatabase` ausgeführt wird, bevor im weiteren Buchungsmethoden ausgeführt werden können. Die Schnittstellen unterscheiden sich also bei Betrachtung der Gesamtmenge von Signaturen im Protokoll, da auf der Provides-Seite zunächst noch eine Datenbankverbindung aufgebaut werden muss.
Im diesen Fällen spricht man von einem „Protokoll Interoperabilitätsfehler“[26]: „Die Anforderung wenigstens einer Aufruffolge im Requires-Protokoll von A wird vom Provides-Protokoll von B nicht angeboten.“
Eine weitere Unterteilung erfolgt in:

- *Fehlende Methodenaufrufe*[27]*:* Die Aufruffolge von A enthält eine Methode, die von B nicht angeboten wird. Dieser Fall gründet bereits auf Problemen auf Signaturlistenniveau, lässt sich also bereits bei vorangegangenen Checks finden.
  Seien A und B Mengen, die die Methoden der Schnittstellen von A bzw. B als Elemente enthalten $a_x \in A \ \wedge \ b_x \in B$, so ist $A \not\subseteq B \ \wedge \ A \cap B \neq \emptyset$.

- *Überflüssige Methodenaufrufe:* Zwar enthält die Aufruffolge von A die richtige Folge von Aufrufen für B, B erwartet aber zusätzliche Aufrufe, um die Aufruffolge anbieten zu können. Hier müsste A also weitere Methoden vom B aufrufen.
  Seien a, b, c Methodenaufruf-Teilsequenzen[28], dann erwartet B Aufrufe der Form $(a, b^n, c)^m$, wird jedoch von A mit b^n aufgerufen, wobei $m, n \in \mathbb{N}$.

26 Vgl. [Reussner04], „protocol interoperability error“

27 Auch „Partial Matching Problem“, vgl. [MCK04]

28 Anlehnung an die Schreibweise für Reguläre Ausdrücke, ~Chomsky-Sprachen.

- *Umgekehrte Methodenaufrufe:* An mindestens zwei Stellen im Protokoll ruft A Methoden in einer umgekehrten Reihenfolge zu B auf. Wie [Reussner04] anmerkt, sind solche Probleme üblicherweise schwer zu adaptieren, da die unterschiedlichen Reihenfolgen im Protokoll auf eine grundsätzlich andere Semantik hindeuten.
  Seien a, b Methodenaufruf-Teilsequenzen, c, d einzelne Methodenaufrufe, dann erwartet A $(a, c, d, b)^n$ als gültige Sequenz, während B $(a, d, c, b)^n$ als gültiges Protokoll akzeptiert, mit $n \in \mathbb{N}$.

Wie ebenfalls zu sehen ist, passen die Methodennamen im Beispiel nicht zusammen. Ein Adapter auf Signaturebene ist folglich eine Voraussetzung für eine weitere Adaption auf der Protokollebene.

- **Nicht-Funktionale Ebene (QoS)**
  Als Beispiel für Interoperabilitätsprobleme auf der QoS-Ebene sei der folgende QML-Vertrag für eine einzelne Komponente gegeben; zunächst der Vertragstyp:

```
type Performance = contract {
     delay : decreasing numeric msec;
     throughput : increasing numeric mb / sec;
}
type Reliabilyty = contract {
     numberOfFailures : decreasing no / year;
     availablity : increasing numeric;
}
```

Ein Beispielvertrag dazu könnte wie folgt aussehen:

```
ComponentPerformance = Performace contract {
     delay < 400 msec;
     throughput > 1.3 mb / sec;
}
ComponentReliability = Reliability contract {
     numberOfFailures < 5 no / year;
     availabilty {
           percentile 100 > 0.5
           mean > 0.9
           variance < 0.2
     }
}
```

Für die Adaption von Komponenten mit definierten QoS-Eigenschaften ist es im Allgemeinen nicht möglich einen Adapter zu erzeugen, der beispielsweise die Performance erhöht. Wie in Kapitel 2.1 beschrieben wurde, hängt der Adapter selber von der Provides-Schnittstelle ab. Ein Adapter, der eine angebotene Methode nutzt, die den Performance-

Anforderungen nicht genügt, wird durch die modifizierte Weiterleitung nur weitere Performanceeinbußen erzeugen[29]. Es ist hier mit Standardmitteln nicht möglich Beschleunigungen zu erzielen.

Sind tatsächlich Verbesserungen von QoS-Eigenschaften wie Performance und Reliability notwendig, reichen die Fähigkeiten eines „einfachen“ Adapters (mit möglicherweise statischen Eigenschaften) nicht mehr aus. In diesen Fälle wären beispielsweise Caching-Strategien notwendig. Diese wiederum hängen stark von der Anwendungsdomäne ab und erfordern umfangreiche Kenntnisse über die Provides-Komponente. Da Caches üblicherweise in Randfällen andere Daten als die Originalkomponente liefern dürften, ist fraglich, ob die Requires-Spezifikation noch exakt eingehalten werden könnte. Entsprechend müssen auch auf dieser Seite genaue Verträge vorliegen, um einen sinnvollen Adapter zu ermöglichen.

Aus den genannten Gründen sind QoS-Eigenschaften für die Adaption aus einer anderen Perspektive interessant: Werden Adapter eingesetzt, die Interoperabilität auf Signatur- oder Protokollebene ermöglichen, so sind die QoS-Merkmale des Adapters selber von Interesse.

Da der Adapter eine Indirektionsebene darstellt, ist es von Bedeutung, ob der Adapter nach Rückgriff auf seine eigenen untergeordneten Komponenten (Provides-Seite der zuvor inkompatiblen Komponenten) durch seinen Performance-Einfluß noch die QoS-Anforderungen der Requires-Seite erfüllen kann. Wenn beispielsweise bei einem Methoden-Name-Mapping die ursprünglichen Performance-Anforderungen der Requires-Seite *unter* dem Angebot der Provides-Seite liegen, ist eine sinnvolle Adaption überhaupt möglich. (Wie bereits erwähnt, machen andere Fälle im Kontext dieses Individuellen Projekts keinen Sinn.)

4.4 Einsatz von Adaptern

Adapter werden in der Praxis genutzt, um die dargestellten Ebenen von Interoperabilitätsproblemen zu lösen. Dies setzt zunächst voraus, dass Interoperabilitäten erkannt werden. Damit alle Interoperabilitäts-Ebenen abgedeckt werden können, setzt dies z. B. die Service-Effect-Automaten / endliche Automaten zur Modellierung von Protokollen oder QML-Notationen zu den zu analysierenden Schnittstellen voraus[30].

Signaturlisten lassen sich grundsätzlich beispielsweise über Reflection-Mechanismen, wie sie im .NET-Framework vorhanden sind, erzeugen. Vor allem die Ermittlung der Provides-

29 Eine eingehendere Beleuchtung der QoS-Eigenschaften von Adaptern (die mit dem Adapter-Generator erzeugt werden können) erfolgt in Kapitel 7, „QoS-Merkmale der Adaptoren“, Seite 46.

30 Vgl. Kapitel 4.2

Schnittstellen wird bereits direkt über die API[31] angeboten[32]. Allerdings ist zu beachten, dass die Semantik der ermittelten Signaturen nicht automatisiert feststellbar ist. So kann die Methode `float square(float x)` in einem Kontext die Ermittlung der Wurzel, im anderen aber das Quadrat der Zahl x bedeuten. In diesem Zusammenhang sind daher externe Informationen, die nicht aus dem Quellcode extrahiert werden können, notwendig.

Allgemein lässt sich daraus für die Generierung höherwertiger[33] Adaptoren die Anforderung ableiten, dass die Schnittstellenbeschreibungen der Komponenten die benötigten Informationen für die Adapation auf einem bestimmten Niveau bereits enthalten muss, damit sich die Erzeugung von Adaptoren automatisiert durchführen lässt.

4.5 Die „Oldenburg-Matrix“

Anders als das in Kapitel 4.2 gezeigte Modell der Schnittstellenhierarchie, klassifiziert die Oldenburg-Matrix (Abbildung 6) die Schnittstellenebenen (Signaturen, Protokolle, QoS) in zwei Dimensionen. Dahinter steht die Erkenntnis, dass sich QoS-Merkmale und andere nicht-funktionale Eigenschaften orthogonal zu den bis dato aufgezeigten Schnittstellenebenen verhalten und die angenommene Abhängigkeit von QoS-Merkmalen nicht von der Erfüllung der niederen Hierarchieebenen alleine abhängt. Vielmehr können alle in der Horizontalen abgetragenen Ebenen eigene nicht-funktionale Anforderungen mit sich bringen.

	Methoden	**Schnittstellen**	**Domäne**
Funktional	Signaturen	Protokolle	Domänenobjekte
Nicht-Funktional	Methodenspezifische Qualität	Schnittstellen-spezifische Qualität	Domänen-anforderungen

Abbildung 6 Oldenburg-Matrix; Klassifikation von Schnittstellenmodellen

(aus [Becker04])

Mit Blick auf das hier vorgestellte Komponentenmodell[34] lässt sich die konsequente Erweiterung um den Domänenbereich feststellen, der die nächsthöhere Stufe nach den Schnittstellenerfordernissen bedeutet. Im ComponentModel entspräche dies einer Komponente, die Funktionalität für eine bestimmte Domäne erfüllt. Entsprechend sind auch

31 Dies bezieht sich auf die API des .NET-Frameworks für C#.

32 Zur Verfahrensweise im Adapter-Generator siehe auch Kapitel 6, Entwurf und Implementierung.

33 Im Sinne höherer Ebenen in der Interface-Hierarchie, Abbildung 4

34 Palladio ComponentModel, Kapitel 3.3, Seite 15

hier differenziert nicht-funktionale Anforderungen aufgeführt, die nur im Bezug auf eine Domäne sinnvoll sind.

Der Adapter-Generator lässt sich in diesem Modell exakt in der ersten Spalte „Methoden“ einordnen. In diesem Modell wird insbesondere auch klar, warum in diesem Individuellen Projekt neben den Signaturen auch die QoS-Merkmale des Adapters auf dieser Ebene sinnvoll betrachtet werden können. Damit decken die Untersuchungen des Individuellen Projekts die Methodenebene in dieser Klassifizierung vollständig ab. Gleichermaßen wird erkenntlich, warum trotz kompletten Matchings und keinerlei Inkompatibilitäten auf der Signaturebene, auf der Protokoll- oder aber Domänenebene dennoch Probleme auftreten können.

5. Entwicklungskonzepte

5.1 Das Adapter-Entwurfsmuster

Das Adapter-Pattern, wie es zum Beispiel in [Gamma] beschrieben wird, ist ein klassen- oder objektbasiertes Strukturmuster. Da es zur Anwendung Vererbung erfordert, ist es jedoch nicht direkt auf Komponenten, wie sie im Falle des Adapter-Generators vorliegen, anwendbar[35]. Dennoch habe viele Aussagen aus der objektorientierten (OO) Welt des Adapter-Musters, wie es [Gamma] beschreibt, auch für Komponenten ihre Gültigkeit und werden daher im Folgenden auf ihre Verwendung im Komponentenkontext hin untersucht. Zugleich soll aufgezeigt werden, wie sich diese Aspekte bei Komponenten darstellen.

Das Verwendungsszenario ist folglich die Vorgabe zweier Klassen respektive Komponenten, deren Schnittstellen nicht zueinander passen und deren unvorhersehbare Wiederverwendung (Deployment) in einer anderen Umgebung.

Das Adapter-Pattern ist auch als Wrapper-Muster bekannt und dient im OO-Original dazu, die Schnittstellen einer Klasse dahingehend zu verändern, dass sie in einem anderen Anwendungsbereich verwendet werden kann. Damit ist die Funktionalität zu dem erforderlichen Konzept für Komponenten identisch. Weiterhin ist auch für den OO-Fall vorgesehen, dass ein Adapter fehlende Funktionalität der zu adaptierenden Klasse (Provides-Seite) ergänzt.

In einer Implementierung der OO-Variante des Adapters in C++[36] würde sowohl von der zu

35 Komponenten verwenden Delegation und keine Vererbung..

36 Auf Grund der Möglichkeit zur Mehrfachvererbung.

adaptierenden Klasse, wie auch von der nutzenden Klasse geerbt. Hier könnte ein Adapter Methoden der vererbenden Klasse überschreiben. In der Komponentenvariante wird der Adapter direkt als eigenständige Komponente zwischen Requires- und Provides-Seite geschaltet und dann direkt von der Requires-Seite angesprochen. Demzufolge kann der Adapter das Originalverhalten der Provides-Komponente beliebig manipulieren (siehe auch Kapitel 6.2.2).

In der Interaktionsfolge rufen die Clients den Adapter auf, woraufhin der Adapter die adaptierte Klasse respektive Provides-Komponente aufruft. Diese Aufruffolge ist im OO- wie Komponenten-Fall identisch.

Im *OO-Fall* wird das Konzept des *„pluggable adapters“*[37] aufgeführt. Als Grundlage dienen bei den zu adaptierenden Klassen schlanke Schnittstellen, die speziell eine spätere Adaption vorsehen. Da sich das Schnittstellenangebot auf die notwendige Größe reduziert, also lediglich die Methoden anbietet, die zwingend für die angebotene Funktionalität erforderlich sind, fallen auch die Adapter sehr kompakt aus.

Für *Komponenten* empfiehlt sich ohnehin ein schlankes Interface, da dies den Integrationsaufwand von Komponenten verringert. Je schlanker Schnittstellen sind, desto weniger wahrscheinlich sind notwendige Anpassungen, etwa über verschiedene Versionen hinweg. Die Größe des Adapters richtet sich sodann nach der Größe der Requires-Interfaces.

Delegationsobjekte sind im *OO-Bereich* ein Ansatz, der dazu verwendet wird, dass ein Adapter-Client ein Objekt seiner Wahl für Anfragen verwenden kann. Ebenso können Client-*Komponenten* im Konstruktor des Adapters Referenzen für die Aufrufe des Adapters definieren. Die Instanz der Referenz wird folglich von der Requires-Komponente bestimmt.

Alle weiteren Ansätze aus dem OO-Muster lassen sich nicht in ähnlicher Form auf Komponenten anwenden und wurden daher an dieser Stelle nicht behandelt. Der geneigte Leser sei auf [Gamma] verwiesen, der das Muster umfassend behandelt.

5.2 Bedienkonzept

Das Bedienkonzept des Adapter-Generators[38] gliedert die Generierung eines Adapters in mehrere Arbeitsschritte, die im folgenden beschrieben werden.

- Im ersten Schritt, der für den Nutzer automatisch geschieht, wird der Adapter-Generator mit seiner GUI geladen. Die über die Schnittstelle[39] spezifizierten IRoles für Provides- und Requires-Seite werden in einer verkürzten Darstellung in die jeweiligen Auswahltabellen

37 Vgl. [Gamma], auch „steckbare Adapter“, ursprünglich aus ObjectWorks\Smalltalk; verfügen über eine integrierte Schnittstellenadaptierung.

38 Screenshots der Anwendung: siehe Kapitel 10.3, „Screenshots des Adapter-Generators“, Seite 53

geladen und zeigen dem Nutzer an, welche Signaturen für das Matching zur Verfügung stehen.

- In nächsten Schritt geht der Nutzer alle Requires-Signaturen einzeln durch und legt fest, welche Provides-Signatur zu der jeweils selektierten Requires-Signatur passt. Dazu gibt es die Möglichkeit die Requires-Signatur direkt über die ListBox auszuwählen („Requires-Komponente“) oder über die Pfeile im mittleren Bereich der GUI zwischen der Signaturen zu wählen.
  Zum Festlegen der passenden Provides-Signatur werden ebenso zwei Möglichkeiten angeboten:
  - Die manuelle Auswahl der passenden Provides-Signatur erfolgt wiederum über eine ListBox auf der Provides-Seite („Provides-Komponente“). Nach dem Klick auf die Signatur wird die ausführliche Typenbezeichnung automatisch in den Bereich „Verfügbar“ geladen und steht für die weiteren Zuweisungen zur Verfügung.
  - Bei der automatischen Auswahl kann der Benutzer sich über die Schaltfläche „bewerten“ vom Adapter-Generator eine Bewertung[40] erzeugen lassen, wie wahrscheinlich eine angebotene Signatur zu der aktuell gewählten Signatur passt. In absteigender Reihenfolge offeriert das System die möglichen Auswahlen.
- Nun, da bereits Requires- und Provides-Signatur ausgewählt sind, erfolgt die Zuordnung der einzelnen Methodenfragmente unter Verwendung von Konvertern zueinander. Auch dieser Schritt kann automatisiert vorgenommen werden.
  - Die manuelle Zuweisung der Methodenfragmente erfolgt per Drag'N'Drop im unteren Bereich der GUI. Möchte ein Benutzer ein einzelnes Fragment der Provides-Seite der Requires-Seite zuweisen, so kann das Fragment per Mausklick auf die gegenüberliegende Seite bewegt werden und lässt sich, durch entsprechende Cursor angedeutet, nur an Stellen einfügen, bei denen der Signatur-Fragmenttyp[41], und Typ übereinstimmen oder für den Typ-Konverter vorliegen.
  - Die automatisierte Variante führt überall dort Zuweisen durch, wo die Signatur-Fragmenttypen passen und sich die Typen, auch unter Zuhilfenahme von Typ-Konverten, matchen lassen. Nicht automatisiert lösbare Konflikte bei der Zuweisung bleiben die Aufgabe des Nutzers, der manuelle Zuweisungen durchführen kann.

    Steht für eine Requires-Typ Provides-Typ Konstellation mehr als ein valider Typen-Konverter zur Verfügung, kann der semantisch passende (oder aber auch evtl. performantere) vom Nutzer in einer ComboBox selektiert werden.

39 Schnittstellenbeschreibung siehe Kapitel 6.2, Seite 39

40 Zur Implementierung dieser Funktion siehe Kapitel 6.5, Seite 42

41 Rückgabewert, Methodenname, Parameter oder Exception

Die Schaltfläche „bewerten & belegen“ führt die beiden zuvor genannten Automatisierungsschritte direkt hintereinander durch und verringert den Aufwand bei der Erzeugung eines Adapters weiter.

Es gilt aber zu bedenken, dass alle Automatisierungen des Adapter-Generators nur die Signaturebene berücksichtigen und auf ein Matching auf dieser Ebene prüfen. Da die Semantik einiger Parameter durchaus trotz gleichen Typs und Namens vollkommen unterschiedlich sein kann, kann der Benutzer an jeder Stelle eingreifen und die Vorschläge des Adapter-Generators ausschlagen um manuelle Zuordnungen vorzunehmen.

Nur der Benutzer ist in der Lage die Semantik zu (er-) kennen, da die Signatur keinerlei semantische Informationen enthält. Daher rührt auch die Bedienphilosophie, die zwar versucht den Erzeugungsvorgang so weit wie möglich zu automatisieren, die automatischen Zuordnungen dem Benutzer aber stets zur Kontrolle vorlegt und zum Beispiel im Falle von „belegen“ mit einer Meldung warnt, wenn nicht alle Zuordnungen vollständig vorgenommen werden konnten[42].

- Im abschließenden Schritt kann der Adapter generiert werden, sofern alle Zuordnungen vollständig durchgeführt wurden. In jedem Fall wird eine Basic Component als Repräsentation des Adapter-Generators im ComponentModel zurückgeliefert. Daneben erhält der Benutzer die Möglichkeit sich auch den Quellcode des Adapters als C#-Datei in einem vorher angegebenen Pfad ausgeben zu lassen. Der Quellcode des Adapters nutzt dabei die eventuell spezifizierten Typ-Konverter.

5.3 Signatur-Missmatches

5.3.1 Vor- und Nachbedingungen des Adapter-Generators

Wie bereits festgestellt wurde, werden dem Adapter-Generator zwei inkompatible IRoles als Eingabe vorgelegt, aus denen dieser einen Adapter erzeugt. Beschreibt man diese Funktionsweise in Form von Verträgen mit Vor- und Nachbedinungen sowie in mengenorientierter Schreibweise für den Signaturumfang der Requires- und Provides-Schnittstelle, ergibt sich ein Bild, wie es Abbildung 7 dargestellt wird.

Zunächst einmal sind mit r_x und p_x die einzelnen Signaturen auf Provides- respektive Requires-Seite gegeben. Eine Schnittstelle besteht aus einer Menge von x = 1..n solcher Signaturen.

42 Zur Automatisierung vgl. auch Kapitel 4.4, Einsatz von Adaptern.

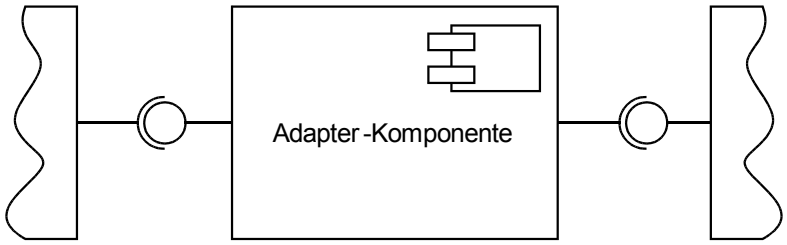
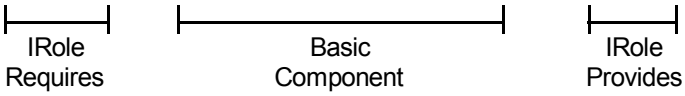
Voraussetzung	$r_x \in R_R$, $p_x \in P_P$ stellen die Signaturen auf der Provides -Seite P_P und Requires-Seite R_R dar. Sie stammen aus der Requires / Provides IRole.
Eingabe	Vertrag: pre $R_R \not\subset P_P$
Adapter-erzeugung	
Component-Model	
Ausgabe	Vertrag: post $R_R \subseteq P_G \wedge R_G \subseteq P_P$ (tatsächlich gilt sogar $P_G \setminus R_R = \varnothing \wedge R_G \setminus P_P = \varnothing$) mit R_G, als Requires- und P_G als Provides-Signaturmenge des generierten Adapters.

Abbildung 7 Mengendarstellung der Adapterfähigkeiten

Die vertragliche Vorgabe des Adapter-Generators für den Input der IRoles bedingt, dass es sich bereits um nicht zueinander passende Interfaces handelt. Dieses Missmatching wird auf Basis des Vertrags angenommen und nicht erneut vom Adapter-Generator überprüft. Da sich ein trivialer, aber auch unnötiger, Adapter ebenfalls mit Hilfe der Adapter-Generators erzeugen lässt, birgt die mögliche direkte Übereinstimmung jedoch keine Probleme. In diesem Fall wäre eine konsistente Behandlung zweier Komponenten möglich, da in jedem Fall ein Adapter in die Verbindung zwischen beiden Komponenten eingefügt werden muss. Obschon diese Verwendung des Adapter-Generators nicht den vertraglichen Vorgaben entspricht.

Die Nachbedingungen für den erzeugten Adapter sichern den verwendenden Komponenten zu, dass die Requires-Schnittstelle der ursprünglich anfordernden Komponente zunächst einmal eine Teilmenge der Provides-Schnittstelle des erzeugten Generators ist. Der Adapter-Generator erzeugt sogar einen Adapter, dessen Schnittstelle exakt den Erfordernissen entspricht. Das heisst, dass es keine Signaturen auf Provides-Adapter-Seite gibt, die keine Verwendung finden.

Auf der Seite der ursprünglichen Provides-Komponente bildet die Requires-Schnittstelle des Adapter-Generators eine Teilmenge des Provides-Angebots. Da die Provides-Schnittstelle unter Umständen sehr groß ausfallen kann, werden jedoch nicht zwingender maßen alle Signaturen tatsächlich verwendet.

5.3.2 Grundproblemklassen

[MCK04] führt unter dem Stichwort „Interface Adapter“ die Klasse von Problemen auf, die größtenteils direkt durch die Funktionalität des Adapter-Generators erfüllt wird. Die häufigsten Gründe, weshalb zwei Signaturen nicht zueinander passen, liegen in

- nicht übereinstimmenden Methodennamen. Die Requires-Seite erwartet einen anderen Methodennamen, als die Provides-Seite anbietet. Obwohl die Semantik zweier Methoden gleich sein kann wie etwa bei `RegistriereKunden` und `FuegeNeuenKundenHinzu`, muss eine Abbildung der Methodennamen aufeinander erfolgen.
- der Reihenfolge der Methodenparameter. Erfordert die Requires-Seite eine Methode mit Parametern in einer Reihenfolge, wie sie auf der Provides-Seite nicht zur Verfügung stehen, erfolgt die korrekte Zuordnung direkt über den Adapter-Generator. Dazu wird der Aufruf von der Requires-Komponente mit veränderter Parameterreihenfolge an die Provides-Seite weitergeleitet.
  Dies ist beispielsweise erforderlich, wenn die eine Requires-Methode die Parameterreihenfolge `(string vorname, string nachname)` erwartet, die Provides-Methode aber lediglich `(string nachname, string vorname)` anbietet. In diesem einfachen Beispiel stimmen die Parametertypen genau überein.

[MCK04] fasst unter „Interface Adapter“ ebenfalls Typinkompatiblitäten des Rückgabetyps und der Parametertypen zusammen. Diese Anforderungen an den Adapter werden beim Adapter-Generator durch Typ-Konverter erfüllt, wie sie in Kapitel 6.2.2 besprochen werden. Über [MCK04] hinausgehend, können auch nicht behandelte Exceptions mit Hilfe von Konvertern adaptiert werden.

5.4 Adapter-Lösungsansätze

Das in [MCK04] angesprochene „Partial Matching Problem“ kann über den Adapter-Generator gelöst werden. Mehrere Provides-Interfaces des ComponentModels, in einer IRole zusammengefasst, können mit einem Requires-Interface gematcht werden. Es besteht somit die Möglichkeit das eine Requires-Schnittstelle durch die Verwendung eine Adapters Zugriff auf mehrere Interfaces auf Provides-Seite erhält, da sich in einer IRole eine Obermenge von Provides-Schnittstellen, die für die Requires-Seite von Bedeutung sind, zusammenfassen läßt.

Im Sinne von [MCK04] lassen sich die erzeugten Adapter somit auch als „Smart Connector“ auffassen. Im Gegensatz zu einem einfachen Adapter beherrscht ein solcher komplexer Adapter auch Mechanismen wie Datenmanipulation, Transformation und Mediation

(Vermittlung von z. B. Methodenaufrufen). In Kapitel 6.2.2 werden die als Konverter-Mechanismen umgesetzten Erweiterungen vorgestellt.

Ergo lassen sich Adapter als intelligente und aktive Komponenten verstehen, da sie eine komplexe Funktionalität enthalten, das Matching vormals nicht passender Komponenten ermöglichen und somit für eine bessere Wiederverwendbarkeit von COTS-Komponenten sorgen.

5.5 Konverter

5.5.1 Verwendung im Adapter-Generator

Konverter bieten beim Adapter-Generator die Möglichkeit, den Rückgabetyp und die Parameter einer einzelnen Methode gezielt zu manipulieren. Eine Beschreibung der definierten Schnittstelle, die eine Erweiterung um weitere eigene Konverter ermöglicht, finden Sie in Kapitel 6.2.2.

Je Parameter oder Rückgabetyp kann genau ein Konverter definiert werden, der Typen kompatibel macht oder die Semantik dieser einzelnen Methodenfragmente[43] verändert.

Fragment	Rückgabe	Methodenname (Konverter unnötig)	Parameter			
ursprüngliche Methoden-Signatur	Typ A	foo	Typ B	Typ C	Typ C	...
Konvertertyp	A > I		B > II	C > III	id	... > ...
konvertierte Methoden-Signatur	Typ I	foo	Typ II	Typ III	Typ C	...

Abbildung 8 Funktionsprinzip der Konverter

Ein Konverter beim Adapter-Generator kann durch einen Dialog mit dem Benutzer

43 *Methodenfragment* bezeichnet die einzelnen Bestandteile einer Methodensignatur. Dazu zählen: Rückgabewert, Methodenname, Parameter und Exceptions.

zusätzliche Informationen aufnehmen[44]. Somit lassen sich die gleichen Konverter mehrfach durch Parametrisierung benutzen.

Wie in Abbildung 8 zu sehen ist, kann für jedes Methodenfragment, mit Ausnahme des Methodennamens, der im Adapter-Generator über andere Mechanismen gematcht wird, ein eigener Konverter verwendet werden. Stehen mehrere mögliche Konverter für die jeweilige Typkonstellation zur Verfügung, kann der ausgewählt werden, der den Erfordernissen entspricht. Implizit wird bei bereits passenden Typen ein Identitätskonverter angenommmen, der keine Typmanipulationen durchführt (im Beispiel bei Typ C angedeutet).

Konverter stellen als Grundfunktionalität eine Konvertierung von einem Typ in einen anderen dar, können darüber hinaus aber auch komplexe Berechnungen und weitere Manipulationen enthalten. Die über den Adapter-Generator realisierbaren Konverter werden im weiteren Verlauf beschrieben.

Konverter können dabei nur von der Provides-Seite für die Requires-Seite Veränderungen vornehmen, nicht anders herum. Existieren zu viele Parameter einer Methode auf Provides-Seite gegenüber der Requires-Seite, ist das Matching folglich ungültig.

5.5.2 Klassifikation von Konvertern

Zum Konzept der Konverter findet sich beispielsweise in der Literatur [MCK04], worin vier Typen von Konnektoren vorgeschlagen werden. Teile dieser Funktionalität, die als „Konnektor“ beschrieben wird, lässt sich auch im Adapter-Generator unter anderem als Konverter wiederfinden. [MCK04] nennt diese Konzepte *Wertebereichs-Transformator*[45], *Interface-Adapter*, *Funktionaler Transformator*[46] und *Workflow-Handler*, die jeweils mit einem eigenen Mechanismus ausgestattet sind, um eine Lücke fehlender Funktionalität der Provides-Komponente zu schließen.

In der Bedeutung des Adapter-Generators wird der *Interface-Adapter* allerdings nicht von einem Konverter realisiert, sondern durch die Art des Mappings im Zusammenspiel mit einer Vielzahl von Konvertern. Die Art der Funktionalität wurde daher bereits im Kapitel 5.3 „Signatur-Missmatches“ behandelt.

Der ebenfalls von [MCK04] vorgeschlagene *Workflow-Handler* erstreckt sich auf den Bereich der Protokollebene und ist daher im Rahmen des IPs nicht von Interesse, da sich die Funktionalitäten auf die Signaturebene beschränken.

Es lassen sich viele Beispiele für typische denkbare Konverter finden, die den folgenden

44 Nähere Informationen zur Funktionsweise des Konverters finden Sie in Kapitel 6.2.2.

45 „value range transformer“ in [MCK04]

46 „functional tranformer“ in [MCK04]

Klassen zuzuordnen sind:

- **Wertebereichs-Transformator** (nach [MCK04]): Stimmen die Wertebereiche von Requires- und Provides-Seite nicht überein, so kann der Wert von einem System in ein anderes transformiert werden. Dazu enthält der Konverter intern eine Formel zur Umrechnung.
  Als Beispiele sind hier z. B. Konvertierungen von Grad Celsius in Fahrenheit, Winkeln im Bogenmaß ins Gradmaß oder einer linearen Skala in eine logarithmische zu nennen.

- **Funktionaler Transformator** (nach [MCK04]): Zur Klassifizierung der möglichen Anwendungsfälle des funktionalen Transformators werden die Schnittstelleninformationen als Menge von Elementen betrachtet. Für den Adapter-Generator muss dabei nach den Elementen einer Schnittstelle, also den *Signaturen*, und den Bestandteilen einer Signatur, den *Methodenfragmenten*, unterschieden werden, da sich Probleme auf der Fragmentebene zu Teilen mit Konvertern lösen lassen. Interessant ist dabei die Mengenbeziehung von Requires- zu Provides-Schnittstelle, wie in Abbildung 9 gezeigt wird. Bei dieser Betrachtung wird zudem eine semantische Übereinstimmung für die sich überschneidenden Bereiche angenommen.

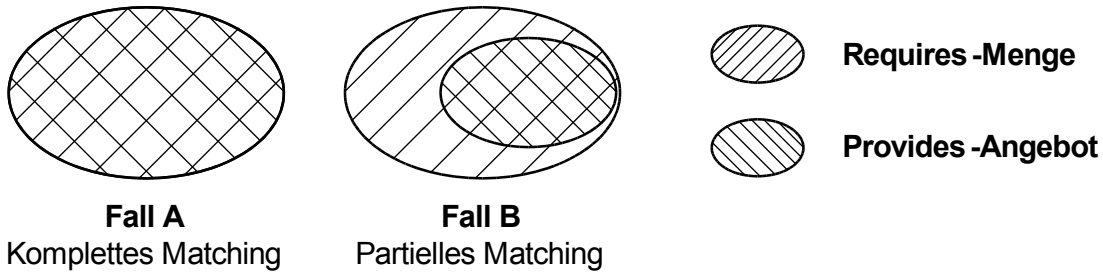


Abbildung 9 Mengendarstellung von Requires-Provides-Mismatches

  - **Fall A:** Dieses ist der einfachste Fall, in dem ein volle Überdeckung von Provides- und Requires-Seite herrscht. Das bedeutet, dass die Signaturen vollständig übereinstimmen und entsprechend auch die Methodenfragmente ein komplettes Matching ermöglichen[47].

  - **Fall B:** Die Provides-Menge ist eine Teilmenge der Requires-Menge. Mit anderen Worten bedeutet dies, dass nicht der vollständige Funktionsumfang, der erforderlich ist, tatsächlich angeboten wird.
    Fehlende Funktionalität auf Ebene der Signaturen lässt sich nicht ergänzen. Gibt es keine (Teil-) Pendant auf Provides-Seite zu einer erforderlichen Signatur, bietet der Adapter-Generator keine Möglichkeit hier direkt einen Konverter zu verwenden. Im Extremfall ist es allerdings möglich, eine beliebige Signatur auszuwählen, und alle Methodenfragmente mit Ausnahme des Methodennamens über Konverter zu emulieren.

47 An dieser Stelle soll nicht näher betrachtet werden, ob das Matching für diesen Fall evtl. durch Parametertausch erzeugt wurde, da dies beim Adapter-Generator keine Konvertertechnik darstellt.

Für diesen Fall sind Konverter für *Methodenfragmente* von Bedeutung, die Funktionalität ergänzen. Dazu gehören die unten genannten Konverter *Konstanter Parameter* und *Extraktor*.

- Der verbleibende (in der Abbildung nicht aufgeführte) Fall bedeutet, dass die Requires-Menge eine Teilmenge der Provides-Menge ist. In diesem Fall verwendet der Adapter nur Teile der Provides-Schnittstelle. Daher ist kein Konverter erforderlich.

- **Konstanter Parameter:** Im Falle gänzlich unpassender Provides-Typen für Methodenfragmente lässt sich ein „konstanter Parameter“ verwenden. Diese Art von Konverter produziert einen konstanten Typen, also einen Typen, der seinen Wert grundsätzlich nicht aus einem Input-Parameter auf der Provides-Seite berechnet. Gleichwohl sind hier Eingaben des Benutzers möglich, die Einfluss auf die Attribute des konkret ausgelieferten Konverters haben.
  Für ein Beispiel werde auf der Requires-Seite ein `int` erwartet, dessen Wert sich jedoch nicht über die Provides-Schnittstelle ermitteln ließe. Man erzeugte einen konstanten „int-Konverter“, der über die Benutzerschnittstelle ermittelt würde, z. B. den Wert „1“ trüge und fortan an das Requires-Interface ausgeliefert würde.
  Ebenso lassen sich komplexere Objekte erzeugen, die gleich mehrere Attribute aufweisen. Gäbe es einen Typen, der die Attribute A, B und C aufwiese, von denen A und B über den Konstruktor initialisiert würden und C über Accessoren veränderbar wäre, dann könnte der Konverter eine Instanz des Typs erzeugen, dabei A und B setzen und abschließend C über einen Setter modifizieren. Das somit erzeugte Objekt würde an die Requires-Schnittstelle, unabhängig von der Provides-Schnittstelle, übergeben.
  Bei dieser Form des Konverters lassen sich allerdings auch deutlich Grenzen aufzeigen, denn nach der Erzeugung des Adapters ist der übermittelte Typ konstant. Damit ist das Einsatzfeld deutlich eingeschränkt, da sich die Requires-Schnittstelle nur mit begrenzter Variabilität bedienen lässt. Dies lässt erwarten, dass der Funktionsumfang der Komponente auf Requires-Seite unter Umständen eingeschränkt ist. Zwar wird die Signaturebene vollständig gematcht, die Auswirkungen auf die Semantik sind jedoch allein durch den Adapter-Generator-Benutzer vorhersehbar.

- **Extraktor:** Konverter nach dem Extraktor-Prinzip sind für den Fall möglich, dass auf der Provides-Seite ein komplexes Objekt vorliegt, dass mehr Informationen enthält, als die Requires-Seite es erfordert. Inbesondere so genannte Container-Objekte vereinen eine Vielzahl von Informationen in einem Typ. Erfordert die Requires-Seite jedoch nur einen Teil dieser Informationen, die z. B. in Attributen vorgehalten werden, extrahiert der Konverter gezielt das erforderliche Attribut. Denkbar sind hier Zugriffe über Accessoren oder Methoden allgemein. Diese Funktion wird auch in Abbildung 10 illustriert.
  Ferner sind auch hier Ergänzungen durch Benutzereingaben möglich. Soll beispielsweise immer der Wert an einer bestimmten Stelle eines Array verwendet werden, so kann der Index über den Benutzerdialog ermittelt werden. Dies ist vor allem sinnvoll, wenn ein

solches Array eine feste Länge hat und unter einem festen Index Informationen mit fester Semantik[48] anbietet.
Da ein Methodenfragment einer Provides-Signatur durch den Adapter-Generator mehrfach verwendet werden kann, können entsprechend auch mehrere Konverter unterschiedliche Teilwerte aus ein und demselben Methodenfragment extrahieren.

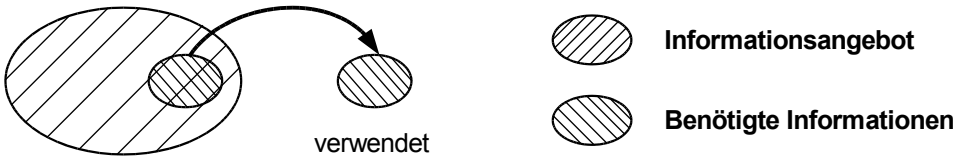


Abbildung 10 Funktionsweise des Extraktor-Konverters

- **Typ-Konverter:** Dieser Konverteransatz läßt sich am Einfachsten am Beispiel der einfachen Datentypen `int` und `boolean` erklären. Wird ein `int` (wie beispielsweise in der Programmiersprache C üblich) dazu benutzt, einen boolschen Wert zu repräsentieren, kann ein Typ-Konverter eine Übersetzung der üblichen Werte 0 und 1 in einen `boolean`-Typ vornehmen. Dadurch sind beide Parameter oder Rückgabewerte direkt adaptierbar und führen zu einem gültigen Matching.
  Des Weiteren sind diese „Übersetzungen“ auch für komplexe Typen möglich. Der Konverter konstruiert aus den vorliegenden Typinformationen und Attributen automatisch ein Objekt der erwarteten Form und reicht dieses weiter. Einfache Attributtypen der Objekte können z. B. nach dem vorangehend beschriebenen Verfahren konvertiert werden.
  Bei der Verwendung von Konvertern stellt dieses Vorgehen die wahrscheinlich am meisten genutzte Form dar, da alle nicht-semantischen Änderungen eine Typkonvertierung (mit unterschiedlichen Typen als Ein- und Ausgabe) bedingen.

Darüber hinaus sind noch beliebige weitere Konverter denkbar, die sich im Kern auf die Konvertierung eines Typs in einen anderen Typ beschränken. Sie können auch eine Kombination aus den vorangehend genannten Konzepten bilden. Dabei kann die Semantik nahezu beliebig modifiziert werden und sich z. B. auch aus anderen Datenquellen erschließen. Nicht zuletzt das Konzept von individuellen Benutzereingaben je Konverter erlaubt auch mächtige *semantische* Konvertierungen.

48 Beispiel: Es sei ein zweidimensionales Array gegeben mit der Semantik [0][x]: Name, [1][x]: Vorname, ... Dann liefert der Index „0“ dieses Arrays liefert bei Enumeration über x stets Namen.

6. Entwurf und Implementierung

Im Bereich der Implementierung soll es in diesem Kapitel vor allem um ausgewählte Aspekte gehen, die für die Funktionalität des Adapter-Generators von besonderer Bedeutung sind und einer Kommentierung bedürfen.

6.1 Architektur

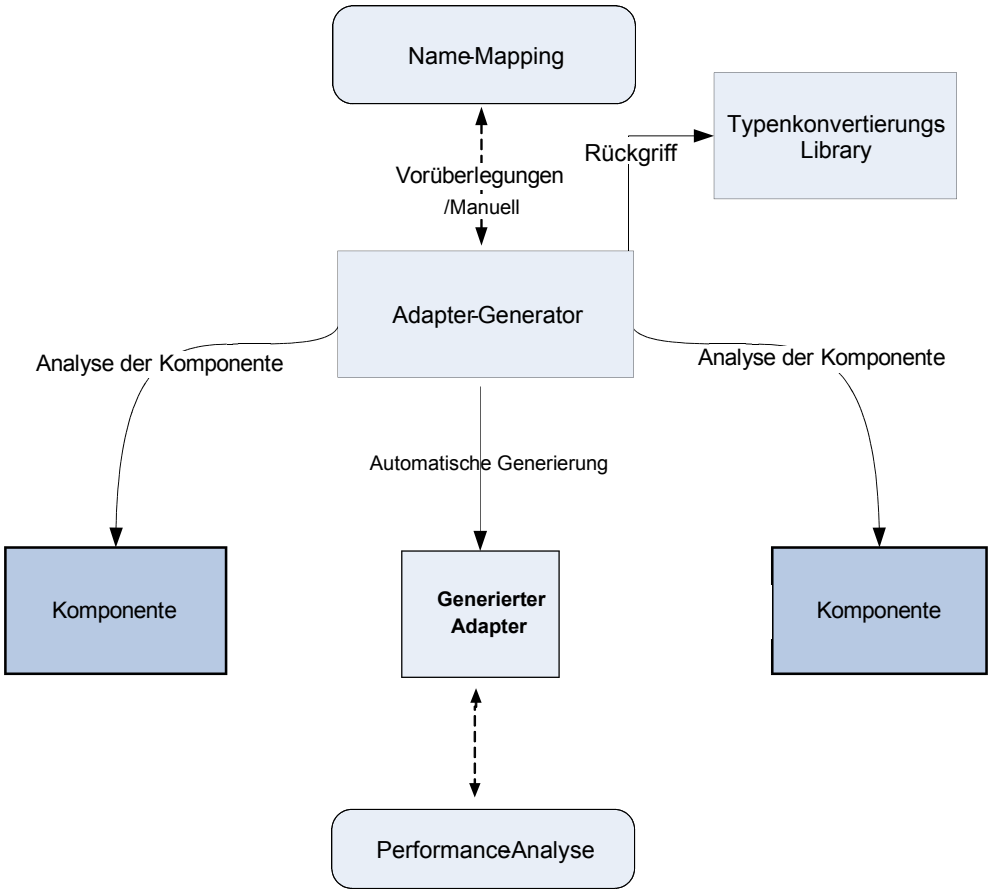


Abbildung 11 Vorgehensmodell des Adapter-Generators

Die Architektur des Adaptergenerators richtet sich vor allem nach dem Erzeugungsprozess (Abbildung 11) der Adapter. Daher sei an dieser Stelle das Vorgehen zur Erzeugung eines Adapters skizziert. Für die einzelnen Prozessteile existieren jeweils eigene Teilbereiche im Adapter-Generator, die dann jeweils die Ergebnisse der Prozessschritte über `CurrentSettings` weiterreichen.

Im ersten Schritt steht das Name-Mapping, dieses wird über die automatischen Zuordnungs- und Auswahlmechanismen (siehe Kapitel 6.5) realisiert. Diese Mechanismen wiederum gehen aus den Vorüberlegungen hervor. Die manuelle Ergänzung dieser Eingaben erfolgt durch die

Interaktion mit dem Benutzer, der das Name-Mapping per Drag'N'Drop vornimmt.

Die Konverter (Kapitel 6.2.2) stellen eine Rückgriff-Möglichkeit dar, damit das Mapping komplettiert werden kann. Sie stammen aus einer Bibliothek (dll), in der sie gebündelt vorliegen. Daneben verarbeitet der Adaptergenerator als Eingabe die Analysedaten über die Schnittstelleninformationen (Kapitel 6.2) der beiden Komponenten.

Auf der Ausgabeseite liefert der Adaptergenerator den generierten Adapter als Quellcode und als ComponentModel-Repräsentation aus. Hierauf beziehen sich die Abschätzungen in der Performance-Analyse (Kapitel 7).

6.1.1 Einordnung in das Palladio ComponentModel

Abbildung 12 zeigt eine *mögliche* Verwendung des Adapter-Generators zusammen mit dem ComponentModel. Dazu wurden in der Darstellung ebenfalls Erweiterungen des ComponentModel angegeben, die für die Gesamt-Adaptionsprozess erforderlich sind.

Als Eingabe für den Gesamtprozess dienen zwei Komponenten (C1 und C2), deren Requires- bzw. Provides-Schnittstellen nicht zueinander passen. Hinter dem Akronym MBEL verbirgt sich eine Komponente, die ursprünglich in Java geschrieben wurde, und ausführbare .NET-Dateien und Libraries parsen, erzeugen, editieren und manipulieren kann. In der ursprünglichen Funktionalität ist es recht ähnlich zur Byte Code Engineering Library (BCEL), das Java-Bytecode verarbeitet. Im Kontext von Palladio wurde MBEL derart ergänzt, dass es eine im ComponentModel verwendbare Repräsentation des Ursprungscodes erzeugt[49].

Damit lässt sich ein konkretes ComponentModel direkt aus vorliegenden Dateien heraus erzeugen. Daneben besteht aber auch die Möglichkeit das ComponentModel über vorhandene Factory-Methoden zu erzeugen, wie in Kapitel 3.3 beschrieben wurde.

Das ComponentModel selber greift wiederum auf Palladio FSM (Finite State Machines) zurück, um die Protokollebene mit Hilfe von endlichen Automaten modellieren zu können. Zur Erzeugung der Adaptoren auf Signaturebene sind die FSMs jedoch nicht von Bedeutung und wurden daher nur der Vollständigkeit halber erwähnt.

Aus dem ComponentModel heraus werden die Beschreibungen der beiden Komponenten als IRoles entgegengenommen und vom Adapter-Generator verarbeitet. Der Adapter-Generator verwendet jedoch auch intern das ComponentModel zur Darstellung der gegebenen Signaturinformationen.

49 z. T. noch in der Implementierungsphase

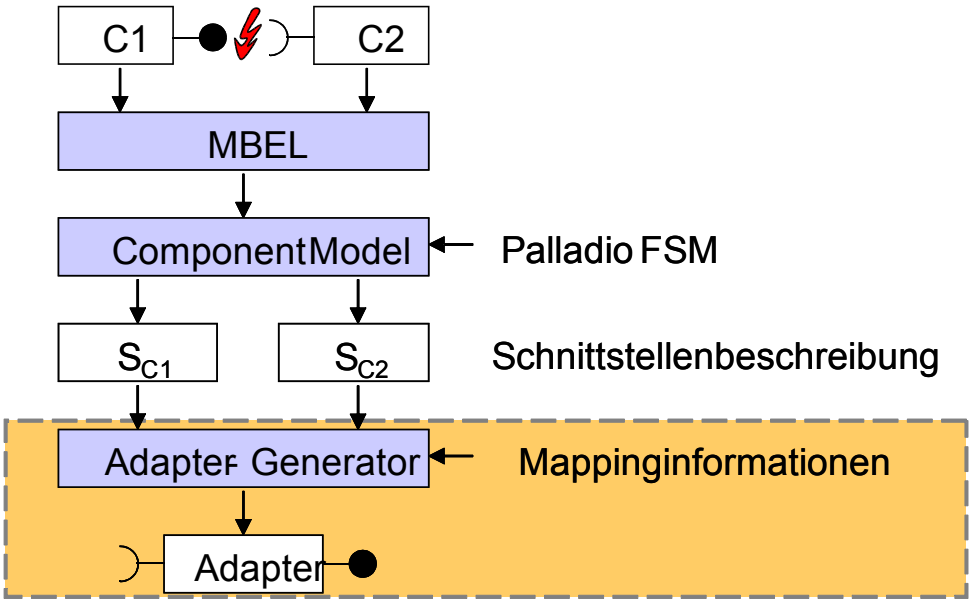


Abbildung 12 Einordnung des Adapter-Generators in das Palladio ComponentModel

Die Verwendung des ComponentModel ist intern für die Requires- und Provides-Seite nahezu identisch, so dass hier nur ein einzelnes Verfahren beschrieben wird. Zunächst wird mit Hilfe der IInterface-Beschreibung aus dem IRole-Objekt eine neue BasicComponent (IComponent) mit Hilfe der Factory-Methode erstellt. Die BasicComponent wird intern aus allen Bereichen referenziert und bietet daher Zugriff auf die darunter liegenden Komponenteninformationen, wie Abbildung 15 zeigt. Aus der IComponent wird die Signaturliste (ISignatureList über IInterface) unter anderem verwendet, um die Daten für die angebotenen Methoden einer Komponente im Fenster „Requires-Komponente“ / „Provides-Komponente“ anzuzeigen, wie im Anhang in Kapitel 10.3 abgebildet.

Die in der Signaturliste enthaltenen ISignatures werden unter anderem für die Detailansichten „Zuordnungen“ / „Verfügbar“ verwendet. Insgesamt wird das ComponentModel an allen Stellen (bis auf wenige Ausnahmen in denen die GUI direkt andere Datenquellen verlangt) verwendet, an denen die Komponenten oder ihre Bestandteile intern abgebildet werden müssen.

Nicht zuletzt liefert der Adapter-Generator auch eine IComponent (neben Quellcode) zurück, die die Adapter-Komponente repräsentiert. Damit können alle externen Komponenten, die den Adapter-Generator verwenden auf eine konsistente Komponentendarstellung für Ein- und Ausgaben zurückgreifen, was eine einfache Verwendung ermöglicht.

Damit ergibt sich als „Aufgabenbereich“ für den Adapter-Generator der gestrichelt umrahmt dargestellte Bereich in Abbildung 12. Die ebenfalls in diesem Bereich dargestellten „Mappinginformationen“ rühren von zwei Eingaben her. Auf der einen Seite sind Vorüberlegungen in der Implementierung des Adapter-Generators als Eingabe zu sehen.

Sowohl bei der Auswahl der richtigen Methode auf Provides-Seite (vgl. Kapitel 6.5, „Automatisierung“) als auch bei der Zuweisung von Methodenfragmenten zueinander fließen Erkenntnisse ein, die zum Implementierungszeitpunkt verfügbar waren. Auf der anderen Seite erfolgen Eingaben über das korrekte Matching in Interaktion mit dem Benutzer. Es sei an dieser Stelle erneut darauf hingewiesen, dass nur der Benutzer letztlich Auskünfte über die Semantik von Signaturen liefern kann, im Umkehrschluss alle Vorschläge durch die Automatismen also lediglich die über die Signatur gegebenen Informationen berücksichtigen können.

6.2 Externe Schnittstellen

6.2.1 Adapter-Generator

Der Adapter-Generator liegt zur Verwendung durch andere Komponenten als DLL-Bibliothek vor. Über die Klasse `AdaptergenBuilder` stellt der Adapter-Generator die folgende kompakte Schnittstelle zur Verfügung:

```
public IComponent createAdapter(IRole requiredRole, IRole
     providedRole)
```

Nach dem Aufruf der angegebenen Schnittstelle werden direkt die GUI-Komponenten instanziert, womit der Adapter-Generator verwendbar ist. Wie erkennbar ist, liefert der Adapter-Generator in jedem Fall eine IComponent zurück, die eine Repräsentation des Adapters enthält.

Als Eingaben dienen dem Adapter-Generator lediglich zwei IRole-Objekte, die die Provides- bzw. Requires-Seite enthalten.

Als weitere „Schnittstelle“ nach außen darf die Möglichkeit angesehen werden, dass der Quellcode des Adapters im Dateisystem abgelegt werden kann. Damit ist ebenfalls eine Möglichkeit gegeben, den Adapter in realen Systemen einzusetzen oder in anderen Kontexten weiter zu verarbeiten.

6.2.2 Konverter

Konverter dienen, wie bereits angesprochen, dazu, die Typen auf der Requires- und Provides-Seite zu manipulieren. Da beliebige Konverter denkbar sind, die je nach Einsatzszenario des Adapter-Generators sehr stark variieren können, wurde ein Interface definiert, dass die Konverter implementieren müssen. Damit ist es möglich, eigene Konverter zu ergänzen.

Die Schnittstelle ist wie folgt definiert:

```
public interface ITypeConverter
{
     IType ConcreteRequiredType { set; get; }
     IType ConcreteProvidedType { set; get; }
     IType RequiredType { get; }
     IType ProvidedType { get; }
     string converterName();
     string converterDescription();
     void performConvertion();
}
```

Über die Methode `converterName()` liefert ein Konverter seinen Namen zurück, der dann dem Benutzer angezeigt werden kann. Ebenfalls zu Informationszwecken dient die `converterDescription()`-Methode, die eine ausführliche Beschreibung des konkreten Konverters enthalten sollte, die unter anderem die genaue Funktion erklärt.

Daneben sind die Accessoren `RequiredType` und `ProvidedType` definiert, die einen IType zurückliefern und damit angeben, für Konvertierungen von welchem zu welchem Typen sie geeignet sind. Entsprechend darf diese Zugriff auch nur lesend erfolgen.

Für die eigentliche Konvertierung sind zunächst einmal die Accessoren `ConcreteRequiredType` und `ConcreteProvidedType` vorgesehen. Sie müssen vor der Durchführung der Konvertierung gesetzt werden und enthalten eine Referenz auf die konkret zu konvertierenden Typen. Die Werte dieser beiden Typen werden folglich verändert. Die Manipulation wird über die Methode `performConvertion()` angestoßen.

Damit Konverter flexibler eingesetzt werden können, wird vor dem Ausführen der Konvertierung die Methode `showInputDialog()` aufgerufen, die vom Konverter zum Beispiel mit Abfragen über Konvertierungsparameter in Form eines GUI-Fensters gefüllt werden kann. So ist für einen Währungskonverter die Abfrage eines Wechselkurses denkbar.

Zur Erleichterung der Erstellung von Abfragen über die GUI wurde das `IInputDialog`-Interface definiert:

```
public interface IInputDialog
{
     string showDialog();
     string CaptionText { set; }
     string QuestionText { set; }
}
```

Dazu existiert die Implementierung im Form des `InputDialog`s, der bereits eine Form erzeugt, auf der ein Fragetext anzeigt wird. Der Wert des integrierten Antwortfelds wird als Ergebnisstring zurückgeliefert und kann dann verarbeitet werden.

Da auch für den Konverter Interaktionen mit dem Benutzer vorgesehen sind, kann dadurch ebenfalls die Wertsemantik aller Typen, für die ein Konverter verwendet werden kann, erfasst werden.

Konverter sollten im Allgemeinen zustandslos sein, damit sie sich bei jeder (automatischen) Verwendung durch den Adapter-Generator respektive Adapter identisch verhalten und keine unerwünschten Seiteneffekte erzeugen.

Zur Verwendung neuer Konverter muss lediglich die `Adapters.dll` erneuert werden. Sämtliche Konverter werden dynamisch zur Laufzeit des Adapter-Generators erkannt und per Reflection aus der Bibliothek gelesen. Damit stehen sie unmittelbar für die Konvertierung zur Verfügung. Folglich lassen sich neue Konverter sehr leicht in den Adaptionsprozess einbinden.

6.3 Datenhaltung

Die Datenhaltung der Komponentenrepräsentation erfolgt beim Adapter-Generator vor allem in einer zentralen Containerklasse `CurrentSettings`, die Referenzen auf die IRoles der Provides- und Requires-Seite hält, und damit auch auf alle darunter liegenden Signaturinformationen. Daneben werden in der Klasse auch die Referenzen auf die Mapping- und sonstigen Informationen, die den derzeitigen Zustand des Adapter-Generators darstellen, gehalten.

Sollte es nötig werden, dass der Adaptionsprozess zwischengespeichert wird, so lassen sich prinzipiell von dieser Klasse aus alle Zustandsinformationen serialisieren und somit zum Beispiel in eine Datei schreiben.

6.4 Darstellung

Die zentrale Darstellungskomponente des Adapter-Generators bildet das `DragDropDataGrid`, eine von der `DataGrid`-Klasse aus dem ADO.NET abgeleitete Klasse, die um Funktionalität für Drag'N'Drop erweitert wurde.

Die Basiskomponente des DataGrids eignet sich besonders zur Darstellung komplexer Daten und bietet bereits von Haus aus Mechanismen zur Trennung von Darstellung und Datenhaltung. Durch ihre Wurzeln im Bereich von ADO.NET, also dem Bereich .NETs zur Datenbankanbindung, sind aber andererseits auch weniger strikte Mappings von Darstellung auf die Daten möglich, denn Datenquellobjekte können auch über ihre Namen in String-Repräsentation referenziert werden.

Für DataGrids können in gewisser Weise Masken (`ColumnStyles`) definiert werden, die festlegen, welche Teile der internen Datenstruktur in welcher Weise angezeigt werden sollen. Damit lassen sich auch ComboBoxes in einzelnen Zellen realisieren, die auf die bestehenden Datenstruktur zugreifen und manipulieren können.

Damit die DataGrids in der Umgebung des Adapter-Generators möglichst einfach verwendet werden können, wurden sie u. a. um einen Drag'N'Drop-Mechanismus erweitert, der Drags nur in das jeweils andere DataGrid zulässt und Drops nur an den Stellen erlaubt, an denen die Typen der Requires- und Provides-Seite sinnvoll unter möglicher Verwendung eines Konverters abgelegt werden können. Die passenden Stellen werden mit Standard-Drag'N'Drop-Optik kenntlich gemacht und ermöglichen eine intuitive Einordnung.

6.5 Automatisierung

Zur Unterstützung des Adaptionsvorgangs wurden, wie bereits angesprochen, Automationsmechanismen implementiert. Die Automatisierung greift dabei an zwei Stellen im Generierungsprozess ein: bei Auswahl einer passenden Signatur zu der aktuell gewählten und beim Zuweisen von Methodenfragmenten der Provides- zur Requires-Seite, wie bereits oben besprochen wurde. Im Folgenden soll vor allem auf den Bewertungsmechanismus zur Auswahl der passenden Signatur eingegangen werden.

6.5.1 Bewertungsalgorithmus

Als Ausgangsbasis für die Erzeugung eines Ratings dient die aktuell gewählte Requires-Signatur. Danach werden der Reihe nach alle Signaturen auf der Provides-Seite durchgegangen und einzeln bewertet. Der jeweils ermittelte Wert wird danach dem Benutzer in einem Dialog präsentiert.

Die Berechnung des Rating-Werts lässt sich (zunächst einmal fest codiert) in der `MethodFitsRating`-Klasse parametrisieren, womit sich die erzielbaren Werte weiter optimieren lassen. Dazu ist ein Maximalwert je Bewertungsbestandteil definierbar, mit dem die Erfüllung eines Kriteriums berechnet wird.

Bei der Berechnung finden folgende Faktoren Berücksichtigung:

- Methodenname: Die Ähnlichkeit des geforderten mit dem angebotenen Methodennamen wird bewertet. In der bisherigen Implementierung findet die auf String-Objekten definierte `CompareTo`-Methode Anwendung. Für spätere Verbesserungen lassen sich hier jedoch auch wörterbuchbasierte Vergleiche, die Reflexions- und Stammformen der Namen berücksichtigen denken. Da Methodennamen oftmals natürlichsprachige Bezeichnungen sind, die durch CamelCase-Schreibweise eine Worttrennung implizieren, dürfte dieses Verfahren eine deutliche Verbesserung bringen. So sind die Methodennamen `WortErgänzung` und `ErgänzeUmWort` sehr ähnlich und könnten unter Umständen mit einem solchen Verfahren erkannt werden.
  Zwar liefert der Methodenname den größten Hinweis auf die Semantik einer Methode, zugleich birgt die Berücksichtigung des Methodennamens eine große Gefahr, dass die Methoden (trotz eines möglicherweise identischen Namens) eine komplett andere Semantik haben. Das Beispiel `sqr` wurde in diesem Zusammenhang bereits genannt und verdeutlicht, dass es sich bei den gleichen Methodennamen sowohl um die Wurzelberechnung, wie auch um das Quadrat einer Zahl handeln kann.
  Daraus folgt, dass die Bewertung des Methodennamens nicht der entscheidende Hinweisfaktor auf eine passende Semantik sein darf, da ansonsten möglicherweise die falschen Methoden vorgeschlagen würden, bei denen evtl. andere Bewertungsparameter untergewichtet blieben. Anders herum ist der Name im Allgemeinen sehr aussagekräftig, so dass daraus resultierende Übereinstimmungen eine deutliche Bewertung im Rating finden sollten.

- Rückgabetyp: Wie bei allen weiteren Ratings von Typen, findet bei der Bewertung Beachtung, ob die Typen direkt passen, (`equals`) oder über einen vorhandenen Konverter modifiziert werden können. Ist der Einsatz eines Konverters notwendig, erfolgt eine definierbare Abwertung. Passt der Requires-Typ direkt zum Provides-Typ, wird die volle Bewertung vergeben.

- Anzahl der Parameter: Da durch den Einsatz von Konvertern eine freiere Anzahl von

Parametern ermöglicht wird, erfolgt eine Zusatzbewertung nur, sofern die Parameteranzahl der Methode exakt übereinstimmt.

- Parameter: Die Parameter werden zunächst einmal einzeln wie auch der Rückgabetyp bewertet. Auch hier erfolgt die Unterscheidung nach direkt und indirekt passenden Typen. Zusätzlich werden jedoch auch die Parameternamen bewertet. Stimmen die Namen exakt überein, wird das Rating für den entsprechenden Parameter aufgewertet.
- Exceptions: Exceptions finden zunächst noch keine Bewertung, da hier prinzipiell immer ein Abfangen mit Hilfe entsprechender Konverter möglich ist. Entsprechend wird der Ratingwert nicht durch Exceptions beeinflußt.

Aus der Summe der Einzelratings ergibt sich zuletzt ein Gesamtwert für eine Signatur, der dann ausgewiesen wird.

Offenbar handelt es sich bei der Bewertung um einen verbesserungswürdigen Variationspunkt des Adapter-Generators. Gerade die Einführung eines Wörterbuchs könnte signifikante Verbesserungen hervorbringen. Daneben sind weitere Kriterien denkbar, die in die Bewertung einfließen können. Daher wurde das IMethodFitsRating-Interface eingeführt, das weitere Implementierungen der Bewertungsfunktion erleichert:

```
public interface IMethodFitsRating {
     RatedSignatureList rate(CurrentSettings currentSettings);
}
```

6.5.2 Zuweisungsalgorithmus

Gegenüber dem Bewertungsalgorithmus arbeitet der Zuweisungsalgorithmus vergleichsweise einfach. Da vor der Zuweisung der Parameter zueinander bereits die automatisierte Bewertung durchgeführt, und diese vom Benutzer geprüft wurde, ist die Annahme gerechtfertigt, dass die Methodenfragmente bereits relativ gut passen.

Daher erfolgt die Zuordnung in einer Schleife, worin der Reihe nach die Methodenfragmente der Requires-Seite durchgegangen werden. Passt ein Parametertyp der gewählten Provides-Komponente (unter Berücksichtigung der verfügbaren Konverter), wird dieser zugewiesen.

Der Algorithmus warnt im Übrigen, wenn eine Zuordnung nicht vollständig automatisch durchgeführt werden konnte. Sollten bei der Zuordnung Fehler auftreten, so können diese dann nachträglich über den Drag'N'Drop-Mechanismus behoben werden.

6.6 Tool-Einsatz

Die Software des Individuellen Projekts wurde in der Microsoft .NET-Sprache C# entwickelt. Die Entscheidung für C# erfolgte aus zwei Gründen:

- Das zu verwendende Palladio ComponentModel ist in C# implementiert. Durch die Verwendung von C# war die Kommunikation mit den vorliegenden Komponenten problemlos möglich. Der Einsatz in der integrierten Entwicklungsumgebung war damit komfortabel und übersichtlich[50].
- Zudem unterstützt der umfangreiche Reflection-Namespace die Arbeit bei der Laufzeitinitialisierung von Typen, das Auslesen von Typinformationen und die Verwendung der als Assembly-Teil vorliegenden Typ-Konverter. Die Ergänzung neuer Typen oder das Hinzufügen neuer Konverter erfordert die Neucompilierung der Typen bzw. des Konverter-Assemblies, braucht aber nicht auf den eigentlichen Adapter-Generator zurückzugreifen.

Als integrierte Entwicklungsumgebung wurde Microsoft Visual Studio .NET 2003, dass über MSDNAA[51] zur Verfügung stand, verwendet. Da die vorliegende Version über keine Refactoring-Unterstützung verfügt, wurde diese über IntelliJ ReSharper als Plug-In ergänzt. Zwar befand sich die Version über den Zeitraum der Entwicklung hinweg im Stadium einer Alpha-Software[52], lief für die benötigten Aufgaben jedoch ausreichend stabil und ermöglichte durch die Funktionalitäten eine produktive Entwicklung der Software.

Zur Erstellung der Dokumentation zum Projekt wurde OpenOffice 1.1.2 von Sun Microsystems benutzt. Die quelloffene Software eignet sich auch für größere Dokumente und erlaubt umfangreiche Querverweise, automatische Verzeichnisse und schemagebundene Formatierungen von Texten und Grafiken. Gegenüber einer LaTeX-Umgebung heben sich vor allem mehr gestalterische Eingriffsmöglichkeiten, integrierte Rechtschreibprüfungen etc. hervor.

Die Grafiken wurden primär mit Microsoft Visio 2003 erstellt. Die vorhandenen Ausgabeformate ermöglichten eine problemlose Einbindung in das vorliegende OpenOffice-Dokument.

50 Auto-Syntax-Completion, etc.

51 MSDNAA: Microsoft Developer Network Academic Alliance

52 Im Rahmen eines EAP: Early Access Program

7. QoS-Merkmale der Adaptoren

Die in diesem Kapitel vorgestellten Abschätzungen der QoS-Merkmale der erzeugten Adapter basieren auf einer Analyse der Aufrufstruktur. Da insbesondere keine Tests mit den konkreten Adaptern durchgeführt wurden, sind die hier vorgestellten Metriken nicht validiert, sondern zeigen vielmehr Klassen auf, in die bestimmte Techniken grob eingeordnet werden müssen. Damit die Analyse nachvollziehbarer wird, werden jeweils die entsprechenden Fragmente aus dem Quellcode der erzeugten Adaptoren wiedergegeben.

Da QoS eine große Menge von möglichen Werten beinhaltet, soll hier vor allem auf die Verzögerung der Aufrufe der Provides-Komponente durch die Einsatz von Adaptern eingegangen werden. Dieser Wert dürfte einen der Hauptmerkmale des Adapters ausmachen, da eine der Kernfunktionalitäten die Umleitung von Methodenaufrufen ist.

7.1 Konstruktor

Da jeder Adapter zunächst einmal eine Indirektionsstufe darstellt, darf angenommen werden, dass jeder Adapter mindestens eine konstante Aufrufverzögerungen für den Adapter beinhaltet.

```
private ProvidesKomponente providesKomponente;
public Adapter (ProvidesKomponente providesKomponente) {
     this.providesKomponente = providesKomponente;
     ...
```

Diese ergibt sich durch den Aufruf der Adapterkomponente durch die Requires-Komponente (Konstruktor), die dann wiederum die Provides-Komponente anspricht. Da in diesem Schritt zwei Komponenten geladen und initialisiert werden müssen, was intern unter anderem Speicherverwaltungsaufgaben und Objektverwaltung anstößt, wird dies vergleichsweise viel Zeit kosten. Zur Betrachtung der Eigenschaften der Adapter-Komponente, ist jedoch der Zeitbedarf zur Initialisierung der *Provides*-Komponente nicht von Interesse, da dieser Wert je nach Größe und Komplexität der zu ladenden Komponente schwankt und zudem von der Requires-Komponente übernommen wird.

7.2 Methodenaufrufe

Nachdem die Provides-Komponente im Adapter als Referenz vorliegt, können darauf Methoden ausgeführt werden. Auch hier läßt sich ein konstanter Anteil ausmachen, der mindestens in jedem Methodenaufruf enthalten ist.

Zur Unterscheidung der Varianten mit und ohne Rückgabewert wird hier eine an reguläre Ausdrücke erinnernde Schreibweise verwendet:

```
public [ReturnType|void] adapterMethode(Typ1 param1, Typ2 param2, ...
{
     ...
     [return]+ providesKomponente.providesMethode(param2, paramX, ...);
     ...
```

Betrachtet man auch hier nur die relevante Aufrufzeit, die durch den Adapter verursacht wird, so ist vor allem die untere Zeile von Interesse, in der der Aufruf an die Provides-Komponente weitergeleitet wird. Der fixe Anteil wird durch die Weiterleitung des Methodenaufrufs an die Provides-Komponente verursacht.

Es sei an dieser Stelle angemerkt, dass für die vorliegende Adaptervariante in C# davon ausgegangen wird, dass keine hochoptimierenden Compiler verwendet werden, die möglicherweise den Adapter vollständig „wegoptimieren“, und somit eine einfache Variante des Adapters fast vollständig ohne Verzögerung ausführen könnten.

Wie leicht nachvollziehbar sein dürfte, richtet sich die Aufrufverzögerung vor allem nach der Zahl der übergebenen Parameter. Der variable Anteil der Verzögerung ergibt sich folglich entsprechend [Zahl der Parameter] * [Zeitbedarf je Parameter]. In der umgekehrten Richtung wird sich eine Verzögerung durch die Weiterleitung der Rückgabewerte (sofern diese nicht vom Typ `void` sind) ergeben.

Damit ist dieser Teil eines erzeugten Adapters insgesamt relativ leicht abzuschätzen. Die statische Berechnung von Werten dürfte relativ genaue Werte liefern. Da prinzipiell Referenzen auf Objekte übergeben werden, dürfte sich die berechnete Zeit auch nicht durch die Größe der Datenstrukturen, die referenziert werden, verändern.

7.3 Konverter

Die Konverter sind bei weitem am schwersten bezüglich ihrer Performance abzuschätzen, da ihre Funktionalität von einer einfachen Konvertierung eines Integer von einer Einheit in eine andere, bei der ein Integer mit einem festen Faktor multipliziert wird, bis hin zu komplexen beliebigen Berechnungen reichen kann. Daher kann der variable Anteil der Konverter sehr

stark schwanken und hängt von jedem einzelnen Konverter ab. Zusätzlich sind weitere dynamische Effekte zu erwarten, die in Abhängigkeit von der zu verarbeitenden Datenmenge auftreten. Da Konverter tatsächlich auf den Daten operieren, dürfte die Laufzeit massiv von diesem Faktor verändert werden.

Der fixe Anteil wird wiederum von den Aufrufen der Konverter (Initialisierung) gebildet, sofern ein Konverter für einen Typ verwendet wird. Wird keine Konverter verwendet, ergeben sich natürlich auch keine Performance-Effekte.

7.4 Weitere QoS-Merkmale

QoS-Merkmale wie Reliability dürften durch den eigentlichen Adapter nicht beeinflußt werden, da der Adapter selber keinen Zugriff auf externe Ressourcen vornimmt. Werden hingegen Konverter eingesetzt, so ist auch dieses Verhalten vollständig von den Konvertern abhängig. Nimmt ein Konverter beispielsweise über das Internet Zugriff auf entfernte Ressourcen, dürfte sich die Zuverlässigkeit signifikant verschlechtern.

Die durchschnittliche Antwortzeit wird sich ähnlich wie die oben aufgeführte Verzögerung ergeben. Da in der Regel nicht davon auszugehen ist, dass die Adaptoren auf Echtzeitsystem eingesetzt werden, wofür sich C# auch nur schwerlich eignen dürfte (Stichwort: Garbage Collection), lässt sich keine obere Schranke für die garantiert Antwortzeit angeben.

Anhand des Beispiels des Adapaptergenerators zeigt sich bereits, von wie vielen Faktoren die Bestimmung von z. B. Performance-Werten abhängig ist. Daher würden sich in der Praxis Werte, die mit Hilfe von Tests validiert würden, ergeben, die eher eine Klassifizierung des Laufzeitverhaltens ermöglichen, denn eine exakte Vorhersage. Vor allem das Konverterkonzept verlangt eine exakte Bestimmung der QoS-Attribute eines jeden Konverters, die zudem noch über die Eingabewerte dynamisiert werden müssten.

8. Fazit

8.1 Vergleich mit den Fähigkeiten anderer Adapterkonzepte

Zum Abschluss dieser Ausarbeitung soll betrachtet werden, wie sich die beschriebenen Fähigkeiten des Adapter-Generators im Vergleich mit anderen Adapterkonzepten, wie sie in

der Literatur zu finden sind, darstellen. In [MCK04] findet sich eine Übersicht über verschiedene Ansätze zur Adaption. Diese wurde in Abbildung 13 um die Spalte „Adapter-Generator“ erweitert und ermöglicht einen guten Gesamtüberblick.

Vergleichskriterium	Spitznagel (1)	Mehta	Spitznagel (2)	Smart Connector	Adapter-Generator
„Glueing Components“ / Verbinden von Komponenten	●	●	●	●	●
wertebereichsmodifizierend	○		○	●	●
Adaptieren von Komponenteninterfaces	●	○	●	●	●
Modifikation von funktionalem Verhalten	○		●	●	●
Hinzufügen neuer Funktionalität	●	○	●	●	●
Veränderungen des Kontrollflusses	○	●	○	●	
Kombination verschiedener Konnektoren / Konverter	●		○	●	○

● vollständige Unterstützung ○ partielle Unterstützung

Abbildung 13 Vergleich mit anderen Adapterkonzepten (erweitert nach [MCK04], Table 1)

Zu bedenken ist, dass der Adapter-Generator auf Signaturebene arbeitet und daher keine sinnvollen Veränderungen des Kontrollflusses vornehmen kann. Ergo findet sich auch keine Unterstützung in der unten abgebildeten Tabelle. Die Kombination verschiedener Konverter ist beim Adapter-Generator nicht explizit vorgesehen. Eine Kombination von Konvertern ist gleichwohl implizit über Programmiersprachenkonstrukte wie Vererbung möglich und bleibt alleine dem Konverter überlassen.

Es bleibt festzustellen, dass vor allem die Einführung von Konvertern die größte Flexibilisierung für den Adapter-Generator bringt. Die meisten der in der Tabelle genannten Kriterien werden erst durch Konverter möglich, die zu deutlich intelligenteren Adaptern führen.

8.2 Projektverlauf

Der im [Proposal] vorgesehene Projektplan sah die Entwicklung des Adapter-Generators in

drei Inkrementen vor, die nach der Entwicklung eines Prototypen vorgesehen waren. Der *Prototyp* sollte dabei unter Vorbehalt sinnvoller Verwendungsmöglichkeit bereits die Entwicklungsbasis für die weiteren Iterationsschritte bilden. Vorgesehen war die Möglichkeit, die Requires- zur Provides-Schnittstelle manuell zuordnen zu können. Erreicht wurde ein GUI-Prototyp, der manuelle Zuordnungen der einen DataGrid-Seite zur anderen ermöglichte und dabei bereits auf eine interne Repräsentation einer einzelnen Schnittstelle zurückgreifen konnte. In diesem Schritt wurde vor allem das mögliche Bedienkonzept zur manuellen Erzeugung eines Signaturmappings erprobt.

Anstelle der im Proposal vermuteten diskreten Entwicklungsschritte wurden im Verlauf der Entwicklung stetig Erweiterungen im Funktionsumfang vorgenommen, so dass nicht eindeutige Inkremente am Ende eines Entwicklungsschritts standen. Im CVS wurde dabei die jeweils aktuell lauffähige Version angeboten.

Im *ersten Iterationsschritt* sollte der Adapter-Generator die syntaktischen Analysen der Komponenten automatisiert durchführen können. Durch die Verwendung des Palladio ComponentModel wurde die Analyse der gegebenen Komponenten nach außen verlagert. Die Eingabe von IRoles ermöglichte den direkten Zugriff auf die darunter liegende Signaturbeschreibung der Schnittstellen. Daher wurde in diesem Iterationsschritt vor allem das ComponentModel in den Adapter-Generator integriert. Dies umfasste die Eingabe der Komponentenbeschreibung wie auch die interne Repräsentation von Datenstrukturen mit Hilfe der ComponentModel-Konstrukte.

Da das ComponentModel in der Zeit des IPs deutlich erweitert und restrukturiert wurde, hatte dies zur Folge, dass der Adapter-Generator an die neuen Schnittstellen und Datencontainer angepasst werden musste. Auf der anderen Seite bot der erweiterte Funktionsumfang eine einfachere Verwendung des ComponentModels.

Für den *zweiten Iterationsschritt* war die Erfassung von QoS-Merkmalen der erzeugten Adapter vorgesehen. In Absprache mit dem Betreuer wurde die Behandlung von QoS-Merkmalen auf den Theorieteil in dieser Ausarbeitung verlagert, wie sie in Kapitel 7 nachzulesen ist. Da die QoS-Merkmale nicht zuletzt durch das Konzept der Konverter maßgeblich beeinflußt werden, wurde dieser Schritt daher an das Ende der Projektphase verschoben.

Der abschließende *dritte Iterationsschritt* sollte ursprünglich weitere Ergänzungen im Funktionsumfang des Adapter-Generators mit sich bringen. Dazu gehört unter anderem das komplette Konverterkonzept aber auch der Tausch von Parameterreihenfolgen, das Matching von Methodennamen usw. Da sich diese Funktionen jedoch als elementar für den Adapter-Generator herausstellten, wurden die Implementierungen dieses Schritts bereits nach dem ersten Iterationsschritt durchgeführt, womit der zweite Iterationsschritt zum letzten geplanten wurde.

Als Erweiterung zu der bestehenden Implementierung des Adapter-Generators wurde jedoch die Erzeugung weiterer Typ-Konverter aufgenommen, damit sich der Adapter-Generator breiter testen und verwenden läßt. Zudem kann die Implementierung der bestehenden Konverter als Beispiel für weitere Ergänzungen verwendet werden.

Insgesamt zeigte sich, dass der Zeitplan unrealistisch geschätzt worden war, und die frühen Iterationsschritte deutlich mehr Zeit in Anspruch nahmen, als zunächst angenommen. Daher wurde als Ergänzung zum [Proposal] eine Aktualisierung des Zeitplans verfasst, der der realen Entwicklung des Projekt eher entsprach. Zusammengefasst wurde die Mehrzahl der Ziele des Individuellen Projekts erreicht.

9. Ausblick

Da der Adapter-Generator an einigen Stellen gewollt Erweiterungsmöglichkeiten läßt, bietet es sich vor allem an, weitere Typ-Konverter zu erstellen, damit weitere Anwendungsdomänen für die Nutzung mit dem Adapter-Generator erschlossen werden.

Ebenso eignet sich die Implementierung weiterer Bewertungsalgorithmen für die Verbesserung der Präzision von Signaturenratings. Damit lassen sich zugleich die automatischen Fähigkeiten des Adapter-Generators erweitern und das Handling verbessern, worin die effizientere Erzeugung von Adaptoren resultiert.

Neben diesen Verbesserungen unter der Oberfläche sind auch noch weitere Optimierungen des User-Interfaces möglich. So wird bisher nicht grafisch mit einem veränderten Mauscursor angezeigt, welche Elemente per Drag'N'Drop beweglich sind. Weitergehende Veränderungen der GUI sollten maßgeblich durch eine Nutzerstudie fundiert werden, damit die Entwicklungen in die richtige Richtung vorangetrieben werden.

Ein wichtiges Ziel dürften jedoch die im zweiten Iterationsschritt vorgesehenen Erweiterungen um eine Unterstützung der Berechnung von QoS-Eigenschaften sein. Da das ComponentModel ausdrücklich die Erfassung von QoS-Merkmalen zu Schnittstellen als AuxilaryInformation vorsieht, ließen sich damit präzise Aussagen über die Adaptereigenschaften im Zusammenspiel mit den verwendeten Requires- / Provides-Komponenten treffen lassen.

10. Anhang

10.1 RADL-Komponentendarstellung

Neben der UML2 zur Darstellung von Komponentenbeziehungen ist ebenfalls die RADL-Notation (DSTC/Monash, Melbourne), einem Nachfolger von DARWIN (IC, London) gebräuchlich (siehe Abbildung 14). Bei dieser Darstellung gibt die Pfeilrichtung die Kontrollflussrichtung an.

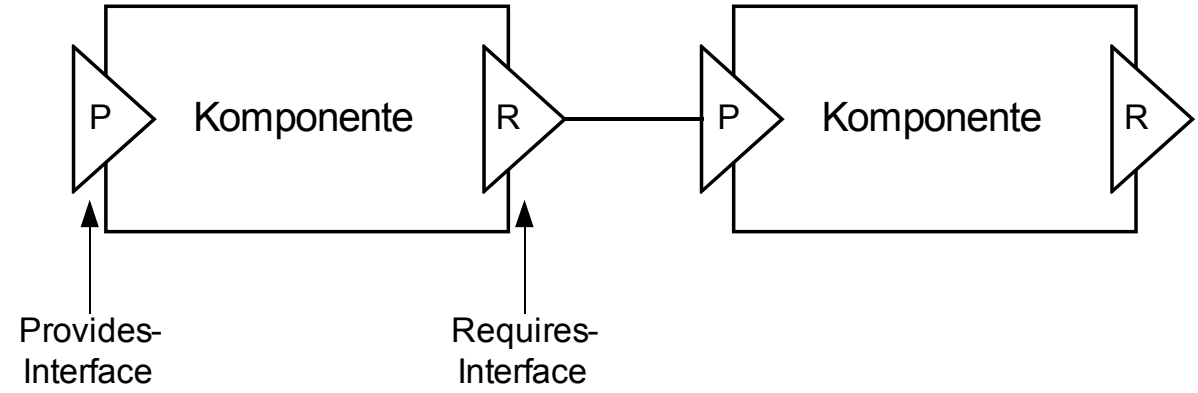


Abbildung 14 RADL-Komponentendarstellung

10.2 ComponentModel

In Abbildung 15 wird grob die Hierarchie der Strukturelemente des ComponentModels skizziert. Abgebildet sind nur diejenigen Elemente, die für den Adapter-Generator eine Bedeutung haben.

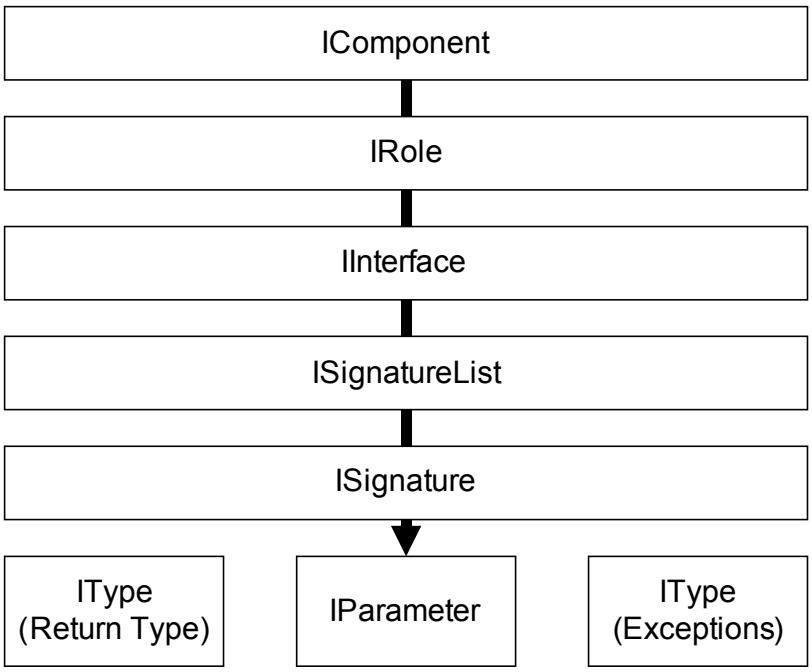


Abbildung 15 Grobe hierarchische Darstellung der Elemente des ComponentModels

10.3 Screenshots des Adapter-Generators

In Abbildung 16 und 17 finden die zwei Hauptdarstellungsformen des Adapter-Generators.

Abbildung 17 liefert ein Rating über die wahrscheinlich passendste Signatur auf Provides-Seite zu der aktuell gewählten Requires-Signatur. Dargestellt wird ein Rating-Index, der die konkrete Bewertung einer Signatur darstellt. Höhere Werte bedeuten eine bessere Bewertung. Die Werte werden dabei automatische absteigend sortiert. Vorgewählt ist der am besten bewertete Eintrag. Damit der Nutzer den Vorschlag komplett bewerten kann, werden die vollständige Requires- und auch Provides-Signatur angezeigt.

Abbildung 16 ist das Hauptfenster des Adapter-Generator und ermöglicht die Selektion der zuzuordnenden Signaturen und die konkrete Zuordnung unter der möglichen Verwendung eines Konvertes. Manuelle Zuordnungen werden hier über einen Drag'N'Drop-Mechanismus vereinfacht, der gleichzeitig einen Typcheck gegen die gewünschte Zuordnung durchführt.

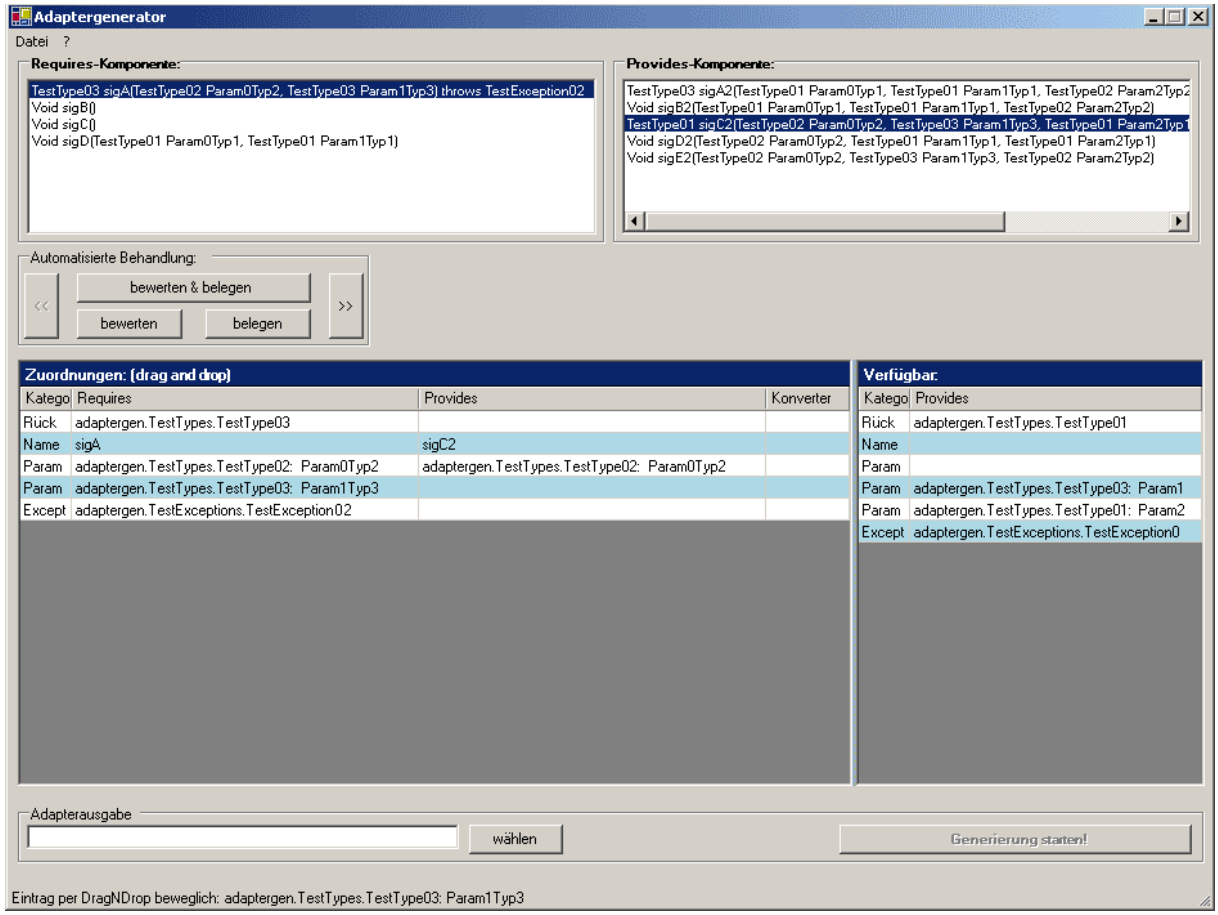


Abbildung 16 Darstellung des Hauptfensters der Adapter-Generator GUI

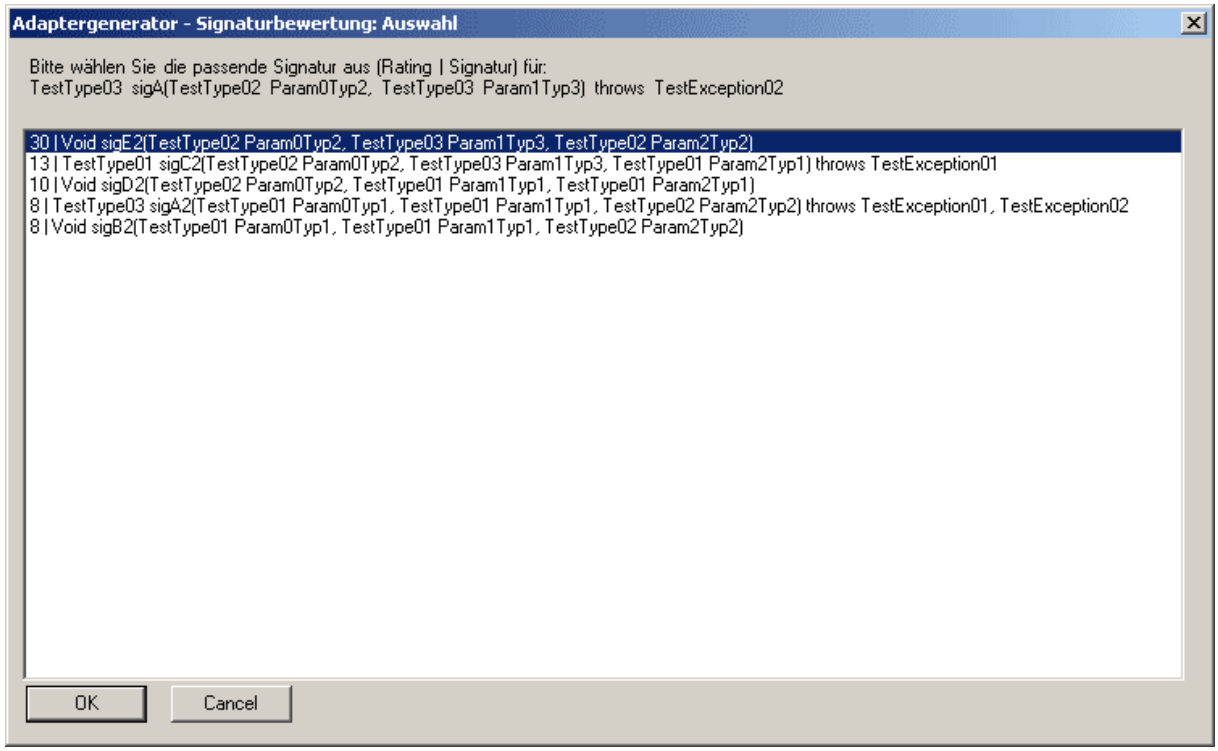


Abbildung 17 Bewertungsdialog der einzelnen Signaturen

11. Literaturverzeichnis

Becker04: Steffen Becker, Sven Overhage, Ralf H. Reussner, *Classifying Software Component Interoperability Errors to Support Component Adaption*, 2004, Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Band 3054, Springer Verlag

Beugnard: A. Beugnard, J. M. Jézéqeul, N. Plouzeau, D. Watkins, *Making components contract aware*, 1999

CM04: Steffen Becker, *The Palladio Component Model*, DFG Palladio, Oldenburg, 2004, (unveröffentlicht)

Gamma: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Entwurfsmuster*, Addison Wesley Verlag, 2003

MCK04: Hyun Gi Min, Si Won Choi, Soo DOng Kim, *Using Smart Connectores to Resolve Partial Matching Problems in COTS Component Acqusition*, Springer-Verlag Berlin Heidelberg, 2004

Proposal: Klaus Krogmann, Proposal - Generierung von Adaptoren, 2004

Reussner04: Ralf H. Reussner, Steffen Becker, *A Classification of Interoperability Problems and its use for Component Adaptors*, DFG Palladio, Oldenburg, 2004, (unveröffentlicht)

RFB: Ralf H. Reussner, Viktoria Firus, Steffen Becker, *Parametric Performance Contracts for Software Components and their Compositionality*, 2004, Proceedings of the 9. International Workshop on Component-Oriented Programming (WCOP)

TechRew7-04: Technologie Review, Thomas Vasek, *Rechner auf Rädern*, Heise Zeitschriften Verlag GmbH & Co. KG, 7/2004

12. Abbildungsverzeichnis

13. Stichwortverzeichnis

14. Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche erkenntlich gemacht worden. Diese Arbeit wurde weder in dieser noch einer ähnlichen Form einer anderen Prüfungsbehörde vorgelegt oder bisher veröffentlicht.

Harkebrügge, 04.09.04

_____

Klaus Krogmann