Ein modellbasierter Ansatz für adaptierbare und selbstadaptive Komponenten

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Steffen Göbel
geboren am 15. Mai 1976 in Görlitz

Gutachter:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill	TU Dresden
	Prof. Dr. rer. nat. habil. Uwe Aßmann	TU Dresden
	Prof. Dr. rer. nat. habil. Heinrich Hußmann	LMU München

Tag der Verteidigung: 2. Oktober 2006

Dresden im Oktober 2006

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als Doktorand bei SAP Research CEC Dresden sowie als Mitarbeiter am Lehrstuhl Rechnernetze der Technischen Universität Dresden. An dieser Stelle möchte ich mich bei allen herzlich bedanken, die zum Gelingen dieser Arbeit beigetragen haben.

An erster Stelle geht mein Dank an Prof. Dr. Alexander Schill für die gute Zusammenarbeit, die angenehme Arbeitsatmosphäre am Lehrstuhl Rechnernetze und für die zahlreichen Diskussionen und Hinweise während der Erstellung des Manuskripts. Prof. Dr. Aßmann und Prof. Dr. Hußmann danke ich ebenfalls für wertvolle Ratschläge und für die Übernahme der Begutachtung dieser Arbeit.

Allen derzeitigen und ehemaligen Mitarbeitern des Lehrstuhls Rechnernetze und von SAP Research CEC Dresden möchte ich für die Unterstützung bei meiner Arbeit danken. Insbesondere danke ich Kay Kadner und Dr. Thomas Springer für die vielen inhaltlichen Diskussionen und für das Korrekturlesen des Manuskripts.

Abschließend möchte ich mich bei meiner Partnerin Stefanie Meißner bedanken, die mich stets motiviert hat und vor allem für Ihr Verständnis und Unterstützung in der „heißen Phase“ der Dissertation.

Dresden, Oktober 2006 Steffen Göbel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die komponentenbasierte Softwareentwicklung hat sich in den letzten Jahren immer mehr als Verfahren der Softwareentwicklung etabliert. Die Verfügbarkeit einer Reihe von Komponentenplattformen für unterschiedliche Programmiersprachen hat wesentlich zu diesem Trend beigetragen. JavaBeans und J2EE sind dabei im Wesentlichen für die Programmiersprache Java und Microsoft COM für C++ zu erwähnen. Diese Komponentenplattformen decken unterschiedliche Bereiche der Anwendungsentwicklung ab, von der Entwicklung von Benutzerschnittstellen, über verteilte Anwendungen bis zu Server-Anwendungen mit Datenbankanbindung.

Ein wesentliches Ziel der komponentenbasierten Softwareentwicklung ist die verbesserte Wiederverwendbarkeit und die „Industrialisierung“ der Entwicklung von Software. Entsprechende Ideen dazu wurden von McIlroy [McI68] bereits vor fast 40 Jahren formuliert. In diesem Zusammenhang wird oft der Vergleich mit Mikrochips bzw. standardisierten Bauteilen im Maschinenbau angeführt (z. B. [Cox90]), um zu verdeutlichen, dass Softwarekomponenten („*Software-ICs*“) im Idealfall genauso einfach wiederverwendet werden sollen. Um dies zu erreichen, werden Softwarekomponenten durch explizit definierte Schnittstellen von der Umgebung abgegrenzt. Diese Schnittstellen werden später zur Komposition von Komponenten zu kompletten Anwendungen verwendet. Diese Eigenschaften fasst Clemens Szyperski in seiner bekannten Definition von Softwarekomponenten zusammen:

> *„A Software Component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“ [Szy98]*

Zur Verbesserung der Wiederverwendbarkeit müssen Softwarekomponenten möglichst *flexibel* komponiert werden können, sowohl innerhalb einer bestimmten Anwendung als auch in unterschiedlichen Anwendungen. Beide Prozesse werden in der vorliegenden Arbeit als *Adaption* von Komponenten bezeichnet. In einer Anwendung bezieht sich die Flexibilität auf die Fähigkeit der Komponenten zur Anpassung an geänderte Umgebungsbedingungen bzw. andere Anwendungseinstellungen. Sie wird durch Parametrisierung von Komponenten oder Änderung der Anwendungsstruktur erreicht. Die Anpassung kann dabei sowohl von außerhalb als auch durch die Komponente selbst gesteuert werden, wobei die eigenständige Adaptation als Selbstadaption bezeichnet wird [OGT+99]. Bei Wiederverwendung in unterschiedlichen Anwendungen müssen sich Komponenten zusätzlich zu den Adaptionsaspekten innerhalb einer Anwendung auch mit anderen Komponenten komponieren lassen. Dazu sind teilweise Anpassungen an den Schnittstellen oder am Verhalten notwendig, um Inkompatibilitäten zu beseitigen.

Neben der komponentenbasierten Softwareentwicklung wurden in den letzten Jahren Methoden der generativen [CE00] und der aspektorientierten Softwareentwicklung [KLM+97] sowie die *Model-driven Architecture* (MDA [Fra03, MB04]) entwickelt. Obwohl diese Methoden prinzipiell unabhängig von der komponentenbasierten Softwareentwicklung angewendet werden können, ist eine Vereinigung der verschiedenen Ansätze Erfolg versprechend. Damit kann eine weitere Trennung der Verantwortlichkeiten (*Separation of Concerns* [Dij76]) insbesondere bei der Entwicklung von Komponenten mit den gewünschten Flexibilitätsanforderungen erreicht werden.

1.1 Motivation und Problemstellung

Die Problemstellung dieser Arbeit kann mit drei Fragen zusammengefasst werden: Wie können adaptierbare Komponenten entwickelt werden? Wie kann die Entwicklung durch Modellierungstechniken und geeignete Werkzeugunterstützung vereinfacht und der Entwicklungsaufwand reduziert werden? Wie können die Adaptionskonzepte für Komponenten in bestehende Anwendungen integriert werden?

Viele heute bekannte Methoden und Techniken, die im weiteren Sinne zur Adaption von Komponenten und Anwendungen eingesetzt werden können, unterstützen nur bestimmte Phasen der Anwendungsentwicklung im Bereich von Analyse über Entwurf, Entwicklung und Installation bis zur Laufzeit. Software-Produktlinien und die generative Softwareentwicklung (siehe Abschnitt 2.6) konzentrieren sich auf den Zeitraum von Analyse bis maximal zur Installation, unterstützen jedoch keine Adaption zur Laufzeit. Mechanismen zur Modifikation von Quelltexten oder Bytecode arbeiten in der Regel nur zur Installationszeit oder beim Laden von Komponenten. Am anderen Ende des Spektrum unterstützen Techniken wie Adapter [GHJV95] und Reflection bzw. *Metaobject Protocols* [KRB91] die Adaption ausschließlich zur Laufzeit. Es fehlt ein Verfahren zur Adaption, das auf ähnliche Weise zu allen genannten Zeiten eingesetzt werden kann und das die unterschiedlichen entwickelten Adaptionsmechanismen integriert.

Komponentenparameter[1] werden schon lange als ein gängiges Verfahren zum Einstellen von Eigenschaften und damit als eine einfache Form der Adaption verwendet. Zur effizienteren Entwicklung von Komponenten fehlt jedoch ein Verfahren, mit dem die Auswirkungen von Parameteränderungen auf Abläufe innerhalb der Komponente explizit modelliert und insbesondere strukturelle Änderungen beschrieben werden können. Der entsprechende Programmcode zur Auswertung von Parametern wird heute oft manuell zusammen mit der sonstigen Programmlogik entwickelt. Die Komponentenadaption muss stattdessen als ein weiterer Aspekt bzw. eine weitere Sicht während der Komponentenentwicklung aufgefasst und durch Werkzeuge unterstützt werden.

Anstelle von Komponentenparametern werden auch verschiedene Formen von Konfigurationsdateien eingesetzt, um Einstellungen bereits bei der Installation von Komponenten vorzunehmen (z. B. Deskriptoren bei EJB-Komponenten). Auch in diesen Fällen werden die Konfigurationsdateien durch speziellen Programmcode ausgewertet und die Auswirkungen der Konfigurationsdateien auf Abläufe und die Struktur innerhalb von Komponenten wird nicht explizit modelliert.

[1] Komponentenparameter werden in der Literatur oft auch als *Properties* bzw. Attribute bezeichnet.

Einige verwandte Arbeiten (z. B. OpenORB Unterabschnitt 2.5.4) versuchen die mangelhafte Unterstützung für Adaption durch die Entwicklung neuer Komponentenplattformen zu lösen. Dabei tritt jedoch das Problem auf, dass damit die große Menge von existierenden komponentenbasierten Anwendungen nicht berücksichtigt werden kann. Softwareentwicklung besteht in der Praxis jedoch zum größten Teil aus der Weiterentwicklung von existierenden Produkten. Nur in wenigen Fällen wird wegen einer neuen Technologie eine komplette Neuentwicklung durchgeführt. Der Nutzen muss dabei den enormen Aufwand rechtfertigen. In den meisten Fällen wird eine Migrationsstrategie benötigt, um neue Technologien einzuführen und gleichzeitig die bisherigen Entwicklungen weiterverwenden zu können.

1.2 Ziele der Arbeit

In der vorliegenden Arbeit soll ein Verfahren zur Modellierung und Umsetzung von Adaptionsaspekten für Komponenten entwickelt werden. Das Gesamtkonzept wird als *Adaptierbare Komponenten* bezeichnet. Zur Lösung der identifizierten Probleme im heutigen Stand der Technik werden die folgenden drei Kernziele definiert:

1. *Entwicklung von plattformunabhängigen Adaptionskonzepten für Komponenten:*
   Die Konzepte sollen unabhängig von einer bestimmten Komponentenplattform und Programmiersprache definiert werden, damit der mögliche Einsatzbereich nicht von Anfang an eingeschränkt wird. Die Adaption soll durch Komponentenparameter erreicht werden, die damit unterschiedliche interne Konfigurationen der Adaptierbaren Komponenten festlegen. Grundlage dafür ist ein hierarchisches Komponentenmodell, wobei eine Adaptierbare Komponente eine Menge von Subkomponenten sowie weitere Bestandteile wie z. B. Adaptionsoperatoren enthält. Die Adaption der Komponenten soll zu verschiedenen Zeitpunkten des Lebenszyklus – bei der Installation, beim Erzeugen von Komponenteninstanzen und während der Laufzeit – mit den gleichen Mechanismen unterstützt werden. Im Sinne der aspektorientierten Programmierung soll der Aspekt der Adaption bei der Entwicklung der Komponenten separiert und gekapselt werden.

   Bei der Entwicklung der Konzepte Adaptierbarer Komponenten sollen spätere Erweiterungen oder Änderungen berücksichtigt werden. Voraussetzung dafür ist ein modularer Aufbau, um einzelne Konzepte bzw. Bestandteile relativ unabhängig voneinander wegzulassen, auszutauschen oder weiterentwickeln zu können. Damit soll auch erreicht werden, dass existierende und zukünftige Adaptionsmechanismen für Komponenten integriert werden können.

2. *Entwicklung von Modellierungstechniken für Adaptierbare Komponenten:*
   Adaptierbare Komponenten sollen so weit wie möglich auf Basis einer UML-basierten graphischen Notation modelliert werden können. Auf diese Weise können zum einen existierende UML-Modellierungswerkzeuge direkt oder mit einigen Erweiterungen wiederverwendet werden und zum anderen hat sich UML in den letzten Jahren zum Defacto-Standard für die Softwaremodellierung etabliert. Die Ergebnisse der Modellierung sollen in einem MOF-basierten Metamodell gespeichert werden, um damit den Austausch von Modellen in Form von XMI (*XML Metadata Interchange*) zu unterstützen und die Anbindung von Werkzeugen zu erleichtern. Die Adaptionslogik soll unter Nut-

zung der Modellierungsergebnisse für die Komponenten automatisch erzeugt werden, so dass Entwickler von dieser Aufgabe entlastet werden.

3. *Modelltransformation und Plattformunterstützung für Adaptierbare Komponenten:* Damit Adaptierbare Komponenten in bestehende oder neue komponentenbasierte Anwendungen integriert werden können, müssen die plattformunabhängigen Konzepte und Modelle auf die jeweilige Komponentenplattform transformiert werden. Zu diesem Zweck soll zunächst ein allgemeines Verfahren für die Definition und Implementierung einer Plattformunterstützung entwickelt werden. Ein plattformunabhängiges Framework mit Funktionen für die wesentlichen Konzepte Adaptierbarer Komponenten soll die Entwicklung für neue Komponentenplattformen vereinfachen und beschleunigen. Unter Nutzung des Verfahrens und des Frameworks sollen beispielhaft Unterstützungen für einige weit verbreitete Komponentenplattformen implementiert werden.

1.3 Abgrenzung

Eine Reihe von Problemstellungen in Zusammenhang mit dem Thema der vorliegenden Arbeit werden nicht weiter betrachtet und sind damit mögliche Anknüpfungspunkte für weiterführende Arbeiten (siehe Abschnitt 6.2).

- Eine Verhaltensspezifikation von Komponenten wird nicht berücksichtigt und unterstützt. Die Funktionsfähigkeit einer Zusammenarbeit von Komponenten und die Korrektheit von Rekonfigurationen kann nicht bewiesen werden.
- Es werden keine neuen formalen Mechanismen zur Beschreibung von Rekonfiguration entwickelt.
- Es werden keine neuen Adaptionsmechanismen für Komponenten auf Basis von Quelltext- oder Bytecode-Änderungen entwickelt. Ziel ist vielmehr die Integration vorhandener Mechanismen innerhalb des Konzepts von Adaptierbaren Komponenten.
- Modelltransformationen werden nicht formal beschrieben und es werden auch keine neuen formalen Verfahren dafür entwickelt.

1.4 Gliederung der Arbeit

Im Folgenden wird kurz der Aufbau dieser Arbeit beschrieben:

Kapitel 2 stellt wesentliche verwandte Arbeiten vor und grenzt sie gegen die eigene Arbeit ab. In diesem Zusammenhang wird der aktuelle Stand der Technik zu Komponentenplattformen und Adaptionsmechanismen für Komponenten in einem kurzen Überblick vorgestellt.

Kapitel 3 beschreibt die Konzepte und Modelle für Adaptierbare Komponenten. Dazu gehören insbesondere visuelle Modellierungsverfahren für Konfigurationen und die Definition der Abbildung von Komponentenparameterwerten auf Konfigurationen. Außerdem werden verschiedene Detailprobleme diskutiert, die bei der Rekonfiguration zur Laufzeit beachtet werden müssen.

Kapitel 4 beschreibt die notwendigen Schritte vom Metamodell bis zur Laufzeitunterstützung für Adaptierbare Komponenten, jedoch immer noch unabhängig von einer bestimmten Komponentenplattform. Dazu wird zunächst gezeigt, wie verschiedene Eigenschaften für Komponentenplattformen ermittelt werden können, die für die Unterstützung von Adaptierbaren Komponenten benötigt werden. Diese Eigenschaften werden anschließend systematisch zur Konstruktion einer Laufzeitunterstützung für Adaptierbare Komponenten verwendet. Zu diesem Zweck wird auch ein Framework beschrieben, das wichtige Funktionen für die Implementierung bereitstellt.

Kapitel 5 untermauert die praktische Einsetzbarkeit der in Kapitel 3 und 4 entwickelten Verfahren. Dazu wird im ersten Teil die Unterstützung der drei Komponentenplattformen JavaBeans, EJB und Microsoft COM sowie als Spezialfall die Unterstützung von Web-Services detailliert beschrieben, um die Modelltransformation auf reale Komponentenplattformen zu demonstrieren. Durch die Implementierung einer Kryptographiekomponente wird gezeigt, dass sich Adaptierbare Komponenten ohne Änderungen der Komponentenplattformen realisieren lassen. Im zweiten Teil werden die Modellierungsmöglichkeiten von Adaptierbaren Komponenten anhand von zwei Fallstudien näher untersucht. Damit wird gezeigt, dass mit Adaptierbaren Komponenten konkrete Probleme effizienter gelöst werden können als mit bisher verfügbaren Techniken.

Kapitel 6 fasst die wesentlichen Ergebnisse dieser Arbeit noch einmal im Überblick zusammen. Anschließend werden noch offene Forschungsfragen und damit Anknüpfungspunkte für zukünftige Arbeiten diskutiert.

Kapitel 2

Verwandte Arbeiten

Die Thematik der vorliegenden Arbeit enthält Bezüge zu verschiedenen Forschungsbereichen der Softwaretechnik und verteilten Systemen. In diesem Kapitel werden verschiedene Arbeiten aus diesen Bereichen kurz vorgestellt und Gemeinsamkeiten und Unterschieden zu dieser Arbeit hergestellt.

Tabelle 2.1 gibt einen Überblick über die aus verschiedenen Forschungsbereichen übernommenen Konzepte für Adaptierbare Komponenten. Darauf aufbauend wurde in dieser Arbeit eine Reihe von neuen Konzepten entwickelt:

- Komponentenparameter werden auf unterschiedliche interne Konfigurationen von Adaptierbaren Komponenten abgebildet. Die Parameter und die Konfigurationen können unabhängig voneinander definiert werden.
- Die Komponentenadaption kann zu unterschiedlichen Zeiten (Entwicklung, Installation, Instanziierung, Laufzeit) durchgeführt werden, ohne dass die Adaptionsbeschreibung angepasst werden muss.
- Die Beschreibung von Adaptierbaren Komponenten erfolgt unabhängig von bestimmten Programmiersprachen und Komponentenplattformen (*Overlay* Komponentenmodell) und wird durch Modelltransformationen auf existierende Komponentenplattformen abgebildet.
- Eigenschaften von Komponentenplattformen werden systematisch ermittelt, um ein allgemeines Verfahren zur Definition von Modelltransformationen zu definieren.

2.1 Architekturbeschreibungssprachen (ADL)

Architekturbeschreibungssprachen[1] werden zur Beschreibung von Software- und Systemarchitekturen [SG96, HNS00] verwendet, wobei das Abstraktionsniveau in der Regel über dem von Programmiersprachen liegt. Die Fähigkeit zur expliziten Modellierung von Konfigurationen unterscheidet ADLs von anderen Entwurfsansätzen wie Programmiersprachen, objektorientierter Modellierung und formalen Spezifikationssprachen. Seit mehr als 20 Jahren wird dieses Thema im Rahmen von zahlreichen Forschungsprojekten in teils unterschiedlichen Richtungen weiterentwickelt [MT00].

Komponenten, Ports und Verbindungen bzw. Konnektoren finden sich als Grundkonzepte bei fast allen ADLs. Diese Konzepte werden auch zur Modellierung der internen Konfigurationen von Adaptierbaren Komponenten verwendet (siehe Abschnitt 3.4). Komponenten

[1] *Architecture Description Language* (ADL)

Forschungsgebiet	Übernommene Konzepte
ADLs	explizite Modellierung der Architektur mit Komponenten, Ports und Verbindungen; hierarchische Dekomposition
Beschreibung von Rekonfigurationen	Modellierung von Konfigurationen und Variationen
AOP	Joinpoints für Aspektoperatoren
Self-Healing-Systems	Idee für Selbstadaptivität, Kontextmodelle und Steuerung
Produktlinien	Modellierung von Änderungen zu einer Basiskonfiguration, Variationsmechanismen durch Parameter, Konfigurationen und Erweiterungen
MDA	Plattformunabhängige und Plattformspezifische Modelle, Modelltransformation; Metamodelle auf Basis von MOF
Adaptionsmechanismen	Adaptionsoperatoren, Implementierung von Interceptoren, Aspekt- und Adaptionsoperatoren

Tabelle 2.1: Übernommene Konzepte für Adaptierbare Komponenten

mit einer Menge von Ports modellieren Orte der Informationsverarbeitung und Verbindungen modellieren Kommunikationsbeziehungen zwischen bestimmten Komponentenports zusammen mit ihren Eigenschaften. Eine Architekturkonfiguration ist ein Verbindungsgraph zwischen Komponenten. Komponenten können oft auch aus anderen Komponenten zusammengesetzt werden (Subarchitektur) und erlauben damit eine hierarchische Komposition von Systemen mit jeweils unterschiedlichen Detaillierungsgraden. Die verschiedenen ADLs unterscheiden sich hauptsächlich darin, welche Systemeigenschaften wie z. B. Echtzeiteigenschaften oder Verhaltensbeschreibungen damit modelliert werden können.

ACME [GMW97] versucht die Konzepte einer Reihe von existierenden ADLs (Aesop, Adage, C2, MetaH, Rapide, SADL, UniCon und Wright) zu integrieren, um ein einheitliches Austausch- und Repräsentationsformat zu realisieren. xADL [DvdHT02a] verfolgt einen ähnlichen Ansatz und beschreibt eine Infrastruktur zur Entwicklung von neuen ADLs auf Basis von XML. Seit Version 2.0 können auch mit UML (*Unified Modeling Language* [uml05b]) die wesentlichen Konzepte von ADLs (Komponenten, Verbindungen, *required*- und *provided*-Ports, hierarchische Komposition) graphisch modelliert werden. UML-Komponentendiagramme werden als Basis für die Modellierung von Konfigurationen und Variationen Adaptierbarer Komponenten verwendet.

Die meisten ADLs erlauben nur die statische Modellierung von Software-Architekturen. Lediglich C2 [MORT96], Darwin [MDEK95], Koala [OLKM00], Rapide [Luc96], and Weaves [GR91] berücksichtigen und beschreiben dynamische Architekturänderungen.

Darwin, Koala und Rapide unterstützten nur Architekturänderungen zur Laufzeit, die bereits zur Entwicklung vordefiniert wurden. C2 und Weaves unterstützen beliebige Rekonfigurationen zur Laufzeit, wobei die Konsistenz durch die Laufzeitumgebung sichergestellt wird. C2 enthält dazu eine Architktur-Modifikationssprache, die eine Menge von Operationen

zum Hinzufügen, Entfernen und Verbinden von Komponenten enthält. Weaves realisiert die Rekonfiguration auf ähnliche Weise durch die Bereitstellung einer Programmierschnittstelle.

Adaptierbare Komponenten unterstützen sowohl vordefinierte Rekonfigurationen mit der Beschreibung von expliziten Konfigurationen und Variationen wie bei C2 und Weaves als auch (indirekt) beliebige Änderungen durch Manipulation der Konfigurationen und Variationen mittels einer Programmierschnittstelle (Unterabschnitt 4.3.9).

Koala erlaubt die Modellierung von strukturellen Änderungen mit Hilfe von Schaltern (*Switches*), indem eine Verbindung einer *required*-Schnittstelle zwischen zwei verschiedenen *provided*-Schnittstellen umgeschaltet werden kann. Konfigurationen können sowohl zur Übersetzungszeit als auch zur Laufzeit festgelegt werden. Koala-Komponenten sind lediglich Abstraktionen für C-Module, d. h. die Konzepte sind eng an die Programmiersprache C gekoppelt. Sie können mit einer graphischen Notation zur Konfigurationsbeschreibung ähnlich wie ICs modelliert werden.

Schnittstellen können miteinander verbunden werden, wenn die *provided*-Schnittstelle mindestens alle Funktionen der *required*-Schnittstelle unterstützt (*Interface compatibility*). Wenn sich lediglich die Methodennamen in zwei Schnittstellen unterscheiden, können Methoden paarweise zugeordnet werden (*Function Binding*). Spezieller Glue-Code kann zur Verknüpfung von Komponenten eingesetzt werden. Diese Konzepte werden auf ähnliche Weise auch von Verbindungen in Adaptierbaren Komponenten (siehe Unterabschnitt 3.3.3) unterstützt, ohne allerdings eine Kopplung an die Programmiersprache C vorauszusetzen.

Diversity Interfaces sind spezielle *required*-Schnittstellen von Komponenten, die Zugriffe auf Parameterwerte kapseln. Im Sinne einer Produktlinie (Abschnitt 2.6) repräsentieren sie Variationspunkte einer Komponente. Wenn bestimmte Parameterwerte bereits zur Übersetzungszeit festgelegt werden, kann der Compiler verschiedene Optimierungen durchführen und z. B. nicht verwendete Komponenten weglassen. Parameter bei Adaptierbaren Komponenten können in ähnlicher Weise angewendet werden. Konfigurationen und Parameter sind jedoch entkoppelt, werden nur durch die Parameterabbildung miteinander verknüpft und können damit in einem gewissen Umfang unabhängig voneinander entwickelt und geändert werden.

Java/A [BHH+05] integriert ein abstraktes Komponentenmodell für die architekturbasierte Programmierung (*Architectural Programming*) in die Programmiersprache Java. Damit werden Konzepte von Architekturbeschreibungssprachen direkt in die Programmiersprache integriert um den Bruch zwischen den beiden Beschreibungen zu überwinden. Das Verhalten von Komponenten wird formal definiert und kann damit auch verifiziert werden. Java/A-Programme werden durch einen Compiler in Java-Klassen transformiert, wobei gleichzeitig die Kompatibilität von Verhaltensbeschreibungen zusammenarbeitender Komponenten überprüft wird. ArchJava [Ald03, JAN05] verfolgt einen ähnlichen Ansatz, beschränkt sich aber auf Strukturbeschreibungen und unterstützt keine Verhaltensbeschreibungen. Beide Ansätze sind im Gegensatz zu Adaptierbaren Komponenten an die jeweilige spezielle Programmiersprache gebunden.

2.2 Beschreibung von Rekonfigurationen

In diesem Abschnitt werden verschiedene Verfahren zur Beschreibung von Rekonfigurationen vorgestellt. Dabei gibt es auch Querbezüge zum vorhergehenden Abschnitt, da einige Beschreibungsmöglichkeiten in ADLs integriert wurden.

2.2.1 Operationale Beschreibung

Gerel (*Generic Reconfiguration Language* [End94]) ist eine einfache imperative Programmiersprache für Rekonfigurationen, die sowohl zur Entwicklung von generischen Änderungsskripten als auch für Ad-hoc Rekonfigurationen verwendet werden kann. Die Sprache enthält Kommandos zum Hinzufügen und Entfernen von Komponenten bzw. Verbindungen, Kommandos zur Steuerung des Kontrollflusses sowie eine Abfragesprache (Gerel-SL) zur Beschreibung von Bedingungen für Kontrollfluss-Kommandos und von Vorbedingungen (*preconditions*) für Rekonfigurationen.

Architektur-Modifikationssprachen (AML) beschreiben Änderungen einer Architekturbeschreibung (ADL) durch Kommandos wie bei Gerel. Vertreter solcher Sprachen sind AML für C2 [Med96] und Clipper [AHP94]. Die ADLs Concic und Darin (Abschnitt 2.1) enthalten auch einfache Beschreibungssprachen für Rekonfigurationen, die aber weniger Ausdrucksmöglichkeiten als Gerel unterstützen und nicht generisch für unterschiedliche Konfigurationen verwendet werden können.

Die Beschreibung von Variationen (Unterabschnitt 3.4.4) als Teil der Konfigurationsbeschreibung von Adaptierbaren Komponenten kann auch als graphische Beschreibungssprache aufgefasst werden und integriert die Kommandos zur Rekonfiguration der vorgestellten operationalen Beschreibungssprachen. Neu hinzugekommen ist die UML-basierte Notation zur Modellierung.

Inverardi und Wolf [IW95] verwenden die *Chemical Abstract Machine* (CHAM) zur formalen Spezifikation und Analyse von Software-Architekturen. Dabei wird eine Architektur als abstrakte Maschine modelliert, die Analogien mit der Chemie und chemischen Reaktionen herstellt. *Moleküle* sind die Grundbausteine und daraus bestehende *Lösungen* (Multimengen von Molekülen) definieren Zustände einer CHAM. Durch Transformationen, die chemische Reaktionen modellieren, werden Lösungen in andere Lösungen umgewandelt. *Membranen* kapseln eine Menge von Lösungen und können damit hierarchische Systeme modellieren. CHAM übernimmt und vereinigt damit Konzepte von Zustandsautomaten und Graphgrammatiken und kann zur Modellierung und Analyse hochgradig paralleler System verwendet werden. [Wer98] nutzt CHAM zur Beschreibung von Architektur-Stilen und zur Steuerung von Rekonfigurationen.

2.2.2 Graph-Transformationen und Graph-Grammatiken

Architekturen bestehend aus Komponenten und Verbindungen (siehe Abschnitt 2.1) können formal als gerichtete Graphen beschrieben werden. Meist werden Komponenten als Knoten und Verbindungen als Kanten des Graphen modelliert. Lediglich in [HIM99, Hir03] werden Komponenten mit ihren Ports als Hyper-Kanten und Verbindungen als Knoten in einem Hyper-Graphen beschrieben.

Graph-Grammatiken eignen sich insbesondere zur formalen Beschreibung von *Architecture Styles* [Mét96, Mét98], welche eine Menge von Architekturen mit ähnlichen Eigenschaften und Kommunikationsstrukturen (z. B. *Pipeline* oder Client-Server) definieren. Durch eine Menge von Graph-Ersetzungsregeln (*Productions*) teilweise auch mit Nebenbedingungen, die jeweils einen Teilgraphen durch einen anderen Teilgraphen ersetzen, können alle möglichen Architekturgraphen iterativ bestimmt werden. Im Rahmen von Adaptierbaren Komponenten werden

keine Architektur-Stile betrachtet und daher wurden Graph-Grammatiken nicht weiter berücksichtigt.

2.2.3 Prozess-Algebren

Prozess-Algebren (auch als Prozess-Calculus bezeichnet) sind formale Ansätze zur Beschreibung von parallelen Systemen. Damit können Kommunikationsbeziehungen und Synchronisationen zwischen unabhängigen Prozessen[2] beschrieben werden. Durch die Verwendung einer Algebra können Eigenschaften von Systemen formal bewiesen werden. Prozess-Algebren werden aus diesen Gründen auch zur formalen Beschreibung einiger ADLs eingesetzt.

Die ersten Vertreter von Prozess-Algebren waren der *Calculus of Communicating Systems* (CCS [Mil80]) und *Communicating Sequential Processes* (CSP [Hoa78]). Die ADL Wright [All97, ADG98] verwendet CSP als formale Basis zur Beschreibung des Verhaltens von Komponenten und Konnektoren, um die Konsistenz von Systemen zu überprüfen. Die Semantik von Conic wurde mit CCS und CSP formalisiert [MKS89]. Damit konnte aber im Wesentlichen nur das Verhalten von statischen Systemen modelliert und analysiert werden, nicht jedoch Rekonfigurationen.

Der π-Calculus [Mil89] ist eine Weiterentwicklung von CCS, um auch Mobilität in verteilten und parallelen Systemen formal zu modellieren. Das operationale Modell der ADL Darwin wurde mit dem π-Calculus beschrieben [MK96] und kann als Nachfolger von Conic auch Rekonfigurationen formal beschreiben. Die Kompositionssprache Piccola [LSNA97] basiert ebenfalls auf dem π-Calculus.

Prozess-Algebren benötigen Verhaltensbeschreibungen von Komponenten, zumindest hinsichtlich der Kommunikationen. Da dieses Thema in der Arbeit nicht berücksichtigt wurde (siehe Abschnitt 1.3), kamen Prozess-Algebren nicht zur Beschreibung von Adaptierbaren Komponenten in Betracht.

2.2.4 Logik-basierte Beschreibung

Aguirre und Maibaum [AM03a] definieren eine temporale Logik basierend auf einer Manna-Pnueli-Logik [MP92], um Verhaltenseigenschaften von dynamisch rekonfigurierbaren Systemen formal zu beschreiben. Komponenten werden durch Assoziationen miteinander verbunden und mehrere Komponenten können Subsysteme bilden. Bestimmte Operationen auf diesen Subsystemen können Rekonfigurationen der Subkomponenten auslösen, die durch temporale Logik formal beschrieben werden. Im Unterschied zu Graph-Grammatiken oder CHAM werden Rekonfiguration deklarativ beschrieben und bieten damit erweiterte Möglichkeiten zur Charakterisierung von Systemen und dem Beweisen von Eigenschaften.

Logik-basierte Beschreibungen benötigen Verhaltensbeschreibungen für Komponenten. Da dieses Thema in der Arbeit nicht berücksichtigt wurde (siehe Abschnitt 1.3), kamen Logik-basierte Beschreibungen nicht zur Beschreibung von Adaptierbaren Komponenten in Betracht.

[2] Komponenten können in diesem Zusammenhang auch als Prozesse aufgefasst werden.

2.3 Self-Managed Systeme

Self-Managed bzw. *Self-Adaptive* Systeme[3] reagieren eigenständig auf Änderungen von Umgebungsbedingungen und passen ihre Struktur bzw. ihr Verhalten entsprechend an. Der Teilbereich der *Self-Healing* Systeme soll eigenständig Fehler und Probleme erkennen und sie anschließend selbst wieder beseitigen. Viele Analogien stammen dabei aus der Biologie, wo Organismen bestimmte Verletzungen auch selbstständig heilen können.

In [OGT+99, Wil04] werden typische Bestandteile von Self-Managed Systemen vorgestellt und verschiedene Implementierungsmöglichkeiten klassifiziert. Sensoren oder Monitore sind für die Erfassung von Umgebungsinformationen verantwortlich. Diese Informationen müssen von einer Systemkomponente ausgewertet werden, um Änderungen zu planen. Anschließend können diese Änderungen von einer weiteren Systemkomponente ausgeführt werden. In [BCDW04] werden verschiedene Verfahren zur formalen Beschreibung von Self-Managed Systemen untersucht, die größtenteils auch schon in Abschnitt 2.2 vorgestellt wurden. Außerdem wurden die folgenden Schritte für Self-Managed Systeme identifiziert:

1. Initiierung der Architekturänderung z. B. durch den Systemzustand, vordefinierte Ereignisse, Änderungen von Umgebungsbedingungen
2. Auswahl der Architektur-Transformation (vordefinierte Auswahl, eingegrenzte Auswahl aus einer vordefinierten Menge von Konfigurationen oder beliebige Auswahl)
3. Ausführung der Rekonfiguration
4. Bewertung der Architektur nach der Rekonfiguration

Verschiedene Arbeiten [OGT+99, DvdHT02b, GMK02, SG02] realisieren Self-Managed Systeme durch Änderungen bei dynamischen Systemarchitekturen. Sie basieren dabei im Wesentlichen auf Vorarbeiten und Erweiterungen von Forschungen zu ADLs (siehe Abschnitt 2.1).

Die Realisierung der Selbstadaptivität bei Adaptierbaren Komponenten (Unterabschnitt 3.4.7) wird ebenfalls durch Architekturänderungen realisiert. Allerdings werden diese Änderungen nur indirekt durch Parameteränderungen ausgeführt.

2.4 Aspektorientierte Programmierung

Motivation für die Entwicklung der aspektorientierten Programmierung (AOP [KLM+97]) war eine weitere Verbesserung des *Separation of Concerns*. Man hatte festgestellt, dass bestimmte Funktionen von Software wie z. B. Logging, Fehlerbehandlung, Synchronisierung oder Autorisierung nicht in einer Methode gebündelt werden können. Stattdessen sind diese so genannten *cross-cutting concerns* über viele Methoden bzw. Klassen verteilt (*scattered code*) und mit anderem Programmcode vermischt (*tangled code*). AOP versucht diese Vermischung zu entflechten (*untangle*) und den eigentlich zusammengehörenden Programmcode in Form von *Aspekten* wieder zusammenzufassen. Kiczales et. al. [KLM+97] definieren einen Aspekt als eine klar modularisierte Implementierung eines *cross-cutting concerns*. Diese Aspekte werden später während der Übersetzung oder auch erst zur Laufzeit wieder an definierten Stellen

[3] Die beiden Bezeichnung werden gleichberechtigt in der Literatur verwendet.

in das Programm eingefügt. Mögliche Stellen für das Einfügen von Aspekten sind die so genannten *Joinpoints*. Mit Hilfe von *Pointcuts*, die über verschiedene Verfahren eine Menge von Joinpoints auswählen, werden Aspekte an bestimmte Joinpoints gebunden. Der Vorgang des Einfügens von Aspekten wird als *Weaving* bezeichnet.

Verschiedene Ansätze von AOP unterscheiden sich hauptsächlich darin, welche Joinpoints unterstützt werden, wie Pointcuts definiert werden und auf welche Weise bzw. wann das Weaving durchgeführt wird. Im Rahmen der aspektorientierten Softwareentwicklung (AOSD) werden die Konzepte von AOP nicht nur in der Implementierungsphase, sondern in allen Phasen (Anforderungsanalyse, Entwurf) der Softwareentwicklung angewendet.

In dieser Arbeit werden keine neuen Techniken zur AOP entwickelt. Aspektoperatoren (siehe Unterabschnitt 3.3.5) integrieren jedoch einige Kernkonzepte wie Joinpoints in Adaptierbare Komponenten.

AspectJ AspectJ [KHH+01] ist eine Erweiterung von Java um aspektorientierte Konzepte, die seit 2002 im Rahmen eines Eclipse-Projektes weiterentwickelt wird. Mögliche Joinpoints sind z. B. vor und nach Methodenaufrufen, bei Exceptions und beim Zugriff auf Membervariablen von Objekten. Die integrierte Pointcut-Sprache von AspektJ erlaubt sowohl die statische Auswahl von Joinpoints durch Angabe von Namen bzw. regulären Ausdrücken als auch die dynamische Auswahl in Abhängigkeit des Kontrollflusses des Programms. Mehrere Pointcut-Ausdrücke können durch logische Operationen (AND, OR, NOT) miteinander kombiniert werden. Der durch einen Pointcut eingefügte Java-Programmcode wird als *Advice* bezeichnet. Die Konzepte von AspektJ wurden auf andere Programmiersprachen wie z. B. AspectL für Common Lisp, AspectR für Ruby, AspectC++ für C++ portiert.

Hyperspace-Programmierung Hyper/J realisiert ein Verfahren zur *multi-dimensionalen Separation of Concerns* [TOHS99] für Java-Programme, das auch allgemein als *Hyperspace*-Programmierung bezeichnet wird. Ein Java-Programm kann in verschiedene *Concerns* zerlegt werden, die dabei flexibel in separaten Modulen gekapselt werden. Eine Spezifikation legt fest, wie einzelne Java-Klassen, Felder und Methoden bestimmten *Concerns* zugeordnet werden, welche *Concerns* komponiert werden und in welcher Beziehung sie dabei zueinander stehen, um die Komposition durchzuführen. Hyper/J arbeitet auf Basis von Java-Class-Dateien und erzeugt auch wieder Class-Dateien, ohne dafür Java-Quelltext zu benötigen.

Mit einer Menge von Modulen (Hyper-Module), die unterschiedliche *Concerns* kapseln, kann das *Compositor*-Werkzeug von Hyper/J neue Programme bzw. Komponenten erzeugen, die alle oder eine Teilmenge der *Concerns* integrieren. Damit können Erweiterungen, Software-Konfigurationen und Anpassungen realisiert werden, ohne den ursprünglichen Quelltext zu modifizieren. Diese Methode kann sowohl für Neuentwicklungen als auch für existierende Java-Programme, die ursprünglich ohne Hyper/J entwickelt wurden, angewendet werden.

Im Gegensatz zu AspektJ wird nicht zwischen dem „Haupt"-*Concern* in Form von Klassen und Methoden und einer Menge von *cross-cutting concerns* in Form von Aspekten unterschieden, sondern alle *Concerns* werden gleich behandelt und können gekapselt werden. Hyper/J arbeitet dabei auf der Granularität von kompletten Methoden und unterstützt damit keine Zerteilung des Programmcodes innerhalb von Methoden. Wie bei AspectJ arbeitet Hyper/J zur Übersetzungszeit und erlaubt damit im Gegensatz zu Adaptierbaren Komponenten keine Rekonfiguration von Anwendungen zur Laufzeit.

Composition Filters *Composition Filters* (CF [ABV92, BA01]) beschreiben eine modulare und orthogonale Erweiterung von Objekten zur Kapselung von *cross-cutting concerns*. Eine CF-Klasse aggregiert Null oder mehr interne Klassen und komponiert ihr Verhalten durch einen oder mehrere enthaltene Input- und Output-Filter. Das Filter-Konzept ist programmiersprachenunabhängig definiert und erlaubt die Manipulation von gesendeten und empfangenen Nachrichten (Methodenaufrufe) der CF-Klasse, d. h. Joinpoints sind in diesem Fall vor und nach Methodenaufrufen vorgesehen. Filter können auf diese Weise verschiedene *Concerns* wie z. B. Vererbung, Delegierung, Synchronisation und Echtzeit-Anforderungen implementieren. Erste Versionen von CF betrachteten nur *cross-cutting concerns* bei einem Objekt. Später wurden durch die Verwendung des Superimposition-Konzepts [Bos99] auch mehrere Objekte berücksichtigt [BA01]. Implementierungen von CF existieren für verschiedene Programmiersprachen wie z. B. Java (ComposeJ) und .NET (Compose*.NET).

Jede empfangene Nachricht (Methodenaufrufe) wird von allen Input-Filtern und jede gesendete Nachricht von allen Output-Filtern der CF-Klasse bearbeitet. Mit Filter-Ausdrücken, die eine einfache Nachrichten-Manipulationssprache darstellen, können deklarativ bestimmte Nachrichten für einen Filter ausgewählt werden (*matching*), der sie manipulieren und z. B. an eine aggregierte Klasse weiterleiten kann. Dabei können in den Filter-Ausdrücken auch Bedingungen (*Conditions*) in Form von booleschen Ausdrücken verwendet werden. Nicht ausgewählte Nachrichten werden an den nächsten Filter in der Kette weitergereicht. Verschiedene Filtertypen erlauben z. B. die Weiterleitung von Nachrichten (Dispatch-Filter), die Auslösung von Exceptions (Error-Filter), die zeitweise Blockierung von Nachrichten (Wait-Filter), die Reifikation von Nachrichten zur Weiterleitung an andere Objekte (Meta-Filter) und die Manipulation von Realtime-Eigenschaften des Thread (Realtime-Filter). Weitere Filtertypen können bei Bedarf eingeführt werden.

Adaptierbare Komponenten verwenden wie CF ein hierarchisches Komponentenmodell. Aspektoperatoren, insbesondere die Joinpoints *preMethod* und *postMethod*, können ähnliche Aufgaben wie Filter übernehmen. Filter-Ausdrücke und verschiedene Filtertypen werden jedoch nicht unterstützt. Die Integration der CF-Konzepte in Adaptierbare Komponenten wäre denkbar, wurde aber in dieser Arbeit nicht berücksichtigt.

Adaptive Programming *Adaptive Plug-and-Play Components* (APPC [ML98]) realisieren ein Ansatz für generische und komponierbare Komponenten, die die Kollaboration zwischen mehreren Objekten kapseln. Sie sind eine Weiterentwicklung des *Adaptive Programmings* [Lie96]. Das zugrunde liegende Verfahren des Kollaborations- bzw. Rollen-basierten Entwurfs (*Collaboration-based Design*) zerlegt objektorientierte Anwendungen in eine Menge von Klassen und eine Menge von Kollaborationen (*Colloboration-based Decomposition*), d. h. die Aspekte Anwendungsstruktur und Kommunikationsverhalten werden separiert. Jede Klasse kann unterschiedliche Rollen in verschiedenen Kollaborationen mit anderen Klassen übernehmen, wobei jede Rolle einen unterschiedlichen Aspekt des Verhaltens der Klasse definiert. APPC bestehen aus einem *Interface Class Graph*, der Beziehungen zu anderen Klassen beschreibt, und einer Verhaltsdefinition, die das Verhalten bzw. die Kommunikation mit anderen Klassen beschreibt. Kollaborationen zwischen Klassen werden damit im Gegensatz zu objektorientierten Sprachen explizit gekapselt und nicht über mehrere Methoden und Klassen verstreut (*cross-cutting concern*). Integrierte Techniken zur Whitebox- und Blackbox-Komposition erleichtern die evolutionäre Softwareentwicklung.

Aspectual Collaborations (AC [LLO03]) und der Vorgänger *Aspectual Components* [LLM99] sind Weiterentwicklungen von APPC, die Konzepte von AOP, ADLs und komponentenbasierter Softwareentwicklung integrieren. Jede AC definiert einen formalen Klassen-Graphen, der die zusammenarbeitenden Klassen (*Participants*) beschreibt. Jede dieser Klassen enthält wie bei der objektorientierten Programmierung Felder und Methoden sowie zusätzlich *expected* Member, ähnlich wie *required*-Ports, und *aspectual* Methoden. *Expected* Member können mit anderen Collaborations verknüpft werden (*attach*) und *aspectual* Methoden kapseln *cross-cutting concerns* ähnlich wie bei AspectJ. Ein Compiler übersetzt den Quelltext von ACs in Java-Bytecode.

Die verschiedenen Ansätze zum Adaptive Programming arbeiten im Gegensatz zu Adaptierbaren Komponenten auf der Ebene von Quelltext im Sinne einer speziellen Programmiersprache und können nur zur Entwicklungszeit verwendet werden. Eine Rekonfiguration zur Laufzeit wurde nicht betrachtet. Die Trennung von Verhalten und Struktur wurde bei ACs dafür klarer umgesetzt, auch wenn bei Adaptierbaren Komponenten Verbindungen zwischen Subkomponenten explizit modelliert werden können.

2.5 Komponentenmodelle und Komponentenplattformen

In diesem Abschnitt werden verschiedene Komponentenmodelle bzw. -plattformen kurz vorgestellt und dabei insbesondere hinsichtlich der Unterstützung von verschiedenen Formen der Adaption sowie von Konzepten, die auch von Adaptierbaren Komponenten aufgegriffen werden, untersucht. Im Gegensatz zu Adaptierbaren Komponenten sind die Mechanismen in den Komponentenplattformen integriert und können somit nur von speziell dafür entwickelten Komponenten verwendet werden.

2.5.1 Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB, siehe auch Abschnitt 5.1) unterstützen in Abhängigkeit des jeweiligen Applikationsservers die Installation bzw. Aktualisierung von Komponenten zur Laufzeit, jedoch nicht für einzelne Komponenteninstanzen, sondern nur für Komponententypen (siehe Definitionen in Unterabschnitt 3.3.1). Vor einer Aktualisierung von Komponenten müssen alle Komponenteninstanzen deaktiviert werden. Eine Übertragung von Zustandsinformationen, wie in Abschnitt 3.6 vorgestellt, wurde dabei nicht vorgesehen.

BARK (*Bean Automatic Reconfiguration frameworK*) [RAC+02] beschreibt eine mögliche Erweiterung von EJB, um einfache Rekonfigurationen inklusive des Herunterladens von Komponenten bei mehreren EJB-Servern automatisch durchzuführen. BARK übernimmt dabei ausschließlich die Rolle eines Management-Werkzeuges, das Rekonfigurationen steuert, ohne jedoch laufende Anwendungen zu analysieren oder Vorschläge für Rekonfigurationen zu unterbreiten. Auch bei dieser Erweiterung bezieht sich die Rekonfiguration nur auf Komponententypen und nicht auf Komponenteninstanzen.

Eine andere Arbeit [JDL02] erweitert EJB um Mechanismen zur Adaption von Services bei geänderten Umgebungsbedingungen. Eine *Adaption Engine* steuert Rekonfigurationen, die durch *Adaptation Policies* in Form von zusätzlichen XML-Deskriptoren beschrieben werden. Ein einfaches *Monitoring Framework* kann verschiedene Umgebungsbedingungen wie CPU-Auslastung, Ladezustand der Batterie oder Netzwerkbandbreite ermitteln, die als Auslöser

für Rekonfigurationen diesen. Die Erweiterung wurde in einen existierenden EJB-Server (JOnAS) integriert und ist damit abhängig von der Komponentenplattform im Gegensatz zu Adaptierbaren Komponenten.

2.5.2 JavaBeans

JavaBeans (siehe auch Abschnitt 5.2) als leichtgewichtige Komponentenplattform besitzen keine integrierten Mechanismen zur Adaption von Komponenten. Da für die Komponenten aber prinzipiell der komplette Funktionsumfang der Java-Klassenbibliothek zur Verfügung steht, können eigene Adaptionsmechanismen damit entwickelt werden.

2.5.3 Fractal

Fractal [BCS02] realisiert wie Adaptierbare Komponenten ein hierarchisches Komponentenmodell (siehe Unterabschnitt 3.3.1), das programmiersprachen- und plattformunabhängig definiert wurde. Zugehörige Komponentenplattformen wurden bereits für verschiedene Programmiersprachen wie z. B. Java, C++ und Smalltalk entwickelt.

Eine Fractal-Komponente besteht aus zwei Teilen: einem Controller und dem Inhalt aus Subkomponenten, die *required*- und *provided*-Schnittstellen unterstützen. Der Controller steuert verschiedene nicht-funktionale Aspekte der Komponente, wie Introspektion, Konfiguration und Rekonfiguration und kann zu diesem Zweck eine Reihe von unterschiedlichen Control-Schnittstellen implementieren. Dabei wird zwischen internen und externen Schnittstellen unterschieden, die nur von internen Subkomponenten bzw. von anderen externen Komponenten aufgerufen werden können.

Komponenten werden über primitiven oder komplexen Verbindungen (*bindings*) miteinander verbunden. Primitive Verbindungen stellen Verbindungen zwischen gleichrangigen Komponenten (*normal binding*) sowie zwischen internen und externen Schnittstellen von zusammengesetzten Komponenten her (*export* und *import bindings*), die sich alle im gleichen Adressraum befinden. Diese Konzepte werden in ähnlicher Weise auch von Adaptierbaren Komponenten unterstützt und um zusätzliche Mechanismen zum Methoden-Verknüpfung und der Behandlung von unterschiedlichen Schnittstellentypen erweitert (siehe Unterabschnitt 3.3.3). Komplexe Verbindungen bestehen aus einer Menge von Komponenten und primitiven Verbindungen und fungieren als Konnektoren, um z. B. entfernte Methodenaufrufe über das Netzwerk zu unterstützen. Mit der *BindingController*-Schnittstelle können Verbindungen zwischen Komponenten analysiert und geändert werden.

Konfigurationen von Komponenten können wahlweise programmtechnisch, mit einer XML-basierten ADL oder mit einem graphischen Werkzeug interaktiv definiert werden. Mit der *LifeCycleController*-Schnittstelle des Controllers können Komponenten vor der Durchführung von Rekonfigurationen explizit deaktiviert bzw. aktiviert werden. Im Gegensatz zu Adaptierbaren Komponenten werden Rekonfigurationen nicht transparent durchgeführt, sondern müssen explizit vorgesehen und im Controller programmiert werden. Zur Realisierung von komplexeren Rekonfigurationen können mit der *ContentController*-Schnittstelle Subkomponenten zu einer zusammengesetzten Komponente hinzugefügt bzw. entfernt werden.

2.5.4 OpenORB

OpenORB [BCRP98, CBCP02] ist eine Komponentenplattform, die sich besonders durch die unterstützte Funktionalität zur Reflection auszeichnet. Die Komponentenplattform selbst wurde komponentenbasiert mit C++ implementiert und verwendet ebenso wie Anwendungskomponenten das entwickelte leichtgewichtige und effiziente Komponentenmodell OpenCOM, das auf Microsoft COM [Rog97] basiert und es um Mechanismen und Schnittstellen zur Realisierung von Reflection und zusammengesetzten Komponenten erweitert.

Alle Komponenten unterstützen vier verschiedene Meta-Interfaces, um auf die Meta-Ebene zuzugreifen, die als *Meta-space* bezeichnet wird. Damit werden vier verschiedene Metamodelle des Meta-space unterstützt: Mit dem Schnittstellen-Metamodell können unterstützte *provided*- und *required*-Schnittstellen von Komponenten sowie ihre jeweiligen Typen ermittelt werden. Das Architektur-Metamodell erlaubt den Zugriff auf die Implementierung einer Komponente, um damit Verbindungen zwischen Komponenten zu analysieren und auch zu ändern. Mit dem Interception-Metamodell können zur Laufzeit Interceptoren bei bestimmten Schnittstellen registriert werden, die anschließend bei jedem Methodenaufruf aufgerufen werden. Das Ressourcen-Metamodell erlaubt den Zugriff auf Mechanismen zur Ressourcenreservierung des darunterliegenden Ressourcenmanagements, insbesondere zur Realisierung von Multimedia Anwendungen mit QoS-Garantien.

Der Meta-space selbst besteht ebenfalls aus Komponenten, die auch wieder Meta-Interfaces unterstützen. Auf diese Weise entsteht ein potentiell unendlicher Turm der Reflection. Die Meta-Komponenten werden aber erst beim Zugriff erzeugt und benötigen so im Normalfall keine Ressourcen.

Einige Konzepte von OpenORB werden auch von Adaptierbaren Komponenten in ähnlicher Weise unterstützt: das hierarchisches Komponentenmodell, die expliziten Verbindungen zwischen Verbindungen und die Unterstützung von Rekonfiguration und Reflection. Allerdings müssen bei OpenORB Komponenten selbst bestimmte Meta-Schnittstellen implementieren und Verbindungen und Rekonfiguration werden nicht modelliert, sondern programmtechnisch gesteuert. Außerdem sind alle Mechanismen abhängig von der Komponentenplattform und darin integriert.

2.5.5 SOFA

Die SOFA (*SOFtware Appliances*) Komponentenplattform [PBJ98, KT02] unterstützt ein verteiltes und hierarchisches Komponentenmodell mit besonderer Funktionalität zum dynamischen Herunterladen und Aktualisieren von Komponenten zur Laufzeit (*download & update*). SOFA Komponenten besitzen *provided*- und *required*-Schnittstellen und werden wie bei ADLs über Konnektoren miteinander verbunden. Komponenten werden prinzipiell als Blackbox betrachtet, zusammengesetzte Komponenten gestatten jedoch einen Einblick in die interne Architektur (Greybox). Zusätzlich werden Versionsinformationen eingeführt, um damit Aktualisierungen und Abhängigkeiten von Komponenten zu steuern. Mit einer integrierten ADL (genannt SOFA CDL) werden diese Informationen explizit beschrieben. Für Architektur von zusammengesetzten Komponenten und die Komponentenschnittstellen können zusätzlich Verhaltensprotokolle definiert werden [Pla05].

Die so genannte DCUP-Architektur (*Dynamic Component UPdate*) von SOFA unterteilt Komponenten in einen permanenten und einen aktualisierbaren Teil. Der permanente Teil

kann zur Laufzeit nicht geändert werden und enthält den *Component Manager*, der in Zusammenarbeit mit *Updaters* und einer einheitlichen Control-Schnittstelle die Steuerung der Aktualisierung der Komponente übernimmt. Im aktualisierbaren Teil steuert der *Component Builder* die Initialisierung bzw. das Beenden von allen weiteren Komponenten. Dabei erzeugt er z. B. Verbindungen zwischen Komponenten entsprechend der Definitionen in der ADL. Bei einer Aktualisierung wird der *Component Builder* zusammen mit allen weiteren Komponenten des aktualisierbaren Teils der Komponente ausgetauscht. Die SOFA-Implementierung in Java verwendet für alle Bestandteile des aktualisierbaren Teils einen eigenen Class-Loader, um unterschiedliche Versionen von gleichnamigen Implementierungsklassen zu ermöglichen.

Mit Hilfe eines Zwischenspeichers (*StateStore*) im permanenten Teil einer Komponente kann der *Component Manager* auch Zustandsinformationen von alten zu neuen Versionen von Komponenteninstanzen übertragen. Im Gegensatz zu dem Verfahren in Abschnitt 3.6 wird der Zustandstransfer nicht deklarativ, sondern in Form von Java-Quelltext beschrieben.

2.6 Software-Produktlinien

Software-Produktlinien beschäftigen sich mit der „Produktion“ von Produktfamilien. Die Analogie mit gekannten Begriffen aus der Industrie soll die Übertragung dort etablierter Konzepte und Verfahren auf die Softwareentwicklung verdeutlichen, wie bereits McIlroy [McI68] vorschlug. Eine erste Definition von Produktfamilien wurde von David Parnas bereits vor 30 Jahren formuliert:

> *„We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of individual family members.“ [Par76]*

Methoden zu Software-Produktlinien beschäftigen sich mit der Analyse und der Modellierung von Gemeinsamkeiten und Unterschieden einer Menge von Software-Produkten, die auch als Produktfamilie bezeichnet wird. Außerdem werden Werkzeuge für die Modellierung und die Erzeugung bzw. Generierung von Software-Produkten entwickelt. Generell wird zwischen dem *Domain Engineering* und dem *Application Engineering* unterschieden, die sich mit der Analyse, der Entwicklung und dem Test von Software-Produktlinien selbst bzw. von Anwendungen auf Basis von Software-Produktlinien beschäftigen.

Mit FODA (*Feature-Oriented Domain Analysis* [KCH$^+$90]) werden Unterschiede der verschiedenen Produkte einer Software-Produktlinie in einer bestimmten Domäne systematisch analysiert und mit einem Feature-Diagramm modelliert. Dieser Prozess wird auch als Feature-Modellierung [CE00, CHE04] bezeichnet. Ein *Feature* beschreibt eine bestimmte definierte Funktionalität einer Software und wird als Änderung bzw. Verfeinerung (Delta) einer Ausgangsstruktur beschrieben, meist in Form von Subklassen oder Mixins [BC90]. Dadurch entsteht eine Hierarchie von Erweiterungen, die auch aufeinander aufbauen können.

Ein Feature-Modell, das durch ein Feature-Diagramm graphisch ausgedrückt wird, beschreibt mit einer hierarchischen Baumstruktur die Zusammenhänge zwischen den Features eines Produktes. Gemeinsame (*common*) Features sind in allen Produkten einer Produktfamilie vorhanden, während variable Features Unterschiede modellieren. Einzelne variable Features können als optional oder obligatorisch gekennzeichnet werden und mehrere Features

einer Gruppe können als Alternativen (1 aus n) oder als Teilmenge (n aus m) ausgewählt werden. Auf diese Weise wird die Variabilität in einer Produktfamilie explizit beschrieben und es werden gültige Kombinationen verschiedener Features definiert. UML kann ebenfalls zur Modellierung von Produktlinien verwendet werden [Gom04].

Knoten im Feature-Baum mit variablen Features werden als Variationspunkte (*Variation Points*) [IJ97] bezeichnet. Variationspunkte werden durch bestimmte Variationsmechanismen implementiert, z. B. durch Vererbung, Erweiterung, Parameter, Konfigurationen, Template-Instantiierung oder Generierung. Adaptierbare Komponenten verwenden in diesem Sinne Parameter (Unterunterabschnitt 3.3.1.2), Konfigurationen (Abschnitt 3.4) sowie Erweiterungen in Form von Adaptions- und Aspektoperatoren (Unterabschnitt 3.3.4) als Variationsmechanismen.

Die Möglichkeiten zur Beschreibung von Variabilitäten sind bei den verschiedenen vorgestellten Techniken von Software Produktlinien ausdrucksmächtiger als die in Abschnitt 3.4 entwickelten Verfahren für die Konfigurationsbeschreibung von Adaptierbaren Komponenten. Allerdings werden die Techniken für Software Produktlinien im Gegensatz zu Adaptierbaren Komponenten nur zur Analyse-, Entwurfs- bzw. Entwicklungszeit angewendet, aber nicht für Rekonfigurationen zur Laufzeit, d. h. die Konfiguration einer Software-Produktline kann zur Laufzeit nicht geändert werden.

Generative Programming [CE00] fasst verschiedene Konzepte insbesondere zu Software-Produktlinien zu einer Entwicklungsmethode zusammen:

> *„Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.“ [CE00]*

Greenfield und Short beschreiben mit *Software Factories* [GS04] eine ähnliche Vision für die Industrialisierung der Software-Entwicklung wie *Generative Programming*. Hauptprobleme sind für sie monolithische Konstruktionen, übertriebene Verallgemeinerungen, Einmalentwicklungen und unausgereifte Prozesse bei der Softwareentwicklung. Als Ausweg sollen im Wesentlichen bereits bekannte Methoden und Technologien aus der Software-Technik sinnvoll miteinander kombiniert werden, um eine automatisierte Softwareentwicklung in der Praxis zu erreichen. Eine *Software Factory* definieren sie wie folgt:

> *„A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components.“ [GS04]*

Aber auch Software-Factories beschränken sich auf die Definition von Software-Produkten zur Entwicklungszeit und sehen keine speziellen Verfahren zur Rekonfiguration zur Laufzeit vor, wie bei Adaptierbaren Komponenten.

2.7 Modellgetriebene Softwareentwicklung

Verschiedene Modelle werden schon seit vielen Jahren zur Softwareentwicklung eingesetzt, in der Praxis bisher insbesondere zur Dokumentation während der Analyse- und Entwurfsphase. Die eigentliche Software wurde anschließend mit verschiedenen Programmiersprachen entwickelt und die Modelle dienten nur noch als Orientierungsvorlage bei der Programmierung. Mit der modellgetriebenen Softwareentwicklung [SV05] soll der offensichtliche Bruch zwischen Modellen und Quelltexten beseitigt werden. Angestrebt werden eine vollständige Beschreibung von Software auf der Modellebene und eine anschließende möglichst automatische Erzeugung von Anwendungen. Modellgetriebene Softwareentwicklung hat viele Querbezüge zu den im vorigen Abschnitt beschriebenen Software-Produktlinien. Eine klare Trennung der beiden Forschungsbereiche ist daher nicht immer möglich. Stattdessen ist eine zunehmende Verschmelzung zu erwarten.

2.7.1 UML und Metamodellierung

Die *Unified Modeling Language* (UML [uml05a, uml05b]) der Object Management Group (OMG) hat sich in den letzten Jahren als Standard für die Modellierung von Anwendungen etabliert. Die verschiedenen Diagrammtypen (z. B. Klassen-, Komponenten-, Zustands-, Aktivitätsdiagramme) können zur graphischen Modellierung von unterschiedlichen Sichten (z. B. Struktur und Verhalten) und Abstraktionsniveaus verwendet werden. Die anfänglich mehr informelle Beschreibung der Software wurde insbesondere durch die Integration der *Object Constraint Language* (OCL [OCL05]) zunehmend in Richtung einer exakten und formalen Beschreibung weiterentwickelt. Mit UML-Profilen kann UML domänenspezifisch erweitert werden, beispielsweise für Anwendungen auf Basis von EJB [uml04] oder Echtzeitanwendungen [UML05c]. Ein UML-Profil wird dabei als Erweiterung des UML-Metamodells um neue Stereotypen, *Tagged Values* und (OCL-)*Constraints* definiert. Diese Erweiterungen werden zunehmend auch von UML-Entwicklungswerkzeugen unterstützt, die damit direkt zur Modellierung verwendet werden können.

Seit der Version 2.0 wurde das Metamodell von UML vollständig mit der *MetaObject Facility* (MOF [MOF06]) definiert. MOF definiert ein Metameta-Modell, das zur Beschreibung von beliebigen Metamodellen verwendet werden kann. XMI (*XML Metadata Interchange* [XMI05]) beschreibt ein allgemeines, XML-basiertes Austauschformat für beliebige Modelle auf Basis von MOF und ermöglicht damit die Interoperabilität zwischen verschiedenen Modellierungswerkzeugen. Der Modellierungsprozess in Verbindung mit MOF gemäß den Vorgaben der OMG sieht Modelle auf vier verschiedenen Ebenen vor: M3 als Metameta-Modell, M2 für Metamodelle, M1 für Modelle und M0 für Instanzen. Modelle einer bestimmten Ebene sind jeweils Instanzen der nächst höheren Ebene. Nur Modelle aus M3 sind selbstbeschreibend, um keine unendliche Rekursion einzuführen.

Metamodelle und UML-Profile definieren die abstrakte Syntax von Modellierungssprachen. Sie können zur Entwicklung domänenspezifischer Sprachen (*Domain-Specific Language* – DSL) verwendet werden, wenn zusätzlich auch noch die konkrete Syntax, z. B. eine bestimmte graphische Notation, definiert wird.

Executable UML [MJ02] ist ein erster Ansatz zur vollständigen Entwicklung von Anwendungen ausschließlich auf Modellebene. UML wird dabei um Verhaltensbeschreibungen für

Methoden erweitert, damit ein Modell-Compiler Anwendungen automatisch aus Modellen generieren kann.

Explizite Konfigurationen und Variationen von Adaptierbaren Komponenten werden mit Hilfe der graphischen Notation von UML-Komponentendiagrammen und einigen Erweiterungen (hinzugefügte/entfernte Verbindungen bzw. Komponenten) modelliert. Das Metamodell wurde mit MOF definiert (Unterabschnitt 4.3.1 und Anhang B) und als Speicherformat für Modellinstanzen wurde XMI verwendet. Anstelle eines eigenen Metamodells auf Basis von MOF, wäre es auch möglich gewesen, Adaptierbare Komponenten ausschließlich durch das UML-Metamodell erweitert durch ein UML-Profil zu modellieren. Um die Komplexität zur Laufzeit zu reduzieren, wurde jedoch das relativ einfache eigene Metamodell anstelle das sehr umfangreichen UML-Metamodells verwendet.

2.7.2 MDA

Die *Model-Driven Architecture* (MDA [mda03, MB04]) ist eine Standardisierungsinitiative der OMG für ein Vorgehensmodell zur modellgetriebenen Softwareentwicklung. Software soll damit durch eine Reihe von Modellen auf unterschiedlichen Abstraktionsniveaus und Detaillierungsgraden entwickelt werden und anschließend im Idealfall automatisch generiert werden. MDA unterstützt dabei die Spezifizierung von Systemen unabhängig von einer bestimmten Plattform, die Spezifizierung von Plattformen, die Auswahl einer bestimmten Systemplattform und die Transformation der Systemspezifikation auf eine bestimmte Plattform. Portabilität, Interoperabilität und Wiederverwendbarkeit sind die drei Hauptziele der MDA.

Die MDA unterscheidet drei unterschiedliche Sichten (*MDA Viewpoints*):

- Das Domänenmodell (*Computation Independent Model* – CIM) beschreibt das System aus der Sicht und mit den Begriffen der jeweiligen Domäne, jedoch insbesondere unabhängig von der Struktur des Softwaresystems, dass später zur Verarbeitung verwendet werden soll. Zur Modellierung können domänenspezifische Sprachen (*Domain Specific Language* – DSL [MHS05]) definiert und angewendet werden.

- Das plattformunabhängige Modell (*Platform Independent Model* – PIM) beschreibt das System unabhängig von einer bestimmten Plattform, damit es später auf eine Vielzahl von Plattformen transformiert werden kann.

- Das plattformspezifische Modell (*Platform Specific Model* – PSM) erweitert ein PIM um Details einer bestimmten Plattform.

MDA integriert eine Reihe von anderen Standards der OMG: MOF wird zur Definition von Metamodellen, XMI als Speicher- und Austauschformat für Metamodelle und UML in Zusammenarbeit mit OCL zur graphischen Modellierung verwendet. *Query/View/Transformation* [qvt05] beschreibt eine Sprache und ein Verfahren zur Definition von Modelltransformationen. Damit können Beziehungen zwischen unterschiedlichen Modellen und den einzelnen Modellbestandteilen hergestellt werden, um Modelltransformationen automatisch durchführen zu können.

Die Unterstützung verschiedener Komponentenplattformen für Adaptierbare Komponenten orientiert sich an dem MDA Prozess. Die Modellierungstechniken (siehe Kapitel 3) und das zugehörige Metamodell (siehe Anhang B) können als eine DSL aufgefasst werden.

2.8 Adaptionsmechanismen

In diesem Abschnitt werden verschiedene Adaptionsmechanismen vorgestellt und klassifiziert. Die Auswahl erfolgte dabei unter dem Gesichtspunkt, dass sich die Mechanismen zur Adaption von Softwarekomponenten anwenden lassen, auch wenn das nicht immer in den jeweiligen Originalpublikationen beschrieben bzw. angedacht wurde. Die Verfahren unterscheiden sich im Zeitpunkt der Anwendung (Entwicklungszeit bis Laufzeit), im Adaptionsgegenstand (Quelltext, Bytecode, Objekte zur Laufzeit), in der Art der Adaptionsbeschreibung (Programm, Deklarativ, API) und im Umfang der möglichen Adaptionen.

Die meisten Verfahren arbeiten mit Java. Es ist aber denkbar, die Konzepte in ähnlicher Weise auch für andere Programmiersprachen wie z. B. C# anzuwenden.

2.8.1 Modifikation von Bytecode

Eine Reihe von Projekten beschäftigt sich mit der Erzeugung und Manipulation von Java-Bytecode. Die meisten Verfahren können sowohl zur Installationszeit als auch zur Laufzeit angewendet werden.

Binary Component Adaptation (BCA [KH98]) beschreibt als erste Arbeit ein Verfahren zur Modifikation von Java-Bytecode während des Ladens, um Änderungen an Java-Komponenten vorzunehmen, ohne die Binär-Kompatibilität zu verletzen. Verschiedene Änderungen wie das Umbenennen von Klassen, Methoden und Feldern sowie das Hinzufügen von Schnittstellen, Methoden und Feldern zu Klassen werden mit Hilfe von *Delta File* beschrieben und beim Laden der `class`-Dateien entsprechend ausgeführt. BCA nimmt Änderungen an der Java VM vor, um sich in den Ladevorgang zu integrieren. Die in Unterabschnitt 3.3.4 beschriebenen Adaptionsoperatoren verwenden einen ähnlichen Ansatz zur Beschreibung von Änderungen, allerdings ohne den Zeitpunkt und das Verfahren für die Durchführung der Änderungen festzulegen.

Die *Byte Code Engineering Library* (BCEL [Dah01]) ist eine Open-Source Java-Bibliothek, die im Rahmen eines Projektes der Apache Foundation [bce05] weiterentwickelt wird. Sie stellt zahlreiche Funktionen zur Analyse, Erzeugung und Manipulation von Java-Bytecode zur Verfügung. Die Informationen einer `class`-Datei werden dazu durch Objekte repräsentiert, um einen einfachen Zugriff auf alle darin enthaltenen symbolischen Informationen sowie auf Methoden, Felder und Bytecode-Instruktionen zu erhalten. Diese systemnahen (*low-level*) Funktionen von BCEL können zur Implementierung von darauf aufbauenden Adaptionsmechanismen verwendet werden und ermöglichen beispielsweise die Erzeugung von Java-Bytecode zur Laufzeit und die Modifikation von Java-Bytecode mit Hilfe von speziellen Class-Loadern während des Ladens.

JOIE (*Java Object Instrumentation Environment* [CCK98]) unterstützt ähnlich wie BCA die Transformation von Java-Klassen während des Ladens durch einen speziellen Class-Loader. Im Gegensatz zu BCA sind dafür allerdings keine Änderungen an der Java-VM erforderlich. Die Transformationen werden als Programm innerhalb von normalen Java-Klassen definiert, die eine bestimmte Schnittstelle implementieren. Durch die Registrierung beim JOIE-Class-Loader werden sie dann während des Ladens von Java-Klassen ausführt.

Javassist [Chi00] erweitert die Standard-Reflection API von Java um Funktionen zur Unterstützung von struktureller Reflection und damit eines einfachen Metaobject Protokolls. Damit können Klassen zur Laufzeit verändert und auch erzeugt werden, ohne detailliertes

Wissen über den Aufbau von Bytecode vorauszusetzen. Die Erweiterung wird ähnlich wie bei JOIE durch die Veränderung des Bytecodes von Java-Klassen während des Ladens realisiert. In späteren Arbeiten wurde Javassist auch zur Implementierung von dynamischer aspektorientierter Programmierung [CST03] (siehe Abschnitt 2.4) verwendet. Es wird mittlerweile als ein Teilprojekt des EJB-Applikationsservers JBoss [FR03] weiterentwickelt.

JMangler [KCA01] ist ein Java-Framework zur Transformation von `class`-Dateien während des Ladens, das die Vorteile von BCA, JOIE und Javassist miteinander kombiniert und erweitert sowie gleichzeitig einige Einschränkungen beseitigt. JMangler arbeitet unabhängig von einer bestimmten Java VM und setzt auch nicht die Verwendung eines speziellen Class-Loaders voraus, kann dadurch allerdings auch keine System-Klassen modifizieren. Mehrere *Transformer* – das sind spezielle Java Klassen, die Bytecode-Modifikationen durchführen – können unter Berücksichtigung möglicher Abhängigkeiten miteinander kombiniert werden, um komplexere Transformationen durchzuführen.

2.8.2 Modifikation von Quelltext

Eine in der Praxis wahrscheinlich immer noch sehr oft verwendete Adaptionstechnik ist *Copy & Paste*. Dabei kopieren Entwickler einfach den Quelltext einer Komponente und nehmen anschließend manuell die gewünschten Änderungen vor. Dieses Verfahren führt zu mehreren Kopien des Quelltextes, die anschließend unabhängig voneinander weiterentwickelt werden. Damit steigt insbesondere die Wahrscheinlichkeit, dass neue Fehler eingeführt werden oder dass gefundene Fehler nicht in allen Kopien beseitigt werden. Mit dem Konzept der Vererbung in objektorientierten Programmiersprachen können Änderungen und Erweiterungen an Klassen vorgenommen werden, ohne den Quelltext zu duplizieren. Schnittstellen können dabei aber nicht wesentlich modifiziert werden.

RECODER [rec06] ist ein Java-Framework zur programmtechnischen Analyse und Transformation von Java-Quelltexten. RECODER parst Java-Quelltext-Dateien, repräsentiert sie als Objektmodell zur Durchführung beliebiger Manipulationen und kann sie anschließend wieder in Form von Quelltext abspeichern. Auf diese Weise wird eine statische Metaprogrammierung auf Basis von Java-Quelltext unterstützt.

Invasive Software Composition [Aßm03b] beschreibt eine Technik, die verschiedene Konzepte von generativer Softwareentwicklung (Abschnitt 2.6) und AOP (Abschnitt 2.6) miteinander kombiniert und erweitert. Software-Komponenten werden als *Greyboxes* aufgefasst, die durch Kompositionsprogramme miteinander verknüpft und gleichzeitig angepasst werden können. So genannte *Fragment Boxes* kapseln verschiedenartige Code-Templates, die über *Hooks* modifiziert werden können. Eine *Fragment Box* kann Pakete, Klassen, Methoden oder Teile von Methoden (einen Block von Anweisungen oder einzelne Anweisungen) enthalten, wobei mit Hooks jeweils unterschiedliche Eigenschaften modifiziert werden können. Mit implizite Hooks, die vergleichbar sind mit Joinpoints bei der aspektorientierten Programmierung, können Änderungen an einer *Fragment Box* durchgeführt werden, ohne dass dies bei der Entwicklung vorgesehen werden muss. Explizite Hooks, die vergleichbar sind mit Template-Ausdrücken in C++ oder Ports in ADLs, werden dagegen bereits bei der Entwicklung als Stellen für Änderungen bzw. Erweiterungen vorgesehen. COMPOST (*Software COMPOsition SysTem*) implementiert die Konzepte von Invasive Software Composition auf Basis von Java-Quelltext und nutzt dabei das RECODER-Framework.

Alle vorgestellten Mechanismen können nur zur Entwicklungszeit, spätestens aber bei der

Installation angewendet werden. Zur Realisierung von Rekonfigurationen zur Laufzeit sind sie damit nicht geeignet.

2.8.3 Adaptionsmechanismen zur Laufzeit

Wrapper und Adapter [GHJV95] sind typische Implementierungsverfahren für Adaptionsmechanismen zur Laufzeit. Ein zusätzliches Wrapper- oder Adapter-Objekt kann die Funktionalität eines zugehörigen anderen Objektes erweitern oder anpassen, ohne das Ursprungsobjekt direkt zu ändern. Adaptions- und Aspektoperatoren verwenden diesen Ansatz als eine mögliche Implementierungsstrategie (Unterabschnitt 4.3.5).

Metaobject Protocols (MOP [KRB91]) definieren ein weiteres Verfahren, um Änderungen zur Laufzeit durchzuführen. Sie wurden ursprünglich für objektorientierte Programmiersprachen entwickelt, um durch eine Menge von (Meta-)Klassen und Methoden das Verhalten der Programmiersprache zu analysieren (*Introspection*) und zu modifizieren (*Intercession*). In komponentenorientierten Systemen können entsprechend die Eigenschaften von Komponenten manipuliert werden, wie z. B. bei OpenORB [CBCP02] oder Fractal [BCS02]. In Adaptierbaren Komponenten werden die Konzepte übernommen, um in begrenztem Umfang das Adaptionsverhalten zu modifizieren (siehe Unterabschnitt 4.3.9).

Lasagne [TVJ+01, JT03] verwendet Wrapper unter Nutzung des Decorator- und Rollen-Entwurfsmusters für Client-spezifische Adaptionen von Komponenten zur Laufzeit. Für jeden Client kann dabei eine bestimmte Kette von Wrapper-Objekten ausgewählt werden, die damit einen bestimmten Aspekt (siehe Abschnitt 2.4) realisieren und kapseln. Um die Erzeugung von neuen Wrapper-Objekten für jeden Client zu vermeiden und damit die Skalierbarkeit zu erhöhen, können bestimmte Wrapper-Objekte für verschiedene Clients wiederverwendet werden. In einer Folgearbeit [NJ04] wird mit Lasagne/J eine Spracherweiterung für Java vorgestellt, die die gleichzeitige und unabhängige Erweiterung von Objekten unterstützt.

Superimposition [Bos99] erlaubt die Adaption von Black-Box-Komponenten mit Hilfe von so genannten *Adaptation Types*, die wesentlich mehr Möglichkeiten bieten als die *Delta Files* von BCA. Damit wird vordefinierte, aber konfigurierbare Funktionalität der vorhandenen Funktionalität einer Komponente „aufgezwungen“ (*impose*) bzw. hinzugefügt. Die Komponente und die Funktionalität zur Adaption der Komponente sind dabei voneinander entkoppelt. Die Superimposition wurde mit Hilfe eines *Layered Object Model* (LayOM) implementiert.

Alle vorgestellten Verfahren lassen sich prinzipiell auch als Implementierungsstrategie von Adaptions- und Aspektoperatoren einsetzen, wurden im Rahmen dieser Arbeit aber nicht berücksichtigt.

Kapitel 3

Konzepte und Modellierung Adaptierbarer Komponenten

In diesem Kapitel werden die Modellierung und die zugrunde liegenden Konzepte Adaptierbare Komponenten unabhängig von konkreten Komponentenplattformen (siehe Abschnitt 2.5) beschrieben. Ergebnis der Modellierung ist ein plattformunabhängiges Modell[1] Adaptierbarer Komponenten, für das ein auch passendes Metamodell entwickelt wurde (siehe Anhang B). Die Modelltransformation und beispielhafte Implementierung bei einigen existierenden Komponentenplattformen ist anschließend Gegenstand von Kapitel 4 und Kapitel 5.

Zunächst werden in Abschnitt 3.1 die generellen Konzepte und Bestandteile von Adaptierbaren Komponenten im Überblick vorgestellt. Im Anschluss daran werden die einzelnen Modellbestandteile detailliert beschrieben. Darauf aufbauend wird in Abschnitt 3.4 die Konfigurationsbeschreibung erläutert, die überhaupt erst die Adaptivität der Komponenten ermöglicht. Die verbleibenden Abschnitte dieses Kapitels beschäftigen sich mit verschiedenen Detailproblemen wie z. B. der Synchronisierung der Rekonfiguration.

3.1 Übersicht

Die in der vorliegenden Arbeit entwickelten Adaptierbaren Komponenten zeichnen sich durch eine Reihe von Eigenschaften und Konzepten aus, die im Folgenden kurz vorgestellt werden.

Abbildung von Parametern auf Konfigurationen Wie bereits in der Einleitung in Abschnitt 1.2 beschrieben, besteht die Grundidee von Adaptierbaren Komponenten in der Abbildung von Komponentenparametern auf unterschiedliche interne Konfigurationen. Adaptierbare Komponenten nutzen ein hierarchisches Komponentenmodell, d. h. sie setzen sich aus einer Menge von Subkomponenten zusammen. Die Subkomponenten sind entweder atomare Komponenten, die selbst keine Subkomponenten enthalten, oder wiederum Adaptierbare Komponenten. Auf diese Weise können auch komplexe Anwendungen schrittweise in immer kleinere und damit besser handhabbare Komponenten zerlegt werden (*hierarchische Dekomposition*). Dieses Konzept von zusammengesetzten Komponenten wurde insbesondere bei ADLs (siehe Abschnitt 2.1) entwickelt und wird auch bei einigen Komponentenplattformen, wie z. B. Fractal (siehe Unterabschnitt 2.5.3) unterstützt.

Plattformunabhängigkeit Die Konzepte und die Modellierung Adaptierbarer Komponenten sind unabhängig von einer bestimmten Komponentenplattform. Sie werden erst später auf

[1] *Platform Independent Model* – PIM

entsprechende Modelle und Konzepte von Zielkomponentenplattformen abgebildet oder geeignet emuliert. Die dafür notwendige Modelltransformation und Laufzeitunterstützung ist Gegenstand von Kapitel 4.

Anwendbarkeit zu verschiedenen Zeiten im Entwicklungszyklus Bei der Modellierung von Adaptierbaren Komponenten werden keine Annahmen gemacht, wann bestimmte Funktionen realisiert werden. Das gilt z. B. für das Setzen von Parametern oder das Ausführen von Adaptionsmechanismen. Diese Funktionen können je nach konkreten Anforderungen zur Entwicklungs-, Installations- oder Laufzeit realisiert werden. Auf diese Weise sind verschiedene Optimierungen möglich, ohne das Ausgangsmodell ändern zu müssen.

Visuelle Entwicklung Wie bereits in der Einleitung in Abschnitt 1.2 gefordert, werden Adaptierbare Komponenten so weit wie möglich auf Basis einer UML-basierten graphischen Notation [uml05b] modelliert, da sich UML in den letzten Jahren zum De-facto-Standard für die Softwaremodellierung etabliert hat und eine Reihe von Modellierungswerkzeugen verfügbar sind. Für die Speicherung der Modellierungsergebnisse (Metamodellierung) gibt es mindestens zwei verschiedene Ansätze (siehe Unterabschnitt 2.7.1): die Verwendung des UML-Metamodells [uml05b] oder die Entwicklung eines eigenen MOF-basierten Metamodells [MOF06]. Für Adaptierbare Komponenten wird ein MOF-basierten Modell definiert (siehe Anhang B), da das UML-Metamodell sehr umfangreich ist und nur ein sehr kleiner Teil (Komponentendiagramme) zur Modellierung benötigt werden. Der Speicherplatzbedarf kann nicht vernachlässigt werden, da das Metamodell auch von der Laufzeitunterstützung verwendet wird (siehe Unterabschnitt 4.3.1).

Integration von Adaptionsmechanismen Bei der Komposition von mehreren Komponenten zu einer Adaptierbaren Komponente müssen teilweise Änderungen an den Komponenten vorgenommen werden, um eine Zusammenarbeit zu ermöglichen. Für diesen Zweck können verschiedene Adaptionsmechanismen (siehe Abschnitt 2.8) in die Adaptierbare Komponente integriert werden.

Explizite Verknüpfung von Komponenten Innerhalb einer Adaptierbaren Komponente werden Abhängigkeiten und Beziehungen zwischen Komponenten explizit beschrieben. Komponenten kommunizieren mit ihrer Umwelt ausschließlich über Ports. *Provided*-Ports definieren Funktionalität einer Komponente, die anderen Komponenten zur Verfügung gestellt wird, wohingegen *required*-Ports benötigte Funktionalität ausdrücken. Verbindungen koppeln die unterschiedlichen Ports zur Laufzeit. Sie stellen außerdem die Verknüpfung von externen Ports der Adaptierbaren Komponente mit internen Ports von Subkomponenten her (*Interface Binding*). Damit wird eine Trennung zwischen der Programmlogik der Komponenten und ihrer Verknüpfung erreicht.

Annahmen Bei der Entwicklung von Adaptierbaren Komponenten wurden einige Annahmen zugrunde gelegt:

- Die externe Sicht einer Adaptierbaren Komponente (Ports bzw. funktionale Komponenten-Schnittstellen und Parameter) bleibt konstant und ist nicht von der Adaption

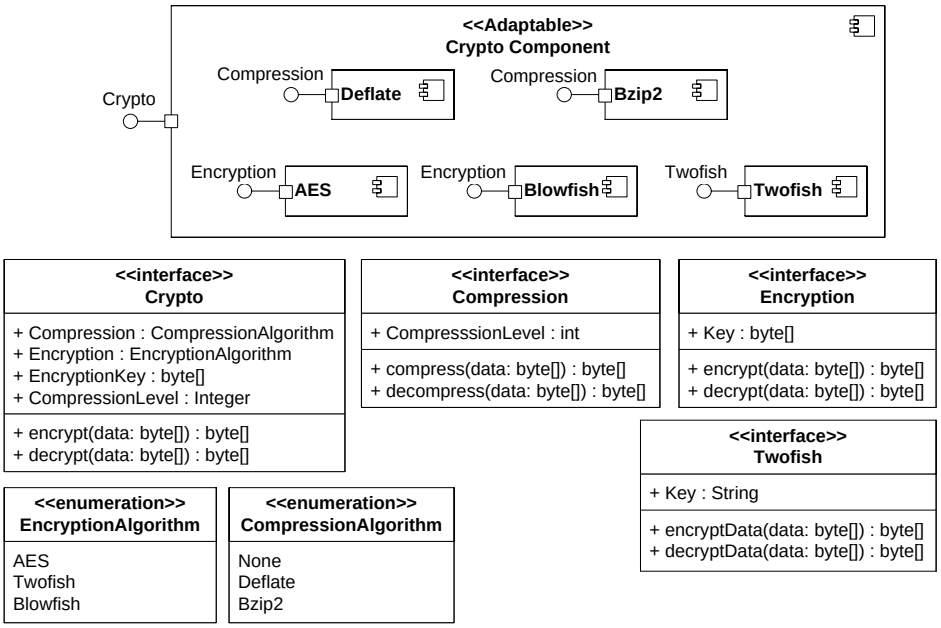

Abbildung 3.1: Aufbau der Kryptographiekomponente

betroffen. Dadurch bleiben Auswirkungen von Rekonfigurationen zur Laufzeit immer auf die jeweilige Adaptierbare Komponente beschränkt.

- Für die Implementierung von Komponenten ist keine formale Verhaltensbeschreibung vorhanden. An dieser Stelle bieten sich Anknüpfungspunkte für weiterführende Arbeiten (siehe Abschnitt 6.2).
- Alle Methodenaufrufe bei Adaptierbaren Komponenten werden in endlicher Zeit abgearbeitet.

3.2 Einführendes Beispiel

In diesem Abschnitt wird ein relativ einfaches Beispiel einer Adaptierbaren Komponente vorgestellt, das dennoch ausreichend komplex ist, um die wesentlichen Konzepte zu demonstrieren. Im weiteren Verlauf dieses Kapitels wird das Beispiel immer wieder aufgegriffen, um vorgestellte Detailkonzepte daran zu illustrieren.

Als Beispiel wird eine Kryptographiekomponente verwendet (siehe Abbildung 3.1). Sie stellt über ihre Schnittstelle `Crypto` zwei wesentliche Funktionen zur Verfügung: das Verschlüsseln und das Entschlüsseln von Daten. Zusätzlich besteht noch die Möglichkeit, die Daten vor dem Verschlüsseln zu komprimieren bzw. nach dem Entschlüsseln wieder zu dekomprimieren. Eine Komprimierung reduziert zum einen den Speicherplatz bzw. die übertragene Datenmenge und erhöht zum anderen die Entropie der Daten.

Die Kryptographiekomponente unterstützt die Verschlüsselungsalgorithmen AES [DR02], Blowfish [Sch94] und Twofish [SKW+99] sowie die Datenkompressionsalgorithmen Deflate [Deu96] und Bzip2 [Sew05]. Diese unterschiedlichen Algorithmen liegen als Subkomponenten

Compression / Encryption	Bzip2	Deflate	None
AES	Bzip2 + AES	Deflate + AES	AES
Blowfish	Bzip2 + Blowfish	Deflate + Blowfish	Blowfish
Twofish	Bzip2 + Twofish	Deflate + Twofish	Twofish

Tabelle 3.1: Mögliche Konfigurationen der Kryptographiekomponente

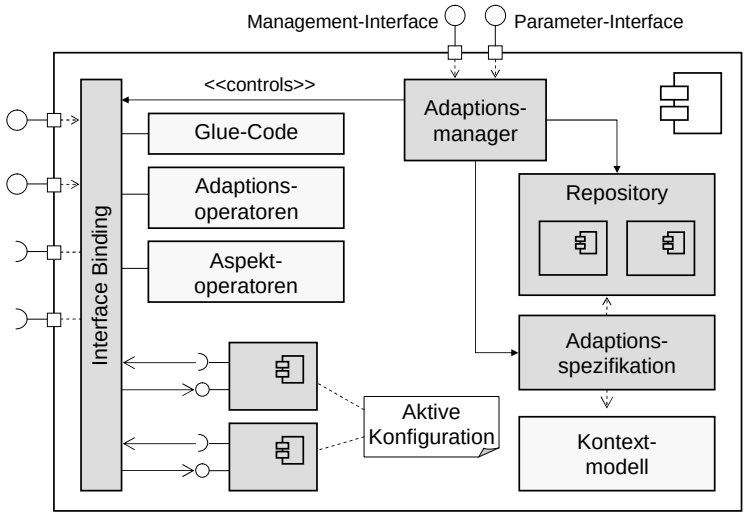

Abbildung 3.2: Bestandteile einer Adaptierbaren Komponente

vor und werden innerhalb der Kryptographiekomponente miteinander kombiniert. Die AES- und Blowfish-Komponente implementieren beide die `Encryption`-Schnittstelle, während die Twofish-Komponente die `Twofish`-Schnittstelle implementiert. Beide Schnittstellen definieren im Prinzip die gleichen Methoden. Durch die unterschiedlichen Typen und Methodennamen sind sie jedoch ohne zusätzliche Anpassungen nicht zueinander kompatibel. Die Auswahl der Algorithmen bzw. der entsprechenden Subkomponenten erfolgt mit Hilfe von Parametern der Kryptographiekomponente. Der Parameter `Compression` legt den verwendeten Datenkompressionsalgorithmus und der Parameter `Encryption` den verwendeten Verschlüsselungsalgorithmus fest.

Tabelle 3.1 zeigt die möglichen Werte der beiden Parameter und die sich daraus ergebenen Konfigurationen der Kryptographiekomponente. Die Verknüpfung und Auswahl der Subkomponenten bleibt für den Anwender transparent.

3.3 Strukturelle Bestandteile von Adaptierbaren Komponenten

Eine Adaptierbare Komponente besteht intern aus einer Reihe von strukturellen Bestandteilen, wie in Abbildung 3.2 dargestellt. Die Bestandteile haben jeweils auch zugeordnete Klassen im Metamodell Adaptierbarer Komponenten (siehe Anhang B), um die Eigenschaften zu modellieren. Viele dieser Bestandteile sind optional und werden nur bei Bedarf eingesetzt, um damit eine flexible Anpassung an die jeweiligen Anforderungen von Anwendungen zu ermöglichen.

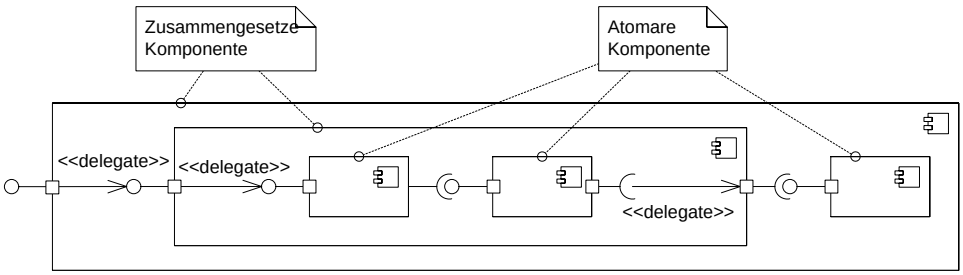

Abbildung 3.3: Beispiel für eine zusammengesetzte Komponente

Definition 3.1. Ein Typ einer Adaptierbare Komponente wird durch ein Tupel $AdKomponente = (\underline{Po}, \underline{P}, \underline{Konf}, Mapping)$ definiert mit $\underline{Po} \subseteq \underline{Ports}$ als Menge der externen Ports, $\underline{P} \subseteq \underline{Parameter}$ als Menge der Komponentenparameter, $\underline{Konf} \subseteq \underline{Konfigurationen}$ als Menge aller internen Konfigurationen und $Mapping \in \underline{Mappings}$ als Definition der Abbildung von Parameterwerten auf Konfigurationen.

Die Mengen $\underline{Ports}$, $\underline{Parameter}$, $\underline{Konfigurationen}$ und $\underline{Mappings}$ werden in den Definitionen 3.4, 3.7, 3.17 und 3.26 definiert.

In den folgenden Abschnitten werden die einzelnen Modellbestandteile detailliert beschreiben.

3.3.1 Subkomponenten

Subkomponenten implementieren die Funktionalität einer Adaptierbaren Komponente, die durch die Verknüpfung mit externen Ports (siehe Unterabschnitt 3.3.3) aufgerufen werden kann. Das Prinzip der Kapselung sorgt dafür, dass andere Komponenten außerhalb der Adaptierbaren Komponente nicht direkt auf die Subkomponenten zugreifen können. Die Bezeichnung „Subkomponente“ drückt keinen semantischen Unterschied zu „Komponenten“ aus, sondern beschreibt lediglich die Kompositionsbeziehung von Subkomponenten innerhalb von Adaptierbaren Komponenten. Subkomponenten selbst können sowohl wiederum Adaptierbare Komponenten, als auch atomare Komponenten repräsentieren, wobei atomare Komponenten keine Subkomponenten enthalten. Auf diese Weise entsteht der bereits im Übersichtsabschnitt erwähnte hierarchische Charakter des Komponentenmodells (siehe Abbildung 3.3).

Eine Komponente besteht aus unterschiedlichen Artefakten, je nachdem zu welchem Zeitpunkt sie betrachtet wird (siehe Abbildung 3.4):

- Zur **Entwicklungszeit** besteht eine Komponente aus einer Menge von Quelltextdateien, Metainformationen abhängig von der Komponentenplattform (z. B. XML-Deskriptoren bei EJB) und zusätzlichen Ressourcen wie Programmbibliotheken und Daten. Der Quelltext steht zum Zeitpunkt der Komposition Adaptierbarer Komponenten nicht immer zur Verfügung, insbesondere wenn die Komponente von einem anderen Hersteller gekauft wurde.

- Zur **Installationszeit** besteht eine Komponente aus einer Menge von Binärcode (z. B. Java Bytecode), der durch Übersetzung der Quelltextdateien entsteht. Metainformationen und Ressourcen sind in veränderter oder unveränderter Form wie zur Entwicklungszeit vorhanden.

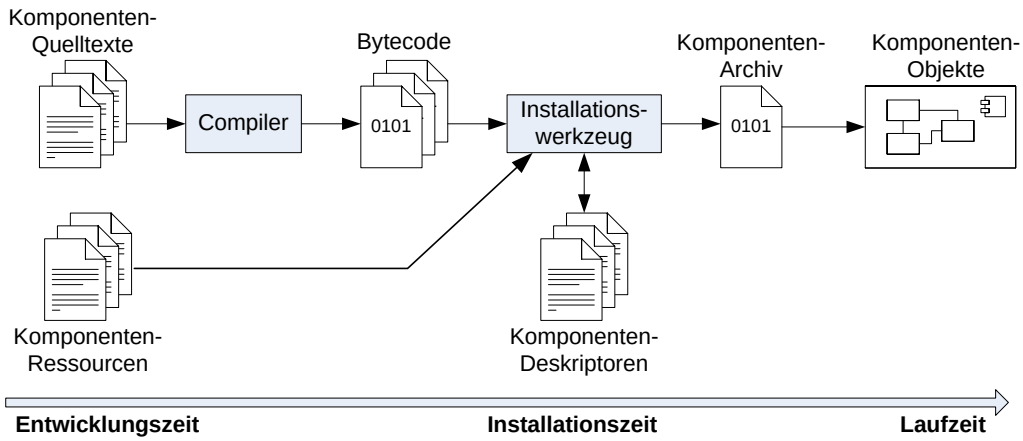

Abbildung 3.4: Artefakte von Komponenten während des Entwicklungsprozesses

- Zur **Laufzeit** besteht eine Komponente aus geladenem Programmcode und Daten, bei einer objektorientierten Programmiersprache wie Java oder C++ also aus einer Menge von Klassen, Interfaces und Objekten.

Diese Differenzierung kann bei der Modelltransformation (siehe Kapitel 4) und bei der Implementierung von Adaptions- und Aspektoperatoren (siehe Abschnitte 3.3.4 und 4.3.5) berücksichtigt und zur Optimierung verwendet werden.

Im Zusammenhang mit der Konfiguration einer Adaptierbaren Komponente (siehe Abschnitt 3.4) muss zwischen Subkomponententypen und Instanzen derselben [2] unterschieden werden. Diese Beziehung entspricht der Unterscheidung zwischen Klassen und Objekten in objektorientierten Programmiersprachen. Komponenteninstanzen existieren nur zur Laufzeit und werden aus Komponententypen erzeugt. Eine Instanz einer Adaptierbare Komponente enthält einen bestimmten Subkomponententyp maximal einmal, aber sie kann durchaus mehrere Instanzen dieses Typs gleichzeitig in einer Konfiguration verwenden. Eine Komponenteninstanz kann dabei Zustandsinformationen enthalten (siehe Abschnitt 3.6).

Definition 3.2. Ein **Komponententyp** ist ein Tupel $Komponententyp = (Name, \underline{Po}, \underline{P})$ mit *Name* als eindeutigem Bezeichner des Komponententyps, $\underline{Po} \subseteq \underline{Ports}$ als Menge aller *provided*- und *required*-Ports des Komponententyps, $\underline{P} \subseteq Parameter$ als Menge aller Komponentenparameter des Komponententyps und *Code* als Programmcode der Komponentenimplementierung.

$\underline{Komponententypen}$ bezeichnet die Menge aller Komponententypen, $\underline{Ports}$ die Menge aller Ports (siehe Definition 3.4) und *Parameter*die Menge aller Komponentenparameter (siehe Definition 3.7).

Definition 3.3. Eine **Komponenteninstanz** ist ein Tupel $Komponente = (Name, Typ)$ mit *Name* als eindeutigem Bezeichner der Komponenteninstanz und $Typ \in \underline{Komponententypen}$ als Komponententyp.

$\underline{Komponenten}$ bezeichnet die Menge aller Komponenteninstanzen.

[2] Der Begriff „Komponente“ wird in dieser Arbeit verwendet, wenn sowohl Komponententypen als auch Komponenteninstanzen gemeint sind.

Die Hilfsfunktion *fPorts* : *Komponente* → *Po* ermittelt die Menge aller *provided*- und *required*-Ports einer bestimmten Komponenteninstanz.

3.3.1.1 Ports und Methoden

Ports definieren die Kommunikationsschnittstelle von Komponenten, über die jede ein- und ausgehende Kommunikation[3] mit anderen Komponenten erfolgen muss. Angebotene Funktionalität wird dabei anderen Komponenten über *provided*-Ports zur Verfügung gestellt. Wenn eine Komponente Funktionen einer anderen Komponente aufrufen will, muss sie dazu einen entsprechenden *required*-Port deklarieren und benutzen. Zur Laufzeit verknüpfen Verbindungen (siehe Unterabschnitt 3.3.3) *required*- und *provided*-Ports miteinander und sorgen dafür, dass Methodenaufrufe an die richtige Komponente weitergeleitet werden. Dieses Prinzip hat den großen Vorteil, dass die Verknüpfung von Komponenten getrennt von der sonstigen Programmlogik erfolgt (*Separation of Concerns*).

Komponenten können beliebig viele *required*- und *provided*-Ports verwenden. Allerdings kann durch die Zielkomponentenplattform die Anzahl der *provided*-Ports auf eins beschränkt werden. Ein spezieller *provided*-Port mit dem Namen *Default* – als *Komponentenport* bezeichnet – wird von jeder Adaptierbaren Komponente unterstützt. Dieser Port dient zum Zugriff (Lesen und Ändern) auf Komponentenparameter und alle anderen Ports. Optional und in Abhängigkeit des verwendeten plattformspezifischen Modells (siehe Abschnitt 4.1) können auch weitergehende Funktionen zur Manipulation einer Adaptierbaren Komponente in Form eines einfachen Metaobject Protocols (siehe Unterabschnitt 2.8.3) angeboten werden.

Jeder Port besitzt einen eindeutigen Namen im Namensraum der zugehörigen Komponente und einen bestimmten Typ, der durch die implementierte Schnittstelle definiert ist.

Definition 3.4. Ein **Komponentenport** ist ein Tupel $Port = (Name, Typ, Art)$ mit *Name* als eindeutigem Bezeichner des Ports, $Typ \in \underline{Typen}$ als Bezeichner der Portschnittstelle und $Art \in \{req11, req1n, req01, req0n, prov\}$ als Art des Ports.

Typen bezeichnet die Menge aller Porttypen und *Ports* die Menge aller Komponentenports.

Klassifikation von Ports Es werden vier verschiedene Arten von *required*-Ports unterschieden:

Einfach, optional Ports akzeptieren genau eine Verbindung zu einem passenden *provided*-Port. Die Komponente kann aber auch ohne eine Verbindung arbeiten ($Art = req01$, im UML-Komponentendiagramm mit der Multiplizitätsangabe `0..1` gekennzeichnet).

Einfach, obligatorisch Ports akzeptieren genau eine Verbindung zu einem passenden *provided*-Port. Ohne diese Verbindung kann die zugehörige Komponente nicht arbeiten ($Art = req11$, im UML-Komponentendiagramm mit der Multiplizitätsangabe `1` gekennzeichnet).

Mehrfach, optional Ports akzeptieren keine, eine oder mehrere Verbindungen zu passenden *provided*-Ports ($Art = req0n$, im UML-Komponentendiagramm mit der Multiplizitätsangabe `*` gekennzeichnet).

[3] Dazu zählen insbesondere Methodenaufrufe, aber auch beliebige Datenübertragungen.

Mehrfach, obligatorisch Ports akzeptieren eine oder mehrere Verbindungen zu passenden *provided*-Ports. Ohne mindestens eine Verbindung kann die zugehörige Komponente nicht arbeiten ($Art = req1n$, im UML-Komponentendiagramm mit der Multiplizitätsangabe `1..*` gekennzeichnet).

Die verschiedenen Arten von *required*-Ports korrespondieren mit den unterschiedlichen Kardinalitäten (0, 1, *), die bei Assoziationen innerhalb von UML-Klassendiagrammen benutzt werden.

Provided-Ports ($Typ = prov$) werden dagegen nicht weiter klassifiziert. Sie können prinzipiell mit allen vier Arten von *required*-Ports verbunden werden, sofern die Typen kompatibel sind (siehe Unterabschnitt 3.3.3) und die jeweilige maximale Anzahl von Verbindungen eingehalten wird.

Port-Typen und Methodensignaturen Der Typ eines Ports wird durch seine Schnittstelle definiert. Ein Typ besitzt einen eindeutigen Namen, eine optionale Menge von Super-Typen[4] und eine Menge von Methoden mit bestimmten Signaturen, die die operationale Schnittstelle definieren. Das Tupel (Methodenname, Signatur) ist eindeutig innerhalb einer Schnittstelle.

Eine Methode[5] realisiert eine Funktion $F : Z \times I_1 \times \ldots \times I_n \rightarrow Z' \times E \times O_1 \times \ldots \times O_m$ mit Z, Z' als interne Zustände der Komponente vor bzw. nach dem Funktionsaufruf, $n, m \in \mathbb{N}$ als Anzahl der Eingabe- bzw. Ausgabeparameter, I_i als getypte Menge der Eingabeparameter, E als Menge der möglichen Exceptions und O_i als Menge der Ausgabeparameter. Die getypten Mengen der Ein- und Ausgabeparameter definieren dabei die *Methodensignatur*. Die Mengen der Ein- und Ausgabeparameter sind nicht immer disjunkt, d. h. ein Parameter kann auch sowohl Ein- als auch Ausgabeparameter sein. Der Typ von Ein- und Ausgabeparametern legt auch fest, ob die Übergabe als Wert (*by value*) oder als Referenz (*by reference*) erfolgt.

In Java z. B. werden Methodenparameter immer als Referenz übergeben, außer bei den primitiven Datentypen `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`. Durch die standardmäßige Parameterübergabe als Referenz kann in Java prinzipiell jeder Eingabeparameter (außer die erwähnten primitiven Datentypen) auch zur Ausgabe verwendet werden, wenn das zur Referenz gehörende Objekt innerhalb der Methode manipuliert wird. Die Methodensignatur genügt hier nicht zur Unterscheidung. In C++ dagegen können mit dem Schlüsselwort `const` in der Methodensignatur gekennzeichnete Eingabeparameter nicht verändert werden und sind damit keine Ausgabeparameter. Der Rückgabewert einer Methode ist ausschließlich ein Ausgabeparameter.

Mit Hilfe der Funktion F einer Methode können die folgenden Eigenschaften definiert werden:

Definition 3.5. Wenn $\forall i_1 \in I_1 \ldots i_n \in I_n, z \in Z : \; F(z, i_1 \ldots i_n) = (z, o_1 \ldots o_m)$ gilt, d. h. wenn der Zustand bei beliebigen Eingaben einer Methode unverändert bleibt, dann hat die Funktion keinen Einfluss auf den Zustand der Komponente und ist **zustandsunveränderlich**.

Wenn $\forall i_1 \in I_1 \ldots i_n \in I_n : \; \forall z_1, z_2 \in Z : \; F(z_1, i_1 \ldots i_n) = F(z_2, i_1 \ldots i_n)$ gilt, d. h. wenn eine Methode unabhängig vom Zustand bei gleicher Eingabe das gleiche Ergebnis liefert, dann

[4] Ein Super-Typ wird auch als Basisklasse oder Oberklasse bezeichnet.

[5] Die Begriffe *Methode*, *Elementfunktion* und *Memberfunktion* werden in der Literatur oft als Synonyme verwendet.

ist die Funktion unabhängig vom Zustand der Komponente und wird als **zustandsunabhängig** bezeichnet.

Definition 3.6. Wenn alle Methoden einer Komponente zustandsunabhängig sind, dann wird die Komponente als **zustandslos** bezeichnet. Anderenfalls ist die Komponente **zustandsbehaftet**.

3.3.1.2 Komponentenparameter

Komponentenparameter dienen zur Anpassung einer Komponente an bestimmte Anwendungsszenarien und Umgebungsbedingungen. Sie sind damit das Mittel zur Steuerung der Adaption in dem Komponentenmodell (siehe auch Abschnitt 3.4). Jeder Komponentenparameter besitzt innerhalb der zugehörigen Komponente einen eindeutigen Namen, der zum Zugriff benötigt wird. Werte von Parametern können prinzipiell zu unterschiedlichen Zeitpunkten innerhalb des Lebenszyklus einer Komponente – von der Entwicklung über Installation und Instanziierung bis zur Laufzeit (siehe auch Abschnitt 4.1) – festgelegt und geändert werden. Durch das gewählte plattformspezifische Modell kann die Änderung von Parametern allerdings auf bestimmte Zeiten begrenzt werden. So kann z. B. bei einem Modell ohne Unterstützung von Laufzeit-Rekonfiguration das Ändern von Parametern zur Laufzeit verboten werden.

Definition 3.7. Ein **Komponentenparameter** ist ein Tupel $Parameter = (Name, Typ, \underline{Wertebereich}, Default)$ mit $Name$ als eindeutigem Bezeichner des Parameters, $Typ \in \{Real, Integer, Enumeration, Boolean, Map, Array, Any\}$ als Typ des Parameters, $\underline{Wertebereich}$ als Definitionsmenge des Parameters[6] in Abhängigkeit des Parametertyps und $Default \in \underline{Wertebereich} \cup \epsilon$ als optionaler voreingestellter Wert dieses Parameters, wenn kein anderer Wert zugewiesen wurde.

$\underline{Parameter}$ ist die Menge aller Komponentenparameter.

Definition 3.8. Eine **Wertzuweisung eines Komponentenparameters** ist ein Tupel $Parameterwert = (Parameter, Wert)$ mit $Parameter \in \underline{Parameter}$ und $Wert \in \underline{Wertebereich}$ als gültiger Wert in Abhängigkeit des Parametertyps und des Wertebereichs. $\underline{Parameterwerte}$ bezeichnet die Menge aller Wertzuweisungen bei Komponentenparametern.

Es gibt verschiedene Typen von Komponentenparametern, die jeweils für unterschiedliche Anwendungsfälle besonders geeignet sind (siehe Unterabschnitt 3.4.6). Prinzipiell wird durch einen bestimmten Typ aber kein Anwendungsfall ausgeschlossen. Die im Folgenden angegebenen Anwendungsfälle sind daher nur Beispiele.

Enumeration-Parameter besitzen eine endliche Menge von möglichen Werten, wovon genau ein Wert ausgewählt werden kann. Die Definition des Wertebereichs enthält die Menge von möglichen Strings. Damit eignen sich diese Parameter besonders für eine direkte Abbildung eines bestimmten Wertes auf eine bestimmte Konfiguration einer Adaptierbaren Komponente.

[6] Damit kann die Definitionsmenge eines bestimmten Parametertyps weiter eingeschränkt werden, z. B. ein Wertebereich von 0–10 beim Typ Integer.

Numerische Parameter (Integer oder Real) repräsentieren einen beliebigen numerischen Wert (Integer: *Wertebereich* = $\mathbb{N}$; Real: *Wertebereich* = $\mathbb{R}$). Durch die optionale Angabe eines Wertebereichs bestehend aus oberer und unterer Schranke können die gültigen Parameterwerte auf dieses Intervall eingeschränkt werden. Numerische Parameter können dafür verwendet werden, einen Wertebereich auf eine bestimmte Konfiguration der Adaptierbaren Komponente abzubilden. Alternativ kann damit die Anzahl der Anwendungen einer Mehrfach-Variation (siehe Abschnitt 3.4) festgelegt werden.

Map-Parameter definieren eine Menge von Name-Wert-Paaren. Mit der Definition des Wertebereichs werden die gültigen Namen und die zugeordneten Typen des Wertes festgelegt. Map-Parameter werden insbesondere zur Parametrisierung von Variationen (siehe Unterabschnitt 3.4.4) verwendet. Die einzelnen Werte werden in diesem Fall bestimmten Parametern von Subkomponenten zugewiesen, die durch die Variation hinzugekommen sind. Die zugeordnete Variation wird nicht angewendet, wenn der Map-Parameter nicht definiert ist (`null`).

Array-Parameter repräsentieren eine Menge von Map-Parametern. Sie werden zur Parametrisierung von Mehrfach-Variationen (siehe Unterabschnitt 3.4.4) verwendet. Jeder Map-Parameter des Arrays wird dabei genau einer Mehrfach-Variation zugeordnet.

Beliebige Parameter sind alle Parameter, die sich nicht in die ersten drei Klassen einordnen lassen. Diese Parameter werden entweder auf Parameter von Subkomponenten abgebildet oder zur Auswahl von Variationen verwendet. Dabei wird die entsprechende Variation nur ausgewählt, wenn dem Parameter ein beliebiger Wert (außer `null`) zugewiesen wird.

Jedem Parameter kann ein Standardwert zugewiesen werden, der zur Anwendung kommt, wenn kein anderer Wert explizit zugewiesen wurde. Diese Möglichkeit kann z. B. angewendet werden, um eine Standardkonfiguration einer Adaptierbaren Komponente auszuwählen.

3.3.2 Glue-Code

Glue-Code ist ein optionaler Bestandteil des Komponentenmodells. Er wird immer dann benötigt, wenn das einfache Verbinden von Subkomponenten sowie die Anwendung von Adaptions- und Aspektoperatoren zur Erstellung einer Adaptierbaren Komponente nicht ausreichen. In diesem Fall kann mit Hilfe von Glue-Code fehlende Funktionalität implementiert werden.

Glue-Code unterscheidet sich im Aufbau, in den Eigenschaften und in der Verwendung nicht von anderen Subkomponenten (siehe Unterabschnitt 3.3.1), d. h. er kommuniziert ebenso über Ports (siehe Unterunterabschnitt 3.3.1.1) mit der Außenwelt. Der einzige Unterschied besteht darin, dass Glue-Code explizit für eine bestimmte Adaptierbare Komponente entwickelt wird und in der Regel auch nicht in anderen Komponenten wiederverwendet werden kann. Subkomponenten können und sollen dagegen nicht nur in einer Anwendung verwendet werden.

Da sich Glue-Code in seinem Verhalten nicht von Subkomponenten unterscheidet, werden auch keine speziellen Mechanismen zur Integration innerhalb von Adaptierbaren Komponenten benötigt. Glue-Code wird ganz normal über Verbindungen mit anderen Subkomponenten bzw. mit externen Ports verknüpft.

3.3.3 Verbindungen

Verbindungen[7] definieren, wie die durch *provided*-Ports angebotene Funktionalität einer Adaptierbaren Komponente durch die Zusammenarbeit von Subkomponenten realisiert wird. Unterschiedliche interne Verbindungen zwischen Subkomponenten sind damit ein Mittel zur Realisierung der Adaption.

Es werden prinzipiell zwei Arten von Verbindungen unterschieden: Verbindungen zwischen *required*- und *provided*-Ports von Subkomponenten[8] und Verbindungen zwischen externen Ports einer Adaptierbaren Komponente und internen Ports von Subkomponenten[9]. Die Menge aller Verbindungen zwischen internen und externen Ports wird als **Interface Binding** bezeichnet.

3.3.3.1 Verbindungen zwischen Subkomponenten

Eine Verbindung besteht immer zwischen genau einem *provided*- und einem *required*-Port und garantiert, dass beim Methodenaufruf an einem *required*-Port die entsprechende Methode in der Komponenteninstanz des verbundenen *provided*-Ports aufgerufen wird. Es ist dem Komponentencontainer[10] überlassen wie in Abhängigkeit der Zielkomponentenplattform eine Verbindung realisiert wird.

Da alle Ports typisiert sind (siehe Unterunterabschnitt 3.3.1.1), d. h. sie implementieren eine bestimmte Schnittstelle, können Verbindungen nur zwischen typkompatiblen Ports hergestellt werden. Zwei Ports sind auf jeden Fall kompatibel, wenn sie den gleichen Typ besitzen. Da dies aber keine notwendige Bedingung für Verbindungen ist, wird im Folgenden eine Typ-Relation definiert, die zur allgemeinen Beschreibung der Typkompatibilität verwendet wird. Damit wird bereits ein einfacher Adaptionsoperator (siehe Unterabschnitt 3.3.4) realisiert.

Definition 3.9. Eine **Typ-Relation** $A \xrightarrow{T} B$ mit $A, B \in \underline{Typen}$ über der Menge aller Typen legt fest, dass A ein Subtyp von B ist, d. h. A übernimmt alle Eigenschaften von B und erweitert sie durch neue Methoden. Damit kann nach dem Liskovschen Substitutionsprinzip [LW93] der Typ B durch den Typ A ersetzt werden. Die Typ-Relation ist reflexiv, antisymmetrisch und transitiv.

Die Kompatibilität kann als eine gerichtete Assoziation zwischen genau zwei Partnern (z. B. Port, Methode, Parameter) aufgefasst werden:

- Der **Anbieter** stellt ein Artefakt (z. B. Port oder Methode) mit bestimmten Eigenschaften zur Verfügung.
- Der **Benutzer** erwartet ein Artefakt, das bestimmte Eigenschaften erfüllt.

Angewendet auf Verbindungen stellt ein *provided*-Port einen Anbieter und ein *required*-Port einen Benutzer dar. Ein *required*-Port mit dem Typ A ist damit genau dann kompatibel zu einem *provided*-Port mit dem Typ B, wenn gilt: $B \xrightarrow{T} A$. Das bedeutet, die Ports sind kompatibel, wenn sie den gleichen Typ besitzen oder wenn B ein Subtyp von A ist, also eine

[7] In der UML-Terminologie werden Verbindungen als *Connectors* bezeichnet.
[8] in UML als *Assembly Connectors* bezeichnet
[9] in UML als *Delegation Connectors* bezeichnet
[10] Die Laufzeitumgebung von Komponenten wird als Komponentencontainer bezeichnet.

von A abgeleitete Schnittstelle implementiert. In objektorientierten Programmiersprachen wird die Typrelation durch eine Vererbungsbeziehung ausgedrückt, in Java z. B. mit dem Schlüsselwort `extends`.

Definition 3.10. Eine **Verbindung** zwischen zwei Ports wird definiert durch ein Tupel $Verbindung = (pK, pP, rK, rP)$ mit $pK \in \underline{Komponenten}$ als Komponenteninstanz des *provided*-Ports, $pP = (Name_p, Typ_p, Art_p) \in \underline{Ports}$, $Art_p \in \{prov\}$, $pP \in fPorts(pK)$ als *provided*-Port, der in der Menge der Ports des Komponententyps von pK enthalten ist, $rK \in \underline{Komponenten}$ als Komponenteninstanz des *required*-Ports und $rP = (Name_r, Typ_r, Art_r) \in \underline{Ports}$, $Art_p \in \{req01, req0n, req11, req1n\}$, $rP \in fPorts(rK)$ als *required*-Port, der in der Menge der Ports des Komponententyps von rK enthalten ist.

Weiterhin gilt $Typ_p \xrightarrow{T} Typ_r$, d. h. der Typ des *provided*-Ports ist typkompatibel zum Typ des *required*-Ports.

$\underline{Verbindungen}$ bezeichnet die Menge aller Verbindungen zwischen *provided*- und *required*-Ports.

3.3.3.2 Verbindungen zwischen internen und externen Ports

Das hierarchische Komponentenmodell der Adaptierbaren Komponenten erfordert eine Verknüpfung von Ports der zusammengesetzten Komponente (bezeichnet als externe Ports) mit passenden Ports eingeschlossener Subkomponenten (bezeichnet als interne Ports). Auf diese Weise wird die Funktionalität einer Komponente auf einer Ebene in der Komponentenhierarchie durch die Zusammenarbeit ein oder mehrerer Komponenten der nächst tieferen Ebene realisiert (siehe Abbildung 3.3).

Für die Verknüpfung von externen und internen Ports werden zwei Varianten unterstützt:

Port-Verknüpfung Ein externer *provided*- bzw. *required*-Port wird mit einem typkompatiblen internen *provided*- bzw. *required*-Port einer Subkomponenteninstanz verbunden.

Methoden-Verknüpfung Alle Methoden der Schnittstelle eines externen *provided*-Ports werden mit Methoden von internen *provided*-Ports verknüpft. Dabei gilt, dass die Methodensignaturen von miteinander verknüpften Methoden kompatibel sein müssen. Die Namen der Methoden dürfen sich dabei auch unterscheiden. Mit Methoden-Verknüpfungen können die einzelnen Methoden einer externen Schnittstelle auf unterschiedliche Subkomponenten abgebildet werden.

Definition 3.11. Eine **Port-Verknüpfung** zwischen einem externen und einem internen Port wird definiert durch ein Tupel $Verknuepfung = (eK, eP, iK, iP)$ mit eK als Komponententyp einer Adaptierbaren Komponente, $eP = (Name_e, Typ_e, Art_e) \in \underline{Ports}$, $eP \in \underline{Po}$ als externer Port, der in der Menge der Ports des Komponententyps eK enthalten ist, $iK \in \underline{Komponenten}$ als Komponenteninstanz des *required*-Ports, die in der Menge der Komponenteninstanzen des Komponententyps eK enthalten ist und $iP = (Name_i, Typ_i, Art_i) \in \underline{Ports}$, $iP \in fPorts(iK)$ als interner Port, der in der Menge der Ports des Komponententyps von iK enthalten ist.

Weiterhin gilt $Art_e = Art_i$, $Art_i = prov \implies Typ_i \xrightarrow{T} Typ_e$ und $Art_i \neq prov \implies Typ_e \xrightarrow{T} Typ_i$, d. h. die Arten von internem und externem Port sind gleich, bei *provided*-Ports ist der Typ des internen Ports typkompatibel zum externen Port und bei *required*-Ports ist der Typ des externen Ports typkompatibel zum internen Port.

Für die Realisierung von Methoden-Verknüpfungen muss Kompatibilität auf der Ebene von Methodensignaturen definiert werden. Diese Definitionen sich auch Grundlage für die im nächsten Abschnitt beschriebenen Adaptionsoperatoren. Ähnliche Fragestellungen werden mit Signaturmorphismen auch im Bereich der formalen Spezifikationsmethoden untersucht, z. B. in ASL [Wir86].

Methodensignaturen sind auf jedem Fall kompatibel, wenn Ein- und Ausgabeparameter sowie Exceptions in Anzahl, Typ und Reihenfolge übereinstimmen, die Methodensignaturen also identisch sind. Anderenfalls müssen Typ-Relationen der Ein- und Ausgabeparameter berücksichtigt werden. Es gelten die folgenden Regeln:

- Für jeden Eingabeparameter $e_1 \in I_1 \dots e_n \in I_n$ einer Methode des externen Ports[11] gilt für den zugeordneten Parameter $i_1 \in I_1 \dots i_n \in I_n$ der zugeordneten Methode des internen Ports $\forall_{j=1}^{n} e_j \xrightarrow{T} i_j$. Auf diese Weise kann ein Eingabeparameter der Methode des externen Ports immer dem Eingabeparameter der Methode des internen Ports zugewiesen werden.

- Für jeden Ausgabeparameter $e_1 \in O_1 \dots e_m \in O_m$ einer Methode des externen Ports gilt für den zugeordneten Parameter $i_1 \in O_1 \dots i_m \in O_m$ der zugeordneten Methode des internen Ports $\forall_{j=1}^{m} i_j \xrightarrow{T} e_j$. Auf diese Weise kann ein Ausgabeparameter der Methode des internen Ports immer dem Ausgabeparameter der Methode des externen Ports zugewiesen werden.

- Für jede Exception $i \in E$ aus der Menge der möglichen Exceptions einer Methode des internen Ports gibt es mindestens eine Exception $e \in E$ aus der Menge der möglichen Exceptions des externen Ports für die $i \xrightarrow{T} e$ gilt. Damit kann eine mögliche Exception des internen Ports immer einer Exception des externen Ports zugewiesen werden.

Einige objektorientierte Programmiersprachen, wie z. B. Java, C++ oder C# unterscheiden zwischen primitiven Datentypen und Objekttypen. Zwischen primitiven Datentypen werden dabei keine Vererbungsbeziehungen definiert, da es sich ja nicht um Objekte handelt. Zur Beschreibung der Kompatibilität von Methodensignaturen ist es jedoch sinnvoll, die Typ-Relation (siehe Definition 3.9) auch für einige primitive Datentypen zu definieren. So kann z. B. ein Java `byte`-Parameter mit einem Wertebereich von $[-128 \dots 127]$ immer auch einem `short`-Parameter mit einem Wertebereich von $[-32.768 \dots 32.767]$ zugewiesen werden. Für die Typ-Relation bei primitiven Datentypen zur Beschreibung der Kompatibilität von Methodensignaturen gilt, dass der Wertebereich des als Subtyp betrachteten Parameters eine Teilmenge des Wertebereichs des anderen Parameters bilden muss.

Auf diese Weise ergeben sich für Java die folgenden Typ-Relationen bei primitiven Datentypen:

$\texttt{byte} \xrightarrow{T} \texttt{short} \xrightarrow{T} \texttt{int} \xrightarrow{T} \texttt{long}$

$\texttt{float} \xrightarrow{T} \texttt{double}$

[11] Der externe *provided*-Port kann wie ein *required*-Port einer Subkomponente betrachtet werden.

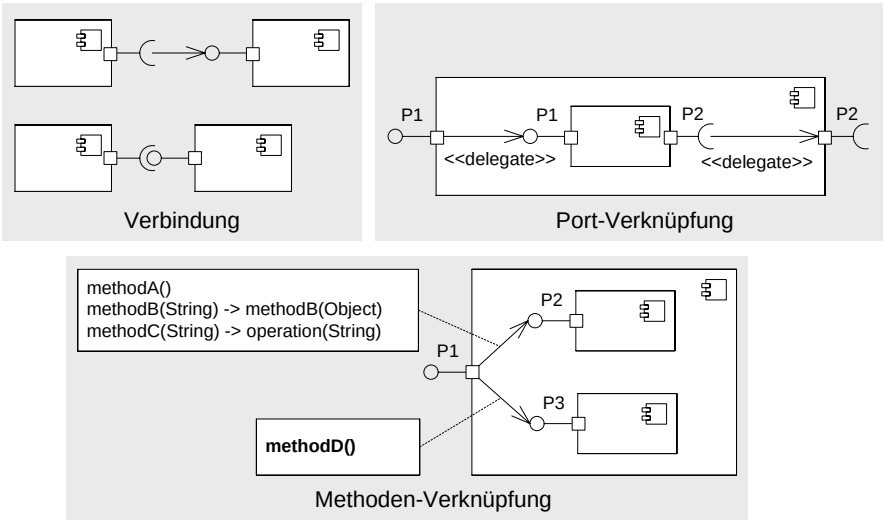

Abbildung 3.5: UML-Darstellung von Verbindungen und Verknüpfungen

Entsprechend ergeben sich für C++ die folgenden Beziehungen bei primitiven Datentypen:

`unsigned char` $\xrightarrow{T}$ `unsigned int` $\xrightarrow{T}$ `unsigned long`

`char` $\xrightarrow{T}$ `short int` $\xrightarrow{T}$ `int` $\xrightarrow{T}$ `long`

`float` $\xrightarrow{T}$ `double` $\xrightarrow{T}$ `long double`

Definition 3.12. Eine **Methoden-Verknüpfung** zwischen einer Methode eines externen und einer Methode eines internen Ports wird definiert durch ein Tupel $\mathit{Verknuepfung} = (eK, eP, eS, iK, iP, iS)$ mit eK, eP, iK, iP wie bei Definition 3.11, eS als Methodensignatur des externen Ports eP und iS als Methodensignatur des internen Ports ip.

Weiterhin gilt $Art_e = Art_i = prov$ und $iS \xrightarrow{T} eS$, d. h. der interne und externe Port sind *provided*-Ports und die Methodensignaturen sind typkompatibel.

Ein externer *provided*-Port eP ist **vollständig verknüpft**, wenn zu jeder Methode von eP eine Methoden-Verknüpfung definiert wurde.

Abbildung 3.5 zeigt die graphischen Darstellungen der verschiedenen Verbindungen und Verknüpfungen. Bis auf die Darstellung von Methoden-Verknüpfungen entsprechen sie der UML-Spezifikation [uml05b]. Methoden-Verknüpfungen sind in der UML nicht definiert und werden daher in Form von Delegationen mit zugeordneten Assoziationsklassen dargestellt, um bei einer Delegation anzugeben, welche Methoden verknüpft werden. Für jede verknüpfte Methode mit der zugehörigen Methodensignatur ist genau ein Eintrag in der Assoziationsklasse enthalten.

Standardmäßig besitzen Verbindungen eine synchrone Aufrufsemantik, d. h. ein Methodenaufruf blockiert die aufrufende Komponente bis die Methode der aufgerufenen Komponente vollständig abgearbeitet und ein optionales Ergebnis zurückgeliefert wurde. Dies entspricht der Semantik bei Methodenaufrufen in vielen Programmiersprachen, wie z. B. Java und C++.

Verbindungen können in plattformspezifischen Modellen (siehe Abschnitt 4.1) mit weiteren Attributen versehen werden, die auch die Aufrufsemantik beeinflussen können (z. B.

asynchrone Methodenaufrufe). Diese Attribute müssen dann entweder von Werkzeugen zum Entwicklungs- bzw. Installationszeitpunkt oder vom Komponentencontainer zur Laufzeit berücksichtigt werden. Beispiele für solche speziellen Verbindungen sind Stromverbindungen in COMQUAD (siehe Abschnitt 5.6).

3.3.4 Adaptionsoperatoren

Adaptionsoperatoren sind optionale Bestandteile von Adaptierbaren Komponenten. Sie werden verwendet, um definierte Änderungen an Subkomponententypen vorzunehmen. Diese Änderungen können prinzipiell alle Elemente eines Subkomponententyps (siehe Definition 3.2) betreffen, d. h. die Ports, die Komponentenparameter und den Programmcode.

Definition 3.13. Ein **Adaptionsoperator** definiert eine Funktion
$AdOp : \underline{Komponententypen} \rightarrow \underline{Komponententypen}$, die genau einen Komponententyp als Argument erhält und genau einen angepassten Komponententyp als Resultat zurückliefert.

Existierende Komponenten können nicht immer ohne Änderungen zu einer Adaptierbaren Komponente kombiniert werden. Ursache dafür sind meist inkompatible Komponentenports, die oft dadurch entstehen, dass die Komponenten von unterschiedlichen Entwicklern stammen. Zwei Ports sind inkompatibel, wenn keine Verbindung zwischen ihnen hergestellt werden kann (siehe Unterabschnitt 3.3.3). Adaptionsoperatoren bieten in solchen Fällen die Möglichkeit, die Inkompatibilitäten zu überwinden, ohne die Komponenten direkt anpassen zu müssen. Damit sind Adaptionsoperatoren ein wichtiges Hilfsmittel, um die Wiederverwendung von Komponenten zu steigern. Dabei wird vorausgesetzt, dass die erwartete und tatsächlich von einer Komponente zur Verfügung gestellte Funktionalität übereinstimmt. Dies kann nur von einem Entwickler beurteilt werden und nicht automatisch entschieden werden.

Für die Entwicklung eines Adaptionsoperators sind im Allgemeinen die angebotenen und die für die Zusammenarbeit benötigten Eigenschaften einer Komponente bekannt. Der gesuchte Adaptionsoperator muss genau die Abbildung von angebotenen auf benötigte Eigenschaften realisieren. Als Ergebnis erzeugt ein Adaptionsoperator einen neuen Komponententyp, der wie jede andere Subkomponente innerhalb einer Adaptierbaren Komponente verwendet werden kann.

3.3.4.1 Ausgewählte Ursachen für Inkompatibilitäten zwischen Ports

Anpassungen zwischen inkompatiblen Schnittstellen sind durch Adaptionsoperatoren in Form von speziell geschriebenen Programmen möglich. Hier sollen jedoch die wesentlichen Eigenschaften solcher Programme deklarativ beschrieben werden, um die Adaption auf einer höheren Abstraktionsstufe handhaben zu können und damit die Entwicklung von Adaptionsoperatoren zu vereinfachen. Die folgenden Ursachen für Inkompatibilitäten von Ports sollen im Folgenden näher betrachtet werden:

- Alle Methodennamen und Methodensignaturen der beiden Portschnittstellen stimmen überein, aber die Namen der Typen bzw. die Menge der Supertypen sind verschieden.

  Beispiel 3.1.
  ```
  public interface A {
          public byte[] encrypt(byte[] data);
          public byte[] decrypt(byte[] data);
  }
  ```

```
public interface B {
        public byte[] encrypt(byte[] data);
        public byte[] decrypt(byte[] data);
}
```

- Mindestens eine Methode hat unterschiedliche Namen bei sonst gleicher Methodensignatur in den beiden Portschnittstellen.

  Beispiel 3.2.
  ```
  public interface A {
          public byte[] encryptData(byte[] data);
  }
  public interface B {
          public byte[] encrypt(byte[] data);
  }
  ```

- Mindestens eine Methode hat unterschiedliche Methodensignaturen in beiden Portschnittstellen. Die Ein- und Ausgabeparameter der Methode sind aber kompatibel oder können durch eine Funktion aufeinander abgebildet werden.

  Beispiel 3.3.
  ```
  public interface A {
          void setKey(String key)
  }
  public interface B {
          void setKey(byte[] key)
  }
  ```

- Für mindestens eine Methode des einen Zielports gibt es keine Entsprechung beim Ausgangsport. Entweder ist die entsprechende Funktionalität nicht vorhanden oder der Aufruf einer Methode im Zielport entspricht dem Aufruf einer Sequenz von Methoden im Ausgangsport.

  Beispiel 3.4.
  ```
  public interface A {
          public byte[] encrypt(String key, byte[] data);
  }
  public interface B {
          public void setKey(String key);
          public byte[] encrypt(byte[] data);
  }
  ```

Diese Fälle können auch beliebig miteinander kombiniert auftreten.

3.3.4.2 Beschreibung von Adaptionsoperatoren durch Adaptionsschritte

Entsprechend der gerade betrachteten Ursachen für die Inkompatibilität von Ports werden im Folgenden eine Reihe von **Adaptionsschritten** definiert, die zur Beschreibung eines Adaptionsoperators verwendet werden können. Die Adaptionsschritte legen fest, welche Änderungen an einem *provided*-Port vorgenommen werden müssen. Sie legen jedoch *nicht* fest, wie und wann diese Änderungen durchgeführt werden müssen. Prinzipiell könnten die gleichen Adaptionsschritte auch für *required*-Ports angewendet werden. Da es aber zum gleichen Ergebnis führt, wenn bei einer Verbindung entweder der *provided*- oder der *required*-Port angepasst wird, wurde die Anwendbarkeit von Adaptionsschritten auf *provided*-Ports beschränkt. Ein

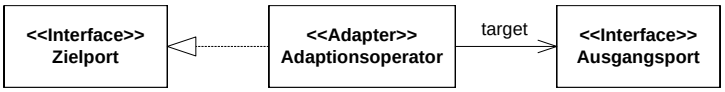

Abbildung 3.6: Realisierung eines Adaptionsoperators durch einen Adapter

Adaptionsoperator fasst eine geordnete Menge von Adaptionsschritten in einer Einheit zusammen.

Ausgangspunkt eines Adaptionsoperators ist ein *provided*-Port einer Komponente, der im Folgenden als *Ausgangsport* bezeichnet wird. Das Ergebnis des Adaptionsoperators, also der adaptierte Port, wird als *Zielport* bezeichnet. Die Adaptionsschritte beschreiben, wie man den Ausgangsport schrittweise in den Zielport transformiert. Zur Veranschaulichung kann man sich dabei vorstellen, dass durch die Adaptionsschritte ein spezieller Adapter (*Object Adapter*-Muster aus [GHJV95]) erzeugt wird, der Methodenaufrufe beim Typ des Zielport auf Methodenaufrufe beim Typ des Ausgangsports abbildet (siehe Abbildung 3.6).

Adapter sind aber nur eine mögliche Implementierung von Adaptionsoperatoren. Auf der Modellebene wird bewusst von Implementierungsdetails abstrahiert, d. h. Adaptionsoperatoren beschreiben ausschließlich *was* adaptiert werden soll, aber nicht *wie* die Implementierung erfolgen muss. Die Implementierung wird erst durch plattformspezifische Modelle (siehe Abschnitt 4.1) definiert. Durch diese Vorgehensweise kann je nach Einsatzgebiet der Adaptierbaren Komponente eine optimierte Implementierungsstrategie verwendet werden, ohne dass die Adaptionsbeschreibung angepasst werden muss. Verschiedene Implementierungsstrategien von Adaptionsoperatoren werden in Unterabschnitt 4.3.5 vorgestellt.

Umbenennung des Porttyps Wenn alle Methodennamen, die zugehörigen Methodensignaturen und die Menge der Supertypen des Ausgangs- und Zielports übereinstimmen, kann der Porttyp umbenannt werden. Jeder Aufruf einer Methode des Zielports wird damit auf die entsprechende Methode des Ausgangsports abgebildet. Der Typ des Zielport muss bereits an einer anderen Stelle definiert worden sein (z. B. als Java-Interface).

Der Adaptionsschritt wird wie folgt definiert: *renameType(oldType, newType)*

Umbenennung einer Methode Wenn die Signatur einer Methode im Ausgangs- und Zielport übereinstimmt bzw. kompatibel zueinander ist und sich lediglich der Methodenname unterscheidet, wird mit der Umbenennung eine Zuordnung dieser Methoden realisiert, wie sie in ähnlicher Weise auch schon in Unterabschnitt 3.3.3 für die Methoden-Verknüpfung beschrieben wurde. Ein Aufruf der Methode mit dem neuen Namen führt zum gleichen Resultat wie zuvor ein Aufruf mit dem alten Namen.

Der Adaptionsschritt wird wie folgt definiert: *renameMethod(oldMethod, newMethod)*

Hinzufügen einer Methode zu einem Komponentenport Zusammen mit der neuen Methode wird auch neue Funktionalität zum Ausgangsport hinzugefügt. Der Methodenname zusammen mit der Signatur darf zuvor nicht im Ausgangsport enthalten sein. Die neue Methode kann auf zwei Wegen hinzugefügt werden:

- Die Implementierung der Methode wird über Quelltext in der Programmiersprache der Komponente definiert. Dieser Quelltext darf nur den Methodenrumpf enthalten und

lediglich Methoden des Ausgangsports aufrufen. Den in der neuen Methode definierten Ausgabeparametern müssen im Quelltext Werte zugewiesen werden.

Der Adaptionsschritt wird wie folgt definiert: *addMethod(Signatur, Programmcode)*

- Die neue Methode wird über eine Aufrufsequenz von existierenden Methoden des Ausgangsports definiert. Die Rückgabewerte der aufgerufenen Methoden, mit Ausnahme der letzten, werden nicht berücksichtigt. Sofern für die neue Methode Ausgabeparameter definiert wurden, müssen sie mit den Ausgabeparametern der letzten aufgerufenen Methode in der Sequenz übereinstimmen. Als Eingabeparameter können nur Parameter der neuen Methode verwendet werden.

  Der Adaptionsschritt wird wie folgt definiert: *addMethod(Signatur, Sequenz von Methodenaufrufen)*

Änderung der Methodensignatur Wenn die Parameter einer Methode in Ausgangs- und Zielport nicht kompatibel zueinander ist (siehe Unterabschnitt 3.3.3), müssen sie durch eine Funktion aufeinander abgebildet werden. Dabei gilt, dass jedem Eingabeparameter der Methode des Ausgangsports ein Wert zugeordnet werden muss. Analog muss auch jedem Ausgabeparameter der Methode des Zielports ein Wert zugeordnet werden. Denkbar sind Unterschiede im Parametertyp, in der Reihenfolge, in verwendeten Einheiten (z. B. Millisekunden statt Sekunden) und die Ersetzung von Einzelparametern durch eine Datenstruktur. Die Abbildung wird in Form einer Zuweisungsoperation unter Benutzung der Syntax der Programmiersprache der Komponente definiert.

Wenn eine Methode des Ausgangsports Exceptions deklariert, die beim Zielport nicht vorhanden sind, muss Programmcode für die Behandlung dieser Exceptions definiert werden.

Der Adaptionsschritt wird wie folgt definiert: *addMethod(Signatur im Zielport, Signatur im Ausgangsport, Code für Abbildung, Code für Exception)*

Beispiele Im Folgenden werden einige Beispiele für die Definition von Adaptionsoperatoren aus einzelnen Adaptionsschritten gegeben:

- Die Inkompatibilitäten aus Beispiel 3.2 können wie folgt ausgeglichen werden:

```
renameMethod(encrypt, encryptData);
renameType(B, A)
```

- Die Inkompatibilitäten aus Beispiel 3.4 können wie folgt ausgeglichen werden:

```
addMethod("public byte[] encrypt(String key, byte[] data)",
                        {setKey(key), return encrypt(data)});
renameType(B, A)
```

3.3.5 Aspektoperatoren

Aspektoperatoren sind optionale Bestandteile von Adaptierbaren Komponenten. Wie Adaptionsoperatoren werden sie zur Anpassung von Subkomponenten an den Kontext einer Adaptierbaren Komponente eingesetzt. Die wesentlichen Unterschiede liegen jedoch darin,

dass Aspektoperatoren auf eine Menge von Subkomponenten gleichzeitig angewendet werden können und dass Komponentenports und Parameter nicht verändert werden.

Definition 3.14. Ein **Aspektoperator** definiert eine Funktion
$AsOp : \underline{Komponententypen}^n \rightarrow \underline{Komponententypen}^n$ mit $n > 0 \ \wedge \ n \in \mathbb{N}$, die eine endliche Menge von Komponententypen als Argument erhält und eine Menge von Komponententypen mit gleicher Kardinalität als Ergebnis zurück liefert. Nur der Programmcode der Komponenten wird dabei verändert, Ports und Parameter bleiben unverändert.

Jeder Komponententyp der Ergebnismenge kann innerhalb der Adaptierbaren Komponente wie jeder andere Subkomponententyp verwendet werden. Das bedeutet auch, dass auf eine oder mehrere Komponenten der Ergebnismenge wiederum ein Aspektoperator angewendet werden kann. Die Mehrfachanwendung von Aspektoperatoren ist dabei nicht immer kommutativ[12].

Aspektoperatoren unterstützen einige Konzepte und Möglichkeiten der Aspektorientierten Programmierung (siehe Abschnitt 2.4) auf Komponentenebene. Dazu werden die folgenden *Joinpoints* für Komponenten definiert (siehe auch Abbildung 3.7), an denen das Verhalten durch den Aspektoperator beeinflusst werden kann:

Erzeugen von Komponenteninstanzen (*create*) Mit diesem Joinpoint kann ein alternativer Programmcode zur Erzeugung einer Komponenteninstanz realisiert werden. Damit wird die Standardimplementierung der Laufzeitunterstützung für Adaptierbare Komponenten (siehe Abschnitt 4.3) überschrieben.

Nach dem Erzeugen von Komponenteninstanzen (*postCreate*) Mit diesem Joinpoint kann zusätzlicher Programmcode für die Initialisierung einer Komponenteninstanz hinzugefügt werden, wenn die Standardimplementierung der Laufzeitunterstützung nicht ausreicht. Beispielsweise kann damit eine Datenbankverbindung erzeugt werden oder eine Konfigurationsdatei eingelesen werden.

Vor und nach Methodenaufrufen (*preMethod, postMethod*) Bei allen definierten Methoden in *provided*- und *required*-Ports kann zusätzlicher Programmcode vor und nach der Ausführung eingefügt werden. Bei Vor-Methoden-Joinpoints kann dieser Code kann auf Eingabeparameter und bei Nach-Methoden-Joinpoints auf Ausgabeparameter zugreifen und diese auch verändern. Mit diesen Joinpoints können beispielsweise Zugriffskontrollen, die Protokollierung von Ereignisse und Fehlerbehandlungen realisiert werden.

Vor und nach dem Ändern eines Parameters (*preParameter, postParameter*) Mit diesem Joinpoint kann zusätzlicher Programmcode vor und nach der Änderung eines Parameterwertes durchgeführt werden. Damit können z. B. zulässige Parameterwerte überprüft werden.

Vor und nach Rekonfiguration (*preReconfiguration, postReconfiguration*) Eine Komponente, die von einer Rekonfiguration beeinflusst wird, kann vor und nach der Durchführung der eigentlichen Rekonfigurationsoperationen zusätzlichen Programmcode ausführen.

[12] Die Erforschung gegenseitiger Beeinflussungen von unterschiedlichen Aspekten ist Gegenstand des Forschungsgebiets *Aspect Interference*

Vor dem Löschen von Komponenteninstanzen (*preDelete*) Mit diesem Joinpoint können zusätzliche Aufräumaktionen vor dem endgültigen Löschen einer Komponenteninstanz durchgeführt werden. Beispielsweise können Ressourcen wie Datenbankverbindungen, die bei der Komponentenerzeugung reserviert wurden, wieder freigegeben werden.

Löschen von Komponenteninstanzen (*delete*) Mit diesem Joinpoint kann ein alternativer Programmcode zum Entfernen einer Komponenteninstanz realisiert werden. Damit wird die Standardimplementierung der Laufzeitunterstützung für Adaptierbare Komponenten (siehe Abschnitt 4.3) überschrieben.

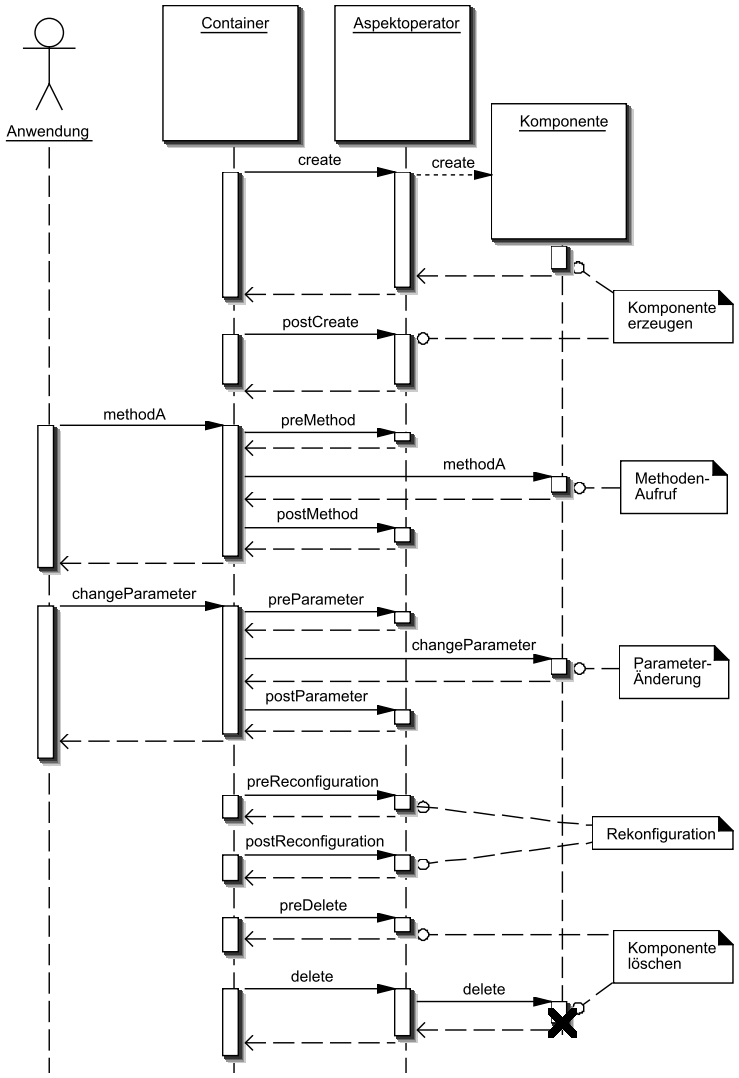

Abbildung 3.7: Sequenzdiagramm zum Aufruf der verschiedenen Joinpoints bei einer Komponente

Beispiel 3.5. Adaptionsoperator mit Komponente mit eigenem Thread
Eine Email-Komponente soll innerhalb einer Adaptierbaren Komponente verwendet werden.

Sie besitzt einen eigenen Thread, um eingestellte Nachrichten asynchron zu versenden. Durch den eigenen Thread verletzt sich jedoch die Annahme für die Durchführung von Rekonfigurationen, dass alle Methodenaufrufe in endlicher Zeit abgearbeitet werden (siehe Abschnitt 3.1). Mit Hilfe eines Adaptionsoperators kann die Email-Komponente so angepasst werden, dass sie die Annahme wieder erfüllt. Dazu wird vor einer Rekonfiguration mit dem Joinpoint *preReconfiguration* der Thread angehalten und nach der Rekonfiguration mit dem Joinpoint *postReconfiguration* der Thread wieder gestartet. Durch den Adaptionsoperator muss die Mail-Komponente selbst nicht geändert werden.

3.3.6 Kontextmodell

Mit Hilfe eines optionalen Kontextmodells kann eine Adaptierbare Komponente auf Umgebungsinformationen zugreifen. Auf diese Weise können selbstadaptive Komponenten entwickelt werden, die auf Änderungen innerhalb der jeweiligen Anwendung oder des Systems eigenständig reagieren (siehe Abschnitt 2.3 und Unterabschnitt 3.4.7). Umgebungsinformation, oder auch Kontextinformationen genannt, sind prinzipiell beliebige Daten, die strukturiert in dem Kontextmodell abgespeichert werden. Einfache Beispiele sind der verfügbare freie Speicher, die maximal verfügbare Netzwerkbandbreite oder der aktuelle Status der Internet-Anbindung (online/offline).

Das Kontextmodell wird von der Laufzeitumgebung der Komponenten implementiert (siehe Unterabschnitt 4.3.8). Diese Implementierung verwaltet eine Menge von Kontextquellen, die Werte von Kontextinformationen zur Laufzeit ermitteln. Implementierungen für Kontextquellen sind entweder bereits in der Laufzeitumgebung integriert oder werden zusammen mit Adaptierbaren Komponenten mitgeliefert, um auch domänenspezifische Kontextinformationen zu unterstützen. Adaptierbare Komponenten können zur Laufzeit über die Schnittstelle des Kontextmodells auf die Kontextinformationen zugreifen.

Jede Kontextinformation besitzt einen eindeutigen Namen und einen bestimmten Datentyp. Weitere Strukturierungen der Kontextinformationen, z. B. als Baum oder zweistufig mit Namensbereich (*Name Space*) und Name, können implementierungsspezifisch durch Namenskonventionen[13] definiert werden. Es werden drei verschiedene Datentypen für Kontextinformationen unterstützt: ganze Zahlen (*Integer*), reelle Zahlen (*Real*) und boolesche Werte (*Boolean*). Der Definitionsbereich dieser Datentypen kann durch zusätzliche Bedingungen weiter eingeschränkt werden, z. B. ganze Zahlen zwischen 0 und 10.

Definition 3.15. Eine **Kontextinformation** ist ein Tupel $Kontext = (Name, Typ, DB)$ mit $Name$ als eindeutigem Bezeichner der Kontextinformation, $Typ \in \{Integer, Real, Boolean\}$ als Datentyp und DB als Definitionsbereich der Kontextinformation (Teilmenge des Definitionsbereichs des Datentyps).

Definition 3.16. Ein **Kontextwert** ist ein Tupel $Kontextwert = (Kontext, Wert)$ mit $Kontext$ als Kontextinformation und $Wert$ als aktueller Wert der Kontextinformation mit einem Datentyp entsprechend des Datentyps der Kontextinformation und $Wert \in DB$.

[13] Beispielsweise können Namensbestandteile durch Punkte getrennt werden, um damit einen Zugriffspfad in einer Baumstruktur zu beschreiben (z. B. `network.bandwidth.max`).

3.4 Konfigurationsbeschreibung

Die Konfigurationsbeschreibung übernimmt drei wesentliche Funktionen, die zusammen den Aufbau und das Verhalten einer Adaptierbaren Komponente bestimmen:

Definition der Konfigurationen: Es wird die Menge aller möglichen internen Konfigurationen der Adaptierbaren Komponente definiert, die später durch die Änderung von Komponentenparametern ausgewählt werden können. Eine Konfiguration legt damit den *internen Zustand* einer Adaptierbaren Komponente fest.

Definition der Parameter-Abbildung: Es werden Werte von Komponentenparametern auf bestimmte Konfigurationen abgebildet. Auf diese Weise wird die Adaptivität der Komponente realisiert.

Definition der Selbstadaptivität Als optionales Konzept können sich Adaptierbare Komponenten unter Nutzung eines Kontextmodells auch eigenständig an Änderungen der Umgebung anpassen. Dabei wird auf die definierten Konfigurationen und die Parameter-Abbildung zurückgegriffen.

In den folgenden Unterabschnitten werden die drei Aufgabenbereiche der Konfigurationsbeschreibung detailliert vorgestellt. Dazu wird zuerst die Konfiguration einer Adaptierbaren Komponente genau definiert. Danach werden vier verschiedene Möglichkeiten zur Beschreibung von Konfigurationen vorgestellt. Jede dieser Varianten eignet sich für unterschiedliche Anwendungsfälle. Ziel ist es dabei, die Beschreibung für den Entwickler so einfach wie möglich zu gestalten, gleichzeitig aber eine hohe Flexibilität zu erreichen. Die letzten zwei Unterabschnitte behandeln schließlich die Beschreibung von Parameter-Abbildungen und der Selbstadaptivität.

3.4.1 Definition und Gültigkeit von Konfigurationen

Definition 3.17. Die **Konfiguration** einer Adaptierbaren Komponente ist eindeutig definiert durch das Tupel $\mathit{Konfiguration} = (\underline{K}, \underline{V}, \underline{PW})$. Dabei ist:

- $\underline{K} \subseteq \underline{\mathit{Komponenten}}$ eine nichtleere Menge von Subkomponenteninstanzen; (Die Ergebnisse von Adaptions- und Aspektoperatoren sowie Glue-Code werden auch als Subkomponenten betrachtet.)
- $\underline{V} \subseteq \underline{\mathit{Verbindungen}}$ eine Menge von typkompatiblen Verbindungen zwischen Ports von Subkomponenten bzw. zwischen externen Ports der Adaptierbaren Komponente und Ports von Subkomponenten;
- $\underline{PW} \subseteq \underline{\mathit{Parameterwerte}}$ eine Menge von Wertzuweisungen für Parameter von Subkomponenten.

Zu jedem Zeitpunkt während der Laufzeit einer Adaptierbaren Komponente ist immer genau eine Konfiguration *aktiv*. Mit Hilfe der Konfigurationsbeschreibung werden alle möglichen aktiven Konfigurationen definiert.

Definition 3.18. Eine Konfiguration ist nur dann *gültig*, wenn sie alle folgenden Bedingungen erfüllt:

- Jede Konfiguration enthält mindestens eine Subkomponenteninstanz ($|\underline{K}| \geq 1$).
- Alle obligatorischen *required*-Ports ($Art \in \{req11, req1n\}$) aller Subkomponenteninstanzen $K \in \underline{K}$ sind über mindestens eine Verbindung $V \in \underline{V}$ entweder mit einem typkompatiblen *provided*-Port einer anderen Subkomponente $K \in \underline{K}$ oder einem externen *required*-Port der Adaptierbaren Komponente verbunden.
- Alle externen *provided*-Ports $P \in \underline{ExternalPorts}$ sind entweder über eine Verbindung $V \in \underline{V}$ mit einem typkompatiblen *provided*-Port einer Subkomponente $K \in \underline{K}$ verbunden oder über eine Menge von Methoden-Verknüpfungen $V \in \underline{V}$ mit einem oder mehrere *provided*-Ports von Subkomponente $K \in \underline{K}$ verbunden.
- Alle Verbindungen $V \in \underline{V}$ verbinden entweder einen typkompatiblen *provided*-Port einer Subkomponente $K_1 \in \underline{K}$ mit einer anderen Subkomponente $K_2 \in \underline{K}$, einen typkompatiblen *provided*-Port einer Subkomponente $K \in \underline{K}$ mit einem externen *provided*-Port $P \in \underline{ExternalPorts}$ oder einen typkompatiblen *required*-Port einer Subkomponente $K \in \underline{K}$ mit einem externen *required*-Port $P \in \underline{ExternalPorts}$.

3.4.2 Beschreibung von expliziten Konfigurationen

Die einfachste Möglichkeit zur Beschreibung einer Konfiguration einer Adaptierbaren Komponente besteht darin, alle für das Tupel *Konfiguration* benötigten Mengen (Komponenteninstanzen, Verbindungen und Parameterwerte) explizit zu definieren.

Definition 3.19. Eine **explizite Konfiguration** ist ein Tupel $VK = (id, \underline{K}, \underline{V}, \underline{PW})$ mit id als eindeutigem Bezeichner und $\underline{K}$, $\underline{V}$ und $\underline{PW}$ als Mengen wie in Definition 3.17. Eine explizite Konfiguration VK ist **vollständig**, wenn sie die Gültigkeitsbedingungen erfüllt (siehe Definition 3.18) und direkt als Konfiguration einer Adaptierbaren Komponente (siehe Definition 3.17) verwendet werden kann. Anderenfalls ist VK **unvollständig** und muss durch Variationen oder Template-Variationen zu einer vollständigen Konfiguration ergänzt werden.

Damit wird eine *explizite Konfiguration* in folgenden Schritten modelliert:

1. Alle notwendigen Subkomponenteninstanzen werden ausgewählt. Falls erforderlich, können auf diesen Subkomponenten Adaptions- bzw. Aspektoperatoren angewendet werden. Nach diesem Schritt ist die nichtleere Menge von Subkomponenteninstanzen $K \subseteq \underline{Komponenten}$ definiert.
2. Verbindungen zwischen *required*- und *provided*-Ports von Subkomponenteninstanzen werden festgelegt.
3. Verknüpfungen zwischen internen Ports von Subkomponenteninstanzen und externen Ports der Adaptierbaren Komponente werden festgelegt.
4. Allen Parametern der Subkomponenteninstanzen können optional entweder feste Werte oder externe Parameter zugeordnet werden. Bei der Zuordnung von externen Parametern der Adaptierbaren Komponente übernimmt der jeweilige Parameter einer Subkomponenteninstanz den Wert des externen Parameters zur Laufzeit.
5. Die explizite Konfiguration erhält einen eindeutigen Namen im Kontext der Adaptierbaren Komponente.

Visuelle Modellierung von expliziten Konfigurationen mit UML Zur Modellierung von expliziten Konfigurationen wird die Notation von UML-Komponentendiagrammen übernommen. Lediglich zur Beschreibung von Methoden-Verknüpfungen sind Erweiterungen notwendig (siehe Abbildung 3.5). Eine explizite Konfiguration wird als Komponentendiagramm innerhalb eines Rahmens (*Frame*) in Form eines Rechtecks dargestellt, wobei in der linken oberen Ecke als Name des Rahmens der Text „Config“ gefolgt vom Namen der expliziten Konfiguration steht. Innerhalb des Rahmens[14] befindet sich genau eine Adaptierbare Komponente mit ihren externen Ports. Alle Subkomponenteninstanzen (inklusive Adaptions- und Aspektoperatoren), Verbindungen und Verknüpfungen werden entsprechend der UML-Notation aus Abschnitt 3.3 innerhalb der Adaptierbaren Komponente eingezeichnet.

Bewertung Mit der ausschließlichen Verwendung von expliziten Konfigurationen können prinzipiell alle möglichen Konfigurationen einer Adaptierbaren Komponente modelliert werden. Da jede vollständige explizite Konfiguration mit genau einer aktiven Konfiguration zur Laufzeit übereinstimmt, ist die Form der Modellierung einfach verständlich für Entwickler.

Als großer Nachteil erweist sich jedoch der hohe Modellierungsaufwand, wenn eine Adaptierbare Komponente sehr viele Konfigurationen enthält und diese Konfigurationen auch viele Gemeinsamkeiten besitzen. Um diesen Nachteil zu beseitigen, werden in den folgenden Abschnitten Template-Konfiguration, Variationen und Template-Variationen eingeführt.

Beispiel Kryptographiekomponente Die Kryptographiekomponente enthält drei verschiedene Verschlüsselungs- und zwei verschiedene Kompressionskomponenten, die jeweils unabhängig voneinander in einer Konfiguration kombiniert werden können (siehe Abschnitt 3.2). Wenn man außerdem berücksichtigt, dass auch Konfigurationen ohne Kompressionskomponente möglich sind, ergeben sich insgesamt neun verschiedene Konfigurationen für die Kryptographiekomponente. Mit der in diesem Abschnitt vorgestellten Beschreibungsmethode können diese neun Konfigurationen jeweils vollständig modelliert werden.

Abbildung 3.8 zeigt die Modellierung von drei der neun Konfigurationen. Alle weiteren Konfigurationen müssen in einer ähnlichen Weise modelliert werden. Damit zeigt sich aber auch schon der wesentlicher Nachteil der Beschreibung von expliziten Konfigurationen: Obwohl sich die einzelnen Konfigurationen der Kryptographiekomponente nicht wesentlich unterscheiden, muss dennoch die vollständige Konfiguration modelliert werden. Bei komplexeren Beispielen mit einer größeren Menge von möglichen Konfigurationen wird damit die ausschließliche Verwendung dieses Ansatzes schnell unpraktikabel für den Entwickler werden.

3.4.3 Beschreibung von Template-Konfigurationen

Sollen in einer Adaptierbaren Komponente mehrere sehr ähnliche Konfigurationen modelliert werden, ist die Beschreibung von expliziten Konfigurationen relativ aufwändig. In solchen Fällen sind Template-Konzepte, wie man sie auch in den Programmiersprachen C++ oder Java[15] findet, sehr nützlich. Die Grundidee besteht darin, an einer bestimmten Stelle einen Platzhalter vorzusehen, der erst später mit einem bestimmten Wert ausgefüllt wird.

[14] In der UML-Terminologie als *Content Area* bezeichnet

[15] Templates heißen bei Java *Generics* und werden erst ab Version 5.0 unterstützt.

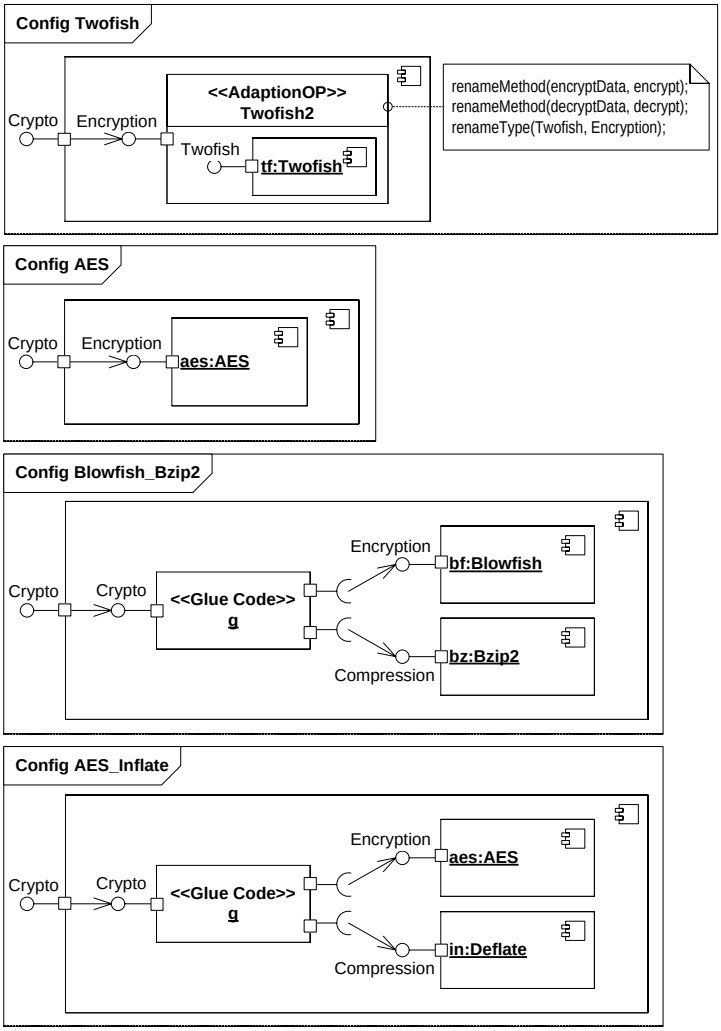

Abbildung 3.8: Modellierung der Kryptographiekomponente mit vollständigen expliziten Konfigurationsbeschreibungen

Angewendet auf Konfigurationen werden Platzhalter für Subkomponenteninstanzen vorgesehen. Für jeden Platzhalter, bezeichnet als Template-Komponente, wird eine nichtleere Menge von möglichen Subkomponententypen spezifiziert. Das bedeutet, dass bereits bei der Definition einer Template-Komponente – im Gegensatz zum Template-Konzept bei Programmiersprachen – alle möglichen Realisierungen in Form von Komponententypen festgelegt werden. Alle diese Komponententypen müssen die gleiche funktionale Schnittstelle implementieren, d. h. sie müssen die gleichen *required*- und *provided*-Ports besitzen.

Definition 3.20. Eine *Template-Komponente* ist ein Tupel $TKomponente = (id, \underline{KT}, \underline{Po}, \underline{P})$ mit id als eindeutigem Bezeichner, $\underline{KT} \subseteq \underline{Komponententypen}$ als Menge möglicher Komponententypen dieser Komponente, $\underline{Po} \subseteq \underline{Ports}$ als Menge aller *provided*- und *required*-Ports und $\underline{P} \subseteq Parameter$ als Menge aller Komponentenparameter. Jeder Komponententyp $K \in \underline{KT}$ besitzt mindestens alle Ports aus $\underline{Po}$ und alle Parameter aus $\underline{P}$.

Definition 3.21. Eine **Template-Konfiguration** ist ein Tupel $TK = (id, \underline{TK}, \underline{K}, \underline{V}, \underline{PW})$ mit id als eindeutigem Bezeichner, $\underline{TK}$ als Menge von Template-Komponenten und $\underline{K}$, $\underline{V}$ und $\underline{PW}$ als Mengen wie in Definition 3.17.

Eine Template-Konfiguration TK steht stellvertretend für $\prod_{i=1}^{|\underline{TK}|} |\underline{KT}_i|$ explizite Konfigurationen (siehe Definition 3.19), wenn $|\underline{KT}_i|$ die Anzahl der möglichen Komponententypen für die i-te Template-Komponente ist. Dazu muss für jede Template-Komponente $K \in \underline{TK}$ ein bestimmter Komponententyp $KT \in \underline{KT}$ aus der Menge der möglichen Komponententypen $\underline{KT}$ festgelegt werden. Dies kann durch eine Menge von Tupeln (K, KT) definiert werden. Alternativ kann es auch in der Form *ID-Template-Konfiguration*<*ID-Template-Komponente* = *ID-Komponententyp*, ...> aufgeschrieben werden, z. B. Compression<e=AES, c=Bzip2> für die Template-Konfiguration aus Abbildung 3.9. Diese Form der Darstellung wird für die Definition von Parameter-Abbildungen in Unterabschnitt 3.4.6 benötigt.

Als Bezeichner von instanziierten Template-Komponenten, deren Typ zur Laufzeit festgelegt wurde, wird die *id* gefolgt von einem Doppelpunkt und dem Namen des ausgewählten Komponententyps verwendet (z. B. `e:AES`).

Visuelle Modellierung von Template-Konfigurationen mit UML Die Modellierung von Template-Konfigurationen entspricht im Wesentlichen der Darstellung von expliziten Konfigurationen (siehe Unterabschnitt 3.4.2), lediglich als Titel wird „Template“ gefolgt vom Bezeichner der Template-Konfiguration verwendet.

Eine Template-Komponente wird als UML-Komponente mit dem neuen Stereotypen „template“ dargestellt. Anstelle des Komponententypen nach dem Bezeichner der Komponenteninstanz werden die Namen der möglichen Komponententypen der Template-Komponente als Komma-separierte Liste in geschweiften Klammern dargestellt (z. B. {Deflate, Bzip2}). Wenn die Liste zu umfangreich ist, kann alternativ auch eine UML-Notiz verwendet werden, welche die Liste enthält und auf die Template-Komponente verweist.

Bewertung Template-Konfigurationen eignen sich besonders, wenn für einzelne Subkomponenteninstanzen mehrere Implementierungen in Form von Komponententypen verfügbar sind und diese unterschiedlichen Implementierungen der einzige Unterschied zwischen Konfigurationen sind. Bestehen dagegen strukturelle Unterschiede zwischen mehreren Konfigurationen, bringt die Verwendung von Template-Konfiguration keine weiteren Vorteile gegenüber expliziten Konfigurationen. Im Übrigen übernehmen Template-Konfigurationen alle Eigenschaften der vollständigen Konfigurationsbeschreibungen.

Beispiel Kryptographiekomponente Mit Hilfe von Template-Konfigurationen kann der als wesentlicher Nachteil herausgestellte hohe Modellierungsaufwand für die Kryptographiekomponente bei vollständigen Konfigurationsbeschreibungen (siehe voriger Abschnitt) beseitigt werden. Wie in Abbildung 3.9 dargestellt, werden lediglich zwei Diagramme benötigt, um alle neun Konfigurationen zu modellieren[16]. In der Template-Konfiguration „Compression“ werden alle Konfigurationen zusammengefasst, die Verschlüsselung und vorherige Kompression bieten, während in „NoCompression“ alle Konfigurationen ohne Kompression modelliert werden.

[16] Es wird angenommen, dass der Komponententyp „Twofish2“ zuvor durch einen Adaptionsoperator wie in Abbildung 3.8 definiert wurde.

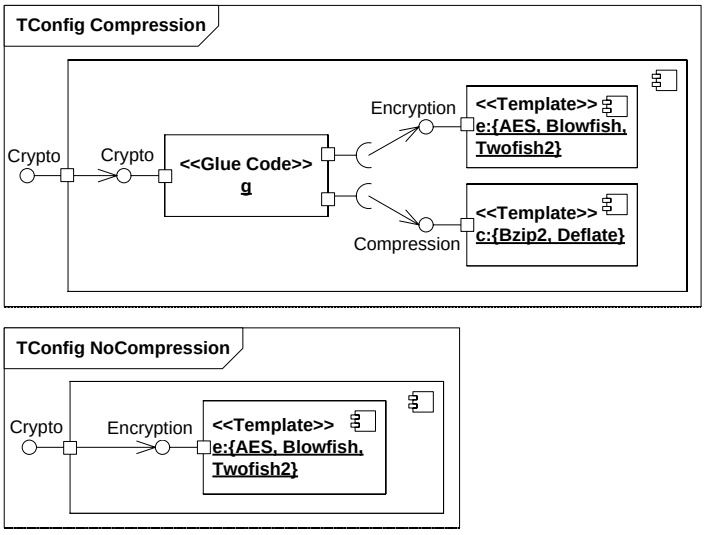

Abbildung 3.9: Modellierung der Kryptographiekomponente mit Template-Konfigurationen

3.4.4 Beschreibung von Variationen

Die bisher vorgestellten Modellierungstechniken betrachten, wie direkt oder indirekt[17] eine vollständige Konfiguration einer Adaptierbaren Komponente definiert werden kann. In vielen Fällen, wie z. B. auch bei der Kryptographiekomponente, sind die Unterschiede zwischen mehreren Konfigurationen jedoch gering. Aus diesem Grund ist es sinnvoll, nur diese Unterschiede zu modellieren.

Variationen beschreiben, wie man durch Anwendung einer Reihe von Änderungsoperationen auf einer Ausgangskonfiguration zu einer neuen Konfiguration gelangt, ohne diese neue Konfiguration explizit zu modellieren. Die Änderungsoperationen werden als *Rekonfigurationsoperationen* bezeichnet.

Definition 3.22. Eine **Rekonfigurationsoperation** ist eine Funktion
$r : \underline{Konfigurationen} \rightarrow \underline{Konfigurationen}$, die eine Änderung an einer Konfiguration (siehe Definition 3.17) einer Adaptierbaren Komponente durchführt und eine neue Konfiguration als Ergebnis liefert. Weder die Ausgangskonfiguration noch die erhaltene Konfiguration müssen vollständig oder gültig (siehe Definition 3.18) sein.

$\underline{RekonfigOps}$ ist die Menge aller Rekonfigurationsoperationen.

Variationen sind damit Funktionen, die eine Konfiguration auf eine andere abbilden. Eine Variation kann auch auf unterschiedliche Ausgangskonfigurationen angewendet werden, sofern dies vom Entwickler vorgesehen wird und alle Rekonfigurationsoperationen bei der Ausgangskonfiguration definiert sind. Eine Variation kann parametrisiert werden, indem eine Menge von Parametern auf Parameter von Subkomponenten abgebildet werden (siehe Rekonfigurationsoperation *setParameter*).

Definition 3.23. Eine **Variation** ist ein Tupel $Variation = (id, \underline{Rekonfig}, DB)$ mit id als eindeutigem Bezeichner, $\underline{Rekonfig}$ als geordnete Menge von Rekonfigurationsoperationen r

[17] mit Hilfe von Template-Komponenten

und *DB* als Menge von Konfigurationen und Variationen, auf denen die Variation definiert ist (Definitionsbereich der Variation).

Eine **Variation** definiert damit eine Funktion $V : \underline{Konfigurationen} \rightarrow \underline{Konfigurationen}$, die eine Ausgangskonfiguration in eine neue Konfiguration überführt. V ergibt sich durch die Nacheinanderausführung aller Rekonfigurationsoperationen $r \in \underline{Rekonfig}$, d. h. $V = r_1 \circ r_2 \circ \ldots \circ r_n$ für $r_1 \ldots r_n \in \underline{Rekonfig}$. Weder die Ausgangskonfiguration noch die erhaltene Konfiguration müssen vollständig oder gültig (siehe Definition 3.18) sein.

Im weiteren Verlauf dieses Abschnittes werden zuerst verschiedene Rekonfigurationsoperationen und ihre visuelle Definition beschrieben. Anschließend werden verschiedene Klassen von Variationen zusammen mit ihren Eigenschaften vorgestellt. Am Ende werden Wechselwirkungen bei der gleichzeitigen Anwendung mehrerer Variationen diskutiert.

3.4.4.1 Rekonfigurationsoperationen

Wie im nachfolgenden noch relativ einfach nachgewiesen wird, genügen fünf verschiedene Typen von Rekonfigurationsoperationen, um jede beliebige Konfiguration einer Adaptierbaren Komponente in eine andere beliebige Konfiguration zu transformieren. Einige Rekonfigurationsbedingungen sind nur dann definiert, wenn die jeweilige Ausgangskonfiguration bestimmte Bedingungen erfüllt, wie im Folgenden beschrieben ist.

Hinzufügen einer Subkomponenteninstanz (*addComponent*): Eine Subkomponenteninstanz aus der Menge aller Komponenteninstanzen der Adaptierbaren Komponente wird zur neuen Konfiguration hinzugefügt. Alle Ports sind zunächst noch unverbunden. In der Ausgangskonfiguration darf keine Komponenteninstanz den gleichen Bezeichner wie die neue Komponenteninstanz besitzen.

$$addComponent : \underline{Konfigurationen} \times \underline{Komponenten} \rightarrow \underline{Konfigurationen}$$
$$(K, k) \mapsto addComponent(K, k) = (\underline{Komponenten} \cup \{k\}, \underline{Verbindungen}, \underline{Parameterwerte})$$

Hinzufügen einer Verbindung (*addConnection*): Eine Verbindung (siehe Unterabschnitt 3.3.3) zwischen zwei internen Ports oder zwischen einem internen und externen Port wird zur neuen Konfiguration hinzugefügt. Diese Operation ist nur definiert, wenn die Ausgangskonfiguration die Ports enthält, die durch die Verbindung verknüpft werden sollen, und die Ports typkompatibel sind.

$$addConnection : \underline{Konfigurationen} \times \underline{Verbindungen} \rightarrow \underline{Konfigurationen}$$
$$(K, v) \mapsto addConnection(K, v) = (\underline{Komponenten}, \underline{Verbindungen} \cup \{v\}, \underline{Parameterwerte})$$

Entfernen einer Subkomponenteninstanz (*removeComponent*): Eine Subkomponenteninstanz wird aus der neuen Konfiguration entfernt. Dies ist nur möglich, wenn alle Ports unverbunden sind und die Subkomponenteninstanz in der Ausgangskonfiguration enthalten ist.

$$removeComponent : \underline{Konfigurationen} \times \underline{Komponenten} \rightarrow \underline{Konfigurationen}$$
$$(K, k) \mapsto removeComponent(K, k) = (\underline{Komponenten} \backslash \{k\}, \underline{Verbindungen}, \underline{Parameterwerte})$$

Entfernen einer Verbindung (*removeConnection*): Eine Verbindung zwischen zwei internen Ports oder zwischen einem internen und externen Port wird aus der neuen Konfiguration

gelöscht. Diese Operation ist nur definiert, wenn die Ausgangskonfiguration die Verbindung zwischen den angegebenen Ports enthält.

$$removeConnection : \underline{Konfigurationen} \times \underline{Verbindungen} \rightarrow \underline{Konfigurationen}$$
$$(K, v) \mapsto addConnection(K, v) = (\underline{Komponenten}, \underline{Verbindungen} \backslash \{v\}, \underline{Parameterwerte})$$

Setzen eines Komponentenparameters (*setParameter*): Der Wert eines Komponentenparameters einer Subkomponenteninstanz wird neu gesetzt. Dabei kann entweder ein fester Wert oder ein Parameter der Adaptierbaren Komponente bzw. einer Variation zugewiesen werden. Bei der Verknüpfung mit einem Parameter wird immer der aktuelle Wert des verknüpften Parameters verwendet. Die Wertzuweisung muss kompatibel mit dem Typ des Parameters sein.

Eine bereits vorhandene Wertzuweisung für den Parameter in der Ausgangskonfiguration wird durch die Operation entfernt. Es ist wichtig, dass eine Wertzuweisung entfernt wird und nicht nur überschrieben, da das Überschreiben eine nicht-kommutative Operation ist.

$$setParameter : \underline{Konfigurationen} \times \underline{Parameterwerte} \rightarrow \underline{Konfigurationen}$$
$$(K, p) \mapsto addConnection(K, p) = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte} \cup \{p\})$$

Für den Beweis, dass diese Typen von Rekonfigurationsoperationen genügen, muss gezeigt werden, dass sich damit jede beliebige Konfiguration in jede beliebige andere Konfiguration in endlich vielen Schritten transformieren lässt.

Beweis. Es seien K_1 und K_2 zwei beliebige Konfigurationen einer Adaptierbaren Komponente. Es existiert immer eine Transformation von K_1 zu K_2, bestehend aus den folgenden Rekonfigurationsschritten:

1. Für alle Verbindungen $v \in V$ wird die Verbindung v aus K_1 entfernt. Nach diesem Schritt sind alle internen und externen Ports unverbunden.
2. Für alle Subkomponenten $k \in K$ wird die Subkomponente k aus K_1 entfernt.
3. Für alle Subkomponenten $k \in K$ wird die Subkomponente k aus K_2 hinzugefügt.
4. Für alle Verbindungen $v \in V$ wird die Verbindung v aus K_2 hinzugefügt.
5. Für alle Wertzuweisungen für Parameter von Subkomponenten wird der neue Wert gesetzt.

Nach der Ausführung dieser Rekonfigurationsschritte wurde K_1 in K_2 transformiert. □

Bei dieser Rekonfiguration werden also zunächst alle Subkomponenten und Verbindungen der Ausgangskonfiguration entfernt, dann die Subkomponenten und Verbindungen der Zielkonfiguration hinzugefügt und schließlich die Werte von Parametern gesetzt. Auch wenn diese Rekonfiguration in der Praxis wahrscheinlich nicht verwendet wird, ist damit beweisen, dass die in diesem Abschnitt definierten fünf Typen von Rekonfigurationsoperationen ausreichen.

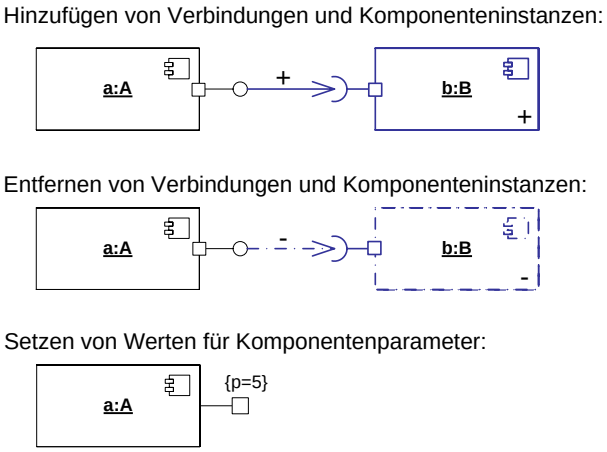

Abbildung 3.10: UML-Darstellung von Rekonfigurationsoperationen

3.4.4.2 Visuelle Definition von Rekonfigurationsoperationen

Für die Definition einer Variation ist keine totale Ordnung der Rekonfigurationsoperationen erforderlich. Es genügt eine partielle Ordnung, die durch die folgenden Schritte bestimmt wird:

- Alle Rekonfigurationsoperationen werden nach ihrem Typ gruppiert, so dass man fünf disjunkte Mengen erhält. Die Reihenfolge der Operationen innerhalb einer Gruppe ist egal.
- Die fünf Gruppen werden in folgender Reihenfolge angeordnet: *removeConnection*, *removeComponent*, *addComponent*, *addConnection*, *setParameter*

Die partielle Ordnung der Rekonfigurationsoperationen ist eine wichtige Voraussetzung für die graphische Definition von Variationen. Damit kann auf die Nummerierung der Operationen verzichtet werden.

Abbildung 3.10 zeigt die entwickelten Erweiterungen von UML zur Modellierung von Rekonfigurationsoperationen. Mit den neuen Stereotypen „+“ und „-“ werden neue bzw. entfernte Verbindungen und Komponenteninstanzen gekennzeichnet. Zur besseren Darstellung können die Änderungen zusätzlich farbig und durch breitere Linien hervorgehoben werden.

Im Folgenden wird nachgewiesen, dass zwei Rekonfigurationen mit der gleichen Menge von Rekonfigurationsoperationen und der gleichen partiellen Ordnung, aber mit unterschiedlichen totalen Ordnungen bei der Anwendung auf die gleiche Ausgangskonfiguration auch zur gleichen Zielkonfiguration führen.

Für alle fünf Typen von Rekonfigurationsoperationen muss gezeigt werden, dass ihre Anwendung kommutativ ist, d. h. $\forall r_1, r_2 \in \underline{RekonfigOps} : \; r_1(r_2(K)) = r_2(r_1(K))$. Dann ist bewiesen, dass eine unterschiedliche Reihenfolge der Anwendung dennoch zu identischen Ergebnissen führt.

addComponent

$$K = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte}), k_1, k_2 \in \underline{Komponenten} \wedge k_1 \neq k2$$

$addComponent(addComponent(K, k_1), k_2)$
$= addComponent((\underline{Komponenten} \cup \{k_1\}, \underline{Verbindungen}, \underline{Parameterwerte}), k_2)$
$= (\underline{Komponenten} \cup \{k_1, k_2\}, \underline{Verbindungen}, \underline{Parameterwerte})$

$addComponent(addComponent(K, k_2), k_1)$
$= addComponent((\underline{Komponenten} \cup \{k_2\}, \underline{Verbindungen}, \underline{Parameterwerte}), k_1)$
$= (\underline{Komponenten} \cup \{k_1, k_2\}, \underline{Verbindungen}, \underline{Parameterwerte})$

removeComponent

$K = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte}), k_1, k_2 \in \underline{Komponenten} \wedge k_1 \neq k2$

$removeComponent(removeComponent(K, k_1), k_2)$
$= removeComponent((\underline{Komponenten} \backslash \{k_1\}, \underline{Verbindungen}, \underline{Parameterwerte}), k_2)$
$= (\underline{Komponenten} \backslash \{k_1, k_2\}, \underline{Verbindungen}, \underline{Parameterwerte})$

$removeComponent(removeComponent(K, k_2), k_1)$
$= removeComponent((\underline{Komponenten} \backslash \{k_2\}, \underline{Verbindungen}, \underline{Parameterwerte}), k_1)$
$= (\underline{Komponenten} \backslash \{k_1, k_2\}, \underline{Verbindungen}, \underline{Parameterwerte})$

addConnection

$K = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte})$
$v_1, v_2 \in \underline{Verbindungen} \wedge v_1 \neq v2$

$addConnection(addConnection(K, v_1), v_2)$
$= addConnection((\underline{Komponenten}, \underline{Verbindungen} \cup \{v_1\}, \underline{Parameterwerte}), v_2)$
$= (\underline{Komponenten}, \underline{Verbindungen} \cup \{v_1, v_2\}, \underline{Parameterwerte})$

$addConnection(addConnection(K, v_2), v_1)$
$= addConnection((\underline{Komponenten}, \underline{Verbindungen} \cup \{v_2\}, \underline{Parameterwerte}), v_1)$
$= (\underline{Komponenten}, \underline{Verbindungen} \cup \{k_1, k_2\}, \underline{Parameterwerte})$

removeConnection

$K = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte})$
$v_1, v_2 \in \underline{Verbindungen} \wedge v_1 \neq v2$

$removeConnection(removeConnection(K, v_1), v_2)$
$= removeConnection((\underline{Komponenten}, \underline{Verbindungen} \backslash \{v_1\}, \underline{Parameterwerte}), v_2)$
$= (\underline{Komponenten}, \underline{Verbindungen} \backslash \{v_1, v_2\}, \underline{Parameterwerte})$

$removeConnection(removeConnection(K, v_2), v_1)$
$= removeConnection((\underline{Komponenten}, \underline{Verbindungen} \backslash \{v_2\}, \underline{Parameterwerte}), v_1)$
$= (\underline{Komponenten}, \underline{Verbindungen} \backslash \{v_1, v_2\}, \underline{Parameterwerte})$

setParameter Der Rekonfigurationsoperator *setParameter* ist nur dann kommutativ, wenn damit verschiedene Parameter gesetzt werden. Das mehrmalige Setzen desselben Parameters innerhalb derselben Rekonfiguration würde bedeuten, dass der letzte gesetzte Wert alle vorigen Zuweisungen überschreibt. Damit würde das Ergebnis von der Reihenfolge abhängen.

$K = (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte})$
$p_1, p_2 \in \underline{Parameterwerte} \wedge p_1 \neq p2$

$setParameter(setParameter(K, p_1), p_2)$
$= setParameter((\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte} \cup \{p_1\}), p_2)$
$= (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte} \cup \{p_1, p_2\})$

$setParameter(setParameter(K, p_2), p_1)$
$= setParameter((\underline{Komponenten}, \underline{Verbindungen} \cup \{p_2\}, \underline{Parameterwerte}), p_1)$
$= (\underline{Komponenten}, \underline{Verbindungen}, \underline{Parameterwerte} \cup \{p_1, p_2\})$

3.4.4.3 Inverse Variationen

Bisher wurde gezeigt, wie eine Variation mit Hilfe einer Menge von Rekonfigurationsoperationen eine Ausgangskonfiguration in eine neue Konfiguration transformiert. Es ist jedoch auch notwendig, eine Variation wieder rückgängig zu machen, um zur Ausgangskonfiguration zurückzukehren. Dazu wird eine weitere Variation definiert, die als *inverse Variation* bezeichnet wird. Das bedeutet, dass die aufeinander folgende Anwendung einer Variation und ihrer inversen Variation wieder zur Ausgangskonfiguration führt.

Definition 3.24. Es sein V eine Variation, die auf einer Menge von Ausgangskonfigurationen definiert ist. V^{-1} wird als die zu V **inverse Variation** bezeichnet, wenn bei der Anwendung auf einer Konfiguration K gilt: $V \circ V^{-1}(K) = K$.

Zunächst muss gezeigt werden, dass zu jeder Variation immer auch eine inverse Variation existiert und wie diese bestimmt werden kann. Da eine Variation aus einer Menge von Rekonfigurationsoperationen besteht, muss eine inverse Variation aus einer Menge von jeweils inversen Rekonfigurationsoperationen bestehen. Folglich muss nachgewiesen werden, dass zu jeder Rekonfigurationsoperation r eine inverse Rekonfigurationsoperation r^{-1} existiert.

Inverse Rekonfigurationsoperationen können durch folgende Regeln bestimmt werden:

$$r(K) = addComponent(K, k) = K' \quad \longrightarrow \quad r^{-1} = removeComponent(K', k) = K \quad (3.1)$$
$$r(K) = removeComponent(K, k) = K' \quad \longrightarrow \quad r^{-1} = addComponent(K', k) = K \quad (3.2)$$
$$r(K) = addConnection(K, v) = K' \quad \longrightarrow \quad r^{-1} = removeConnection(K', v) = K \quad (3.3)$$
$$r(K) = removeConnection(K, v) = K' \quad \longrightarrow \quad r^{-1} = addConnection(K', v) = K \quad (3.4)$$
$$r(K) = setParameter(K, p, v) = K' \quad \longrightarrow \quad r^{-1} = setParameter(K', p, v') = K \quad (3.5)$$

Es ist ersichtlich, dass *addComponent* und *removeComponent* bzw. *addConnection* und *removeConnection* jeweils paarweise zusammen gehören (Operation / inverse Operation) und die inverse Operation sich somit sehr einfach ermitteln lässt. Lediglich *setParameter* bildet in

doppelter Hinsicht eine Ausnahme. Zum einen hat diese Operation sich selbst als inverse Operation, jedoch mit einem anderen Argument für die Parameterzuweisung. Zum anderen kann dieses Argument für die inverse Operation nicht auf Grundlage der ursprünglichen Operation abgeleitet werden. Es ergibt sich aus den Parameterzuweisungen, die in der Ausgangskonfiguration enthalten sind. Damit ist die inverse Operation zu *setParameter* immer von der Ausgangskonfiguration abhängig.

Eine inverse Variation erhält man, indem für jede Rekonfigurationsoperation der ursprünglichen Variation eine inverse Rekonfigurationsoperation nach den weiter oben definierten Regeln bestimmt wird. Anschließend werden die Operationen nach der bereits in diesem Abschnitt definierten partiellen Ordnung sortiert. Eine inverse Variation unterscheidet sich daher im prinzipiellen Aufbau nicht von der zugehörigen Variation.

3.4.4.4 Klassen von Variationen

In Abhängigkeit der verwendeten Rekonfigurationsoperationen innerhalb einer Variation, werden verschiedene Klassen von Variationen unterschieden:

Änderungs-Variationen enthalten mindestens eine *removeComponent* oder *removeConnection* Operation oder eine *setParameter* Operation, die einen Parameter einer Subkomponente der Ausgangskonfiguration ändert. Im Übrigen kann eine Änderungs-Variation beliebige Rekonfigurationsoperationen enthalten. Die Mehrfachanwendung einer Änderungs-Variation auf eine Ausgangskonfiguration ist nicht möglich, d. h. auf die Konfiguration, die durch die Anwendung der Variation entstanden ist, kann die Variation nicht noch einmal angewendet werden.

Additive Variationen enthalten ausschließlich *addComponent* und *addConnection* Operationen. Weiterhin sind *setParameter* Operationen nur bei im Rahmen der Variation hinzugefügten Subkomponenten erlaubt. Die Mehrfachanwendung einer Änderungs-Variation auf eine Ausgangskonfiguration ist nicht möglich.

Mehrfach-Variationen besitzen die gleichen Eigenschaften hinsichtlich der Operationen wie additive Variationen, aber sie können mehrfach auf eine Ausgangskonfiguration angewendet werden. Sie eignen sich damit zur Beschreibung von iterativen Änderungen einer Konfiguration. Der Komponentenentwickler muss festlegen, ob diese Eigenschaft auch zu sinnvollen Ergebnissen führt. Eine notwendige aber nicht hinreichende Voraussetzung ist, dass alle *required*-Ports, zu denen eine Verbindung aufgebaut wird, *mehrfach optional* oder *mehrfach obligatorisch* sind (siehe Unterunterabschnitt 3.3.1.1). Optional kann eine obere Schranke größer als eins[18] für die Anzahl der Anwendungen der Variation definiert werden.

Mehrfach-Variationen erlauben die einfache Modellierung einer großen Anzahl von Konfigurationen. In UML-Diagrammen werden Mehrfach-Variationen mit einem Stern nach dem Namen gekennzeichnet. Ein Beispiel für eine solche Variation ist das Hinzufügen einer Processing-Subkomponente zur Beschleunigung der Abarbeitung von Anfragen (siehe Abbildung 3.11). Diese Variation kann beliebig oft ausgeführt werden, um damit eine Menge von Processing-Subkomponenten mit dem Mehrfach-*required*-Port der Dispatcher-Subkomponente zu verbinden.

[18] Additive Variationen entsprechen Mehrfach-Variationen mit einer oberen Schranke von eins.

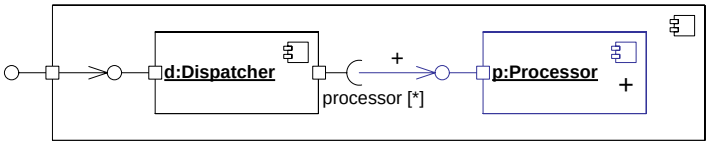

Abbildung 3.11: Mehrfach-Variation zum Hinzufügen einer Processing-Subkomponente

Additive und Mehrfach-Variationen können nicht nur auf Konfigurationen, sondern auch auf Variationen angewendet werden. Damit kann die Änderung einer Konfiguration, ausgedrückt durch eine Variation, selbst nochmals geändert werden. Die endgültige Konfiguration ergibt sich aus der Nacheinanderausführung der verschiedenen Variationen. Die Variationen *Host_ Logger*, *Host_ Realm* und *Host_ Valve* in Abschnitt 5.5 sind Beispiele für solche Variationen.

3.4.4.5 Gleichzeitige Anwendung mehrerer Variationen

Zur Definition einer Variation gehört immer auch die Angabe der Ausgangskonfigurationen, auf denen sie angewendet werden darf. Wenn zwei unterschiedliche Variationen existieren, die für die gleiche Ausgangskonfiguration definiert sind, stellt sich die Frage was passiert, wenn diese beiden Variationen nacheinander angewendet werden. Es können folgende Fälle eintreten:

1. Zwei Mehrfach-Variationen oder eine Mehrfach- und eine Additive Variation: In diesem Fall besteht keine Gefahr einer Beeinflussung, da ausschließlich Verbindungen und Subkomponenten zur Ausgangskonfiguration hinzugefügt werden.

2. Zwei Additive Variationen: Ein Konflikt tritt auf, wenn beide Variationen bei mindestens einem Port der Ausgangskonfiguration eine Verbindung hinzufügen und dieser Port die Art *einfach obligatorisch* oder *einfach optional* besitzt (siehe Unterunterabschnitt 3.3.1.1). In allen anderen Fällen treten keine Beeinflussungen auf.

3. Eine Änderungs-Variation und eine Mehrfach- oder Additive Variation: Ein Konflikt tritt auf, wenn die Änderungs-Variation mindestens eine Subkomponente entfernt, zu der eine Verbindung mit der Mehrfach- oder Additiven Variation hergestellt werden soll. Zusätzlich kann der im Fall (2) beschriebene Konflikt auftreten.

4. Zwei Änderungs-Variationen: Ein Konflikt tritt auf, wenn die Variationen mindestens eine gleiche Verbindung oder Subkomponente entfernen oder einen gleichen Parameter ändern. Zusätzlich können die im Fall (2) und (3) beschriebenen Konflikte auftreten.

Unabhängig von der Art der enthaltenen Rekonfigurationsoperationen beeinflussen sich zwei Variationen nicht, wenn die jeweiligen Mengen der beeinflussten Komponenten, Verbindungen und Parameter disjunkt sind. Für jede Variation müssen die folgenden Mengen bestimmt werden: notwendige Komponenten (NK), gelöschte Komponenten (GK), beeinflusste Ports (Po), beeinflusste Parameter (Pa) und gelöschte Verbindungen (V). Diese Mengen (anfangs werden alle Mengen auf die leere Menge gesetzt) werden iterativ durch Anwendung der folgenden Regeln für alle Rekonfigurationsoperationen einer Variation bestimmt:

- *addComponent(k)*: keine Änderungen
- *removeComponent(k)*: $GK := GK \cup \{k\}$
- *addConnection(v)* mit $v = (k_1, p_1, k_2, p_2)$
  Wenn k_1 in der Ausgangskonfiguration enthalten ist: $NK := NK \cup \{k_1\}$, $Po := Po \cup \{p_1\}$
  Wenn k_2 in der Ausgangskonfiguration enthalten ist: $NK := NK \cup \{k_2\}$, $Po := Po \cup \{p_2\}$
- *removeConnection(v)* mit $v = (k_1, p_1, k_2, p_2)$:
  Wenn k_1 in der Ausgangskonfiguration enthalten ist: $NK := NK \cup \{k_1\}$, $Po := Po \cup \{p_1\}$
  Wenn k_2 in der Ausgangskonfiguration enthalten ist: $NK := NK \cup \{k_2\}$, $Po := Po \cup \{p_2\}$
  $V := V \cup \{v\}$
- *setParameter(k,p)*:
  Wenn k in der Ausgangskonfiguration enthalten ist: $NK := NK \cup \{k\}$, $Pa := Pa \cup \{p\}$

Zwei Variationen A und B beeinflussen sich nicht, wenn gilt:

$$NK_A \cap GK_B = \emptyset \ \wedge \ GK_A \cap NK_B = \emptyset \ \wedge \ Po_A \cap Po_B = \emptyset \ \wedge \ Pa_A \cap Pa_B = \emptyset \ \wedge \ V_A \cap V_B = \emptyset$$

Diese Bedingungen können von Entwicklungswerkzeugen (siehe Unterabschnitt 4.2.3) automatisch überprüft werden.

3.4.4.6 Visuelle Modellierung von Variationen mit UML

Eine Variation wird als Komponentendiagramm innerhalb eines Rahmens (*Frame*) in Form eines Rechtecks dargestellt, wobei in der linken oberen Ecke als Name des Rahmens der Text „Template“ gefolgt vom Namen der Variation steht. Bei Mehrfach-Variationen folgt nach dem Namen ein Stern. Innerhalb des Rahmens werden Komponenteninstanzen (inklusive Adaptions- und Aspektoperatoren) und Verbindungen der Ausgangskonfiguration einer Variation dargestellt. Zusätzlich werden die Rekonfigurationsoperationen der Variation entsprechend Unterunterabschnitt 3.4.4.2 dargestellt.

3.4.4.7 Bewertung von Variationen

Variationen eignen sich besonders gut zur Modellierung von verschiedenen strukturellen Änderungen bei einer Ausgangskonfiguration. Wenn für eine bestimmte Konfiguration x verschiedene Variationen definiert werden, die sich nicht gegenseitig beeinflussen (siehe Unterunterabschnitt 3.4.4.5), können damit bereits mindestens $2^n - 1$ verschiedene Konfigurationen modelliert werden. Bei der Verwendung von Mehrfach-Variationen ist die Zahl der möglichen Konfigurationen sogar unendlich groß. Variationen sind damit eine ideale Ergänzung zu expliziten Konfigurationen und Template-Konfigurationen.

3.4.4.8 Beispiel Kryptographiekomponente

Abbildung 3.12 zeigt eine mögliche Modellierung der Kryptographiekomponente mit Hilfe von Variationen. *Config AES* modelliert eine explizite und vollständige Konfiguration mit der Verschlüsselungskomponente AES, aber ohne zusätzliche Kompression. Zwei weitere explizite Konfigurationen existieren in ähnlicher Weise für Twofish und Blowfish. Die Änderungs-Variation *AES_ Deflate* ist für die explizite Konfiguration *AES* definiert. Sie fügt Glue-Code

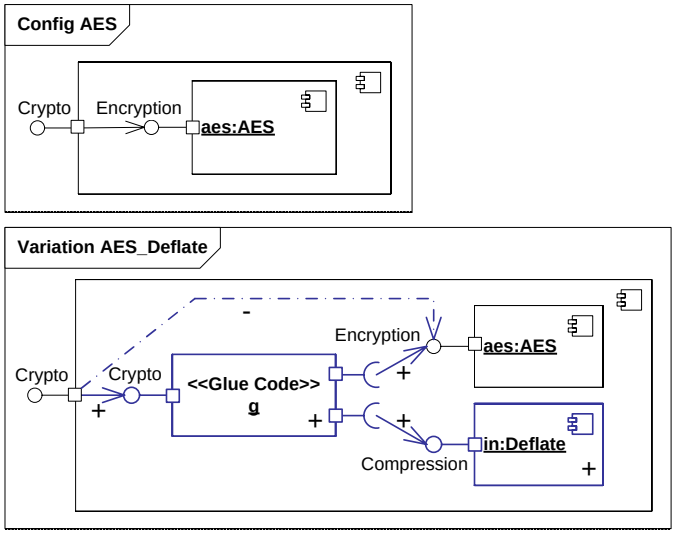

Abbildung 3.12: Modellierung der Kryptographiekomponente mit Variationen

und die Kompressionskomponente Deflate ein. Als Ergebnis entsteht eine Konfiguration, die mit AES verschlüsselt und mit Deflate komprimiert. Eine weitere Variation muss in ähnlicher Weise für Bzip2 modelliert werden.

3.4.5 Beschreibung von Template-Variationen

Wie beim Übergang von expliziten Konfigurationsbeschreibungen zu Template-Konfigurationen, können Template-Konzepte auch bei Variationen eingeführt werden. Auf diese Weise kann eine einzelne Template-Variation eine Menge von Variationen modellieren.

Definition 3.25. Eine **Template-Variation** ist ein Tupel
$TVariation = (id, \underline{TK}, \underline{Rekonfig}, DB)$ mit id als eindeutigem Bezeichner, $\underline{TK}$ als Menge von Template-Komponenten, $\underline{Rekonfig}$ als geordnete Menge von Rekonfigurationsoperationen r und DB als Menge von Konfigurationen und Variationen, auf denen die Variation definiert ist. Die Template-Komponenten werden in einer oder mehreren Rekonfigurationsoperationen verwendet.

Alle sonstigen Eigenschaften übernehmen Template-Variationen von Variationen.

Bei der Anwendung der Template-Variation wird für jede Template-Komponente ein Komponententyp aus der Menge der definierten Komponententypen ausgewählt. Eine Template-Variation beschreibt damit eine Menge von Variationen, ähnlich für eine Template-Konfiguration eine Menge von expliziten Konfigurationen beschreibt.

Visuelle Modellierung mit UML Die Modellierung von Template-Variationen entspricht im Wesentlichen der Darstellung von Variationen (siehe Unterabschnitt 3.4.4), lediglich als Titel wird „TVariation“ gefolgt vom Bezeichner der Template-Variation verwendet.

Eine Template-Komponente wird wie bei der Modellierung von Template-Konfigurationen (siehe Unterabschnitt 3.4.3) dargestellt.

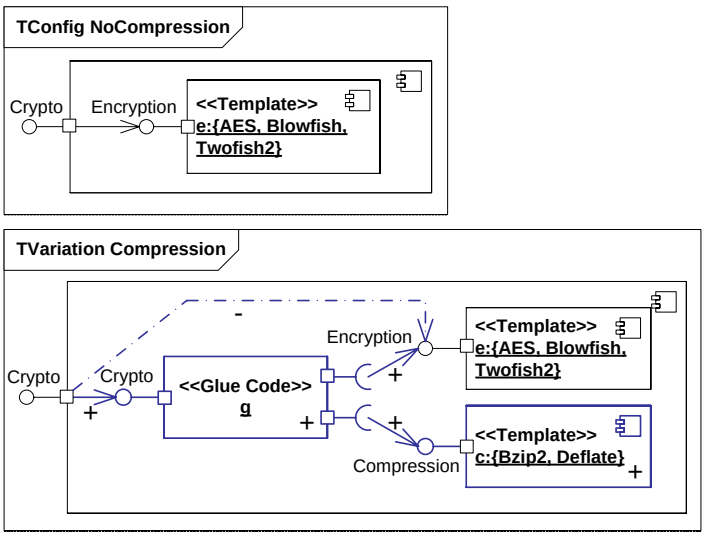

Abbildung 3.13: Modellierung der Kryptographiekomponente mit Template-Variationen

Bewertung von Template-Variationen Template-Variationen übernehmen alle Eigenschaften von Variationen, können aber zusätzlich zur Modellierung von ähnlichen strukturellen Änderungen verwendet werden, die lediglich unterschiedliche Komponententypen verwenden.

Beispiel Kryptographiekomponente Mit Hilfe von zusätzlichen Template-Konzepten kann das Modellierungsbeispiel für Variation weiter vereinfacht werden. Abbildung 3.13 demonstriert, dass nur noch eine Variation (*Compression*) benötigt wird. Die Template-Konfiguration *NoCompression* ist dabei der Ausgangspunkt für die Template-Variation *Compression*.

3.4.6 Abbildung von Komponentenparameter-Werten auf Konfigurationen

Mit der so genannten **Parameter-Abbildung** wird die Zuordnung von Komponentenparameter-Werten auf bestimmte Konfigurationen einer Adaptierbaren Komponente definiert. Eine Konfiguration ergibt sich immer aus der Auswahl genau einer expliziten (Template-) Konfiguration[19] und der optionalen Anwendung einer Menge von Variationen. Die Parameter-Abbildung setzt die Definition aller Komponentenparameter (siehe Definition 3.7) bestehend aus Namen, Typen, Wertebereichen und optionaler Festlegung von Standardwerten voraus.

Die verschiedenen unterstützten Komponentenparameter-Typen wurden zusammen mit ihrer jeweiligen Anwendung bereits in Unterunterabschnitt 3.3.1.2 vorgestellt. Parameter können in Abhängigkeit ihres Typs vier unterschiedliche Aufgaben übernehmen (siehe auch Tabelle 3.2):

- Auswahl einer vollständigen Konfiguration und optional einer oder mehrerer Variationen (*Konfigurationsauswahl*)
- Auswahl einer oder mehrerer Variationen (*Variationsauswahl*)

[19] In diesem Abschnitt sind bei der Erwähnung von vollständigen Konfigurationen und Variationen immer auch Template-Konfigurationen und Template-Variationen gemeint.

	Integer	Real	Enum	Map	Array	Any
Konfigurationsauswahl	X	X	X			
Variationsauswahl	X	X	X	X		X
Variationsanzahl	X				X	
Parameter-Verknüpfung	X	X	X	X	X	X

Tabelle 3.2: Verwendungsmöglichkeiten von Parametertypen

- Festlegung, wie oft eine Mehrfach-Variation angewendet werden soll (*Variationsanzahl*)
- Verknüpfung mit Parametern von Subkomponenten

Die ersten drei dieser Aufgaben werden durch eine Menge von Regeln definiert, die im Folgenden beschrieben werden. Die Verknüpfung mit Parametern von Subkomponenten wurde bereits in Unterabschnitt 3.4.2 beschrieben. Jeder Parameter kann eine der vier Aufgaben übernehmen. Parameter, die primär zur Konfigurations- oder Variationsauswahl oder zur Variationsanzahl verwendet werden, können zusätzlich auch noch zur Parameter-Verknüpfung eingesetzt werden.

Definition 3.26. Die **Parameter-Abbildung** einer Adaptierbaren Komponente wird durch ein Tupel $Mapping = (Default, \underline{Konfigurationsregeln}, \underline{Variationsregeln}, \underline{Anzahlregeln})$ definiert. *Konfigurationsregeln*, *Variationsregeln* und *Anzahlregeln* enthalten jeweils eine Menge von Regeln, die zusammen eine bestimmte Konstellation von Parameterwerten auf eine Konfiguration abbilden. $Default \in \underline{Konfigurationen}$ legt die Standardkonfiguration fest, die nur angewendet wird, wenn durch die Regelmenge keine Konfiguration ausgewählt wurde.

Mappings ist die Menge aller möglichen Parameter-Abbildungen.

Zuordnung von Parameterwerten zu Konfigurationen und Variationen Die Auswahl von expliziten Konfigurationen und die Auswahl von Variationen erfolgt prinzipiell mit den gleichen Konzepten. Eine Menge von Regeln (*Konfigurationsregeln* und *Variationsregeln*) wählt eine explizite Konfiguration und eine Menge von Variationen in Abhängigkeit der aktuellen Parameterwerte aus.

Eine Regel zur Konfigurations- bzw. Variationsauswahl besteht aus einer Prämisse in Form einer konjunktiven Verknüpfung von Wertbedingungen für unterschiedliche Komponentenparameter und einer Konklusion in Form einer Menge von ausgewählten und ausgeschlossenen expliziten Konfigurationen (nur bei Konfigurationsauswahl) bzw. Variationen. Jede Wertbedingung für einen Parameter besteht aus einer Menge von Werten und Intervallen von Werten (nur bei Integer und Real). Eine Wertbedingung gilt als erfüllt, wenn der aktuelle Parameterwert mit einem der angegebenen Werte übereinstimmt oder in einem der angegebenen Intervalle liegt. Wenn alle Wertbedingungen erfüllt sind, wird die entsprechende Regel angewendet.

Für die Variationsauswahl gibt es noch einen Spezialfall mit genau einem Map-Parameter in der Prämisse. Wenn der Map-Parameter definiert wurde (d. h. wenn Werte zugewiesen wurden), werden die einzelnen Werte der verschiedenen Namen innerhalb des Map-Parameters gleichnamigen Parametern der ausgewählten Variation zugewiesen.

Definition 3.27. Es seien P_i Integer-, Real-, Enumeration- oder Any-Parameter ($i = 1 \dots n$) mit $WB(P_i)$ als jeweilige Wertebereiche, AK eine Menge von ausgewählten expliziten Konfigurationen, AV eine Menge von ausgewählten Variationen. Eine **Wertbedingung** $W(P_i)$ für den Parameter P_i ist eine Teilmenge des jeweiligen Wertebereichs $W(P_i) \subseteq WB(P_i)$. Für Any-Parameter werden nur die Werte *definiert* oder *undefiniert* unterschieden, wenn dem Parameter entweder ein beliebiger Wert zugewiesen wurde bzw. kein Wert festgelegt wurde.

Eine **Regel R für eine Konfigurationsauswahl** hat die Form $\bigwedge_{i=1}^{n} W(P_i) \implies AK \cup AV$.

Eine **Regel R für eine Variationsauswahl** hat die Form $\bigwedge_{i=1}^{n} W(P_i) \implies AV$. Dabei kann maximal ein Parameter P_i den Typ Map-Parameter besitzen.

Festlegung der Anzahl von Mehrfach-Variationen Wie bereits in Tabelle 3.2 dargestellt, können Integer und Array-Parameter dazu verwendet werden zu bestimmen, wie oft eine bestimmte Mehrfach-Variation angewendet werden soll. Bei einem Integer-Parameter legt der aktuelle Wert direkt die Anzahl fest. Bei einem Array-Parameter ergibt sich die Anzahl der Anwendungen aus der Anzahl von Map-Einträgen in dem Array. Kein Eintrag in einem Array oder ein Integer-Wert kleiner oder gleich Null führt dazu, dass die entsprechende Mehrfach-Variation nicht angewendet wird. Bei einem Array-Parameter werden für alle enthaltenden Map-Parameter die Werte den Parametern der zugeordneten Mehrfach-Variation zugewiesen (wie bei einem Map-Parameter bei einer Variationsauswahl).

Definition 3.28. Es sei P ein Integer- oder Array-Parameter und MV eine Mehrfach-Variation. Eine **Regel R zur Festlegung der Anzahl von Mehrfach-Variationen** ist ein Tupel (P, MV).

Definition von Regeln Während der Entwicklung einer Adaptierbaren Komponente müssen die Regelmengen (*Konfigurationsregeln*, *Variationsregeln*, *Anzahlregeln*) für die Parameter-Abbildung festgelegt werden. Dieser Prozess kann auf unterschiedliche Arten erfolgen: Entweder werden bestimmten Parameterwerten mögliche Konfigurationen zugewiesen oder, umgekehrt, einer bestimmten Konfiguration die dafür notwendigen Parameterwerte zugeordnet. Die beiden Verfahren lassen sich auch miteinander kombinieren und die Regeln können jeweils direkt abgelesen werden. Die erhaltene Regelmenge ist dabei nicht notwendigerweise identisch, beschreibt aber die gleiche Parameter-Abbildung.

Die Zuordnung von Parameterwerten zu Konfigurationen eignet sich besonders, wenn eine Adaptierbare Komponente nur wenige (< 10) explizite Konfigurationen und Variationen enthält. Umgekehrt ist es bei nur wenigen Parametern zweckmäßiger, den Parameterwerten bzw. -intervallen bestimmte Konfigurationen und Variationen zuzuordnen.

Algorithmus zur Bestimmung der Konfiguration Die Konfiguration wird durch die Anwendung des folgenden Algorithmus bestimmt:

1. Für alle nicht definierten Komponentenparameter[20] wird, sofern vorhanden, der Default-Wert aus der Parameterdefinition verwendet (siehe Definition 3.7).

[20] Jeder Komponentenparameter, dem nicht explizit ein Wert zugewiesen wurde, gilt als *nicht definiert*.

2. Die Menge der ausgewählten expliziten Konfigurationen AK wird mit der Menge aller expliziten Konfigurationen der Adaptierbaren Komponente initialisiert. Die Menge der ausgewählten Variationen AV wird mit der leeren Menge initialisiert.

3. Die erste Regel zur Parameter-Abbildung wird ausgewählt.

4. Wenn die Prämisse der ausgewählten Regel R für die aktuellen Parameterwerte erfüllt ist, werden alle Variationen AV_R der Regel zu AV hinzugefügt ($AV := AV \cup AV_R$) und die Durchschnittsmenge aus AK und der Menge der expliziten Konfigurationen AK_R wird als neuer Wert für AK gesetzt ($AK := AK \cap AK_R$).

5. Die nächste Regel zur Parameterzuordnung wird ausgewählt und der Algorithmus mit Schritt 4 fortgesetzt. Wenn bereits alle Regeln ausgewertet wurden, terminiert der Algorithmus.

Wenn die Regelmenge gültig ist (siehe weiter unten in diesem Abschnitt), enthält AK nach Abschluss des Algorithmus genau eine vollständige Konfiguration und AV eine Menge von darauf anzuwendenden Variationen. Die Laufzeit des Algorithmus hängt linear von der Anzahl der Zuweisungsregeln und der Anzahl Wertebereiche in den Vorbedingungen der Regeln ab, d. h. die Komplexität ist $O(n)$.

Überprüfung der Gültigkeit der Parameter-Abbildung Eine Parameter-Abbildung ist nur dann gültig, wenn für jede mögliche Kombination von Parameterwerten genau eine explizite Konfiguration und eine Menge von Variationen ausgewählt werden. Eine Überprüfung dieser Bedingung[21] muss bereits zur Entwicklungszeit einer Adaptierbaren Komponente bei der Definition der Parameter-Abbildung erfolgen.

Zur Überprüfung werden zunächst alle Parameter P_i bestimmt, die in mindestens einer Regel zur Auswahl einer expliziten Konfiguration vorkommen. Für jeden dieser Parameter P_i werden anschließend alle Werte bzw. Wertebereiche ermittelt, die in mindestens einer Regel vorkommen, wobei $N(P_i)$ die Anzahl der Möglichkeiten bezeichnet. Aus der ermittelten Menge der Parameter mit den jeweiligen unterschiedlichen Werten bzw. Wertebereichen ergeben sich insgesamt $\prod_{i=1}^{n} N(P_i)$ unterschiedliche Kombinationen für Parameterwerte bei den n Parametern. Jede dieser Kombinationen weist somit allen Parametern P_i genau einen Wert aus der Menge der möglichen Werte zu.

Für jede Kombination von Parameterwerten wird durch Anwendung des Algorithmus zur Bestimmung von Konfigurationen ermittelt, ob bei Anwendung aller Regeln maximal eine explizite Konfiguration ausgewählt wird[22]. Wird für eine bestimmte Kombination mehr als eine explizite Konfiguration ausgewählt, ist die Regelmenge ungültig und muss korrigiert werden. Durch die Anwendung des Überprüfungsverfahrens beim jedem Hinzufügen einer neuen Regel während der Entwicklung können Fehler sofort erkannt werden.

Bei der vereinfachten Annahme, dass es k Parameter mit jeweils l unterschiedlichen Werten gibt und damit m verschiedene Regeln überprüft werden sollen, beträgt die Komplexität des Überprüfungsalgorithmus $O(m * l^k)$. Das bedeutet, dass der Algorithmus nur für eine kleine

[21] Wenn die Adaptierbare Komponente nur eine explizite Konfiguration enthält, ist eine Überprüfung nicht erforderlich.

[22] Wenn keine Konfiguration durch die Regeln ausgewählt wurde, wird die Standardkonfiguration verwendet.

Anzahl k von Parametern hinreichend schnell ausgeführt werden kann. Es wurde jedoch angenommen, dass auch bei sehr komplexen Szenarien maximal 3–5 Parameter an der Auswahl von expliziten Konfigurationen beteiligt sind, in vielen Fällen sogar nur ein Parameter. Alle anderen Parameter werden lediglich zur Auswahl von Variationen verwendet. In diesen Fällen kann der Algorithmus ausreichend schnell abgearbeitet werden und daher wurde auf eine eventuell mögliche Optimierung des Algorithmus verzichtet.

Beispiel: Parameter-Abbildung bei der Kryptographiekomponente Für die Kryptographiekomponente wurden in diesem Kapitel bereits vier verschiedene Möglichkeiten zur Beschreibung der Konfiguration vorgestellt (siehe Abschnitt 3.4). Zu jeder dieser Varianten lässt sich eine Abbildung von Parameterwerten auf bestimmte Konfigurationen aufstellen. Weitere Beispiele zur Definition von Parameter-Abbildungen sind in Kapitel 5 beschrieben.

Beispiel 3.6. Vollständige Konfiguration/Template Konfiguration
Die folgende Parameter-Abbildung ordnet die Parameterwerte der Konfigurationsbeschreibung aus Abbildung 3.8 zu:

Prämisse	Konklusion
encryption = AES	AK = {AES_Bzip2, AES_Deflate, AES_None}
encryption = Blowfish	AK = {Blowfish_Bzip2, Blowfish_Deflate, Blowfish_None}
encryption = Twofish	AK = {Twofish_Bzip2, Twofish_Deflate, Twofish_None}
compression = Bzip2	AK = {AES_Bzip2, Blowfish_Bzip2, Twofish_Bzip2}
compression = deflate	AK = {AES_Deflate, Blowfish_Deflate, Twofish_Deflate}
compression = None	AK = {AES_None, Blowfish_None, Twofish_None}

Beispiel 3.7. Variation
Die folgende Parameter-Abbildung ordnet die Parameterwerte der Konfigurationsbeschreibung aus Abbildung 3.12 zu:

Prämisse	Konklusion
encryption = AES	$AK = \{AES\}$
encryption = Blowfish	$AK = \{Blowfish\}$
encryption = Twofish	$AK = \{Twofish\}$
compression = Bzip2	$AV = \{Bzip2\}$
compression = deflate	$AV = \{Deflate\}$

3.4.7 Beschreibung der Selbstadaptivität

Selbstadaptive Anwendungen (siehe Abschnitt 2.3) müssen Änderungen von Kontextinformationen (siehe Unterabschnitt 3.3.6) registrieren, die neuen Kontextinformationen analysieren, geeignete Rekonfigurationen planen und schließlich die Rekonfigurationen ausführen. Im Rahmen der vorliegenden Arbeit wird lediglich ein einfaches Verfahren zur Auswertung von Kontextinformationen und zur Planung von Änderungen eingesetzt, mit dem aber die prinzipielle Machbarkeit von selbstadaptiven Komponenten gezeigt wird. Die Beschreibung und Steuerung der Selbstadaptivität kann in weiterführenden Arbeiten unabhängig von anderen Konzepten Adaptierbarer Komponenten weiterentwickelt werden.

Die Selbstadaptivität wird durch eine Menge von Regeln in Form von bedingten Wertzuweisungen für Komponentenparameter definiert. Jede dieser Regeln besteht aus einem booleschen

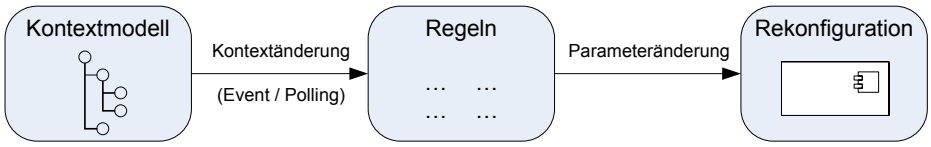

Abbildung 3.14: Auswertung von Kontextinformationen

Ausdruck, der eine Bedingung für einen oder mehrere Kontextwerte definiert, und einer Menge von Wertzuweisungen für einen oder mehrere Komponentenparameter. Wenn der boolesche Ausdruck als „wahr“ ausgewertet wird, werden alle Wertzuweisungen für die Komponentenparameter durchgeführt (siehe Abbildung 3.14). Auf diese Weise können die Mechanismen zur Abbildung von Parameterwerten auf Konfigurationen Adaptierbarer Komponenten (siehe Unterabschnitt 3.4.6) unverändert wiederverwendet werden. Der einzige Unterschied besteht darin, dass Komponentenparameter in diesem Fall von innerhalb der Komponente geändert werden und nicht wie sonst von außerhalb, also z. B. von anderen Komponenten. Die Implementierung des Kontextmodells muss sicherstellen, dass sich die Kontextwerte während der Auswertung von Bedingungen nicht ändern. Dies kann z. B. erreicht werden, in dem vor der Auswertung alle benötigten Kontextwerte ermittelt und zwischengespeichert werden.

Definition 3.29. Es seinen $K_1 \ldots K_n$ Kontextinformationen des Kontextmodells und $WZ = \bigcup_{i=1}^{m} P_i$, $P_i \in \underline{Parameterwerte}$ eine Menge von Wertzuweisungen für Komponentenparameter einer Adaptierbaren Komponente. Eine Wertbedingung $W(K_i)$ für die Kontextinformation K_i ist eine Teilmenge des jeweiligen Wertebereichs.

Eine **Regel** S **zur Selbstadaption** hat die Form $\bigwedge_{i=1}^{n} W(P_i) \implies WZ$.

Bei der Definition der Regeln muss darauf geachtet werden, ein „Schwingen“ zwischen mehreren Konfigurationen zu vermeiden. Dies kann auftreten, wenn unmittelbar nach einer Rekonfiguration durch geringfügig geänderte Umgebungsinformationen wieder eine Rekonfiguration ausgelöst wird. Mögliche Strategien zur Verhinderung solcher endlosen Rekonfigurationen sind die Einführung einer minimalen Zeitdifferenz zwischen zwei Rekonfigurationen, die die regelmäßige Funktionsfähigkeit der Komponente sicher stellen, und die Verhinderungen von weiteren Rekonfigurationen bei der Erkennung von Zyklen.

Unterabschnitt 5.6.3 beschreibt ein Beispiel für eine selbstadaptive Komponente.

3.5 Synchronisation der Rekonfiguration

Eine Parameteränderung zur Laufzeit einer Adaptierbaren Komponente kann in Abhängigkeit der Konfigurationsbeschreibung eine Rekonfiguration auslösen, die durch die Laufzeitumgebung gesteuert werden muss. Bei der Durchführung von Rekonfigurationen müssen eine Reihe von Eigenschaften berücksichtigt werden, die in einer ähnlichen Form insbesondere von Datenbanktransaktionen [HR83] her bekannt sind:

- **Atomizität** – Eine Rekonfiguration wird entweder ganz oder gar nicht ausgeführt.
- **Konsistenz** - Eine Rekonfiguration hinterlässt nach Beendigung einen konsistenten Zustand der Komponente.

- **Isolation** - Mehrere in Ausführung befindliche Rekonfigurationen beeinflussen sich nicht gegenseitig. Außerdem werden Methodenaufrufe nicht durch Rekonfigurationen beeinflusst.

Die Laufzeitumgebung muss außerdem garantieren, dass bei der Durchführung von Rekonfigurationen keine Verklemmungen (*Deadlocks*) auftreten. Ohne besondere Vorkehrungen wären Verklemmungen möglich, wenn mehrere Threads gleichzeitig eine Komponente benutzen – d. h. Methoden aufrufen – und gleichzeitig Rekonfigurationen auslösen können.

Die in diesem Abschnitt beschriebene Lösung zur Synchronisation der Rekonfiguration orientiert sich an dem in [KM90] beschriebenen Verfahren zum *Dynamic Change Management* bei abhängigen Transaktionen. Da das Verfahren jedoch transparent für die Komponenten realisiert werden soll, sind insbesondere Erweiterungen zur automatischen Erkennung des Zustandes von Komponenten und zur Blockierung von Methodenaufrufen erforderlich. Außerdem werden in [KM90] aktive Komponenten, d. h. mit einem eigenen Thread, berücksichtigt, wohingegen Adaptierbare Komponenten keinen eigenen Thread besitzen und nur auf Anfragen (Methodenaufrufe) reagieren.

Modell für Methodenaufrufe Für die weiteren Betrachtungen wird im Folgenden ein Modell für Methodenaufrufe definiert. Ein Methodenaufruf erfolgt immer im Kontext eines bestimmten Threads und wird in endlicher Zeit abgeschlossen, d. h. es darf keine Endlosschleifen geben. Die Laufzeitumgebung kann einen Methodenaufruf blockieren, bevor er von der jeweiligen Komponente abgearbeitet wird[23]. Diese Funktionalität wird zur Erreichung eines definierten Zustandes vor einer Rekonfiguration benötigt und wird im weiteren Verlauf dieses Abschnitts noch näher erläutert. Innerhalb der Methode einer Komponenteninstanz können weitere Methoden bei anderen oder der gleichen Komponenteninstanz (rekursiv) aufgerufen werden.

Zur Laufzeit entsteht für einen Thread T und eine Komponenteninstanz K ein *Aufruf-Stack* $S_{T,K}$, der eine geordnete Menge von Komponenteninstanzen K_i enthält. Eine neue Komponenteninstanz wird hinzugefügt, wenn bei ihr ein Methodenaufruf erfolgte, der nicht von der Laufzeitumgebung blockiert wurde. Wenn der Methodenaufruf abgeschlossen wurde, wird die Komponenteninstanz wieder vom Stack entfernt. Während der Abarbeitung der Methode werden durch Methodenaufrufe weitere Komponenteninstanzen zum Stack hinzugefügt und wieder entfernt. Auf diese Weise ist zu jedem Zeitpunkt erkennbar, bei welchen Komponenteninstanzen im Kontext eines Threads Methodenaufrufe noch nicht abgeschlossen sind. Aus den Aufruf-Stacks $S_{T_i,K}$ aller Threads T_i kann für eine bestimmte Komponenteninstanz K die Menge der *aktiven Threads* TA_K abgeleitet werden: Jeder Thread T_i, in dessen Aufruf-Stack $S_{T_i,K}$ mindestens einmal die Komponenteninstanz K enthalten ist, wird zu TA_K hinzugefügt. Eine Komponenteninstanz K gilt als *inaktiv*, wenn die Aufruf-Stacks $S_{T,K}$ und die Menge der aktiven Threads TA_K leer sind.

Verfahren zur Rekonfiguration Um die geforderten Eigenschaften Konsistenz und Isolation einzuhalten, darf immer nur eine Rekonfiguration bei einer Adaptierbaren Komponente gleichzeitig ablaufen. Außerdem darf während der Rekonfiguration bei keiner daran beteiligten Komponente ein Methodenaufruf abgearbeitet werden. Aus diesen Forderungen kann folgendes Verfahren zur Durchführung einer Rekonfiguration bestimmt werden:

[23] Zur Implementierung kann z. B. das Interceptor-Muster [GHJV95] verwendet werden.

1. Alle direkt von der Rekonfiguration beeinflussten Subkomponenten werden bestimmt.
2. Alle beeinflussten und eventuell weitere indirekt beteiligte Subkomponenten werden für Methodenaufrufe blockiert. Bereits laufende Methodenaufrufe werden trotz Blockierung weiterhin zugelassen, um Verklemmungen zu verhindern.
3. Wenn alle durch die Rekonfiguration beeinflussten Subkomponenten inaktiv sind, wird die eigentliche Rekonfiguration durchgeführt.
4. Die Blockierung von Methodenaufrufen wird aufgehoben und die Rekonfiguration ist abgeschlossen.

Im weiteren Verlauf dieses Abschnittes werden die einzelnen Schritte dieses Verfahrens näher erläutert.

Ein Spezialfall kann auftreten, wenn ein Thread eine Rekonfiguration bei einer Komponenteninstanz auslöst, wo er bei mindestens einer beeinflussten Subkomponente noch aktiv ist. In diesem Fall entsteht eine Verklemmung, bei der der Thread auf sich selbst wartet. Diese Verklemmung kann nur verhindert werden, wenn bereits beim Auslösen der Rekonfiguration durch eine Parameteränderung ein Ausnahmefehler ausgelöst wird und die Rekonfiguration somit nicht stattfindet.

Ermittlung der beeinflussten Subkomponenten einer Rekonfiguration In Abhängigkeit der durchzuführenden Rekonfiguration ermittelt der Adaptionsmanager als Steuerinstanz einer Adaptierbaren Komponente (siehe Unterabschnitt 4.3.2) die Menge der Subkomponenten K_B, die von der Rekonfiguration beeinflusst werden. Für alle Rekonfigurationsoperationen einer Rekonfiguration werden dazu die folgenden Regeln angewendet:

- Für jede Operation *removeComponent(K)* wird K zu K_S hinzugefügt. Außerdem werden alle Subkomponenten zu K_S hinzugefügt, die direkt Methoden von K aufrufen können, d. h. die mindestens einen *required*-Port mit einer Verbindung zu einem Port von K besitzen.
- Für jede Operation *removeConnection(K_1, K_2)* wird K_1, d. h. die Komponente mit einem *required*-Port, zu K_S hinzugefügt, außer wenn K_1 erst innerhalb dieser Rekonfiguration neu hinzugefügt wurde.
- Für jede Operation *addConnection(K_1, K_2)* wird K_1, d. h. die Komponente mit einem *required*-Port, zu K_S hinzugefügt.

Blockierung von Subkomponenten Alle von der Rekonfiguration beeinflussten Subkomponenten müssen inaktiv sein, bevor die Rekonfigurationsoperationen durchgeführt werden können. Um dies zu erreichen müssen neue Methodenaufrufe verhindert werden. Wie in Abbildung 3.15 dargestellt, können aber nicht einfach alle Methodenaufrufe bei Komponenteninstanzen aus K_B ab einem bestimmten Zeitpunkt blockiert werden. Wenn bei einer zu blockierenden Komponenteninstanz K ein bestimmter Thread T in der Menge der aktiven Threads enthalten ist ($T \in TA_K$) und dieser Thread eine weitere Methode bei K aufrufen will, kommt es zu einer Verklemmung: Damit K inaktiv wird, muss der Thread T beendet werden. T kann aber nicht beendet werden, da ein Methodenaufruf bei K blockiert wurde.

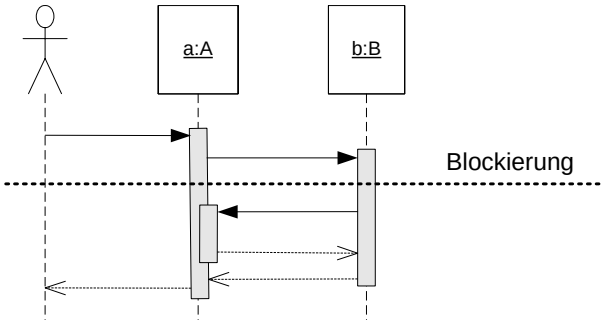

Abbildung 3.15: Sequenzdiagramm zur Veranschaulichung von Verklemmungen bei der Blockierung von Komponenten

Allgemein kann gezeigt werden, dass bei der naiven Implementierung von Blockierungen alle vier Coffman-Bedingungen [CES71] zum Entstehen von Verklemmungen erfüllt sind:

1. Nur der Besitzer einer Ressource kann sie wieder freigeben (*No Preemption*): Die Ressourcen sind hier die Komponenteninstanzen und die Besitzer entweder der Thread eines Methodenaufrufs oder der Adaptionsmanager für die Durchführung einer Rekonfiguration.

2. Prozesse, die bereits Ressourcen besitzen, fordern neue Ressourcen an (*Hold and Wait*): Durch eine Rekonfiguration wird eine Menge von Komponenteninstanzen blockiert und einige davon sind bereits inaktiv. Blockierte aber noch aktive Komponenteninstanzen können sich im Besitz eines Threads befinden, der neue Ressourcen anfordert, indem er Methoden bei inaktiven und blockierten Komponenteninstanzen aufrufen will.

3. Eine Ressource gehört entweder zu einem Prozess oder ist verfügbar (*Mutual Exclusion*): Rekonfigurationen und Threads mit Methodenaufrufen konkurrieren um den exklusiven Besitz von Komponenteninstanzen.

4. Mindestens zwei Prozesse warten gegenseitig auf Ressourcen, die ein anderer Prozess in einem geschlossenen System besitzt *Circular Wait*: Das zirkuläre Warten tritt auf, wenn Threads bei blockierten Komponenteninstanzen und der Adaptionsmanager auf den Abschluss der Methodenaufrufe warten.

Zur Verhinderung von Verklemmungen muss das Eintreten mindestens einer dieser Bedingungen verhindert werden. Die *Circular Wait*-Bedingung kann vermieden werden, wenn auch bei blockierten Komponenteninstanzen Methodenaufrufe von Threads zugelassen werden, die zum Zeitpunkt der Blockierung bei mindestens einer blockierten Komponenteninstanz zur Menge der aktiven Threads gehört. Somit muss ein Methodenaufruf zugelassen werden, wenn gilt: $T \in TA_{K_B}$ mit $TA_{K_B} = \bigcup TA_{K_i},\ K_i \in K_B$.

Durchführung der Rekonfigurationsoperationen Da vorausgesetzt wurde, dass Methodenaufrufe in endlicher Zeit abgearbeitet werden, wird ein Thread $T \in TA_{K_B}$ nach endlicher Zeit alle Methodenaufrufe bei Komponenteninstanzen aus K_B abgeschlossen haben. Danach ist er nicht mehr in der Menge TA_{K_B} enthalten. Das gleiche gilt für alle $T \in TA_{K_B}$ zum Zeitpunkt

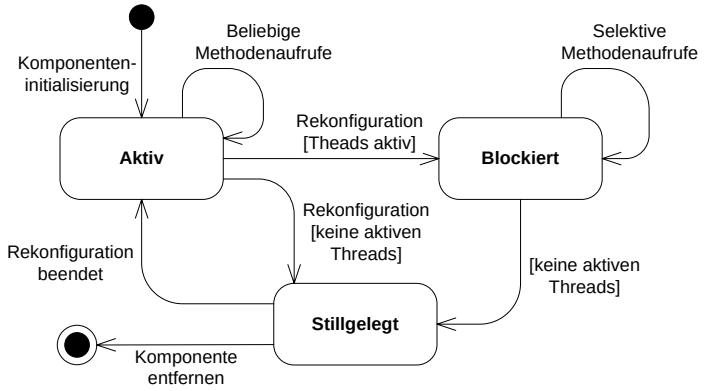

Abbildung 3.16: Zustandsdiagramm mit den Aktivitätszuständen einer Adaptierbaren Komponente

der Blockierung. Da jeder Methodenaufruf eines Thread $T \notin TA_{K_B}$ bei Komponenten aus K_B blockiert wird und damit keine neuen Threads mehr zu TA_{K_B} hinzukommen, sind nach endlicher Zeit alle Komponenten aus K_B inaktiv, d. h. es gilt $TA_{K_B} = \emptyset$.

Sobald dieser Zustand erreicht ist, können alle Rekonfigurationsoperationen ohne Beeinflussung durch Methodenaufrufe durchgeführt werden. Damit wird die Isolationseigenschaft von Rekonfigurationen erfüllt. Neu hinzugekommene Subkomponenten werden automatisch blockiert, d. h. Methodenaufrufe sind noch nicht erlaubt.

Abschluss der Rekonfiguration Alle Blockierungen von Methodenaufrufen werden wieder aufgehoben und während der Rekonfiguration blockierte Methodenaufrufe können fortgesetzt werden. Es ist nicht erforderlich, dass alle Blockierungen gleichzeitig oder in einer bestimmten Reihenfolgen aufgehoben werden, da dadurch keine Verklemmungen auftreten können.

Aktivitätszustände von Komponenteninstanzen Aus dem vorgestellten Verfahren zur Durchführung einer Rekonfiguration ergeben sich die folgenden drei Aktivitätszustände für eine einzelne Komponenteninstanz:

Aktiv Methoden der Komponente können uneingeschränkt von beliebigen Threads aufgerufen werden.

Blockiert Zur Vorbereitung einer Rekonfiguration werden neue Methodenaufrufe nicht mehr zugelassen. Lediglich bereits laufende Threads werden abgearbeitet. In der Menge der zu blockierenden Komponenteninstanzen K_B gibt es noch mindestens einen aktiven Thread.

Stillgelegt Alle Methodenaufrufe wurden abgearbeitet und neue können nicht mehr auftreten. In der Menge der zu blockierenden Komponenteninstanzen K_B gibt es keine aktiven Threads mehr. Die Komponente ist damit inaktiv.

In Abbildung 3.16 sind die drei Aktivitätszustände und die möglichen Zustandsübergänge dargestellt. Der Adaptionsmanager übernimmt die Verwaltung der Zustände.

Mögliche Implementierungen zur transparenten Erkennung von Aktivität bzw. Inaktivität von Komponenten mit Hilfe von Interceptoren werden in Unterabschnitt 4.3.6 beschrieben.

Gleichzeitige Durchführung mehrerer Rekonfigurationen Wenn mehrere Adaptierbare Komponenten innerhalb einer Anwendung verwendet werden, kann es ohne besondere Vorsichtsmaßnahmen passieren, dass zwei oder mehrere unabhängige Rekonfiguration gleichzeitig durchgeführt werden. Beispielsweise ist folgendes Szenario denkbar: Ein Thread ruft Methoden bei einer Komponenteninstanz A auf, während ein andere Thread gleichzeitig Methoden bei einer Komponenteninstanz B aufruft. Die aufgerufenen Methoden in A und B rufen wechselseitig direkt oder indirekt Methoden bei der jeweils anderen Komponenteninstanz auf. Wenn kurz vor diesen weiteren Methodenaufrufen zwei unabhängige Rekonfigurationen bei A und B ausgelöst wurden, kann eine Verklemmung auftreten. Die Methodenaufrufe bei A und B würden jeweils bei der anderen Komponenteninstanz blockieren und die zwei Rekonfigurationen würden auf die Beendigung der beiden Methodenaufrufe warten.

Das weiter oben in diesem Abschnitt beschriebenen Verfahren zur Verhinderung von Verklemmungen ist in diesem Szenario nicht ausreichend. Verklemmungen können immer nur in der kritischen Phase zwischen dem Auslösen einer Rekonfiguration und der damit verbundenen Blockierung von Methodenaufrufen und dem Zeitpunkt, wenn alle Subkomponenten inaktiv sind, auftreten. Die folgenden beiden Verfahren verhindern diese Verklemmungen:

- Eine Verklemmung wird ausgeschlossen, in dem immer nur maximal eine Rekonfiguration gleichzeitig in einer Anwendung die kritische Phase zwischen Start und Inaktivität betreten darf. Dies kann z. B. mit einem anwendungsglobalen Mutex (binäres Semaphor) realisiert werden. Bevor eine Rekonfiguration und damit auch die Blockierung von Subkomponenten gestartet werden kann, muss der Adaptionsmanager das Mutex erwerben. Frühestens nachdem alle blockierten Subkomponenten inaktiv geworden sind, wird es wieder freigegeben.

- Alle Komponenten, die gleichzeitig die kritische Phase einer Rekonfiguration betreten, tauschen Informationen über bereits laufende Threads aus und verhindern, dass Methodenaufrufe dieser Threads blockiert werden. Die Menge der Threads, die auch bei einer Blockierung zugelassen werden muss, ergibt sich aus der Vereinigung der Mengen TA_{K_B} aller Komponenten, bei denen eine Rekonfiguration gleichzeitig stattfindet. Beim zeitlich versetzten Starten von Rekonfigurationen muss jeweils die Menge TA_{K_B} einer neu hinzugekommenen Komponente bei allen anderen Komponenten mit berücksichtigt werden.

3.6 Adaption von zustandsbehafteten Komponenten

Zustandsbehafteten Subkomponenten (siehe Definitionen 3.5 und 3.6) ändern während der Laufzeit durch Methodenaufrufe ihren Zustand und damit auch ihr Verhalten. Im einfachsten Fall entspricht der Zustand einer Komponente einer Menge von Membervariablen in der Implementierungsklasse einer Komponente.

Definition 3.30. Der **Zustand Z einer Komponente** K wird durch eine Menge von Name/Wert-Tupeln ausgedrückt und gespeichert. Jeder Name eines Wertes ist eindeutig für eine bestimmte Komponente.

Bei der Rekonfiguration einer Adaptierbaren Komponente können diese Zustandsinformationen verloren gehen, wenn zustandsbehaftete Subkomponenten aus der aktiven Konfiguration entfernt werden. Dieses Problem kann auf drei unterschiedliche Arten gelöst werden:

1. Nur zustandslose Subkomponenten dürfen in einer Adaptierbaren Komponente verwendet werden. Damit können auch keine Zustandsinformationen verloren gehen. Mit dieser Forderung wird die Anwendbarkeit von Adaptierbaren Komponenten allerdings erheblich eingeschränkt.

2. Alle Subkomponenten werden bereits bei der Entwicklung darauf vorbereitet, dass Zustandsinformationen im Rahmen einer Rekonfiguration verloren gehen können (z. B. durch die Bereitstellung von Default-Werten). Dadurch kann eine Adaptierbare Komponente nach jeder Rekonfiguration nahtlos weiterarbeiten, auch wenn Zustandsinformationen verloren gegangen sind. Diese Lösungsvariante benötigt keine zusätzlichen Mechanismen für Adaptierbare Komponenten. Allerdings lassen sich diese Forderungen möglicherweise nicht für alle Anwendungen erfüllen und der Entwicklungsaufwand erhöht sich u. a. durch zusätzliche Tests.

3. Bestimmte oder alle Zustandsinformationen einzelner Subkomponenten bleiben erhalten und werden gegebenenfalls auf andere (neue) Subkomponenten übertragen. Dazu müssen Adaptierbare Komponenten um geeignete Mechanismen zur Speicherung bzw. Übertragung der Zustandsdaten erweitert werden. Die Semantik der Zustandsinformationen muss bereits bei der Komponentenentwicklung berücksichtigt werden.

Aufgrund der identifizierten Nachteile wurden die ersten beiden Lösungsvarianten nicht weiter verfolgt. Da aber dafür keine Modellerweiterungen notwendig sind, können die Ansätze trotzdem zur Komponentenentwicklung angewendet werden.

Für die Entwicklung von Mechanismen zur Zustandsspeicherung bzw. -übertragung kann man zwei Klassen von Rekonfigurationen unterscheiden, die im Folgenden jeweils spezielle Mechanismen vorgestellt werden.

Eine generelles Problem im Rahmen einer Zustandsübertragung zwischen Komponenteninstanzen wird in dieser Arbeit allerdings nicht behandelt: Auch wenn Zustandsinformationen typkompatibel sind, müssen sie nicht die gleiche semantische Bedeutung in zwei Komponenten besitzen. Ohne detailliertes Wissen über die Implementierung der Komponenten können daher keine Zustandsübertragungen definiert werden. Zur automatischen Überprüfung von Zustandsübertragungen sind auf jeden Fall Verhaltensbeschreibungen notwendig, aber es kann an dieser Stelle nicht geklärt werden, ob eine Überprüfung algorithmisch überhaupt möglich ist. Für die vorgestellten Mechanismen wird daher vorausgesetzt, dass die semantische Kompatibilität der Komponenten durch Entwickler überprüft und getestet wird.

3.6.1 Verfahren zur Zustandsübertragung beim Ersetzen von Komponenten

Wenn im Rahmen einer Rekonfiguration lediglich eine Komponente entfernt und durch eine andere ersetzt werden soll, die im Wesentlichen die gleiche Funktion erfüllt (z. B. der Austausch der AES- durch eine Blowfish-Komponente in der Kryptographiekomponente), müssen Zustandsinformationen nur zwischen diesen beiden Komponenten ausgetauscht werden. Die Bedingung dieser Verfahren wird aber nur von einer Teilmenge aller möglichen Rekonfigurationen erfüllt.

Für die Übertragung der Zustandsinformationen von einer auf eine andere Komponente, d. h. den Austausch einer Komponente, wurden zwei Verfahren entwickelt [Ric04], die im Folgenden beschrieben werden.

Abbildungsfunktion zur Zustandstransformation Für zwei Komponenten K_1 und K_2 wird eine spezielle Abbildungsfunktion definiert, die den Zustand von K_1 auf K_2 abbildet. Die Funktion nimmt alle Zustandsinformationen von K_1 als Eingabeparameter und liefert Werte für die Zustandsinformationen von K_2 als Ergebnis. Dabei können sowohl identische Abbildungen von einer Zustandsinformation von K_1 auf eine andere von K_2, als auch zusätzliche Berechnungen (z. B. Umrechnung von Maßeinheiten) definiert werden. Die Abbildungsfunktion muss speziell für die beiden Komponenten definiert werden und ist auch nicht in der Regel nicht kommutativ. Ein großer Nachteil dieser Lösung besteht im Entwicklungsaufwand, da wenn eine bestimmte Komponente durch n verschiedene andere Komponenten in unterschiedlichen Rekonfigurationen ersetzt werden soll, auch n verschiedene Abbildungsfunktionen definiert werden müssen.

Komponentenfamilie und generischer Zustandstransformation Um die Vielzahl an notwendigen Abbildungsfunktionen zu reduzieren, werden bei dieser Lösung eine Menge von ähnlichen Komponenten zu einer *Komponentenfamilie* zusammengefasst. Für alle Komponenten einer Komponentenfamilie wird ein gemeinsamer Zustand bestehend aus einer Menge von Name/Wert-Tupeln definiert. Für jede Komponente werden anschließend zwei Abbildungsfunktionen definiert, die den Zustand der Komponente auf den gemeinsamen Zustand abbildet und umgekehrt. Aus diese Weise werden für n Komponenten einer Komponentenfamilie immer nur genau $2 * n$ Abbildungsfunktionen benötigt und die Komponenten können dennoch beliebig im Rahmen von Rekonfigurationen ausgetauscht werden.

3.6.2 Verfahren zur Zustandsübertragung bei strukturellen Änderungen

Bei strukturellen Änderungen während einer Rekonfiguration (z. B. ersatzlose Entfernung der Kompressionskomponente in der Kryptographiekomponente), die über einen Austausch von einzelnen Komponenten hinausgehen, können die zuvor beschriebenen Verfahren beim Ersetzen von Komponenten nicht angewendet werden, da es keine 1:1-Zuordnung zwischen einer entfernten und einer neu hinzugefügten Komponente gibt. Daher wurde ein weiteres Verfahren entwickelt, das auch bei größeren strukturellen Änderungen angewendet werden kann.

Die Grundidee des Verfahrens besteht darin, die umgebende Adaptierbare Komponente zur Zwischenspeicherung von Zustandsinformationen von Subkomponenten zu nutzen. Dazu wird einer Adaptierbaren Komponente analog zu Definition 3.30 eine Menge von Name/Wert-Tupeln zugeordnet. Jede Subkomponente kann für eine Teilmenge ihrer Zustandsinformationen eine Zuordnung zu typkompatiblen Name/Wert-Tupeln der zugehörigen Adaptierbaren Komponente definieren. Der Entwickler einer Adaptierbaren Komponente ist dafür verantwortlich, diese Zuordnungen zu definieren. Mehrere Subkomponenten können Zuordnungen zu einem bestimmten Name/Wert-Tupel der Adaptierbaren Komponente besitzen und auf diese Weise Zustandsinformationen im Rahmen einer Rekonfiguration austauschen.

Definition 3.31. Eine **Zuordnung von Zustandsinformationen** ist eine Menge von Tupeln bestehend aus dem Namen eines Zustandswertes der Komponente und dem zugeordneten Namen des Zustandswertes der Adaptierbaren Komponente. Die miteinander verknüpften Werte müssen den gleichen Typ besitzen.

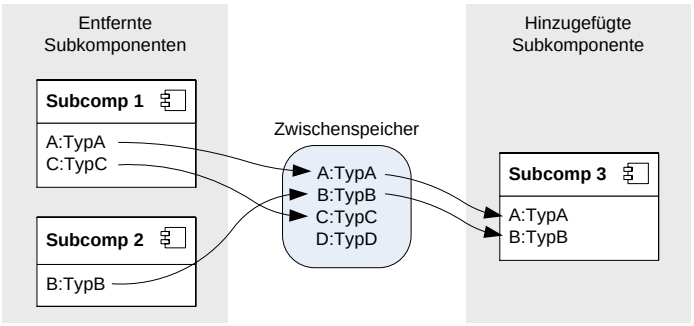

Abbildung 3.17: Zustandsübertragung im Rahmen einer Rekonfiguration

Im Rahmen einer Rekonfiguration werden die folgenden Schritte zur Zustandsübertragung durchgeführt (siehe Abbildung 3.17):

- Beim Entfernen einer Subkomponente aus der aktiven Konfiguration werden, nachdem die Komponente deaktiviert wurde (siehe Abschnitt 3.5), alle Werte von Zustandsinformationen, für die eine Zuordnung definiert wurde, in die zugeordneten Werte der Adaptierbaren Komponente übertragen. Bestehende Werte werden dabei überschrieben.

- Beim Hinzufügen einer neuen Subkomponente zur aktiven Konfiguration werden alle Werte von Zustandsinformationen der Adaptierbaren Komponente, für die eine Zuordnung definiert wurde, für Werte der Subkomponente übernommen. Vorhandene Standardwerte werden dabei überschrieben.

Mit Hilfe von Adaptionsoperatoren (siehe Unterabschnitt 3.3.4) können Typanpassungen entwickelt werden, wenn ein Datenaustausch aufgrund unterschiedlicher Typen sonst nicht möglich wäre. Adaptionsoperatoren müssen dazu die Komponentenschnittstelle um zusätzliche Parameter erweitern, die dann auf vorhandene Parameter der Komponente abgebildet werden. Beispielsweise können die Schlüssel der verschiedenen Kryptographiekomponenten in unterschiedlichen Formaten vorliegen, etwa als String oder Bytearray. Wenn zum Zustandsaustausch in der Adaptierbaren Komponente ein Tupel mit dem Namen „Key“ und dem Typ String vorgesehen wurde, muss für die Verschlüsselungskomponente mit dem Bytearray als Schlüssel ein Adaptionsoperator entwickelt werden, der die Typumwandlung Bytearray $\leftrightarrow$ String vornimmt.

Im Gegensatz zum in Unterabschnitt 3.6.1 vorgestellten Verfahren beim Ersetzen von Komponenten können Zustandsinformationen zwischen beliebig vielen Komponenten ausgetauscht werden. Es ist lediglich erforderlich, dass die ausgetauschten Daten den gleichen Typ verwenden. Als Spezialfall kann mit dem Verfahren aber auch der Austausch von Zustandsinformationen beim Ersetzen einer Komponente modelliert werden. Damit genügt es, wenn Adaptierbaren Komponenten um das in diesem Abschnitt beschriebene Verfahren zur Zustandsübertragung bei strukturellen Änderungen erweitert werden.

Kapitel 4

Modelltransformation und Laufzeitunterstützung für Adaptierbare Komponenten

Die Konzepte der im vorhergehenden Kapitel vorstellten Adaptierbaren Komponenten werden von existierenden Komponentenplattformen nicht direkt unterstützt. Erst durch eine Kombination von Modelltransformation, Werkzeugen (z. B. zur Installation, Entwicklung und Quelltext-Erzeugung) und einer Laufzeitbibliothek können Adaptierbare Komponenten innerhalb von Anwendungen eingesetzt werden.

Zur Unterstützung von Adaptierbaren Komponenten auf einer bestimmten Komponentenplattform werden zwei Phasen unterschieden:

1. Entwicklung von Modelltransformation, Laufzeitbibliothek und Werkzeugen für beliebige Adaptierbare Komponenten

2. Entwicklung, Installation und Ausführung von bestimmten Adaptierbaren Komponenten innerhalb von Anwendungen

Die erste Phase muss für jede Komponentenplattform nur einmal durchgeführt werden, wohingegen die zweite Phase für jede Adaptierbare Komponente erforderlich ist. Für die Ausführung der zweiten Phase werden die Ergebnisse der ersten Phase zwingend benötigt.

In diesem Kapitel werden die Schritte vom plattformunabhängigen Modell Adaptierbarer Komponenten bis zur Implementierungsunterstützung für verschiedene Komponentenplattformen erläutert. Dabei wird nicht eine bestimmte Komponentenplattform behandelt, sondern ein Verfahren entwickelt, um Adaptierbare Komponenten auf beliebige Komponentenplattformen abbilden zu können. Im Kapitel 5 wird dieses Verfahren zur Entwicklung einer konkreten Unterstützung von JavaBeans, EJB, COM und Web-Services angewendet.

Am Anfang des Kapitels werden drei verschiedene Arten von plattformspezifischen Modellen (PSMs) mit unterschiedlichen Eigenschaften eingeführt. Anschließend wird ein Verfahren zur Entwicklung von PSMs und damit auch zur Definition von Modelltransformationen in Abhängigkeit von Eigenschaften der Zielkomponentenplattform beschrieben. Danach wird unter Berücksichtigung der prinzipiell möglichen Eigenschaften von Komponentenplattformen ein generisches Framework vorgestellt, das die Entwicklung und Implementierung von plattformspezifischen Werkzeugen und Laufzeitbibliotheken wesentlich vereinfacht. Unter Nutzung dieses Frameworks wird gezeigt, wie die drei eingangs eingeführten Arten von PSMs für unterschiedliche Komponentenplattformen entwickelt werden können.

Eigenschaft	Installationszeit-PSM	Startzeit-PSM	Laufzeit-PSM
Installations-Konfiguration	ja	ja	ja
Startzeit-Konfiguration	nein	ja	ja
Laufzeit-Rekonfiguration	nein	nein	ja
Zeit für Instanz-Erzeugung	gering	mittel	hoch
Zeit für Methodenaufruf	gering	gering	hoch
Komplexität zur Laufzeit	gering	mittel	hoch
Flexibilität	gering	mittel	hoch

Tabelle 4.1: Eigenschaften der verschiedenen PSMs

4.1 Übersicht zu plattformspezifischen Modellen für Adaptierbare Komponenten

Adaptierbare Komponenten sind in Form eines plattformunabhängigen Modells (*Platform Independent Model*, PIM) definiert. Mit einer Modelltransformation innerhalb eines MDA-Prozesses (siehe Abschnitt 2.7) wird dieses PIM auf ein plattformspezifisches Modell (PSM) abgebildet und damit um Eigenschaften der Zielkomponentenplattform erweitert.

Zur Erhöhung der Flexibilität wird jedoch für eine bestimmte Komponentenplattform nicht nur ein, sondern drei PSMs mit jeweils unterschiedlichen Eigenschaften unterstützt. Der wesentliche Unterschied besteht in den jeweils möglichen Zeitpunkten für die Festlegung bzw. Änderung von Komponentenparametern. Je nach Anforderungen einer Anwendung können Entwickler das jeweils passende PSM auswählen, ohne Änderungen am PIM vornehmen zu müssen. Alle nicht automatisch erzeugten Bestandteile von Komponentenimplementierungen, insbesondere die Quelltexte, bleiben für alle drei PSMs gleich.

Tabelle 4.1 fasst die wesentlichen Eigenschaften dieser Modelle zusammen. Im Folgenden werden die drei PSMs kurz vorgestellt. Eine ausführliche Beschreibung folgt in Abschnitt 4.4. Falls erforderlich können auch noch weitere PSMs mit speziellen Eigenschaften entwickelt werden.

Laufzeit-PSM Dieses PSM bietet die größte Flexibilität zur Laufzeit. Die Parameter einer Adaptierbaren Komponente können sowohl bei der Erzeugung als auch während der Laufzeit geändert werden. Diese Flexibilität wird allerdings auch mit der höchsten Komplexität und der geringsten Performance zur Laufzeit unter den drei PSMs erkauft.

Alle Subkomponententypen, die in der Konfigurationsbeschreibung der Adaptierbaren Komponente enthalten sind, müssen zur Laufzeit verfügbar sein, da prinzipiell jede Konfiguration ausgewählt werden kann.

Startzeit-PSM Dieses PSM erlaubt lediglich die Festlegung von Werten für Komponentenparameter einer Adaptierbaren Komponente zum Zeitpunkt der Erzeugung einer Komponenteninstanz. Eine spätere Änderung dieser Parameter, d. h. die Ausführung einer Rekonfiguration, ist nicht mehr möglich.

Da zum Zeitpunkt der Erzeugung prinzipiell alle definierten Konfigurationen ausgewählt werden können, müssen zur Laufzeit alle Subkomponententypen vorliegen. Die komplexe Lo-

gik zur Steuerung von Rekonfigurationen kann entfallen, da eine Konfiguration später nicht mehr verändert werden kann. Verbindungen zwischen Subkomponenten und die Verknüpfung von internen und externen Ports müssen nur einmal bei der Erzeugung der Adaptierbaren Komponente initialisiert werden und bleiben dann unverändert.

Installationszeit-PSM Bei diesem PSM wird die Konfiguration von Adaptierbaren Komponenten zur Entwicklungs- bzw. Installationszeit festgelegt. Weder beim Erzeugen noch während der Laufzeit kann diese Konfiguration verändert werden. Dadurch müssen nur Subkomponententypen zur Laufzeit verfügbar sein, die auch in der ausgewählten Konfiguration verwendet werden.

Durch die Festlegung auf eine Konfiguration und der damit verbundenen Einschränkung der Flexibilität wird zur Laufzeit die größte Performance unter den drei PSMs erreicht. Außerdem wird nur eine minimale Laufzeitunterstützung benötigt.

4.2 Verfahren zur Entwicklung von plattformspezifischen Modellen

Dieser Abschnitt beschreibt die prinzipielle Vorgehensweise zur Abbildung des Metamodells (siehe Anhang B) und der zugrunde liegenden Konzepte Adaptierbarer Komponenten auf eine bestimmte Komponentenplattform. In Kapitel 5 wird dieses Verfahren beispielhaft für die Entwicklung einer Unterstützung der Komponentenplattformen EJB (Abschnitt 5.1), JavaBeans (Abschnitt 5.2) und COM (Abschnitt 5.3) sowie als Sonderfall für Web-Services[1] (Abschnitt 5.4) angewendet. Weitere Komponentenplattformen können in ähnlicher Weise unterstützt werden.

Als erster Schritt zur Definition einer Modelltransformation müssen die Eigenschaften einer Komponentenplattform in Hinblick auf verschiedene Kriterien ermittelt werden, die für die Unterstützung von Adaptierbaren Komponenten relevant sind. Darauf aufbauend werden dann die Konzepte und Modellbestandteile von Adaptierbaren Komponenten auf verfügbare Konzepte der Komponentenplattform abgebildet. In diesem Zusammenhang muss auch das generische Framework für Adaptierbare Komponenten, das in Abschnitt 4.3 vorgestellt wird, um die plattformspezifischen Teile erweitert werden. Zur Unterstützung einer Komponentenplattform gehört im letzten Schritt auch die Implementierung von Entwicklungs- und Installationswerkzeugen (z. B. Code-Generierung und Packen von Komponenten). Dafür wird ebenfalls auf das Framework zurückgegriffen. Die einzelnen Schritte werden in den folgenden Abschnitten detailliert beschrieben.

4.2.1 Bestimmung der Eigenschaften einer Komponentenplattform

Vor der Erstellung von PSMs müssen zuerst die Eigenschaften der Zielkomponentenplattform ermittelt werden. Dabei sind nur solche Eigenschaften von Interesse, die später für die Umsetzung der Konzepte von Adaptierbaren Komponenten benötigt werden.

Bei der Ermittlung wird zwischen zwei Klassen von Eigenschaften unterschieden:

Merkmale beschreiben den Funktionsumfang, d. h. sie legen fest, ob eine bestimmte Funktion von der Zielkomponentenplattform unterstützt wird oder nicht (Ja- oder Nein-Entschei-

[1] Web-Services sind keine Komponentenplattform, können aber unter bestimmten Bedingungen ähnlich behandelt werden.

dung). Sie können damit sehr einfach für prinzipielle Implementierungsentscheidungen verwendet werden (z. B. in Entscheidungsbäumen). Andere Eigenschaften können von einem bestimmten Merkmal abhängig sein. In diesem Fall sind die abhängigen Eigenschaften nur relevant, wenn das zugehörige Merkmal erfüllt ist, d. h. die abhängigen Eigenschaften sind Verfeinerungen des Merkmals. Neben der Aussage, ob ein Merkmal durch die Komponentenplattform unterstützt wird, muss in vielen Fällen auch ermittelt werden, *wie* das Merkmal verwendet werden kann (z. B. konkrete Programmierschnittstellen).

Erläuterungseigenschaften beschreiben komplexe Vorgänge, Zusammenhänge und Schnittstellen der Zielkomponentenplattform, die individuell bei der Implementierung berücksichtigt werden müssen.

Mit Hilfe von Merkmalen kann die grobe Architektur einer plattformspezifischen Implementierung festgelegt werden (siehe Unterabschnitt 4.2.2). Unter Verwendung des generischen Frameworks (siehe Abschnitt 4.3) könnte dieser Vorgang auch automatisiert werden. Aufgrund der großen Unterschiede zwischen Komponentenplattformen müssen allerdings in jedem Fall die Erläuterungseigenschaften berücksichtigt werden, um vollständige plattformspezifische Implementierungen zu entwickeln.

Im Folgenden werden die verschiedenen Merkmale und Erläuterungseigenschaften von Komponentenplattformen vorgestellt, nochmals orthogonal unterteilt in vier verschiedene Kategorien. Alle Eigenschaften sind nummeriert, um bei Implementierungsentscheidungen in späteren Abschnitten darauf Bezug nehmen zu können. Für jedes Merkmal wird wenn möglich eine Beispiel-Komponentenplattform[2] angegeben, die dieses Merkmal unterstützt.

Komponentenaufbau Diese Kategorie fasst alle Eigenschaften zusammen, die den Aufbau von plattformspezifischen Komponenten aus verschiedenen Artefakten (z. B. Schnittstellen, Parameter und Implementierungsklassen) beschreiben.

E–1 **Unterstützung zusammengesetzter Komponenten** (Merkmal): Eine Komponente kann andere Komponenten enthalten und es gibt einen Mechanismus zur Verknüpfung von externen und internen Ports. Beispiel: Fractal

E–2 **Unterstützung einer variablen Verknüpfung von internen und externen Ports bei zusammengesetzten Komponenten** (Merkmal, abhängig von Eigenschaft E-1): Die Verknüpfung von internen und externen Ports lässt sich zur Laufzeit durch Mechanismen der Komponentenplattform verändern (z. B. durch ein Art *Metaobject Protocol*). Das Merkmal wird nicht erfüllt, wenn die Verknüpfung lediglich statisch, z. B. mittels eines Komponentendeskriptors, festgelegt werden kann. Beispiel: Fractal

E–3 **Trennung von Schnittstellen und Implementierung bei Komponenten** (Merkmal): Die Schnittstellen von *provided*- und *required*-Ports müssen unabhängig von einer konkreten Implementierung definiert werden. In Java bedeutet das beispielsweise, dass die Schnittstelle eines Ports durch ein Java-Interface definiert wird und die Implementierungsklasse dieses Interface implementiert. Insbesondere kann es mehrere Implementierungen für die gleiche Schnittstelle geben. Beispiel: EJB

[2] Verschiedene Komponentenplattformen werden in Abschnitt 2.5 und Kapitel 5 beschrieben.

E–4 **Unterstützung von Komponentenparametern** (Merkmal): Die Komponentenplattform stellt explizite Mechanismen zur Definition und zum Zugriff von Komponentenparametern verschiedener Typen (mindestens Integer, Boolean und String) bereit. Die Komponentenparameter können zur Laufzeit geändert werden. Optional können Parameterwerte auch bereits bei der Installation oder Erzeugung festgelegt werden. Beispiel: JavaBeans

E–5 **Komponentenarten** (Erläuterungseigenschaft): Welche unterschiedlichen Komponentenarten[3] werden von der Komponentenplattform unterstützt und welche besonderen Eigenschaften besitzen sie? Welche dieser Arten können von Adaptierbaren Komponenten bzw. Subkomponenten verwendet werden?

E–6 **Komponentenaufbau** (Erläuterungseigenschaft): Welche Artefakte der Programmiersprache (Klassen und Interfaces) werden zur Entwicklung eine Komponente benötigt? Welche Programmierrichtlinien müssen dabei eingehalten werden (z. B. Namenskonventionen für bestimmte Methoden)?

Kommunikation und Kommunikationsbeziehungen Diese Kategorie fasst alle Eigenschaften zusammen, die die Mechanismen und Konzepte zur Zusammenarbeit von mehreren Komponenten beschreiben.

E–7 **Unterstützung von *required*-Ports** (Merkmal): Mit Hilfe von *required*-Ports können erforderliche Komponenteninstanzen definiert werden (siehe Unterunterabschnitt 3.3.1.1). Es genügt nicht, wenn die Komponentenplattform lediglich Abhängigkeiten zu bestimmten Komponententypen ausdrücken kann. Beispiel: Fractal

E–8 **Unterstützung von *required*-Ports mit mehreren Verbindungen** (Merkmal, abhängig von Eigenschaft E-7): Ein einzelner *required*-Port kann Verbindungen zu mehreren *provided*-Ports speichern (siehe Unterunterabschnitt 3.3.1.1).

E–9 **Unterstützung von mehr als einem *provided*-Port pro Komponente** (Merkmal): Eine Komponente kann mehrere unterschiedliche *provided*-Ports besitzen, die zur Kommunikation mit anderen Komponenten verwendet werden können. Eine Navigation zwischen den verschiedenen Ports wird dabei durch entsprechende Funktionen der Komponentenplattform unterstützt. Beispiel: COM

E–10 **Unterstützung expliziter Verbindungen zwischen Komponenteninstanzen (Merkmal)**: Verbindungen und Abhängigkeiten zwischen verschiedenen Komponenteninstanzen durch Ports können explizit, d. h. außerhalb der Komponenten definiert werden (*Separation of Concerns*), z. B. mit Hilfe von Deskriptoren. Beispiel: Fractal

E–11 **Definition bzw. Änderung von expliziten Verbindungen zur Laufzeit** (Merkmal, abhängig von Eigenschaft E-10): Explizite Verbindungen zwischen verschiedenen Komponenteninstanzen können auch zur Laufzeit mit Hilfe von entsprechenden Funktionen der Komponentenplattform definiert bzw. geändert werden. Beispiel: Fractal

[3] Der sonst in der Literatur verwendete Begriff *Komponententyp* wurde bereits in Unterabschnitt 3.3.1 anderweitig definiert. Um Verwechselungen zu vermeiden, wird daher in dieser Arbeit der Begriff *Komponentenart* verwendet.

E–12 **Verbot von nicht expliziten Verbindungen zwischen Komponenten** (Merkmal, abhängig von Eigenschaft E-10): Verbindungen und Abhängigkeiten zwischen verschiedenen Komponenten dürfen nur explizit durch die Verbindung von *required*- und *provided*-Ports definiert werden. Im Programmcode der Komponenten dürfen nur diese expliziten Verbindungen verwendet werden.

E–13 **Unterstützung synchroner Kommunikation zwischen Komponenten** (Merkmal): Der Aufruf einer Methode bei einer Komponente blockiert den Aufrufer so lange, bis die Methode vollständig abgearbeitet wurde und optional ein Rückgabewert zurückgegeben wurde. Wenn dieses Merkmal nicht unterstützt wird, erfolgt der Nachrichtenaustausch zwischen Komponenten asynchron. Beispiel: JavaBeans

Laufzeitverhalten und Lebenszyklus Diese Kategorie fasst alle Eigenschaften zusammen, die Vorgänge zur Laufzeit von Komponenten beschreiben.

E–14 **Unterstützung zustandsbehafteter (*stateful*) Komponenteninstanzen** (Merkmal): Komponenteninstanzen enthalten Zustandsinformationen, die einerseits durch Methodenaufrufe verändert werden können und andererseits die Ausführung der Methoden beeinflussen (siehe auch Unterabschnitt 3.3.1 und Abschnitt 3.6). Beispiel: EJB

E–15 **Unterstützung von Funktionen zur Ermittlung bzw. Änderung der Eigenschaften einer Komponente** (Merkmal): Die Komponentenplattform stellt Funktionen in der Art eines *Metaobject Protocols* (MOP, siehe Unterabschnitt 2.8.3) bereit, um Komponenteneigenschaften, wie z. B. unterstützte Ports, abzufragen (*Introspection*) und teilweise auch zu ändern (*Intercession*). Beispiel: Fractal

E–16 **Verhinderung von gleichzeitigen Methodenaufrufen (*reentrant*) bei einer Komponenteninstanz** (Merkmal): Zur Vereinfachung der Entwicklung verhindert die Komponentenplattform gleichzeitige Methodenaufrufe durch unterschiedliche Threads bei einer Komponente. Dadurch müssen Fragen zur Synchronisierung und zur Verhinderung von Verklemmungen bei Anwendungen mit mehreren Threads bei der Programmierung (*thread-safe* Programmierung) nicht berücksichtigt werden. Eventuell lässt sich dieses Merkmal auch nur für bestimmte Komponenten aktivieren bzw. deaktivieren (z. B. mittels Komponentendeskriptoren). Beispiel: EJB

E–17 **Unterstützung von dynamischen Proxies und Reflection zum Methodenaufruf** (Merkmal): Die Programmiersprache und die Komponentenplattform erlauben zum einen den Aufruf von beliebigen, zur Übersetzungszeit noch nicht bekannten Methoden mit Hilfe von Reflection. Zum anderen kann zur Laufzeit ein Proxy-Objekt (genannt *dynamic proxy*) erzeugt werden, dass eine beliebige, zur Übersetzungszeit noch nicht bekannte Schnittstelle implementiert. Beispiel: JavaBeans

E–18 **Unterstützung der Modifikation des Programmcodes von Komponenten zur Laufzeit** (Merkmal): Die Programmiersprache und die Komponentenplattform erlauben den Programmcode von Komponenten während des Ladens zur Laufzeit zu verändern. Mit diesen Mechanismen ist es mindestens möglich, zusätzliche Methoden hinzuzufügen, sowie Programmcode vor und nach bestimmten Methoden einzufügen. Beispiel: JavaBeans

E–19 **Verbotene Funktionen für Komponenten** (Erläuterungseigenschaft): Dürfen bestimmte Funktionen der Programmiersprache bzw. zugehöriger Bibliotheken und Frameworks nicht für die Entwicklung von Komponenten verwendet werden (z. B. Laden und Speichern von Dateien, Netzwerkzugriffe, eigene Classloader, Reflection)? Diese Verbote müssen bei der Implementierung der Unterstützung von Adaptierbaren Komponenten berücksichtigt werden.

E–20 **Erzeugung und Löschen von Komponenten** (Erläuterungseigenschaft): Welche Schritte schreibt die Komponentenplattform zur Erzeugung und Initialisierung sowie zum Löschen einer Komponente vor (Komponenten-Lebenszyklus)?

Verwaltung Diese Kategorie fasst alle Eigenschaften zusammen, die den Entwicklungsprozess bis zur Installation von Komponenten beschreiben.

E–21 **Komponentendeskriptoren** (Erläuterungseigenschaft): Wird zusätzlich zum Programmcode noch die Definition von Metainformationen für eine Komponente benötigt (z. B. Komponentendeskriptoren, Eintrag in Verzeichnisse)? Wie können die Metainformationen bei Adaptierbaren Komponenten bzw. Subkomponenten bestimmt werden?

E–22 **Komponentenarchive** (Erläuterungseigenschaft): In welcher Form (z. B. Komponentenarchive oder Verzeichnisstruktur) müssen die Komponenten für die Installation vorliegen? Können mehrere Komponenten für eine gemeinsame Installation in einem Archiv gebündelt werden?

E–23 **Werkzeugunterstützung im Entwicklungsprozess** (Erläuterungseigenschaft): Welche Werkzeuge werden bis zur Installation von Komponenten benötigt? Die Bandbreite der Werkzeuge reicht von Codegeneratoren (z. B. XDoclet [xdo05] bei EJB), über spezielle Editoren für Metainformationen von Komponenten bis zu Installationswerkzeugen.

4.2.2 Abbildung von Modellbestandteilen und Konzepten

Alle Modellbestandteile, die in Abschnitt 3.3 beschrieben wurden, müssen durch Konzepte der Zielkomponentenplattform entweder direkt unterstützt oder geeignet emuliert werden. Als Ergebnis dieses Prozesses muss sowohl die Struktur (z. B. Schnittstellen, Methodensignaturen) als auch das Verhalten (z. B. Aufbau von Verbindungen, Rekonfiguration, Initialisierung) von Adaptierbaren Komponenten bei einer bestimmten Komponentenplattform eindeutig definiert sein. Für die Implementierungsentscheidungen wird auf die im vorigen Abschnitt definierten Eigenschaften von Komponentenplattformen zurückgegriffen.

Insbesondere bei der Emulation von Konzepten ist es oft notwendig, durch die Festlegung von Programmierrichtlinien bestimmte Vorgehensweisen vorzuschreiben bzw. zu verbieten. Diese Richtlinien müssen bereits bei der Komponentenentwicklung eingehalten werden, da eine Überprüfung zur Laufzeit oft nicht möglich ist bzw. zu inakzeptablen Leistungseinbußen führen würde. Es ist aber denkbar, dass Entwicklungswerkzeuge mit Hilfe einer statischen Codeanalyse die Quelltexte auf Verletzungen von Programmierrichtlinien überprüfen und an den Entwickler melden. Aber auch eine Codeanalyse bietet keine hundertprozentige Sicherheit

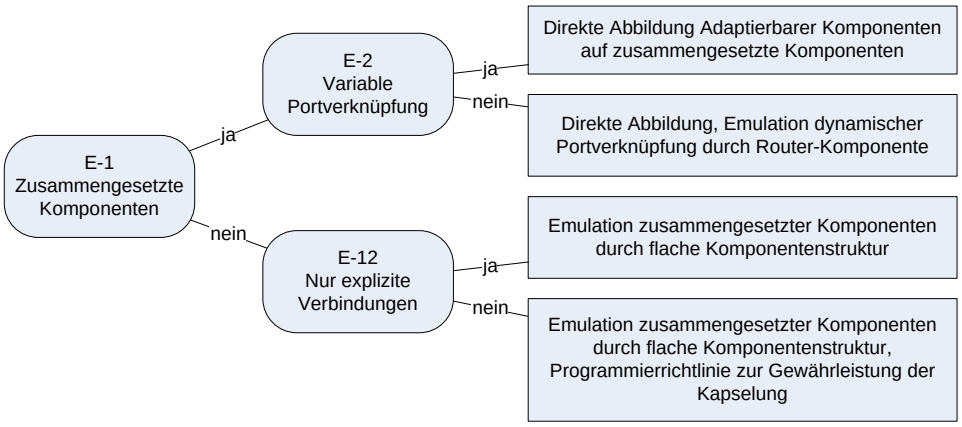

Abbildung 4.1: Umsetzung von zusammengesetzten Komponenten in Abhängigkeit von Merkmalen der Komponentenplattform

für die Einhaltung aller Programmierrichtlinien. Letztendlich sind die Komponentenentwickler dafür verantwortlich, dass alle Programmierrichtlinien berücksichtigt werden.

Im Folgenden wird jeweils beschrieben, wie die Bestandteile und Konzepte von Adaptierbaren Komponenten in Abhängigkeit der Eigenschaften der jeweiligen Zielkomponentenplattform unterstützt werden.

4.2.2.1 Zusammengesetzte Komponenten

Adaptierbare Komponenten verwenden das Konzept von zusammengesetzten Komponenten (auch als *composite* oder *nested components* bezeichnet) zur Kapselung einer Menge von Subkomponenten und unterschiedlicher Konfigurationen. Die Merkmale E-1, E-2 und E-12 sowie die Erläuterungseigenschaft E-5 beeinflussen direkt die Umsetzung des Konzepts durch die Zielkomponentenplattform. Die Auswirkungen dieser Parameter auf die Implementierung sind als Entscheidungsbaum in Abbildung 4.1 dargestellt.

Wenn die Zielkomponentenplattform zusammengesetzte Komponenten unterstützen (Eigenschaft E-1 erfüllt), kann eine Adaptierbare Komponente direkt als zusammengesetzte Komponente realisiert werden. Wenn die Verknüpfungen von externen mit internen Ports nicht dynamisch festgelegt und geändert werden können (Eigenschaft E-2 nicht erfüllt), muss diese Funktion beim Startzeit- und Laufzeit-PSM emuliert werden. Dies kann z. B. durch die Einführung einer speziellen Router-Subkomponente realisiert werden, die mit allen anderen Subkomponenten verbunden wird. Sie kann durch Codegenerierung und/oder die Nutzung von Laufzeitbibliotheken realisiert werden. In Abhängigkeit der gerade aktiven Konfiguration leitet die Router-Komponente Methodenaufrufe an die entsprechenden Subkomponenten weiter.

Bei Komponentenplattformen ohne ein hierarchisches Komponentenmodell (Eigenschaft E-1 nicht erfüllt) muss das Konzept von zusammengesetzten Komponenten emuliert werden. Dazu wird eine spezielle Komponente als Platzhalter für eine Adaptierbare Komponente benutzt. Sie verwaltet eine Liste mit allen ursprünglichen Subkomponenten. Adaptierbare Komponenten und Subkomponenten im PIM werden somit auf die gleiche Art von Komponenten in der Zielplattform abgebildet, d. h. eine ursprünglich hierarchische Struktur wird in eine

flache Struktur umgewandelt. Wenn die Komponentenplattform auch nicht explizit definierte Verbindungen zwischen Komponenten erlaubt (Eigenschaft E-12 erfüllt), kann die Einhaltung des Prinzip der Kapselung (siehe Unterabschnitt 3.3.1), was durch zusammengesetzte Komponenten erreicht wird, jedoch nicht mehr garantiert werden, da prinzipiell jede Komponente auch auf ursprüngliche Subkomponenten innerhalb einer Adaptierbaren Komponente zugreifen kann. Alle Komponenten können sich in diesem Fall nur freiwillig in Form einer Programmierrichtlinie an die Einhaltung der Kapselung halten.

4.2.2.2 Subkomponenten und Glue-Code

Jede Subkomponente[4] und jeder Glue-Code im PIM werden durch eine Komponente der Zielkomponentenplattform implementiert. Wenn die Komponentenplattform mehrere unterschiedliche Komponentenarten unterstützt (Erläuterungseigenschaft E-5), muss geprüft werden, ob alle oder nur einige dieser Arten innerhalb von Adaptierbaren Komponenten unterstützt werden sollen. Nicht erlaubte Arten werden in einer entsprechenden Programmierrichtlinie festgehalten.

Auf Grundlage der Erläuterungseigenschaft E-20 muss plattformspezifischer Programmcode zur Unterstützung des Lebenszyklus von Subkomponenten bzw. Glue-Code entwickelt werden. Für das Erzeugen bzw. Löschen von Komponenten müssen dazu die jeweiligen Verfahren der Zielkomponentenplattform eingehalten werden. Das Framework zur Unterstützung Adaptierbarer Komponenten (siehe Abschnitt 4.3) bietet verschiedene Erweiterungsmöglichkeiten in Form von abstrakten Methoden, die überschrieben werden müssen, um diesen Prozess zu vereinfachen.

4.2.2.3 Komponentenparameter

Komponentenparameter von Adaptierbaren Komponenten werden, wenn Merkmal E-4 erfüllt ist, direkt auf das Parameterkonzept der Zielkomponentenplattform abgebildet. Dafür müssen die entsprechenden Verfahren (z. B. Hilfefunktionen, spezielle Schnittstellen) der Komponentenplattform verwendet werden.

Anderenfalls werden Parameter durch spezielle Zugriffsmethoden in der Komponentenschnittstelle emuliert. Diese Zugriffsmethoden übernehmen das von JavaBeans bekannte Prinzip mit *getter*- und *setter*-Methoden. Das bedeutet, dass für einen Komponentenparameter (siehe Unterunterabschnitt 3.3.1.2) mit dem Namen *param* und dem Typ T die folgenden beiden Methoden definiert werden (in Java-Syntax):

```
public T getParam();            // getter-Methode für Parameter "param"
public void setParam(T value); // setter-Methode für Parameter "param"
```

4.2.2.4 Komponentenports

Für die Umsetzung von Komponentenports müssen die Merkmale E-3, E-7, E-8 und E-9 berücksichtigt werden. Die Auswirkungen dieser Parameter auf die Implementierung sind als Entscheidungsbaum in Abbildung 4.2 dargestellt.

[4] Wenn es sich bei einer Subkomponente um eine Adaptierbare Komponente handelt, gilt zusätzlich die Beschreibung des vorigen Abschnittes.

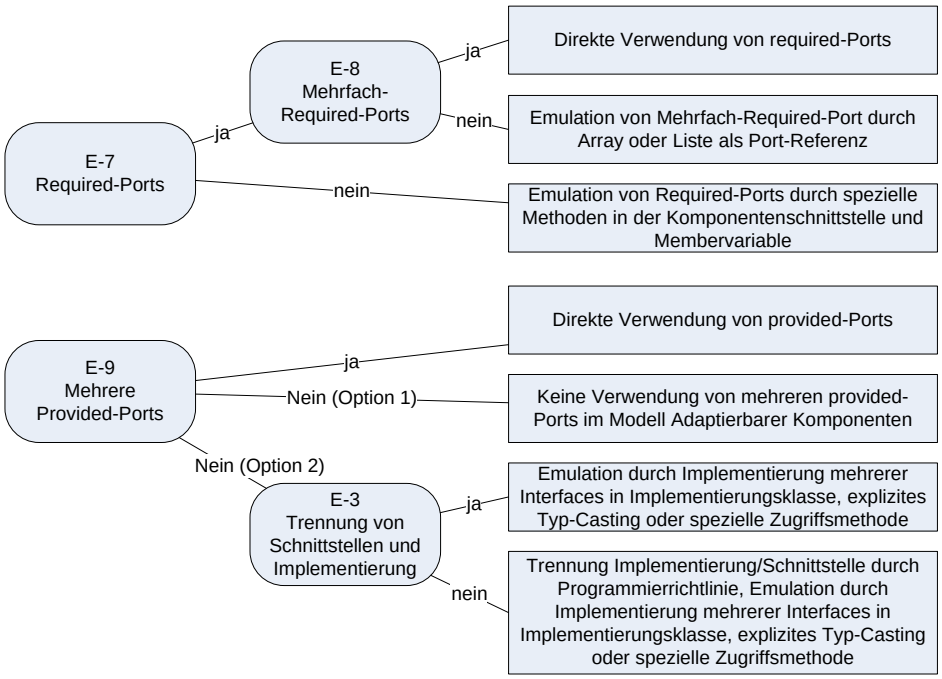

Abbildung 4.2: Umsetzung von Komponentenports in Abhängigkeit von Merkmalen der Komponentenplattform

Unterstützung von *required*-Ports Wenn *required*-Ports (Eigenschaft E-7) und Mehrfach-*required*-Ports (Eigenschaft E-8, siehe Unterunterabschnitt 3.3.1.1) von der Zielkomponentenplattform unterstützt werden, können sie direkt auf das Konzept *required*-Port von Adaptierbaren Komponenten bzw. Subkomponenten abgebildet werden. Wenn lediglich einfache *required*-Ports unterstützt werden (Eigenschaft E-8 nicht erfüllt), kann ein Mehrfach-*required*-Port durch ein Array bzw. einen anderen *List*-Parameter als Typ eines einfachen *required*-Ports emuliert werden. Mehrere Referenzen auf *provided*-Ports können dann über Hilfsmethoden (*getReference*, *addReference*, *removeReference*) dem Array bzw. der Liste zugewiesen werden.

Beispiel 4.1. Eine Komponente mit einem Mehrfach-*required*-Port, der Referenzen des Typs T akzeptiert, wird in der Zielkomponentenplattform auf eine Komponente mit einem *required*-Port des Typs *List* of T oder *Array* of T abgebildet. Im Programmcode der Komponente kann der Wert des *required*-Ports nicht direkt verwendet werden, sondern erst mittels der Hilfsmethode *getReference* kann auf eine bestimmte Referenz in dem emulierten Mehrfach-*required*-Port zugegriffen werden.

Sollte das beschriebene Verfahren zur Emulation von Mehrfach-*required*-Ports bei der Zielkomponentenplattform nicht möglich sein, muss so vorgegangen werden, als wenn *required*-Ports überhaupt nicht unterstützt werden.

Keine Unterstützung von *required*-Ports Wenn *required*-Ports nicht von der Zielkomponentenplattform unterstützt werden, muss dieses Konzept durch spezielle Methoden in der Komponentenschnittstelle emuliert werden. Für einen einfachen *required*-Port des Typs T

wird eine Methode benötigt, mit der die Referenz des *required*-Ports festgelegt werden kann. Innerhalb der Komponenteninstanz muss die Referenz in einer Membervariablen gespeichert werden und optional kann auch noch eine Zugriffsmethode in der Implementierungsklasse der Komponente hinzugefügt werden. In Java kann dies beispielsweise wie folgt realisiert werden:

```
class BeispielKomponente implements KomponentenInterface {
  private T requiredPort; // Membervariable für required"=Port
  ...
  public void setPortname(T ref) { requiredPort = ref; }
  private T getPortname() { return requiredPort; }
  ...
}
```

Bei einem Mehrfach-*required*-Port des Typs T müssen zwei Methoden zum Hinzufügen und Entfernen von Referenzen in die Komponentenschnittstelle aufgenommen werden. Innerhalb der Komponente müssen die Referenzen in Form einer Liste in einer Membervariablen gespeichert werden und optional können auch noch Zugriffsmethoden (z. B. zum direkten Zugriff auf eine bestimmte Referenz bzw. zum Iterieren über alle Referenzen) in der Implementierungsklasse der Komponente hinzugefügt werden. In Java kann dies beispielsweise auf folgende Weise realisiert werden:

```
class BeispielKomponente implements KomponentenInterface {
  private List<T> requiredPorts; // Membervariable für required"=Port
  ...
  public void addPortname(T ref) { requiredPorts.add(ref); }
  public void removePortname(T ref) { requiredPorts.remove(ref); }
  private T getPortname(int i) { return requiredPorts.get(i); }
  private Iterator<T> getPortnameIterator() {
    return requiredPorts.iterator();
  }
  ...
}
```

Mehrere *provided*-Ports Wenn die Komponentenplattform nicht mehrere *provided*-Ports für eine Komponente unterstützt (Eigenschaft E-9 nicht erfüllt), gibt es zwei Möglichkeiten zur Umsetzung:

- Die Modelle von Adaptierbaren Komponenten werden dahingehend eingeschränkt, dass nur noch ein *provided*-Port pro Komponente zugelassen ist. Diese Einschränkung muss allerdings bereits beim Entwurf berücksichtigt werden.
- Für jeden *provided*-Port wird eine Zugriffsmethode (z. B. `get`*Portname*`()`) zur Komponentenschnittstelle hinzugefügt. Wenn die Komponentenplattform eine Trennung von Schnittstelle und Implementierung vorschreibt (Eigenschaft E-3 erfüllt), können mehrere *provided*-Ports mit Hilfe von mehreren implementierten Interfaces in der Implementierungsklasse der Komponente emuliert werden. Anderenfalls muss, wie in Unterunterabschnitt 4.2.2.8 beschrieben, die Trennung von Schnittstelle und Implementierung durch eine Programmierrichtlinie erzwungen werden. Jedes Interface repräsentiert genau einen *provided*-Port. Mit Hilfe von expliziten Typ-Casts kann zur Laufzeit ein bestimmtes Interface und damit ein bestimmter *provided*-Port ausgewählt werden. In

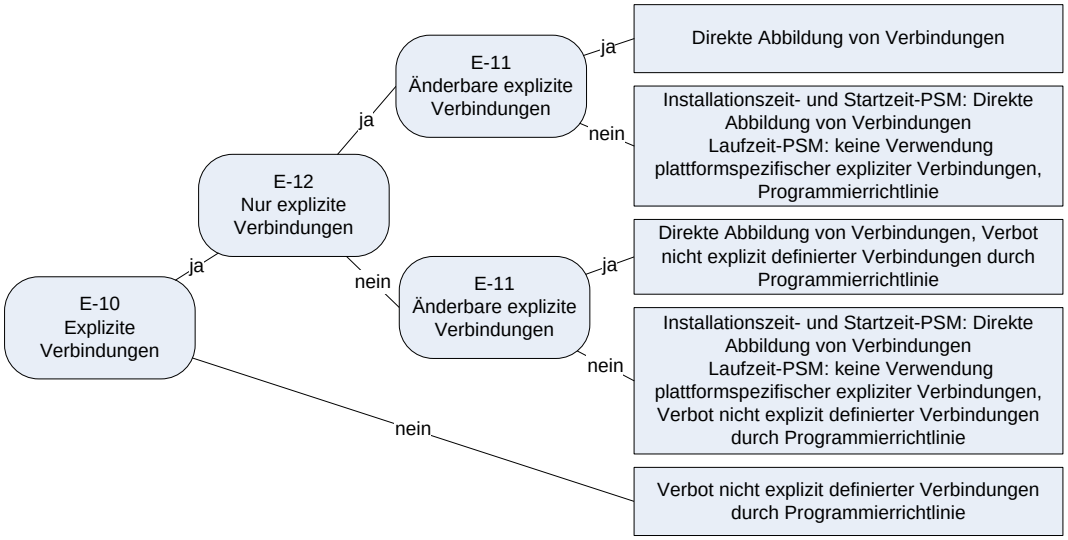

Abbildung 4.3: Umsetzung von Verbindungen in Abhängigkeit von Merkmalen der Komponentenplattform

der Zugriffsmethode wird dazu einfach eine Referenz auf das aktuelle Objekt (`this`) zurückgeliefert.

4.2.2.5 Verbindungen

Für die Umsetzung von Verbindungen müssen die Merkmale E-3, E-10, E-12 und E-13 berücksichtigt werden. Die Auswirkungen dieser Parameter auf die Implementierung sind als Entscheidungsbaum in Abbildung 4.3 dargestellt.

Subkomponenten dürfen nicht selbständig, d. h. ohne Wissen der zugehörigen Adaptierbaren Komponente, Verbindungen zu anderen Komponenten herstellen bzw. neue Komponenten erzeugen, sondern nur über definierte Ports (siehe Unterabschnitt 3.3.3).

Wenn die Komponentenplattform keine expliziten Verbindungen zwischen Komponenten unterstützt (Eigenschaft E-10 nicht erfüllt) oder wenn ausschließlich explizite Verbindungen nicht erzwungen werden können (Eigenschaft E-12 nicht erfüllt), muss eine Programmierrichtlinie nicht explizit definierte Verbindungen zwischen Komponenten verbieten. Zur Laufzeit kann im Allgemeinen nicht zuverlässig überprüft werden, ob Komponenten auch nicht explizit definierte Verbindungen zu anderen Komponenten aufnehmen.

Wenn explizite Verbindungen unterstützt werden (Eigenschaft E-10 erfüllt), können sie im Installationszeit- und Startzeit-PSM uneingeschränkt zur Umsetzung von Verbindungen verwendet werden. Im Laufzeit-PSM, d. h. wenn Rekonfiguration unterstützt wird, können sie nur verwendet werden, wenn sie auch zur Laufzeit geändert werden können (Eigenschaft E-11 erfüllt). Anderenfalls muss auf die Verwendung der Plattformunterstützung für explizite Verbindungen verzichtet werden und Verbindungen durch Programmierrichtlinien hergestellt werden.

4.2.2.6 Adaptionsoperatoren und Aspektoperatoren

Im Gegensatz zu den anderen Konzepten von Adaptierbaren Komponenten können Adaptions- und Aspektoperatoren auch bei einer bestimmten Komponentenplattform durch unterschiedliche Verfahren implementiert werden, die detailliert in Unterabschnitt 4.3.5 diskutiert werden. Die Eigenschaften der Komponentenplattform führen jedoch dazu, dass einige dieser Verfahren nicht angewendet werden können:

- Bei einer fehlenden Unterstützung für die Modifikation von Bytecode (Eigenschaft E-18 nicht erfüllt) können Operatoren nicht durch die Änderung des Bytecodes von Komponenten implementiert werden.
- Bei einer fehlenden Unterstützung für Reflection in der Programmiersprache (Eigenschaft E-17 nicht erfüllt) können Operatoren nicht durch generische Adapter implementiert werden.
- Weitere verbotene Operationen für Komponenten, die in der Erläuterungseigenschaft E-19 ermittelt werden, können die Implementierungsmöglichkeiten der Operatoren weiter einschränken. Allerdings muss hier im Einzelfall geprüft werden, welche verbotene Operatoren zu welchen Auswirkungen führt. Bei Java kann z. B. das Verbot von Classloadern dazu führen, dass die Operatoren keine Bytecode-Modifikation beim Laden einer Komponente verwenden können. Zur Installationszeit wäre aber trotz dieses Verbot eine Bytecode-Modifikation möglich.

4.2.2.7 Kontextmodell

Das Kontextmodell wird nur beim Laufzeit-PSM (siehe Abschnitt 4.1) verwendet und auch nur, wenn es im Modell der Adaptierbaren Komponente definiert wurde (optionaler Bestandteil, Unterabschnitt 3.3.6). Wenn möglich sollten anwendungsglobale Kontextinformationen in einer plattformspezifischen Komponente implementiert werden. Für die Implementierung werden aber z.T. erweiterte System-Zugriffsrechte benötigt, je nachdem, welche Umgebungsinformationen verfügbar gemacht werden sollen (z. B. CPU-Auslastung, freier Speicher). In Abhängigkeit der Eigenschaft E-19 können solche Operationen nicht immer innerhalb einer Komponente implementiert werden.

4.2.2.8 Rekonfiguration

Rekonfiguration wird nur beim Laufzeit-PSM (siehe Abschnitt 4.1) benötigt. Bei Installations- und Startzeit-PSM ist eine Umsetzung dieses Konzepts nicht erforderlich. Nur wenige Komponentenplattformen (z. B. OpenORB, siehe Unterabschnitt 2.5.4) unterstützen eine Rekonfiguration von Komponenten zur Laufzeit (Eigenschaft E-15 erfüllt). Bei allen anderen Komponentenplattformen muss die Rekonfiguration emuliert werden. Die Auswirkungen von Eigenschaften von Komponentenplattformen auf die Umsetzung der Rekonfiguration sind in Form eines Entscheidungsbaumes auch in Abbildung 4.4 dargestellt.

Zur Sicherstellung der geforderten Eigenschaften von Rekonfigurationen (siehe Abschnitt 3.5) steuert ein so genannter *Adaptionsmanager* als Teil der Laufzeitunterstützung für Adaptierbare Komponenten die Rekonfiguration. Mit Hilfe von Interceptoren kann zum einen der Zustand von Komponenten (aktiv oder inaktiv) transparent ermittelt werden und

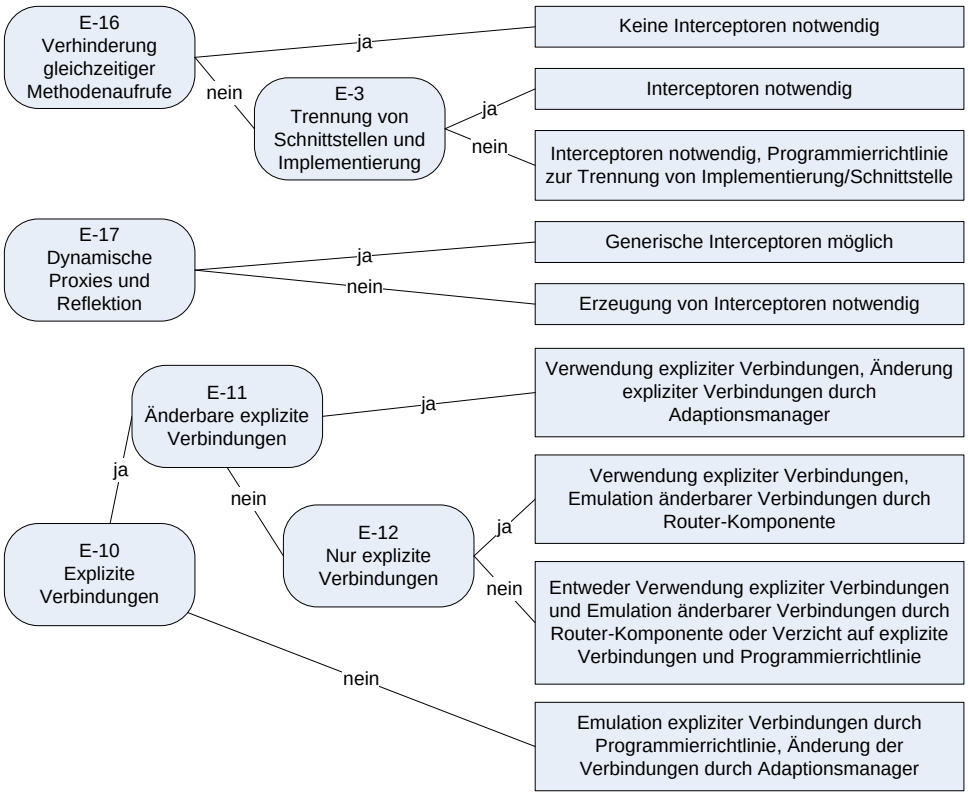

Abbildung 4.4: Umsetzung von Rekonfigurationen in Abhängigkeit von Merkmalen der Komponentenplattform

zum anderen Methodenaufrufe gegebenenfalls blockiert werden. Weitere Implementierungsdetails werden im Abschnitt 4.3 beschrieben.

Interceptoren Interceptoren verwenden zur Steuerung von Rekonfigurationen benötigt, wenn gleichzeitige Methodenaufrufe bei Komponenteninstanzen durch die Komponentenplattform nicht verhindert werden (Eigenschaft E-16 nicht erfüllt). Für den Einsatz von Interceptoren ist eine Trennung von Schnittstellen und Implementierung bei Komponenten erforderlich (Eigenschaft E-3). Wenn dies durch die Komponentenplattform nicht erzwungen wird, muss eine entsprechende Programmierrichtlinie formuliert werden. Generische Interceptoren, die sich für beliebige Komponenten verwenden lassen, erfordern die Unterstützung von Reflection durch die Programmiersprache (Eigenschaft E-17 erfüllt). Andernfalls müssen Interceptoren speziell für eine bestimmte Komponente erzeugt werden.

Für die Änderung der Verknüpfung von internen und externen Ports einer Adaptierbaren Komponente gelten die in Unterunterabschnitt 4.2.2.1 zu Laufzeit-PSM getroffenen Aussagen.

Unterstützung expliziter Verbindungen Wenn die Komponentenplattform explizite Verbindungen unterstützt (Eigenschaft E-10) bzw. sogar explizite Verbindungen vorschreibt (Eigenschaft E-12), müssen diese Verbindungen im Rahmen einer Rekonfiguration auch geändert

werden können. Ist dies der Fall (Eigenschaft E-11 erfüllt), kann die Rekonfiguration auch mit plattformspezifischen Verbindungen realisiert werden.

Wenn ausschließlich explizite Verbindungen erlaubt sind (Eigenschaft E-12 erfüllt), diese aber zur Laufzeit nicht verändert werden können (Eigenschaft E-11 nicht erfüllt), muss eine Router-Komponente eingeführt werden (ähnlich wie bei zusammengesetzten Komponenten in Unterunterabschnitt 4.2.2.1), die Verbindungen mit allen anderen Subkomponenten herstellt. Die Router-Komponente leitet dann in Abhängigkeit der gerade aktiven Konfiguration Methodenaufrufe an die vorgesehenen Empfänger weiter. Bei diesem Verfahren werden im Prinzip neue virtuelle Verbindungen zwischen Komponenten auf Basis von vorhandenen plattformspezifischen Verbindungen realisiert. Der Ansatz verwendet damit einen ähnlichen Ansatz wie Overlay-Netze im Internet, z. B. MBone [SRL96] oder VPNs [YS01].

Wenn die Komponentenplattform zwar explizite Verbindungen unterstützt (Eigenschaft E-10 erfüllt), aber die Verwendung nicht verbindlich vorschreibt (Eigenschaft E-12 nicht erfüllt), kann entweder wie bereits beschrieben eine Router-Komponente eingesetzt werden oder plattformspezifische Verbindungen werden überhaupt nicht verwendet. Bei letzterer Variante wird so vorgegangen, als wenn die Komponentenplattform keine expliziten Verbindungen unterstützt.

Keine Unterstützung von expliziten Verbindungen Wenn die Komponentenplattform keine expliziten Verbindungen unterstützt (Eigenschaft E-10 nicht erfüllt), werden Verbindungen durch den Adaptionsmanager initialisiert und im Rahmen einer Rekonfiguration geändert. Dabei wird wie in Unterunterabschnitt 4.2.2.5 beschrieben verfahren.

4.2.2.9 Zusammenfassung

In Tabelle 4.2 sind noch einmal die Auswirkungen der verschiedenen Merkmale von Komponentenplattformen auf die Implementierung von Adaptierbaren Komponenten dargestellt, die bereits in den vorigen Abschnitten ausführlich beschrieben wurden.

4.2.3 Entwicklung von Werkzeugen

Im Rahmen der Unterstützung einer bestimmten Komponentenplattform müssen auch eine Reihe von Werkzeugen entwickelt werden. Das im nächsten Abschnitt vorgestellte Framework enthält dazu schon eine Reihe von Funktionen, so dass sich der Entwicklungsaufwand reduziert. Im Folgenden werden kurz die wesentlichen Aspekte beschrieben, die mit Werkzeugen unterstützt werden.

- Generierung von Interceptoren: Wenn generische Interceptoren nicht verwendet werden können oder wenn eine maximale Leistung gewünscht wird, müssen Interceptoren speziell für einen bestimmten Komponententyp erzeugt werden.
- Anwendung von Aspekt- bzw. Adaptionsoperatoren: Zur Steigerung der Leistung können Aspekt- und Adaptionsoperatoren bereits zur Installationszeit angewendet werden, indem Quelltexte oder Bytecode von Komponenten modifiziert werden oder indem spezielle Adapterklassen generiert werden.

Merkmal		erfüllt	nicht erfüllt
E–1	Zus.-gesetzte Komponenten	Direkte Abbildung Adaptierbarer Komponenten auf zus.-gesetzte Komponenten	Emulierung durch flache Komponentenstruktur
E–2	Variable Port-Verknüpfung	Nutzung variabler Port-Verknüpfung bei Rekonfiguration	Emulation durch zusätzliche Komponente als „Router“
E–3	Trennung Implementierung und Schnittstelle	Verwendung für Adaptierbare Komponenten und Subkomponenten	Erzwingung der Trennung durch Programmierrichtlinie
E–4	Komponentenparameter	Direkte Abbildung	Emulation durch getter- und setter-Methoden
E–7	*required*-Ports	Direkte Abbildung	Emulation durch Programmierrichtlinie und zusätzlichen Membervariablen
E–8	Mehrfach *required*-Ports	Direkte Abbildung	Emulation durch *required*-Port mit Array-Typ
E–9	Mehrere *provided*-Ports	Direkte Abbildung	Verzicht auf mehrere *provided*-Ports oder Emulation durch Implementierung mehrerer Schnittstellen
E–10	Explizite Verbindungen	Direkte Verwendung	Erzwingung durch Programmierrichtlinie
E–11	Änderung expliziter Verbindungen	Direkte Verwendung für Rekonfiguration	Verzicht bei Startzeit und Laufzeit-PSM
E–12	Nur explizite Verbindungen	Direkte Verwendung	Erzwingung durch Programmierrichtlinie
E–13	Synchrone Kommunikation	Verwendung für Methodenaufrufe	Emulation durch zwei asynchrone Nachrichten
E–14	Zustandsbehaftete Komponenten	Verwendung für Adaptierbare Komponenten und Subkomponenten	Emulation durch Übergabe von Zustandsdaten als Parameter bei Methodenaufrufen
E–15	MOP	Nutzung für MOP von Adaptierbaren Komponenten	Emulation
E–16	Verhinderung gleichzeitiger Methodenaufrufe	Vereinfachter Adaptionsmanager, keine Interceptoren notwendig	Steuerung der Rekonfiguration durch Adaptionsmanager und Interceptoren
E–17	Dynamische Proxies und Reflection	Verwendung für generische Interceptoren und MOP	Codegenerierung für Interceptoren, kein MOP
E–18	Bytecode-Modifikation	Verwendung für Adaptions- bzw. Aspektoperatoren	Nutzung von Adaptern für Adaptions-/ Aspektoperatoren

Tabelle 4.2: Implementierung in Abhängigkeit von Eigenschaften der Komponentenplattform

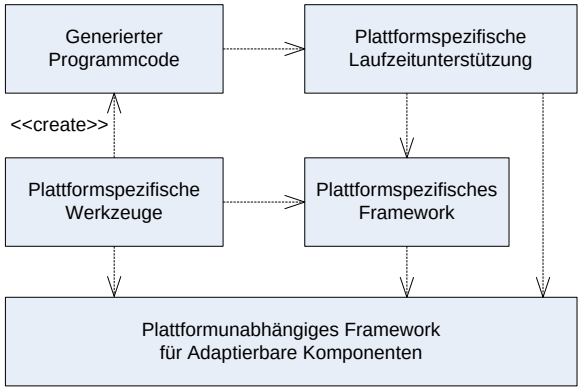

Abbildung 4.5: Nutzung des Frameworks

- Erzeugung von Komponentenarchiven: Alle Komponenten, erzeugter Programmcode und eventuelle Hilfsbibliotheken müssen entsprechend den Anforderungen einer Komponentenplattform (siehe Eigenschaft E-22) in einem Komponentenarchiv gespeichert werden. Mit einem Werkzeug wird dieser Prozess automatisiert.
- Installation von Komponenten: Bei vielen Komponentenplattformen müssen Komponenten vor der ersten Verwendung installiert werden. Wenn möglich kann ein entsprechendes Werkzeug auch gleichzeitig die Erzeugung von Komponentenarchiven als Teilaufgabe mit realisieren.
- Generierung von Codegerüsten für Komponentenimplementierungen: Unter Berücksichtigung von aufgestellten Programmierrichtlinien und dem Modell der jeweiligen Adaptierbaren Komponente können Codegerüste für die weitere Entwicklung erzeugt werden. Diese enthalten im wesentlichen Schnittstellen- und Klassendefinitionen mit leeren Methodenrümpfen.

4.3 Framework zur Unterstützung Adaptierbarer Komponenten

Das Framework zur Unterstützung Adaptierbarer Komponenten bildet die Grundlage für plattformspezifische Implementierungen. Es kann sowohl zur Entwicklung von verschiedenen plattformspezifischen Laufzeitunterstützungen (alle vorgestellten PSMs) als auch für Entwicklungs- und Installationswerkzeuge verwendet werden (siehe Abbildung 4.5). Ziel dabei ist es, möglichst viele Funktionen, die für die Unterstützung von Adaptierbaren Komponenten benötigt werden, in einer komponentenplattformunabhängigen Art zur Verfügung zu stellen. Der Motivation eines Frameworks folgend, müssen dann nur wenige Funktionen durch das Implementieren von abstrakten Methoden in abgeleiteten Klassen an die Spezifika einer bestimmten Komponentenplattform angepasst werden. Neue Komponentenplattformen können somit schneller und mit geringerem Aufwand implementiert werden.

Das Framework wurde in Java entwickelt und kann daher in dieser Form nur für Java-basierte Komponentenplattformen verwendet werden. Alternative Implementierung in anderen Programmiersprachen wie z. B. C++ oder C# sind aber in einer ähnlichen Form denkbar.

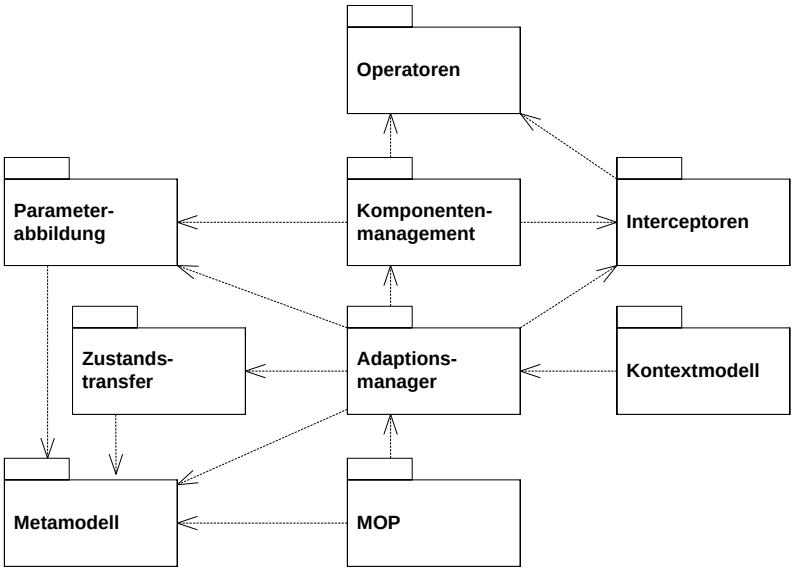

Abbildung 4.6: Paketdiagramm des Frameworks zur Unterstützung Adaptierbarer Komponenten

Abbildung 4.6 zeigt den modularen Aufbau des Frameworks. Die einzelnen Pakete (*Packages*) bieten z.T. unterschiedliche Verfahren für die Realisierung der gleichen Funktionalität an:

- Alle Pakete unterstützen die **direkte Implementierung** von Funktionen. Das bedeutet, dass beim Aufruf einer Methode eine gewünschte Funktion abgearbeitet wird. Mit Hilfe von Reflection können auch Interceptoren in Form einer direkten Implementierung realisiert werden (siehe Unterabschnitt 4.3.6). Diese Flexibilität wird allerdings mit Leistungseinbußen erkauft.
- Einige Pakete bieten zusätzlich eine **indirekte Implementierung** von Funktionen an. Dabei wird eine gewünschte Funktion nicht direkt ausgeführt, sondern stattdessen Java-Quelltext oder Java-Bytecode erzeugt bzw. geändert, der diese Funktion später ausführen kann. Der Vorteil dieser Vorgehensweise besteht darin, dass der generierte Code speziell für die jeweilige Aufgabe optimiert werden kann und daher in vielen Fällen einen erheblichen Performancegewinn gegenüber der direkten Implementierung erzielt.

In den folgenden Abschnitten werden die verschiedenen Pakete des Frameworks näher vorgestellt. Dabei wird auch beschrieben, welche Pakete zusätzlich eine indirekte Implementierung von Funktionen unterstützen.

4.3.1 Metamodell

Das Metamodell-Paket des Frameworks dient zur Speicherung und zum Zugriff auf alle Informationen einer Adaptierbaren Komponente (siehe Kapitel 3 und Anhang B). Das Metamodell wird sowohl für Entwicklungs- und Installationswerkzeuge als auch zur Laufzeitsteuerung von Adaptierbaren Komponenten benötigt. Es ist somit das Bindeglied für alle anderen Pakete des Frameworks.

Für die Implementierung des Metamodells wurde das *Eclipse Modeling Framework* (EMF [emf05]) verwendet. Dabei handelt es sich um ein Werkzeug zur Codegenerierung für *Model Repositories*. Basierend auf einer MOF-kompatiblen Beschreibung [MOF06] des Metamodells erzeugt EMF Java-Quelltext für alle Klassen, die später Instanzen des Metamodells, d. h. Modelle, speichern. Die Metamodelle können mit einem beliebigen Case-Tool entwickelt werden, das XMI-Export [XMI05] unterstützt[5]. Neben der Unterstützung von XMI, erlaubt EMF alternativ auch die Definition der Metamodelle mittels XML-Schema oder speziell annotiertem Java-Quelltext. Bei der Codegenerierung wird nicht einfach bereits vorhandener Quelltext überschrieben, sondern vom Entwickler vorgenommene Änderungen werden beibehalten (*Round Trip Engineering*). Auf diese Weise können Modelländerungen im Laufe des Entwicklungsprozesses mit dem vorhandenen Quelltexten automatisch synchronisiert werden.

Die mittels EMF generierten Java-Klassen des Metamodells stellen ein komfortables API für die Entwicklung zur Verfügung, um die programmtechnische Erzeugung, den Zugriff und das Editieren von Modellen zu unterstützen. Eine detaillierte Beschreibung der Klassen des Metamodells befindet sich in Anhang B. Durch die Unterstützung des Observer-Musters [GHJV95] können sich *Listener* bei bestimmten Objekten des Metamodells anmelden (*event subscription*) und werden dann bei Änderungen automatisch benachrichtigt (*notification*). Modelle Adaptierbarer Komponenten können als XMI gespeichert und gelesen werden.

4.3.2 Adaptionsmanager

Der Adaptionsmanager steuert die Rekonfiguration einer Adaptierbaren Komponente im Laufzeit-PSM (siehe Unterabschnitt 4.4.1). In allen anderen PSMs wird er nicht benötigt, da dort keine Rekonfiguration unterstützt wird.

Der Adaptionsmanager nutzt nahezu alle anderen Pakete des Frameworks zur Durchführung einer Rekonfiguration. Er muss in diesem Zusammenhang sicherstellen, dass alle Eigenschaften von Rekonfigurationen (siehe Abschnitt 3.5) eingehalten werden. Dazu gehört insbesondere, dass zu jedem Zeitpunkt alle externen Ports mit internen Ports verknüpft sind, damit die Funktionsfähigkeit der Adaptierbaren Komponente garantiert ist.

Im Folgenden werden die einzelnen Schritte einer Rekonfiguration und die damit verbundenen Steuerungsaufgaben des Adaptionsmanagers näher vorgestellt, um dabei auch das Zusammenspiel der Framework-Pakete zu erläutern.

Auslösung einer Rekonfiguration Eine Rekonfiguration kann über zwei verschiedene Wege ausgelöst werden: durch Parameteränderungen und durch das MOP (siehe Unterabschnitt 4.3.9) der Adaptierbaren Komponente. Parameteränderungen können entweder außerhalb der Adaptierbaren Komponente durch den Aufruf der entsprechenden Methoden in der Komponentenschnittstelle oder innerhalb durch Änderungen von Kontextinformationen (siehe Unterabschnitt 4.3.8) ausgelöst werden. In beiden Fällen informiert das Komponentenmanagement-Paket den Adaptionsmanager über die Änderungen (*Observer*-Muster).

Bestimmung der neuen Konfiguration Nachdem der Adaptionsmanager über die Parameteränderung informiert wurde, nutzt er das Parameterabbildungs-Paket zur Bestimmung der

[5] EclipseUML von Omondo (http://www.eclipseuml.org) wurde für die Implementierung verwendet, da es direkt EMF unterstützt und in Eclipse integriert ist.

neuen Konfiguration. Wenn die ermittelte neue Konfiguration mit der aktuell aktiven Konfiguration[6] übereinstimmt, wird die Rekonfiguration an dieser Stelle abgebrochen.

Wenn durch das MOP der Adaptierbaren Komponente bereits direkt eine bestimmte Konfiguration ausgewählt wurde, entfällt dieser Schritt.

Ermittlung der notwendigen Rekonfigurationsoperationen Im Folgenden wird ein Verfahren zur Bestimmung von Rekonfigurationsoperationen beschrieben, um die aktuelle in die neue Konfiguration zu überführen. Die Rekonfigurationsoperationen werden zur Durchführung der entsprechenden Rekonfiguration benötigt. Das Verfahren nutzt einen Markierungsalgorithmus zum Finden von Gemeinsamkeiten und Unterschieden zwischen den zwei Konfigurationen.

1. Alle Verbindungen, Subkomponenteninstanzen und Wertzuweisungen der Ausgangs- und Zielkonfiguration werden markiert.

2. Für jede Subkomponenteninstanz der Ausgangskonfiguration wird überprüft, ob sie auch in der Zielkonfiguration vorkommt. Wenn ja, wird die Markierung der entsprechenden Subkomponenteninstanz in beiden Konfigurationen gelöscht.

3. Für jede Verbindung der Ausgangskonfiguration wird überprüft, ob sie auch in der Zielkonfiguration vorkommt und ob die zu den Ports gehörenden Subkomponenteninstanzen in der Ausgangs- und Zielkonfiguration unmarkiert sind. Wenn beide Bedingungen erfüllt sind, wird die Markierung der entsprechenden Verbindung in beiden Konfigurationen gelöscht.

4. Für jede Wertzuweisung der Ausgangskonfiguration wird überprüft, ob sie auch in der Zielkonfiguration vorkommt und ob die zugehörige Subkomponenteninstanz in der Ausgangs- und Zielkonfiguration unmarkiert ist. Wenn beide Bedingungen erfüllt sind, wird die Markierung der entsprechenden Wertzuweisung in beiden Konfigurationen gelöscht.

Die Rekonfigurationsoperationen können wie folgt bestimmt werden:

- Die Menge der zu entfernenden Verbindungen besteht aus allen verbleibenden Markierungen von Verbindungen in der Ausgangskonfiguration. Daraus ergibt sich eine Menge von *removeConnection* Operationen.

- Die Menge der zu entfernenden Subkomponenteninstanzen besteht aus allen verbleibenden Markierungen von Subkomponenteninstanzen in der Ausgangskonfiguration. Daraus ergibt sich eine Menge von *removeComponent* Operationen.

- Die Menge der hinzuzufügenden Subkomponenteninstanzen besteht aus allen verbleibenden Markierungen von Subkomponenteninstanzen in der Zielkonfiguration. Daraus ergibt sich eine Menge von *addComponent* Operationen.

- Die Menge der hinzuzufügenden Verbindungen besteht aus allen verbleibenden Markierungen von Verbindungen in der Zielkonfiguration. Daraus ergibt sich eine Menge von *addConnection* Operationen.

[6] Der Adaptionsmanager speichert immer die gerade aktive Konfiguration.

- Die Menge der Wertzuweisungen für Parameter besteht aus allen verbleibenden Markierungen von Wertzuweisungen in der Zielkonfiguration. Daraus ergibt sich eine Menge von *setParameter* Operationen.

Anschließend werden die Rekonfigurationsoperationen jeweils gruppenweise in folgender Reihenfolge geordnet: *removeConnction*, *removeComponent*, *addComponent*, *addConnection*, *setParameter*. Innerhalb einer Gruppe von Operationen ist die Reihenfolge dabei beliebig (siehe Unterabschnitt 3.4.4).

Ermittlung der an der Rekonfiguration beteiligten Subkomponenten Durch die Anwendung des Verfahrens aus Abschnitt 3.5 ermittelt der Adaptionsmanager alle Komponenteninstanzen, die während der Rekonfiguration deaktiviert werden müssen. Dies sind alle Komponenteninstanzen, die direkt oder indirekt von den im vorhergehenden Schritt ermittelten Rekonfigurationsoperationen beeinflusst werden.

Blockierung von Methodenaufrufen Zur Deaktivierung der ermittelten Komponenteninstanzen müssen alle laufenden Methodenaufrufe beendet und neue blockiert werden. Dazu benachrichtigt der Adaptionsmanager die Interceptoren (Unterabschnitt 4.3.6) der entsprechenden Komponenteninstanzen, um weitere Methodenaufrufe zu blockieren. Eine Referenz zu jedem Interceptor wird bereits bei der Initialisierung von Subkomponenten durch das Komponentenmanagement-Paket im Adaptionsmanager gespeichert. Der Adaptionsmanager wartet anschließend bis alle laufenden Methodenaufrufe bei den zu blockierenden Komponenteninstanzen beendet wurden und sie damit inaktiv sind.

Durchführung der Rekonfigurationsoperationen und Zustandstransfer Der Adaptionsmanager führt alle ermittelten Rekonfigurationsoperationen nacheinander aus. Vor dem Entfernen einer Komponenteninstanz werden die Zustandsinformationen mit Hilfe des Zustandstransfer-Paketes in den Zwischenspeicher übertragen, wenn eine entsprechende Abbildungsvorschrift definiert wurde (siehe Abschnitt 3.6 und Unterabschnitt 4.3.7). In der entgegengesetzten Richtung werden bei neuen Komponenteninstanzen die Zustandsinformationen nach dem Erzeugen gesetzt, ebenfalls wenn eine Abbildungsvorschrift definiert wurde.

Abschluss der Rekonfiguration Der Adaptionsmanager benachrichtigt alle Interceptoren, dass die Blockierung von Methodenaufrufen abgehoben ist. Die neue Konfiguration wird als aktive Konfiguration im Adaptionsmanager gespeichert.

4.3.3 Parameterabbildung

Das Parameterabbildungs-Paket implementiert die Programmlogik zur Auswahl von Konfigurationen einer Adaptierbaren Komponente basierend auf den aktuellen Parameterwerten (siehe Unterabschnitt 3.4.6). Es nutzt das Metamodell-Paket zum Zugriff auf die dort gespeicherte Definition der Parameterabbildung.

Das Parameterabbildungs-Paket stellt zwei unterschiedliche Verfahren (siehe Tabelle 4.3) bereit, die von den verschiedenen PSMs bzw. den Installations- und Entwicklungswerkzeugen verwendet werden können:

	Direkte Auswertung	**Codegenerierung**
Unterstützung von Regeländerungen	ja	nein
Ausführungszeit	hoch	niedrig
Anwendungsgebiete	Laufzeit-PSM mit MOP, Entwicklungswerkzeuge	Start PSM

Tabelle 4.3: Eigenschaften von Verfahren zur Parameterabbildung

Direkte Implementierung der Parameterauswertung Bei diesem Verfahren werden die Regeln zur Parameterabbildung direkt aus dem Metamodell gelesen und entsprechend angewendet. Damit ist sichergestellt, dass immer der aktuelle Stand der Regeln verwendet wird und insbesondere alle eventuellen Änderungen des Metamodells berücksichtigt werden.

Zur Durchführung der Parameterauswertung müssen alle aktuellen Parameterwerte der Adaptierbaren Komponente in Form einer *Map* mit Parametername/Parameterwert als Schlüssel/Wert-Paare übergeben werden. Als Resultat liefert das Paket die den Parameterwerten zugeordnete Konfiguration zurück.

Indirekte Implementierung der Parameterabbildung Bei diesem Verfahren wird basierend auf den Regeldefinitionen des Metamodells Java-Quelltext generiert, der die Abbildung von konkreten Parameterwerten auf Konfigurationen der Adaptierbaren Komponente übernimmt. Der erzeugte Quelltext kann dann wesentlich schneller abgearbeitet werden, als das Verfahren zur direkten Parameterauswertung, da die Regeln nicht aus dem Metamodell ausgelesen und interpretiert werden müssen, sondern direkt als Java-Ausdrücke vorliegen. Allerdings muss bei einer Regeländerung der Quelltext neu erzeugt bzw. angepasst werden. Regeländerungen zur Laufzeit sind damit nicht ohne weiteres möglich[7].

Das Paket erzeugt den Java-Quelltext einer Methode, die genau ein Argument mit dem entsprechenden Datentyp für jeden Parameter der Adaptierbaren Komponente besitzt. Als Resultat wird eine Beschreibung der zugehörigen Konfiguration zurückgegeben. Für die Kryptographiekomponente mit den drei Parametern Compression, Encryption und Key ergibt sich beispielsweise die folgende Methode:

```
public Configuration mapParameter(Compression compression,
                                  Encryption encryption, String key) {
  ... // Zugriff auf Regeln zur Parameterabbildung im Metamodell
      // und Auswertung
  return configuration;
}
```

4.3.4 Komponentenmanagement

Das Komponentenmanagement-Paket enthält eine Reihe von größtenteils abstrakten Methoden für Basisfunktionen und zur Steuerung des Lebenszyklus von Adaptierbaren Komponen-

[7] Theoretisch kann mit Java auch zur Laufzeit Code erzeugt, geladen und ausgeführt werden, was allerdings mit einem relativ hohen Zeitbedarf zur Laufzeit und einem hohen Entwicklungsaufwand verbunden ist.

ten und enthaltenen Subkomponenten. Nahezu alle dieser Methoden müssen von plattformspezifischen Implementierungen überschrieben werden. Da viele andere Pakete des Frameworks diese Methoden benutzen, wird der plattformspezifische Anteil so weit wie möglich in diesem Paket konzentriert. Folgende Funktionen werden durch das Paket bereitgestellt:

- Initialisierung der Adaptierbaren Komponente: In Abhängigkeit der Parameterwerte werden die Konfiguration festgelegt (unter Nutzung des Parameterabbildungs-Pakets), Subkomponenten erzeugt und Verbindungen hergestellt.
- Erzeugen und Entfernen von Subkomponenten (siehe Unterabschnitt 3.3.1): Falls definiert, werden gleichzeitig Adaptions- und Aspektoperatoren initialisiert und hinzugefügt.
- Erzeugen und Entfernen einer Verbindung zwischen zwei Ports (siehe Unterabschnitt 3.3.3). Je nach Bedarf wird im Laufzeit-PSM auf das Interceptor-Paket zugegriffen, um Interceptoren dazwischen zu schalten und beim Adaptionsmanager zu registrieren.
- Verknüpfung von externen Ports der Adaptierbaren Komponente mit Ports von Subkomponenten (*Interface Binding*)
- Lesen und Ändern von Parametern bzw. Zustandsinformationen bei Subkomponenten (siehe Unterunterabschnitt 3.3.1.2): Abstrakte Zugriffsmethoden auf Parameter bzw. Zustandsinformationen erlauben aber die plattformunabhängige Verwendung durch andere Pakete des Frameworks.
- Einlesen von Komponentendeskriptoren und Zugriff auf Informationen: Die Methoden kapseln die unterschiedlichen Verfahren zum Verpacken (*Packaging* und *Deployment*) von Komponenten in den Komponentenplattformen.

Alle abstrakten Methoden müssen speziell für die jeweilige Komponentenplattform implementiert werden. Ganz im Sinne eines Frameworks definieren sie damit die Erweiterungspunkte für plattformspezifische Implementierungen.

Direkte Implementierung Bei der direkten Implementierung muss Java-Reflection benutzt werden, um z. B. neue Objekte zu erzeugen oder Parameterwerte zu ändern, da die Methoden für beliebige, zur Übersetzungszeit unbekannte Komponenten bzw. Java-Klassen anwendbar sein müssen.

Indirekte Implementierung Bei der indirekten Implementierung wird Quelltext bzw. Bytecode speziell für bestimmte Komponenten generiert. Auf diese Weise kann auf die Verwendung von Java-Reflection verzichtet werden, was einen erheblichen Geschwindigkeitsvorteil mit sich bringt (siehe Abschnitt A.1).

4.3.5 Adaptions- und Aspektoperatoren

Das Operatoren-Paket stellt grundlegende Funktionen für die Implementierung von plattformspezifischen Aspekt- und Adaptionsoperatoren bereit. Die beiden Arten von Operatoren

	Generischer Adapter	Bytecode-Generierung	Quelltext-Generierung	Quelltext-Änderung	Bytecode-Änderung
Anwendungszeitpunkt	Laufzeit	Startzeit, Install.	Install., (Startzeit)	Install., (Startzeit)	Startzeit, Install.
Geschw. Erzeugung	++	–	––	––	–
Geschw. Initialisierung	+	+	++	++	+
Geschw. Laufzeit	–	+	+	++	++
Anwendbarkeit	beliebig	beliebig	beliebig	speziell	speziell

Tabelle 4.4: Vergleich von Verfahren für die Implementierung von Adaptions- und Aspektoperatoren

werden in einem Paket des Frameworks zusammengefasst, da für die Implementierung auf die gleichen Strategien zurückgegriffen wird. Prinzipiell werden vier verschiedene Verfahren unterstützt:

- Verwendung von generischen Adaptern (Adapter-Entwurfsmuster nach [GHJV95]) zur Laufzeit
- Generierung (Quelltext bzw. Bytecode) von speziellen Adaptern und Anwendung
- Änderung von Quelltext
- Änderung von Bytecode

Die Implementierung der einzelnen Verfahren wird im Folgenden näher beschrieben.

Generischer Adapter Ein generischer Adapter als eine Form der direkten Implementierung von Adaptions- und Aspektoperatoren kann nur Java-Reflection verwenden, um für beliebige Komponenten anwendbar zu sein. Mit Hilfe eines *dynamic Proxy* [dyn99] kann zur Laufzeit ein Proxy-Objekt erzeugt werden, das ein oder mehrere, ebenfalls erst zur Laufzeit festgelegte Java-Interfaces implementiert. Methodenaufrufe auf diesem speziellen Proxy-Objekt werden an einen so genannten *InvocationHandler* weitergereicht. Das ist eine beliebige andere Klasse, das Interface `InvocationHandler` implementiert. Dort werden Methoden des eigentlichen Adaptions- bzw. Aspektoperators aufgerufen. Die Abläufe bei der Erzeugung und Anwendung eines generischen Adapters zur Implementierung eines Aspektoperators sind in Form eines Sequenzdiagramms in Abbildung 4.7 veranschaulicht. Im Unterschied dazu ruft ein Adaptionsoperator selbst die entsprechende Komponentenmethode auf und übergibt erst danach die Kontrolle wieder an den *InvocationHandler*.

Bei der Verwendung des Adapter-Musters muss darauf geachtet werden, dass eine Komponentenmethode keine Referenz auf das aktuelle Objekt der Implementierungsklasse (`this` in Java bzw. C++) zurück liefert. Ein weiterer Methodenaufruf bei dieser Referenz würde sonst am Adapter vorbei gehen. Durch die explizite Verknüpfung von Komponenten mittels

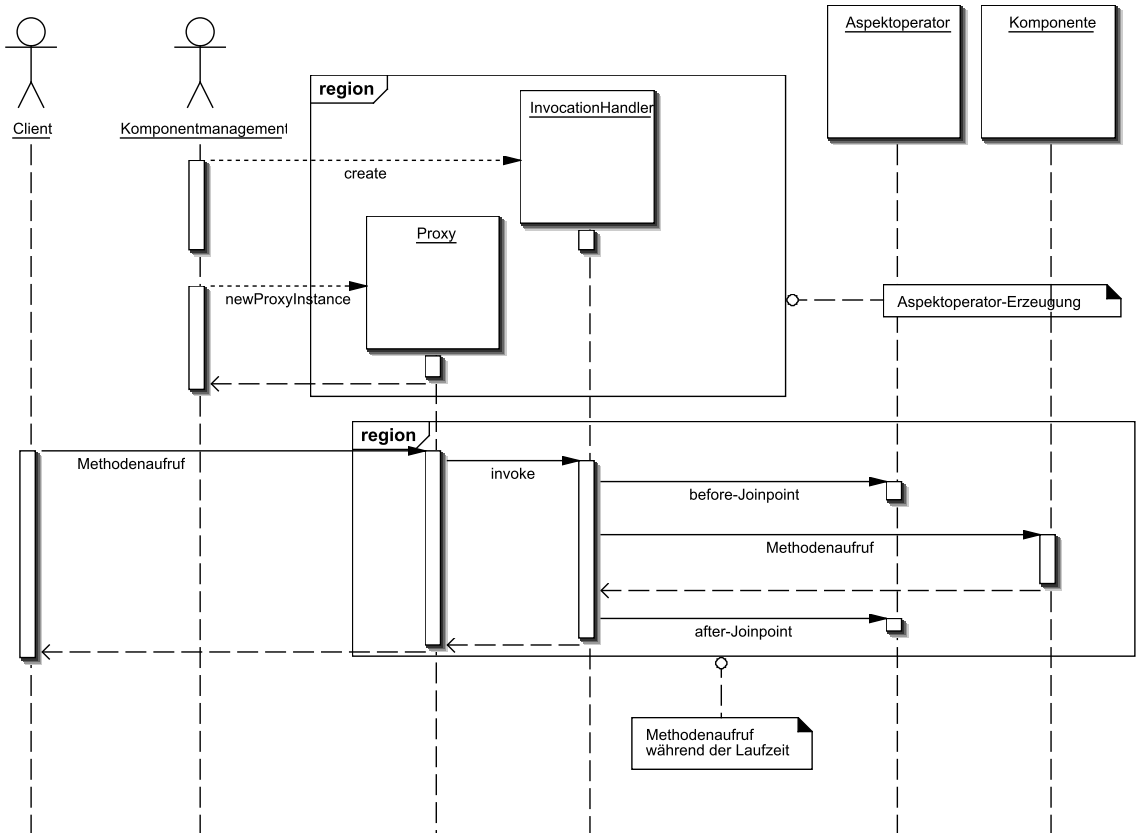

Abbildung 4.7: Sequenzdiagramm für Methodenaufrufe bei dynamischen Proxies

Verknüpfungen zwischen *required*- und *provided*-Ports ist das Übergeben von Referenzen von Komponenten aber sowieso nicht erforderlich.

Mit dem bisher vorstellten generischen Adapter können noch nicht alle möglichen Joinpoints von Aspektoperatoren (siehe Unterabschnitt 3.3.5) unterstützt werden. Jointpoints beim Erzeugung bzw. Entfernen von Subkomponenten und bei Rekonfigurationen werden vom Komponentenmanagement- bzw. Adaptionsmanager-Paket unterstützt, das dazu die entsprechenden Methoden eines Aspektoperators aufruft.

Generierung von Adaptern Anstelle eines generischen Adapters kann auch ein speziell für einen bestimmten Komponententyp zugeschnittener Adapter generiert werden. Dieser Adapter implementiert die Schnittstelle des *provided*-Ports der zugehörigen Komponente. Damit funktionieren Methodenaufrufe bei diesem Adapter ohne die Verwendung von Reflection. Der erzeugte Adapter ruft zur Laufzeit Methoden eines bestimmten Adaptions- bzw. Aspektoperators auf, ebenfalls ohne die Verwendung von Reflection. Auf diese Weise wird im Vergleich zum generischen Adapter bereits an zwei Stellen die zeitaufwändige Verwendung von Reflection durch einen direkten Methodenaufruf ersetzt. Sonst entsprechen die Abläufe zur Laufzeit im Wesentlichen denen von generischen Adaptern.

Die Generierung des Adapters kann zu unterschiedlichen Zeitpunkten und in unterschiedlichen Formen erfolgen:

- Die Generierung des Bytecodes findet **während der Entwicklungs- bzw. Installationszeit** einer Adaptierbaren Komponente statt. Der Bytecode wird entweder direkt generiert, z. B. mit Hilfe der Apache *Byte Code Engineering Library* (BCEL [Dah01]), oder indirekt durch die Übersetzung von erzeugtem Java-Quelltext[8]. Die Generierung von Bytecode ist wesentlich schneller als der Umweg über Java-Quelltext und anschließender Übersetzung. Allerdings kann der erzeugte Bytecode wesentlich einfacher geändert werden, wenn nur der entsprechende Java-Quelltext angepasst werden muss, anstatt ein Generierungsprogramm z. B. unter Nutzung von BCEL zu ändern.

- Die Generierung des Bytecodes findet **zur Laufzeit** beim Erzeugen einer Adaptierbaren Komponente statt. Mit Hilfe eines speziellen Classloaders [LB98] wird der Bytecode erst beim Zugriff auf die Adapter-Klasse erzeugt und anschließend geladen. Auch hier kann der Bytecode entweder direkt mit Hilfe von BCEL oder über den Umweg Java-Quelltext und Übersetzung erzeugt werden. Der direkte Weg mit BCEL sollte allerdings bevorzugt werden, da sonst unnötige Zeitverzögerungen beim Initialisieren einer Adaptierbaren Komponente auftreten.

Die Entscheidung, welche der verschiedenen Verfahren für die Generierung eines Adapters anwendet werden soll, muss in Abhängigkeit der Komponentenplattform, des PSMs und der Anforderungen der jeweiligen Anwendung (z. B. Geschwindigkeit vs. Flexibilität zur Laufzeit) getroffen werden. Einige Komponentenplattformen (z. B. EJB) verbieten die Verwendung von Classloaders für Komponenten. In diesem Fall kann nur die Adaptergenerierung zur Entwicklungs- bzw. Installationszeit angewendet werden (siehe auch Unterunterabschnitt 4.2.2.6).

Änderung von Quelltext Die bisher vorgestellten Verfahren verwenden das Adapter-Muster zur Realisierung der Adaptions- bzw. Aspektoperatoren. Ein Nachteil dieser Lösungen besteht darin, dass zur Laufzeit ein zusätzliches Objekt – das Adapter-Objekt – für jede Komponenteninstanz erzeugt werden muss, das anschließend bei jedem Methodenaufruf durchlaufen wird. Bei Aspektoperatoren kann zur Implementierung eines preMethod- und postMethod-Joinpoints stattdessen auch der Quelltext geändert werden, indem der Aufruf des Aspektoperator direkt an den entsprechenden Stellen in den Quelltext der Komponente eingefügt wird. Dabei müssen folgende Punkte beachtet werden:

- Die Komponente, für die ein Aspektoperator eingefügt werden soll, muss vollständig als Quelltext verfügbar sein. Wenn nur Bytecode vorhanden ist, wie etwa bei kommerziellen Komponenten, lässt sich das Verfahren nicht anwenden.

- Die Komponente besitzt genau eine Klasse, die das Interfaces des *provided*-Ports implementiert. Diese Implementierungsklasse ist durch den Komponentendeskriptor bekannt.

Wenn diese Voraussetzungen erfüllt sind, kann ein Aspektoperator vor und nach jedem Methodenaufruf aufgerufen werden, wie in der folgenden Beispielklasse illustriert wird:

```
1 public class XyzImpl implements Xyz {
2     private AspectOperator _aspectOperator1;
```

[8] Der Compiler des Java SDK besitzt eine Programmierschnittstelle zum Aufruf innerhalb von Java-Programmen.

```
3
4     public void setAspectOperator1(AspectOperator ap) {
5         this._aspectOperator1 = ap;
6     }
7
8     public String method(Object arg) {
9         _aspectOperator1.preMethod(this, "method", new Object[] {arg});
10        try {
11            ... // ursprüngliche Implementierung der Methode
12        } finally {
13            _aspectOperator1.postMethod(this, "method", new Object[] {arg});
14        }
15    }
16 }
```

Die Zeilen 2–6 fügen eine zusätzliche Membervariable und eine Methode zur Initialisierung hinzu[9]. Diese Variable enthält eine Referenz auf den Aspektoperator. Sie wird unmittelbar nach der Erzeugung einer Instanz der Klasse initialisiert. Die Zeilen 9–10 und 12–14 fügen einen Aufruf des Aspektoperators vor und nach Abarbeitung der eigentlichen Methodenimplementierung ein. Ähnlicher Quelltext wird bei allen Methoden eingefügt, die im Interface des *provided*-Ports definiert wurden. Wenn eine bestimmte Methode nur in einer Superklasse implementiert wurde, wird die Methode in der Implementierungsklasse überschrieben, die Aufrufe der Aspektoperatoren eingefügt und die ursprüngliche Methode (super.methode(..)) aufgerufen.

Die Änderung des Quelltextes besitzt gegenüber der Änderung von Bytecode den Vorteil, dass Bytecode und Quelltext weiterhin zueinander passen. Damit wird beim Debugging der Quelltext korrekt angezeigt, sofern Debug-Informationen während des Übersetzens erzeugt wurden. Für die Implementierung der Quelltext-Modifikation können die in Unterabschnitt 2.8.2 vorgestellten Verfahren eingesetzt werden.

Änderung von Bytecode Mit der Änderung von Bytecode wird wie bei der Änderung des Quelltextes die zusätzliche Erzeugung von Adapter-Objekten zur Laufzeit verhindert. Im Gegensatz dazu kann das Verfahren allerdings bei allen Komponenten angewendet werden, insbesondere auch bei kommerziellen Komponenten, die nur als Bytecode verfügbar sind.

Der Bytecode kann ähnlich wie bei der Generierung von Adaptern sowohl zur Entwicklungs- bzw. Installationszeit als auch zur Laufzeit geändert werden. Die Änderung zur Laufzeit mit Hilfe von speziellen Classloadern setzt allerdings voraus, dass dies nicht durch die Komponentenplattform untersagt ist.

Zur Durchführung der Änderung muss das Interface des *provided*-Ports der Komponente von genau einer Java-Klasse implementiert werden und die Klasse muss durch den Komponentendeskriptor bekannt sein. Der Bytecode wird dann so modifiziert, dass am Beginn und am Ende von Methodenimplementierungen die entsprechenden Methoden des Aspektoperators aufgerufen werden. Der erzeugte Bytecode entspricht dabei dem Quelltext bei der Beschreibung von Quelltextänderungen.

Die Realisierung von Aspektoperatoren durch die Änderung von Bytecode wurde im Fra-

[9] Wenn bereits eine Variable mit dem Namen `_aspectOperator1` in der Klasse existiert, wird der Index des Namens solange erhöht, bis kein Konflikt mehr auftritt.

mework nur in Form von abstrakten Methoden vorgesehen, aber noch nicht vollständig implementiert.

4.3.6 Interceptoren

Interceptoren werden für die Unterstützung der Rekonfiguration bei Runtime-PSMs benötigt (siehe Unterabschnitt 4.3.2 und Unterabschnitt 4.4.1), wenn mehrere gleichzeitige Methodenaufrufe bei einer Instanz einer Adaptierbaren Komponente möglich sind (z. B. *Multi-threaded* Anwendungen). Sie dienen zur transparenten Erkennung von Aktivität bzw. Inaktivität von Komponenten und müssen Methodenaufrufe im Rahmen einer Rekonfiguration selektiv blockieren (siehe Abschnitt 3.5). Interceptoren werden sowohl auf der Ebene von Subkomponenten als auch von Adaptierbaren Komponenten benötigt. Im Folgenden werden die verschiedenen Aufgaben von Interceptoren näher vorgestellt.

Erkennung Aktivität/Inaktivität von Komponenten Die transparente Erkennung von Aktivität bzw. Inaktivität von Komponenteninstanzen wird mit Hilfe eines Zugriffszählers in jedem Interceptor realisiert. Jeder Methodenaufruf bei einer Komponente wird von einem Interceptor abgefangen. Dort wird der Zugriffszähler für die zugehörige Komponente um eins erhöht. Nach der Abarbeitung der Methode wird wiederum der Interceptor durchlaufen und dabei der Zugriffszähler um eins vermindert. Auf diese Weise enthält der Zugriffszähler zu jedem beliebigen Zeitpunkt die Anzahl der aktiven Methodenaufrufe bei der Komponenteninstanz. Ein Wert von Null bedeutet damit, dass momentan kein Methodenaufruf abgearbeitet wird ist und die Komponente somit inaktiv ist (siehe Abschnitt 3.5). Dieses Verfahren funktioniert auch bei rekursiven Methodenaufrufen und wenn mehrere Threads gleichzeitig Methoden bei einer Komponente aufrufen. Es muss lediglich sichergestellt werden, dass immer nur ein Thread gleichzeitig den Zugriffszähler verändern darf. Der entsprechende Programmcode zur Synchronisation ist im Interceptor-Paket implementiert. Der Adaptionsmanager kann über eine Schnittstelle beim Interceptor den aktuellen Status einer Komponente (aktiv bzw. inaktiv) abfragen.

Blockierung von Methodenaufrufen Im Rahmen einer Rekonfiguration ermittelt der Adaptionsmanager eine Menge von Komponenten, die deaktiviert werden müssen (siehe Abschnitt 3.5). Anschließend werden die jeweiligen Interceptoren dieser Komponenten angewiesen, Methodenaufrufe bis auf weiteres (selektiv) zu blockieren. Für die Entscheidung, ob ein Methodenaufruf eines bestimmten Threads blockiert werden darf, muss geklärt werden, ob dieser Thread bereits zu einem früheren Zeitpunkt in der Menge der zu blockierenden Komponenten aktiv war. Mit der *Stack-Trace-Analyse* und der *Verwaltung von aktiven Threads* werden weiter unten zwei Entscheidungsverfahren beschrieben.

Wenn ein Methodenaufruf blockiert werden darf, wird der zugehörige Thread angehalten (mit `Object.wait()`). Erst nach Abschluss einer Rekonfiguration werden auf Initiative des Adaptionsmanagers alle Methodenaufrufe wieder fortgesetzt (mit `Object.notifyAll()`).

Stack-Trace-Analyse In Java (und in ähnlicher Weise auch bei anderen Programmiersprachen) wird bei jedem Methodenaufruf die aktuelle Klasse, der Methodenname, lokale Variablen und der aktuelle Programmzähler (*program counter*) auf den Stack gelegt, um nach Abarbeitung der aufgerufenen Methode wieder an diese Stelle zurückkehren zu können. Durch das

Erzeugen eines `Throwable`-Objektes[10], wodurch gleichzeitig ein Schnappschuss des Stacks des aktuellen Threads erzeugt wird, kann man programmtechnisch auf alle Elemente des Stacks zugreifen[11] und erfährt damit, durch welche Methodenaufrufe die gerade aktuelle Methode erreicht wurde und in welchen Klassen der Thread aktiv ist.

Ein Methodenaufruf darf damit nur blockiert werden, wenn keine Implementierungsklasse von allen zu blockierenden Komponenten im aktuellen Schnappschuss des Stacks enthalten ist. Der Zeitaufwand für diese Entscheidung hängt linear von der Anzahl der zu blockierenden Komponenten und der Anzahl der Elemente des Stacks ab. Die Stack-Trace-Analyse muss aber überhaupt nur durchgeführt werden, nachdem eine Blockierung durch den Adaptionsmanager veranlasst wurde. Im Normalfall – ohne eine gerade durchgeführte Rekonfiguration – benötigt das Verfahren keine zusätzliche Rechenzeit bei Methodenaufrufen.

Verwaltung von aktiven Threads Die Stack-Trace-Analyse nutzt spezielle Mechanismen der Java Virtual Machine, die aber nicht immer verfügbar sind (z. B. bei älteren Java-Versionen oder anderen Programmiersprachen). Aus diesem Grund wurde noch ein weiteres Verfahren entwickelt, das sich auch auf andere Programmiersprachen übertragen lässt. Dazu wird in jedem Interceptor eine Menge der aktiven Threads verwaltet. Vor einem Methodenaufruf prüft der Interceptor, ob der aktuelle Thread bereits in dieser Menge enthalten ist. Wenn nicht, wird der aktuelle Thread zusammen mit einem Zähler, der mit dem Wert eins initialisiert wird, hinzugefügt. Anderenfalls wird lediglich der Wert des zum aktuellen Threads gehörenden Zählers um eins erhöht.

Nach einem Methodenaufruf wird der Wert des zum aktuellen Threads gehörenden Zählers wieder um eins vermindert. Erreicht er den Wert Null, wird der Thread aus der Menge der aktiven Threads gelöscht. Die thread-spezifischen Zugriffszähler sind notwendig, um auch rekursive Methodenaufrufe korrekt zu behandeln.

Nach dem Auslösen einer Blockierung durch den Adaptionsmanager wird bei allen weiteren Methodenaufrufen überprüft, ob der aktuelle Thread bereits in der Menge der aktiven Threads bei mindestens einer zu blockierenden Komponente enthalten ist. Ist dies der Fall, wird der entsprechende Thread nicht blockiert.

Dieses Verfahren verursacht im Gegensatz zur Stack-Trace-Analyse auch im Normalfall ohne eine laufende Rekonfiguration zusätzlichen Rechenaufwand bei jedem Methodenaufruf, um die Menge der aktiven Threads zu verwalten. Der Zeitbedarf hängt von der Anzahl der bereits aktiven Threads ab. Ab der Aktivierung der Blockierung kommt zusätzlich noch eine Suchoperation in den Mengen der aktiven Threads bei allen blockierten Komponenten hinzu.

Aufgrund des zusätzlichen Rechenaufwandes (siehe Abschnitt A.2) sollte wenn möglich die Stack-Trace-Analyse für die Implementierung von Interceptoren verwendet werden.

Implementierung und Anwendung zur Laufzeit Die Interceptoren werden im Rahmen der Initialisierung einer Adaptierbaren Komponente durch das Komponentenmanagement-Paket erzeugt und beim Adaptionsmanager registriert. Der Adaptionsmanager kann auf diese Weise die Interceptoren im Rahmen einer Rekonfiguration steuern, in dem Methodenaufrufe blockiert bzw. wieder freigegeben werden (siehe Unterabschnitt 4.3.2). Die zeitlichen Abläufe bei einem Methodenaufruf sind auch in Abbildung 4.8 dargestellt.

[10] `Throwable` ist die Basisklasse für alle (*checked* und *unchecked*) Exceptions in Java.
[11] Der Zugriff mit der Methode `Throwable.getStackTrace()` wird erst ab JDK 1.4 unterstützt.

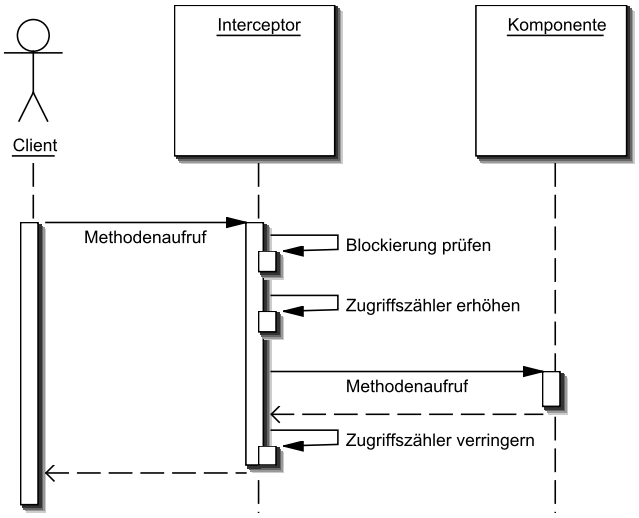

Abbildung 4.8: Sequenzdiagramm für Methodenaufrufe bei Interceptoren

Für die Implementierung von Interceptoren wird auf die Funktionalität des Operatoren-Pakets zurückgegriffen. Ein Interceptor ist nicht anderes als ein spezieller Aspektoperator, der vor und nach jedem Methodenaufruf aufgerufen wird. Damit stehen die vier Implementierungsmöglichkeiten von Aspektoperatoren auch für Interceptoren zur Verfügung. Interceptoren können damit als generischer Adapter, als generierter Adapter und durch die Änderung von Quelltext bzw. Bytecode von Komponenten implementiert werden. Die Vor- und Nachteile dieser Verfahren wurden bereits bei der Beschreibung des Operatoren-Pakets diskutiert (Unterabschnitt 4.3.5). Da Interceptoren bei jedem Methodenaufruf bei einer Komponente involviert sind, wirkt sich eine effiziente Implementierung besonders auf die Geschwindigkeit einer Anwendung aus (siehe Abschnitt A.2).

4.3.7 Zustandstransfer

Das Zustandstransfer-Paket wird im Rahmen einer Rekonfiguration vom Adaptionsmanager (siehe Unterabschnitt 4.3.2) verwendet, um Zustandsinformationen zwischen entfernten und neu hinzugefügten Komponenten zu übertragen (siehe Abschnitt 3.6). Da Rekonfigurationen nur beim Laufzeit-PSM (siehe Abschnitt 4.1) unterstützt werden, wird das Paket auch nur in diesem Zusammenhang benötigt.

Bei einer inaktiven, zu entfernenden bzw. neu hinzugefügten Subkomponente wird mit Hilfe des Metamodells überprüft, ob eine Verknüpfung von Zustandsinformationen definiert wurde. Ist dies der Fall, werden die Daten entweder von einer zu entfernenden Subkomponente gelesen und zwischengespeichert bzw. vom Zwischenspeicher in eine neu hinzugefügte Subkomponente geschrieben. Zum Zugriff auf die Zustandsinformationen wird das Komponentenmanagement-Paket benutzt, um komponentenplattformspezifische Funktionen im Zustandstransfer-Paket zu vermeiden.

Das Zustandstransfer-Paket verwendet intern ein `Map` für die Zwischenspeicherung der Name/Wert-Paare der Zustandsinformationen. Jede Instanz einer Adaptierbaren Komponen-

te besitzt intern maximal einen solchen Zustandsspeicher, der bei der Erzeugung initialisiert wird.

Es wäre denkbar, die Funktionen des Zustandstransfer-Pakets auch indirekt zu implementieren, indem Quelltext für das Lesen und Speichern der Zustandsinformationen in Abhängigkeit des Metamodells erzeugt wird. Der geringe zu erwartende Geschwindigkeitsvorteil, der sich nur für Rekonfiguration bemerkbar macht, rechtfertigt aber nicht den zusätzlichen Implementierungsaufwand.

4.3.8 Kontextmodell

Das Kontextmodell-Paket stellt grundlegende Funktionen für die Bereitstellung und den Zugriff auf Kontextinformationen (siehe Unterabschnitt 3.3.6) bereit. Darauf aufbauend wird die Auswertung von booleschen Ausdrücken und Regeln unterstützt (siehe Unterabschnitt 3.4.7), die zur Anpassung der Adaptierbaren Komponente an geänderte Umgebungsbedingungen definiert wurden.

Jede Kontextinformation wird genau einem von drei Gültigkeitsbereichen zugeordnet:

Komponenteninstanz Die Kontextinformation ist spezifisch für eine bestimmte Komponenteninstanz (z. B. eine Statistik über Methodenaufrufe pro Sekunde).

Komponententyp Alle Komponenten eines bestimmten Komponententyps teilen sich die gleiche Kontextinformation (z. B. Anzahl der erzeugten Komponenten).

Global Alle Komponenten innerhalb einer Anwendung teilen sich die gleiche Kontextinformation (z. B. der verfügbare Hauptspeicher).

Die Kontextinformationen werden intern als Liste verwaltet und die jeweiligen Kontextwerte werden zwischengespeichert. Für die Verwaltung der unterschiedlichen Gültigkeitsbereiche erlaubt das Kontextmodell-Paket ebenfalls Zustandsinformationen pro Komponente, pro Komponententyp und anwendungsglobal.

Eine *Kontextquelle* kann entweder selbst den zugehörigen zwischengespeicherten Wert der Kontextinformation aktualisieren (*push*-Verfahren) oder das Kontextmodell-Paket fragt mit einem eigenen Thread periodisch den Wert ab (*polling*-Verfahren). Durch die Zwischenspeicherung werden Kontextanbieter und Kontextnutzer voneinander entkoppelt und es wird vermieden, dass ein Zugriff auf eine Kontextinformation direkt an die entsprechende Kontextquelle weitergeleitet wird.

Jede Kontextquelle besteht aus einer Implementierungsklasse, die ein bestimmtes Interface implementiert (`ContextSource`). Die Registrierung der Kontextquellen und die Zuordnung zu einer bestimmten Kontextinformation erfolgt über Konfigurationsdateien für jeden Komponententyp bzw. für die Anwendung. Die Implementierung von Kontextquellen für eine bestimmte Komponenteninstanz bzw. einen Komponententyp wird zusammen mit der Adaptierbaren Komponente gebündelt. Dabei müssen Einschränkungen der Zielkomponentenplattform beachtet werden (siehe Unterunterabschnitt 4.2.2.7).

Für jede Anwendung (innerhalb einer JVM) verwaltet das Framework anwendungsglobale Kontextinformationen und dazugehörige Kontextquellen (Singleton-Muster [GHJV95]). Diese werden bei der ersten Verwendung einer beliebigen Adaptierbaren Komponente initialisiert. In einer ähnlichen Weise werden für jeden Komponententyp die zugehörigen Kontextinformationen bei der ersten Erzeugung einer entsprechenden Adaptierbaren Komponente initialisiert.

Kontextnutzer können auf drei verschiedenen, aufeinander aufbauende Abstraktionsebenen auf die Kontextinformationen zugreifen:

- durch die direkte Abfrage des Wertes einer bestimmten Kontextinformation,
- durch die Auswertung eines booleschen Ausdruckes, der eine Bedingung von Kontextwerten definiert, und
- durch die Definition von Regeln in Form von bedingten Zuweisungen, die boolesche Ausdrücke enthalten und Komponentenparameter ändern.

Regeln werden im Gegensatz zu den anderen Zugriffsarten nicht nur einmal ausgewertet, sondern gespeichert. Bei allen Kontextwerten, die von Kontextquellen mit dem *push*-Verfahren aktualisiert werden, lösen Wertänderungen unmittelbar eine erneute Auswertung der Regeln aus. Alle anderen Kontextquellen werden periodisch zuerst nacheinander abgefragt (*polling*-Verfahren) und danach werden die Regeln neu ausgewertet.

Damit bei der Änderung eines Kontextwertes nicht immer alle Regeln neu ausgewertet werden müssen, wird für jede Kontextinformation eine Liste mit abhängigen Regeln verwaltet. Eine Regel ist von einem Kontextwert abhängig, wenn er im booleschen Ausdruck der Bedingung verwendet wird. Die Abhängigkeitsliste wird jeweils beim Hinzufügen bzw. Entfernen von Regeln aktualisiert.

Während der Auswertung von Ausdrücken bzw. Regeln darf keine Aktualisierung der zwischengespeicherten Kontextwerte stattfinden, um zeitliche Änderungen während der Auswertung auszuschließen. Um dies zu erreichen, wird vor der Auswertung das Kontextmodell für Schreibzugriffe gesperrt und erst anschließend wieder freigegeben.

Für die Auswertung der gespeicherten Regeln gibt es zwei verschiedene Verfahren (siehe Abbildung 3.14):

- Bei der **periodischen Auswertung** werden die Kontextwerte in regelmäßigen Abständen abgefragt und die bedingten Zuweisungen aufgrund dieser Werte ausgewertet. Als Nachteile werden bei dieser Lösung Änderungen von Kontextinformationen spätestens erst nach einer Periodendauer registriert und es muss regelmäßig Rechenzeit für die Abfrage der Kontextinformationen aufgewendet werden. Dem steht als Vorteil eine relativ einfache Implementierung des Kontextmodells entgegen. Werte von Kontextinformationen müssen erst bei der Abfrage ermittelt werden.
- Bei der **ereignisgesteuerten Auswertung** werden die bedingten Zuweisungen nur ausgewertet, wenn eine Änderung der Kontextinformationen registriert wurde. Zur Realisierung wird das Observer-Muster [GHJV95] verwendet, d. h. das Kontextmodell informiert einzelne oder alle bedingten Zuweisungen, wenn sich die darin enthaltenen Kontextinformationen geändert haben. Nur dann werden die Bedingungen ausgewertet, was gegebenenfalls zur Änderung von Parametern führt. Der Vorteil dieses Ansatzes besteht darin, dass auf Änderungen der Kontextinformationen unmittelbar reagiert werden kann. Allerdings ist es nicht immer möglich, Änderungen innerhalb des Kontextmodells zu erkennen. Es kann daher passieren, dass die erwähnten Nachteile der periodischen Auswertung nur in die Implementierung des Kontextmodells verlagert werden.

Die Art der Auswertung von bedingten Zuweisungen ist unabhängig von der Implementierung des Kontextmodells. Auch ein Kontextmodell, das intern periodische Messungen durchführt,

kann mit der ereignisgesteuerten Auswertung an die bedingten Zuweisungen gekoppelt werden. Ähnliches gilt für alle anderen Kombinationsmöglichkeiten.

4.3.9 Metaobject Protocol für Adaptierbare Komponenten

Mit einem (einfachen) Metaobject Protocol (MOP) [KRB91] kann das Verhalten von Adaptierbaren Komponenten zur Laufzeit analysiert und verändert werden. Das MOP kann nur zusammen mit dem Runtime-PSM eingesetzt werden, da nur dort die Rekonfiguration zur Laufzeit unterstützt wird. Ein ausschließlich lesender Zugriff auf Komponenteninformationen ist jedoch auch bei Installationszeit- und Startzeit-PSM möglich.

Die Eigenschaften und das Verhalten einer Adaptierbaren Komponente werden durch ihr Modell definiert, das im Metamodell-Paket (siehe Unterabschnitt 4.3.1) gespeichert ist und bei der Komponenteninitialisierung geladen wird. Ohne Verwendung eines MOPs wird dieses Modell zur Entwicklungszeit mit geeigneten Werkzeugen (siehe Unterabschnitt 4.2.3) festgelegt. Mit Hilfe eines MOPs können externe Entitäten (z. B. andere Komponenten oder Werkzeuge) das Modell und damit auch die Eigenschaften von Adaptierbaren Komponenten zur Laufzeit erforschen (*introspection*) und gegebenenfalls ändern.

Folgende Funktionen werden durch das MOP unterstützt:

- Vollständiger lesender Zugriff auf das Modell
- Hinzufügen bzw. Ändern von vollständigen und Template-Konfigurationen, Variationen und Template-Variationen
- Hinzufügen und Entfernen von inaktiven Subkomponenten
- Hinzufügen und Löschen von Regeln der Parameterabbildung

Änderungen am Modell einer Adaptierbaren Komponente können Inkonsistenzen verursachen. Wird z. B. eine neue vollständige Konfiguration und eine entsprechende Regel der Parameterabbildung hinzugefügt, aber noch nicht dafür notwendige Subkomponenten, kann ein Fehler auftreten. Daher dürfen mehrere zusammengehörende Änderungsoperationen, die mit Hilfe des MOPs vorgenommen werden, nicht unterbrochen oder beeinflusst werden. Es gelten somit ähnliche Forderungen, wie schon bei Rekonfigurationen bzw. Datenbank-Transaktionen (siehe Abschnitt 3.5). Zur Realisierung muss ein Anwender des MOP zuerst eine Änderungsberechtigung (*write lock*) anfordern, bevor Änderungsoperationen durchgeführt werden dürfen. Ab diesem Moment ist das Modell für alle anderen Anwender gesperrt, d. h. es kann weder lesend noch schreibend zugegriffen werden. Nach Abschluss aller Änderungen und dem Erreichen eines konsistenten Zustands wird die Änderungsberechtigung wieder freigegeben.

4.4 Verfahren zur Implementierung plattformspezifischer Modelle

Wie bereits in Abschnitt 4.1 vorgestellt, werden für jede Komponentenplattform drei verschiedene PSMs unterstützt. Für ihre Implementierung wird auf das Framework zur Unterstützung Adaptierbarer Komponenten (siehe Abschnitt 4.3) zurückgegriffen. Dabei werden die folgenden Phasen unterschieden:

- Werkzeugunterstützung während der Installation (z. B. Codegenerierung)

- Erzeugung einer Adaptierbaren Komponente zur Laufzeit
- Verhalten zur Laufzeit

Die Festlegung, welches der drei PSMs verwendet werden soll, muss nicht für alle Adaptierbaren Komponenten einer Anwendung gleich sein, sondern kann auch individuell für jede Komponente getroffen werden. Auf diese Weise kann für jede einzelne Komponente der beste Kompromiss aus Flexibilität und Performance gewählt werden.

Im Folgenden werden die Implementierungsdetails der drei Modelle beschrieben, jedoch unabhängig von einer bestimmten Komponentenplattform.

4.4.1 Laufzeit-PSM

Das Laufzeit-PSM könnte eigentlich in zwei separate PSMs aufgeteilt werden, je nachdem, ob MOP unterstützt wird oder nicht. Da die Unterschiede zwischen den beiden Varianten aber gering sind, wurde auf diese Aufteilung verzichtet. Das Laufzeit-PSM benötigt alle Pakete des Frameworks, um eine Rekonfiguration zur Laufzeit zu unterstützen. Lediglich das MOP-Paket wird nur benötigt, wenn MOP auch unterstützt werden soll. Abbildung 4.9 zeigt die prinzipielle Architektur einer Adaptierbaren Komponente, verwendete Werkzeuge und die Nutzung des Frameworks.

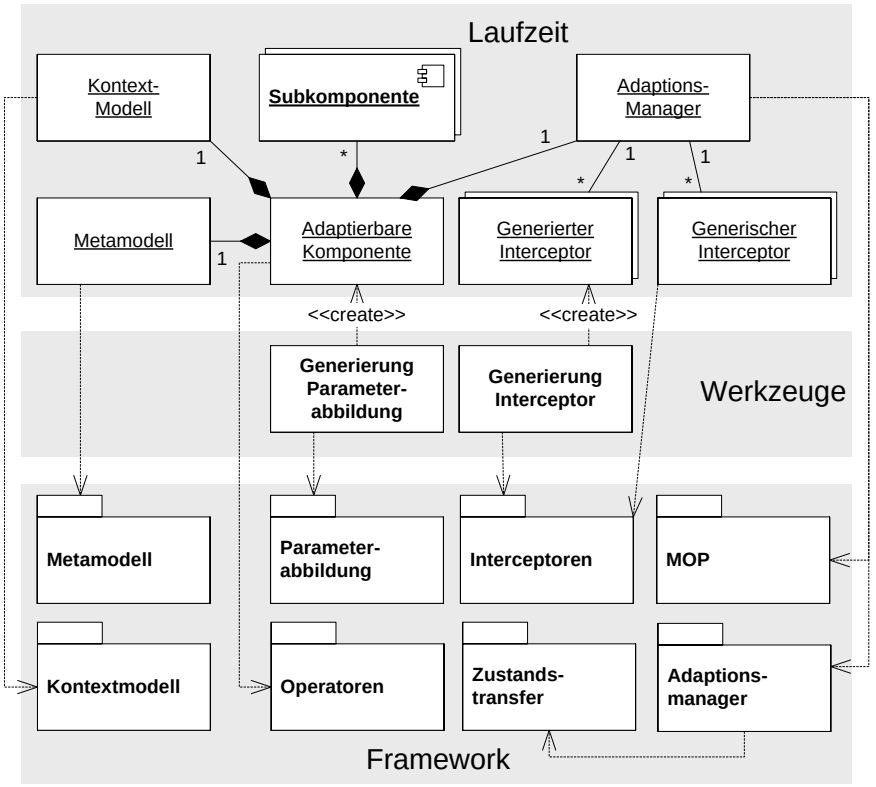

Abbildung 4.9: Laufzeit-PSM

Werkzeugunterstützung während der Installation Beim Laufzeit-PSM werden nahezu alle Entscheidungen zur Konfiguration von Adaptierbaren Komponenten erst zur Laufzeit getroffen. Daher wird eine Werkzeugunterstützung prinzipiell nicht benötigt. Durch die Generierung von Programmcode während der Installation kann jedoch die Laufzeit-Performance von Adaptierbaren Komponenten gesteigert werden, ohne die Flexibilität zu reduzieren. An den folgenden Stellen kann eine werkzeuggestützte Codegenerierung eingesetzt werden:

- Generierung der Logik für die Parameterabbildung: Wenn kein MOP unterstützt wird, sind die Regeln zur Parameterabbildung statisch zur Laufzeit und können daher mit speziell dafür generiertem Programmcode wesentlich schneller ausgewertet werden. Das Parameterabbildungs-Paket des Frameworks (siehe Unterabschnitt 4.3.3) stellt diese Funktionalität bereit.
- Generierung von Interceptoren für alle Subkomponenten: Durch die Generierung von speziell angepassten Interceptoren mit Hilfe des Interceptoren-Pakets (siehe Unterabschnitt 4.3.6) kann auf die Verwendung von Reflection zur Laufzeit verzichtet werden. Bei der Unterstützung von MOP sind zur Laufzeit für neu hinzugefügte Komponenten nur generische Interceptoren verfügbar.
- Modifikation von Komponenten bzw. Generierung von speziellen Adaptern für Adaptions- und Aspektoperatoren: Alle Komponenten, die durch Adaptions- oder Aspektoperatoren in verschiedenen Konfigurationen verändert werden sollen, können auch schon zur Installationszeit bzw. beim Laden durch Bytecode- oder Quelltext-Änderungen mit Hilfe des Operatoren-Pakets (siehe Unterabschnitt 4.3.5) angepasst werden. Bei der Unterstützung von MOP können zur Laufzeit für geänderte Konfigurationen nur generische Adaptions- und Aspektoperatoren angewendet werden.

Erzeugung einer Adaptierbaren Komponente zur Laufzeit Die folgenden Schritte werden durchgeführt:

1. Eine Komponenteninstanz als Stellvertreter für die Adaptierbare Komponente wird durch plattformspezifische Mechanismen erzeugt.
2. Das Komponentenmanagement-Paket erzeugt mit Hilfe des Interceptor-Pakets den Adaptionsmanager und registriert dort einen Interceptor für die Adaptierbare Komponente.
3. Wenn Regeln zur Selbstadaptivität (siehe Unterabschnitt 3.4.7) im Modell der Adaptierbaren Komponente enthalten sind, wird das Kontextmodell mit zugehörigen Kontextquellen initialisiert.
4. Die aktuellen Parameterwerte werden durch das Parameterabbildungs-Paket unter Nutzung des Metamodells bzw. durch den zur Installationszeit generierten Quelltext ausgewertet und damit die Konfiguration ermittelt.
5. Entsprechend der ermittelten Konfiguration und den Informationen aus dem Metamodell erzeugt und initialisiert das Komponentenmanagement-Paket alle Subkomponenten. Falls erforderlich werden dabei auch Adaptions- bzw. Aspektoperatoren angewendet. Die Form der Anwendung (siehe Unterabschnitt 4.3.5) hängt von der Zielkomponentenplattform ab, z. B. ob Bytecode-Modifikationen erlaubt sind (siehe Eigenschaft E-18).

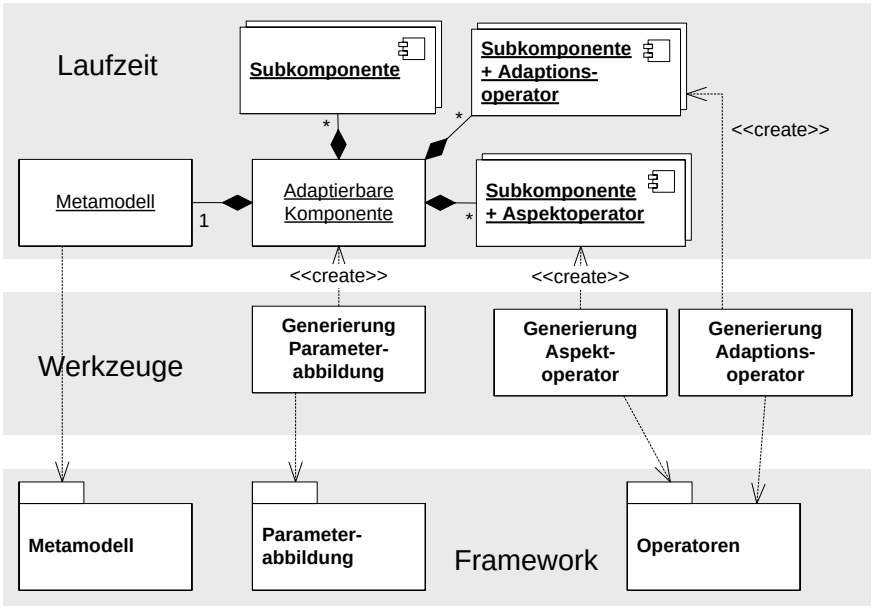

Abbildung 4.10: Startzeit-PSM

6. Für jede Subkomponente werden Interceptoren erzeugt und beim Adaptionsmanager registriert.

7. Entsprechend der ermittelten Konfiguration und den Informationen aus dem Metamodell erzeugt das Komponentenmanagement-Paket alle Verbindungen zwischen internen Ports von Subkomponenten und externen Ports der Adaptierbaren Komponente.

4.4.2 Startzeit-PSM

Da beim Startzeit-PSM auf die Unterstützung von Rekonfiguration verzichtet wird, reduziert sich die Komplexität von Adaptierbaren Komponenten zur Laufzeit und mehrere Pakete des Frameworks (Kontextmodell, Interceptoren und Adaptionsmanager) werden nicht mehr benötigt. Abbildung 4.10 zeigt die prinzipielle Architektur einer Adaptierbaren Komponente, verwendete Werkzeuge und die Nutzung des Frameworks.

Werkzeugunterstützung während der Installation Beim Laufzeit-PSM können bis auf die Generierung von Interceptoren die gleichen Werkzeuge wie beim Laufzeit-PSM angewendet werden (siehe Unterabschnitt 4.4.1).

Erzeugung einer Adaptierbaren Komponente zur Laufzeit Die folgenden Schritte werden durchgeführt:

1. Eine Komponenteninstanz als Stellvertreter für die Adaptierbare Komponente wird durch plattformspezifische Mechanismen erzeugt.

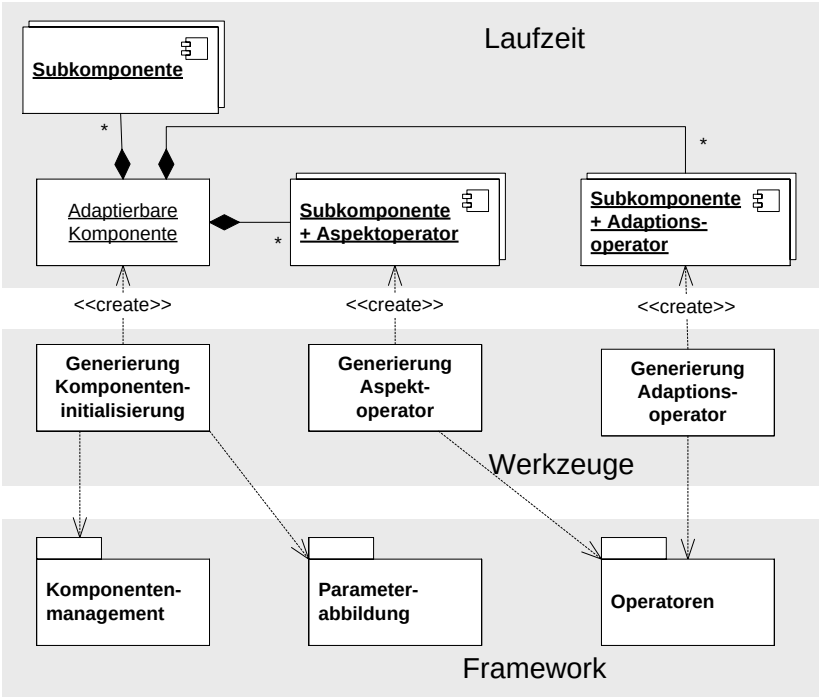

Abbildung 4.11: Installationszeit-PSM

2. Die aktuellen Parameterwerte werden durch das Parameterabbildungs-Paket unter Nutzung des Metamodells bzw. durch den zur Installationszeit generierten Programmcode ausgewertet und damit die Konfiguration ermittelt.

3. Entsprechend der ermittelten Konfiguration und den Informationen aus dem Metamodell erzeugt und initialisiert das Komponentenmanagement-Paket alle Subkomponenten. Falls erforderlich werden dabei auch Adaptions- bzw. Aspektoperatoren angewendet.

4. Entsprechend der ermittelten Konfiguration und den Informationen aus dem Metamodell erzeugt das Komponentenmanagement-Paket alle Verbindungen zwischen internen Ports von Subkomponenten und externen Ports der Adaptierbaren Komponente.

4.4.3 Installationszeit-PSM

Beim Installationszeit-PSM wird die Komplexität von Adaptierbaren Komponenten weitestgehend von der Laufzeit zur Installationszeit verlagert. Dementsprechend nimmt die Komplexität der Werkzeugunterstützung zur Installationszeit zu. Zur Laufzeit wird nur noch das Komponentenmanagement-Paket des Frameworks benötigt, da ansonsten alle notwendigen Funktionen durch generierten Programmcode übernommen werden. Aber auch zur Installationszeit werden nur noch Parameterabbildung, Komponentenmanagement, Aspekt- und Adaptionsoperatoren und das Metamodell-Paket verwendet. Abbildung 4.11 zeigt die prinzipielle Architektur einer Adaptierbaren Komponente, verwendete Werkzeuge und die Nutzung des Frameworks.

Werkzeugunterstützung während der Installation Im Unterschied zu den anderen beiden PSMs ist beim Installationszeit-PSM einer Werkzeugunterstützung zwingend erforderlich. Dazu gehören die bereits beschriebenen Werkzeuge zur Erzeugung der Parameterabbildung und von Aspekt- und Adaptionsoperatoren. Zusätzlich muss auch die Initialisierungslogik für Adaptierbare Komponenten generiert werden, die Subkomponenten erzeugt und feste Verbindungen herstellt.

Als wichtigstes Werkzeug wird beim Installationszeit-PSM ein Mechanismus zur Festlegung von Parameterwerten zur Installationszeit benötigt. Als Ergebnis kann z. B. eine XML-Datei erzeugt werden, die den Parametern einer Komponente bestimmte Werte zuordnet. Mit Hilfe der definierten Parameterwerte kann erst die Konfiguration ermittelt werden, die zur Codegenerierung benutzt werden soll.

Erzeugung einer Adaptierbaren Komponente zur Laufzeit Die folgenden Schritte werden durchgeführt:

1. Eine Komponenteninstanz als Stellvertreter für die Adaptierbare Komponente wird durch plattformspezifische Mechanismen erzeugt.

2. Die generierte Initialisierungslogik erzeugt entsprechend der zugrunde gelegten Konfiguration alle Subkomponenten. Adaptions- bzw. Aspektoperatoren wurden bereits zur Installationszeit angewendet.

3. Die generierte Initialisierungslogik erzeugt alle Verbindungen zwischen internen Ports von Subkomponenten und externen Ports der Adaptierbaren Komponente.

Kapitel 5

Validierung: Unterstützung von Komponentenplattformen und Fallstudien

In diesem Kapitel wird die praktische Anwendung der entwickelten Konzepte und Modelle für Adaptierbare Komponenten (Kapitel 3) sowie der Modelltransformation und Laufzeitunterstützung (Kapitel 4) anhand von zwei unterschiedlichen Kriterien gezeigt:

1. Die Unterstützung des Modells durch verschiedene Komponentenplattformen: Unter Anwendung des Verfahrens aus Abschnitt 4.2 wird die Unterstützung des Modells auf Basis von EJB (Abschnitt 5.1), JavaBeans (Abschnitt 5.2) und Microsoft COM (Abschnitt 5.3) gezeigt. Als Spezialfall wird die Unterstützung von Web-Services (Abschnitt 5.4) beschrieben, auch wenn es sich dabei nicht um eine Komponentenplattform handelt. Alle Implementierungen werden mit einer einfachen Beispielanwendung in Form einer Kryptographiekomponente getestet. Mit der Implementierung der Kryptographiekomponenten auf den jeweiligen Komponentenplattformen wird gezeigt,

2. Die Anwendung der Modellierungstechniken anhand von zwei unterschiedlichen Fallstudien: In der ersten Fallstudie wird ein Modell zur Konfiguration des Open-Source Servlet-Containers Apache Tomcat [Tom05] definiert (Abschnitt 5.5). Die zweite Fallstudie verwendet Adaptierbare Komponenten zur Entwicklung von Komponenten mit mehreren QoS-Profilen (Abschnitt 5.6).

5.1 Unterstützung von EJB

Enterprise JavaBeans (EJB [DeM03]) als Bestandteil der *Java2 Enterprise Edition* (J2EE) hat sich in den letzten Jahren speziell für die Entwicklung von Serveranwendungen und in Verbindung mit Servlets [Ser03] für Web-Anwendungen etabliert. Die in diesem Abschnitt betrachtete EJB-Version 2.1 ist momentan (2005) aktuell, aber Version 3.0 befindet sich bereits in der Entwicklung. Zahlreiche kommerzielle (z. B. IBM Websphere und BEA Weblogic) und Open-Source Applikationsserver (z. B. JBoss und JOnAS) stehen für die Anwendungsentwicklung zur Auswahl. Aus diesen Gründen wurde EJB als wichtige Komponentenplattform für die Unterstützung von Adaptierbaren Komponenten ausgewählt.

5.1.1 Analyse der Eigenschaften der Komponentenplattform EJB

Die Eigenschaften von EJB entsprechend der Kriterien aus Unterabschnitt 4.2.1 sind in Tabelle 5.1 dargestellt. Im Folgenden werden einige ausgewählte Eigenschaften zusammen mit ihren Auswirkungen auf Implementierungsentscheidungen näher erläutert.

Merkmal		Unterstützung
E–1	Zusammengesetzte Komponenten	nein
E–2	Variable Port-Verknüpfung	—
E–3	Trennung Implementierung und Schnittstelle	ja
E–4	Komponentenparameter	nein
E–7	*required*-Ports	ja, mit Einschränkungen
E–8	Mehrfach *required*-Ports	nein
E–9	Mehrere *provided*-Ports	nein
E–10	Explizite Verbindungen	ja, mit Einschränkungen
E–11	Änderbare explizite Verbindungen	nein
E–12	Nur explizite Verbindungen	nein
E–13	Synchrone Kommunikation	ja
E–14	Zustandsbehaftete Komponenten	ja
E–15	MOP für Komponenteneigenschaften	nein
E–16	Verhinderung gleichzeitiger Methodenaufrufe	ja
E–17	Dynamische Proxies und Reflection	ja
E–18	Bytecode-Modifikation	nein

Tabelle 5.1: Ermittlung der Eigenschaften von EJB

***Required*-Ports** EJB unterstützen ein Konzept für *required*-Ports, dass sich jedoch in der Umsetzung von anderen Komponentenplattformen unterscheidet. Zum einen sind die *required*-Ports nicht in der Komponentenschnittstelle ersichtlich. Stattdessen können im Deskriptor einer EJB-Komponente[1] Referenzen zu anderen EJB-Komponenten definiert werden. Zum anderen beziehen sich die Referenzen auf Home-Interfaces von Komponenten, d. h. nicht auf eine bestimmte Komponenteninstanz, sondern auf einen Komponententyp. Erst innerhalb einer Komponentenimplementierung kann mit Hilfe vom speziellen Einträgen im Namensdienst (JNDI-Context `java:comp/env/ejb`) auf diese Home-Interface-Referenzen zugegriffen werden, um damit eine Komponenteninstanz zu erzeugen. Die Verwendung von EJB-Referenzen im Deskriptor wird von der EJB-Spezifikation allerdings nicht zwingend vorgeschrieben (Eigenschaft E-12 nicht erfüllt).

Damit kann das in EJB verwendete Konzept von *required*-Ports nicht zum Herstellen von Verbindungen zwischen bestimmten Komponenteninstanzen verwendet werden. Dies kann nur durch die Übergabe von Referenzen als Argumente bei Methodenaufrufen emuliert werden (siehe Unterunterabschnitt 4.2.2.4).

***Provided*-Ports** Bei EJB-Komponenten werden zwei verschiedene Arten von *provided*-Ports (bei EJB als Komponentenschnittstellen bezeichnet) unterschieden: Remote- und Local-Interfaces. Local-Interfaces können nur von EJB-Komponenten innerhalb des gleichen EJB-Servers bzw. der gleichen JVM verwendet werden. Parameter bei Methodenaufrufen werden dabei als Referenz übergeben (*call by reference*). Remote-Interfaces können dagegen von beliebigen Cli-

[1] Der Komponentendeskriptor `ejb-jar.xml` enthält Informationen zu einer oder mehreren EJB-Komponenten und wird zusammen mit dem Bytecode der Komponenten in einem jar-Archiv gespeichert.

ents aufgerufen werden, insbesondere auf auch von entfernten Clients über Netzwerke. Dafür werden Parameter bei Methodenaufrufen mit Hilfe von Java-Serialisierung als Wert übergeben (*call by value*). Durch den zusätzlichen Aufwand für die Serialisierung sind Methodenaufrufe bei Remote-Interfaces immer erheblich langsamer als bei Local-Interfaces.

Da Subkomponenten und die zugehörige Adaptierbare Komponente in der Regel in der gleichen JVM ablaufen, sollten für Subkomponenten ausschließlich Local-Interfaces verwendet werden, um eine möglichst hohe Leistung zu erreichen. Für externe *provided*-Ports von Adaptierbare Komponenten können dagegen unter Berücksichtigung des jeweiligen Anwendungsszenarios beide Arten von Schnittstellen verwendet werden.

Mehrfach-*provided*-Ports werden von EJB nicht direkt unterstützt. Entweder wird bereits bei der Modellierung auf die Verwendung verzichtet oder es kann zur Emulation das in Unterunterabschnitt 4.2.2.4 beschriebene Verfahren mit expliziten Typ-Casts verwendet werden.

Komponentenparameter EJB unterstützt die Definition von Parametern zur Installationszeit mit Hilfe von Einträgen im Komponentendeskriptor (`env-entry`), die zur Laufzeit über den Namensdienst (JNDI) gelesen werden können. Allerdings können diese Parameter zur Laufzeit nicht geändert werden und sind daher für eine Abbildung auf das Parameterkonzept von Adaptierbaren Komponenten nicht geeignet, da zumindest beim Startzeit- und Laufzeit-PSM auch Parameteränderungen zur Laufzeit unterstützt werden müssen. Komponentenparameter müssen daher durch getter- und setter-Methoden in der Komponentenschnittstelle emuliert werden (siehe Unterunterabschnitt 4.2.2.3).

Trennung von Implementierung und Schnittstellen EJB erzwingt eine Trennung von Implementierungen und Schnittstellen von Komponenten (Eigenschaft E-3 erfüllt). Allerdings wird die Schnittstelle (Local- oder Remote-Interface) nicht, wie bei objektorientierter Programmierung sonst üblich, direkt von der Implementierungsklasse[2] implementiert. Stattdessen wird über Namenskonventionen eine paarweise Beziehung zwischen Methoden der Schnittstelle und der Implementierungsklasse hergestellt. Allen Methoden der Schnittstelle wird auf diese Weise genau eine Methode der Implementierungsklasse zugeordnet. Die Komponentenschnittstelle wird zur Laufzeit von einer Klasse des EJB-Containers implementiert.

Dieses Verhalten hat Auswirkungen auf die Implementierung von Operatoren (siehe Unterabschnitt 4.3.5) bzw. von Interceptoren (siehe Unterabschnitt 4.3.6). Es ist nicht möglich, einen generischen Adapter zwischen die Schnittstelle und Implementierungsklasse zu installieren, da der Adapter, genauer gesagt der dynamische Proxy, nicht die Rolle einer Bean-Klasse im EJB-Deskriptor übernehmen kann. Es ist aber möglich, einen speziell für die Komponentenschnittstelle generierten Adapter anstelle der ursprünglichen Bean-Klasse im EJB-Deskriptor anzugeben. Dieser Adapter kann dann als Operator bzw. Interceptor agieren und leitet Methodenaufrufe anschließend an die eigentliche Implementierungsklasse weiter.

Verhinderung gleichzeitiger Methodenaufrufe Der EJB-Container serialisiert bei Session- und Message-driven Beans alle Methodenaufrufe, so dass keine gleichzeitigen Methodenaufrufe unterschiedlicher Threads bei einer Komponenteninstanz auftreten können (siehe Abschnitte 7.12.10 und 15.4.6 in [DeM03]).

[2] bei EJB üblicherweise als Bean-Klasse bezeichnet

Bei Entity-Beans kann im EJB-Deskriptor individuell festgelegt werden, ob eine bestimmte Komponente gleichzeitige Methodenaufrufe unterstützt oder nicht (siehe Abschnitt 10.5.12 in [DeM03]). Der EJB-Container sorgt für die Einhaltung dieser Festlegung zur Laufzeit.

Verbotene Operationen Die EJB-Spezifikation verbietet die Verwendung einer Reihe von Operationen innerhalb von EJB-Komponenten (siehe Abschnitt 25.1.2 in [DeM03]). Im Folgenden werden alle verbotenen Operationen untersucht, die sich auf die Umsetzung von Adaptierbaren Komponenten auswirken:

- Thread-Operationen (Erzeugen, Stoppen, Synchronisation): Interceptoren (Unterabschnitt 4.3.6) benötigen Mechanismen zur Thread-Synchronisation (z. B. `Object.wait()`), um Methodenaufrufe im Rahmen einer Rekonfiguration blockieren zu können. Durch das Verbot können Interceptoren nicht innerhalb von EJB-Komponenten angewendet werden. Sie werden aber auch nicht benötigt, da gleichzeitige Methodenaufrufe bereits durch den EJB-Container unterbunden werden.
- Classloader: Es ist nicht möglich Aspekt- und Adaptionsoperatoren mit Hilfe von Classloadern zu implementieren, die den Bytecode von Komponenten beim Laden zu verändern (siehe Unterabschnitt 4.3.5).
- Dateisystem-Zugriff mit `java.io`-Paket: Deskriptoren für Adaptierbare Komponenten, etwa zur Speicherung des Modells, müssen im Komponentenarchiv gespeichert werden und über Methoden des Classloaders (`getResource()`) geladen werden. Der direkte Zugriff auf andere Dateien im Dateisystem ist nicht erlaubt.
- Übergabe von `this`-Referenz als Methodenresultat bzw. Parameter bei Methodenaufruf: Beim Herstellen von Verbindungen zwischen EJB-Komponenten muss immer eine Referenz auf die Local- bzw. Remote-Schnittstelle ausgetauscht werden (`SessionContext.getEJBLocalObject()` bzw. `SessionContext.getEJBObject()`).
- Lese- und Schreibzugriff auf statische Felder in Bean-Klassen: Statische Felder dürfen nicht für die Implementierung von Subkomponenten bzw. Adaptierbaren Komponenten verwendet werden. Wenn eine zentrale Instanz (*Singleton*) für alle Adaptierbaren Komponenten benötigt wird, wie z. B. beim Kontextmodell, muss dafür eine spezielle Komponente beim Namensdienst (JNDI) registriert und benutzt werden.

Komponentenarten EJB unterscheidet drei verschiedene Komponentenarten (Erläuterungseigenschaft E-5): Session-Beans, Entity-Beans und Message-driven Beans. Bei Session-Beans wird weiterhin zwischen *stateful* (zustandsbehaftet) und *stateless* (zustandslos) unterschieden.

Message-driven Beans dienen zur asynchronen, nachrichtenbasierten Kommunikation und können daher nicht als Subkomponenten innerhalb von Adaptierbaren Komponenten verwendet werden. Subkomponenten können aber weiterhin auf Message-driven Beans zugreifen, allerdings werden diese Verbindungen nicht explizit modelliert und können demzufolge auch nicht im Rahmen einer Rekonfiguration geändert werden.

Stateless-Session-Beans dürfen keinen *required*-Port verwenden, da die zugehörigen Referenzen auf andere Komponenteninstanzen nicht zwischengespeichert werden können. Wenn

auf die Verwendung eines *required*-Ports nicht verzichtet werden kann, muss eine Stateful-Session-Bean verwendet werden.

Stateful-Session-Beans eignen sich uneingeschränkt zur Verwendung als Subkomponenten. Durch die fehlende Unterstützung von zusammengesetzten Komponenten, werden Adaptierbare Komponenten auf eine generierte Stateful-Session-Bean als Stellvertreter abgebildet, wie in Unterunterabschnitt 4.2.2.1 beschrieben.

Entity-Beans unterscheiden sich in ihrer Handhabung von anderen Komponentenarten, da sie immer einen eindeutigen Schlüssel (*Primary Key*) besitzen und ihre Instanzdaten persistent in einer Datenbank abgespeichert werden. Mit Hilfe des Home-Interfaces können sowohl bereits existierende Instanzen aus der Datenbank geladen (entweder über den *Primary Key* oder spezielle *finder*-Methoden) als auch neue Instanzen erzeugt werden. Das bedeutet, dass bereits bei der Erzeugung einer Entity-Bean ein Wert für den *Primary Key* definiert werden muss. Wenn eine Subkomponente als Entity-Bean implementiert wird, muss daher mindestens ein Komponentenparameter vorgesehen werden, der den Wert des *Primary Keys* enthält.

Session-Beans werden üblicherweise innerhalb von EJB-Anwendungen dazu verwendet, mit Hilfe von Entity-Beans auf persistente Daten zuzugreifen (z. B. Facade-Muster [Mar02]). In diesem Fall ist es nicht sinnvoll, die Entity-Beans als Subkomponenten zu modellieren, wenn der *Primary Key* und damit die Entity-Bean-Instanz erst im Programmcode der Session-Bean bestimmt wird. Stattdessen können Entity-Beans, wie bereits bei Message-driven Beans beschrieben, von Subkomponenten verwendet werden, ohne dass die entsprechenden Verbindungen in der Adaptierbaren Komponente modelliert werden.

Komponenten-Lebenszyklus EJB-Komponenten werden aus Sicht eines Clients immer mit Hilfe des zugehörigen Home-Interfaces erzeugt. Bei Entity-Beans kann darüber auch eine bestimmte Instanz aus der Datenbank geladen werden.

Objekte von Implementierungsklassen (Bean-Klasse), die eine Komponenteninstanz repräsentieren, können für mehrere Clients wiederverwendet werden. Das bedeutet, dass der für die Komponentenentwicklung relevante Lebenszyklus einer Komponenteninstanz nicht mit der Erzeugung des Objektes durch den `new`-Operator beginnt, sondern erst nach der Zuweisung eines Kontextes (`EntityContext` oder `SessionContext`) und anschließenden Callback-Methodenaufrufen (z. B. `ejbActivate`, `ejbLoad`) durch den EJB-Container. Der Lebenszyklus endet ebenfalls nicht erst mit der Löschung des Objektes durch den Java-Garbage-Collector, sondern mit dem Aufruf von Callback-Methoden (z. B. `ejbPassivate()`, `ejbRemove()`) durch den EJB-Container.

Bei der Umsetzung von Adaptierbaren Komponenten dürfen daher Initialisierungsoperationen erst in den dafür vorgesehenen Callback-Methoden ausgeführt werden und nicht bereits im Konstruktor der Bean-Klasse.

Komponentenarchiv Ein oder mehrere EJB-Komponenten werden zusammen mit mindestens einem Komponentendeskriptor[3] (`ejb-jar.xml`) in einem jar-Archiv gebündelt. Beim Laufzeit- und Startzeit-PSM werden alle EJB-Komponenten einer Adaptierbaren Komponente in einem solchen Archiv gebündelt. Beim Installationszeit-PSM können EJB-Komponenten weggelassen werden, die bei der ausgewählten Konfiguration nicht benötigt werden.

[3] Weitere proprietäre Deskriptoren können durch Installationswerkzeuge eines EJB-Servers hinzugefügt werden.

Die Bibliothek der Laufzeitunterstützung muss entweder zu allen Archiven hinzugefügt werden oder der EJB-Server unterstützt die Erweiterung durch Bibliotheken (wie z. B. JBoss), die anschließend für alle Komponenten verfügbar sind.

5.1.2 Laufzeitunterstützung für EJB

In diesem Abschnitt wird ein Gesamtüberblick über die Implementierungsunterstützung von Adaptierbaren Komponenten für die Komponentenplattform EJB gegeben, nachdem im vorhergehenden Abschnitt bereits verschiedene Details der Umsetzung beschrieben wurden.

Aufgrund der ermittelten Eigenschaften von EJB und der in Unterabschnitt 4.2.2 beschriebenen Abbildung von Modellkonzepten ergibt sich die folgende Umsetzung:

- Zusammengesetzte Komponenten: Eine Stateful-Session-Bean wird als Platzhalter für eine Adaptierbare Komponente benutzt und verwaltet eine Liste mit allen EJB-Komponenten, die als Subkomponenten definiert wurden. Auf die Subkomponenteninstanzen darf von EJBs außerhalb der Adaptierbaren Komponente nicht zugegriffen werden (Programmierrichtlinie), insbesondere auch nicht mit Hilfe von JNDI.

- Subkomponenten: Jede Subkomponente und jeder Glue-Code wird je nach Bedarf als Session-Bean oder Entity-Bean implementiert. Subkomponenten mit mindestens einem *required*-Port müssen als Stateful-Session-Bean realisiert werden.

- Komponentenparameter: Alle Komponentenparameter werden durch Zugriffsmethoden in der Komponentenschnittstelle emuliert.

- Komponentenports: Das Local- oder Remote-Interface einer EJB-Komponente ist gleichzeitig ein *provided*-Port. Mehrfach-*provided*-Ports werden durch mehrere implementierte Schnittstellen und explizite Typ-Casts emuliert. *Required*-Ports werden durch spezielle Zugriffsmethoden in der Schnittstelle der EJB emuliert.

- Verbindungen: Verbindungen werden durch die Übergabe einer Referenz auf den *provided*-Port an eine spezielle Methode des *required*-Ports hergestellt. Alle nicht explizit definierten Verbindungen zwischen Komponenten werden durch eine Programmierrichtlinie verboten.

- Adaptions- und Aspektoperatoren: Die Operatoren können nicht durch Änderungen des Bytecodes von Komponenten zur Laufzeit und durch generische Adapter realisiert werden. Alle anderen Implementierungsvarianten (siehe Unterabschnitt 4.3.5) können angewendet werden.

- Rekonfiguration: Da gleichzeitige Methodenaufrufe bei EJB-Komponenten durch den Container verhindert werden, werden keine Interceptoren benötigt. Der Adaptionsmanager initialisiert und ändert Verbindungen im Rahmen einer Rekonfiguration.

Programmierrichtlinien Jede EJB-Komponente, die als Subkomponente innerhalb einer Adaptierbaren Komponente verwendet werden soll, muss die folgenden Voraussetzungen erfüllen:

- Für jeden *required*-Port existiert eine typkompatible Membervariable sowie zugehörige *getter*- und *setter*-Methoden in der Implementierungsklasse der EJB-Komponente. Die

setter-Methode ist darüber hinaus auch im Remote- bzw. Local-Interface enthalten. Für jeden Zugriff auf einen *required*-Port muss entweder die Membervariable oder die zugehörige *getter*-Methode verwendet werden. Die Variable darf innerhalb der Komponente, außer in der *setter*-Methode, nicht geändert werden.

- Im Programmcode einer EJB-Komponente darf nicht, außer in den im vorhergehenden Punkt erwähnten Zugriffsmethoden, mit Hilfe des Namensdiensten (JNDI) auf andere Komponenten der Adaptierbaren Komponente zugegriffen und nicht mit Hilfe von Home-Interfaces neue Komponenten erzeugt werden. Ausgenommen davon bleibt die Erzeugung von Hilfsobjekten wie z. B. Strings oder Maps.

- Stateless-Session-Beans dürfen nur für Subkomponenten ohne *required*-Port verwendet werden, da Referenzen andernfalls nicht gespeichert werden können.

- Werte für alle Komponentenparameter müssen als Argumente einer speziellen `create`-Methode im Home-Interface bei der Komponentenerzeugung angegeben werden.

5.1.3 Alternative Laufzeitunterstützung für EJB

Im Rahmen einer früheren Arbeit [Nes03, GN04] wurde eine alternative Implementierungsstrategie von Adaptierbaren Komponenten für EJB verfolgt. Anstatt fehlende Konzepte einer Komponentenplattform durch Emulation zu unterstützen, wurde dort die Komponentenplattform – im konkreten Fall EJB – entsprechend erweitert. Zu diesem Zweck wurde mit zusammengesetzten Komponenten (*composite component*) eine neue Art von EJB-Komponente eingeführt.

Die Erweiterungen wurden im Open-Source-EJB-Server JBoss [FR03] prototypisch implementiert. Die neue Komponentenart *Composite-EJB* kann eine Menge von anderen EJB-Komponenten als Subkomponenten enthalten, wird aber von außen betrachtet wie eine Stateful-Session-Bean behandelt. Damit können andere EJB-Komponenten in gewohnter Weise auf die neue Komponentenart zugreifen. Der entsprechend angepasste Applikationsserver JBoss sorgt dafür, dass das Prinzip der Kapselung (siehe Unterabschnitt 3.3.1) eingehalten wird, damit Komponenten außerhalb einer Composite-EJB nicht mehr direkt auf Subkomponenten zugreifen können. Da solche Zugriffe technisch verhindert werden, ist eine Programmierrichtlinie zur Durchsetzung nicht mehr erforderlich.

Der EJB-Deskriptor (`ejb-jar.xml`) wurde um Elemente zur Definition der neuen Komponentenart erweitert. Damit kann auch die Verknüpfung von internen und externen Port spezifiziert werden, wenn auch nur statisch. Zu Laufzeit veränderbare Verbindungen wurden mit Hilfe von Interceptoren implementiert, die Methodenaufrufe je nach aktueller Konfiguration an unterschiedliche Ziel-Komponenten weiterleiten.

Es ist denkbar, durch die Erweiterung des Prototypen auch alle weiteren Konzepte von Adaptierbaren Komponenten zu unterstützen, so dass auf eine Emulation und Programmierrichtlinien verzichtet werden kann. Dieser Ansatz besitzt aber eine Reihe von Nachteilen, die dazu geführt haben, dass er nicht weiter verfolgt wurde:

- Die Erweiterungen von EJB sind nur durch die Erweiterung bzw. Änderung eines Open-Source-Applikationsserver möglich (z. B. JBoss). Bei kommerziellen Applikationsservern

sind die Änderungen entweder rechtlich nicht erlaubt oder technisch wegen der Nichtverfügbarkeit von Quelltexten nicht möglich. Eine vollständige Neuentwicklung eines EJB-Servers ist durch den enormen Entwicklungsaufwand nicht praktikabel.

- Durch die Erweiterungen sind damit entwickelte EJB-Komponenten nicht mehr auf anderen EJB-Servern lauffähig. Adaptierbare Komponenten können damit nicht ohne weiteres in existierende Anwendungen integriert werden. Ein wesentliches Ziel der vorliegenden Arbeit könnte damit nicht erfüllt werden.

- Wenn alle Konzepte von Adaptierbaren Komponenten direkt durch den Applikationsserver und damit die Komponentenplattform unterstützt werden sollen, sind weit reichende Änderungen am EJB-Modell erforderlich (z. B. zum Verstellen von Verbindungen zwischen Komponenten), so dass eine Abwärtskompatibilität zu standardkonformen EJB-Komponenten nicht mehr gewährleistet ist.

5.1.4 Implementierung der Kryptographiekomponente mit EJB

Die Laufzeitunterstützung für EJB wurde durch die Erweiterung des Java-Frameworks (siehe Abschnitt 4.3) realisiert. Abbildung 5.1 zeigt die Implementierung der Kryptographiekomponente aus Abschnitt 3.2 in Form eines UML-Klassendiagramms[4]. Alle als Subkomponenten und Glue-Code fungierenden EJB-Komponenten (Aes, Deflate, Gluecode_1) wurden als Stateful-Session-Beans implementiert, die ein Local-Interface zur Kommunikation verwenden. Entsprechend existiert für jede dieser EJB-Komponenten auch ein LocalHome-Interface. Local- und Home-Interfaces können mit Hilfe von XDoclet [xdo05] aus der jeweiligen Bean-Klasse generiert werden.

Die EJB-Komponente „Crypto“ repräsentiert die Adaptierbare Komponente. Sie besitzt ein Remote-Interface, damit auch entfernte Clients darauf zugreifen können. Mit der `create`-Methode des zugehörigen Home-Interfaces kann ein Client eine Instanz erzeugen und dabei auch die gewünschten Parameterwerte zur Auswahl des Verschlüsselungs- und Kompressionsalgorithmus angeben. Mit entsprechenden setter-Methoden kann diese Auswahl zur Laufzeit auch wieder geändert werden, sofern das Laufzeit-PSM verwendet wurde.

Alle EJB-Komponenten der Kryptographiekomponente werden in einer `jar`-Datei gebündelt, die auch einen gemeinsamen Komponentendeskriptor `ejb-jar.xml` enthält. Die Bibliotheken der Laufzeitbibliothek können ebenfalls zur `jar`-Datei hinzugefügt. Im Falls des für die Implementierung verwendeten EJB-Servers JBoss [FR03] wurden sie jedoch global für alle EJB-Komponenten verfügbar gemacht.

5.2 Unterstützung von JavaBeans

Die Komponentenplattform JavaBeans wurde zusammen mit der ersten Version der Java-Programmiersprache entwickelt und ist in die Standard-Klassenbibliothek integriert. Ursprünglich waren JavaBeans hauptsächlich für die komponentenbasierte Entwicklung von graphischen Benutzerschnittstellen konzipiert, sie können aber auch für beliebige Anwendungen

[4] Die Subkomponenten Bzip, Blowfish und Twofish zusammen mit dem Adaptionsoperator wurden aus Platzgründen weggelassen.

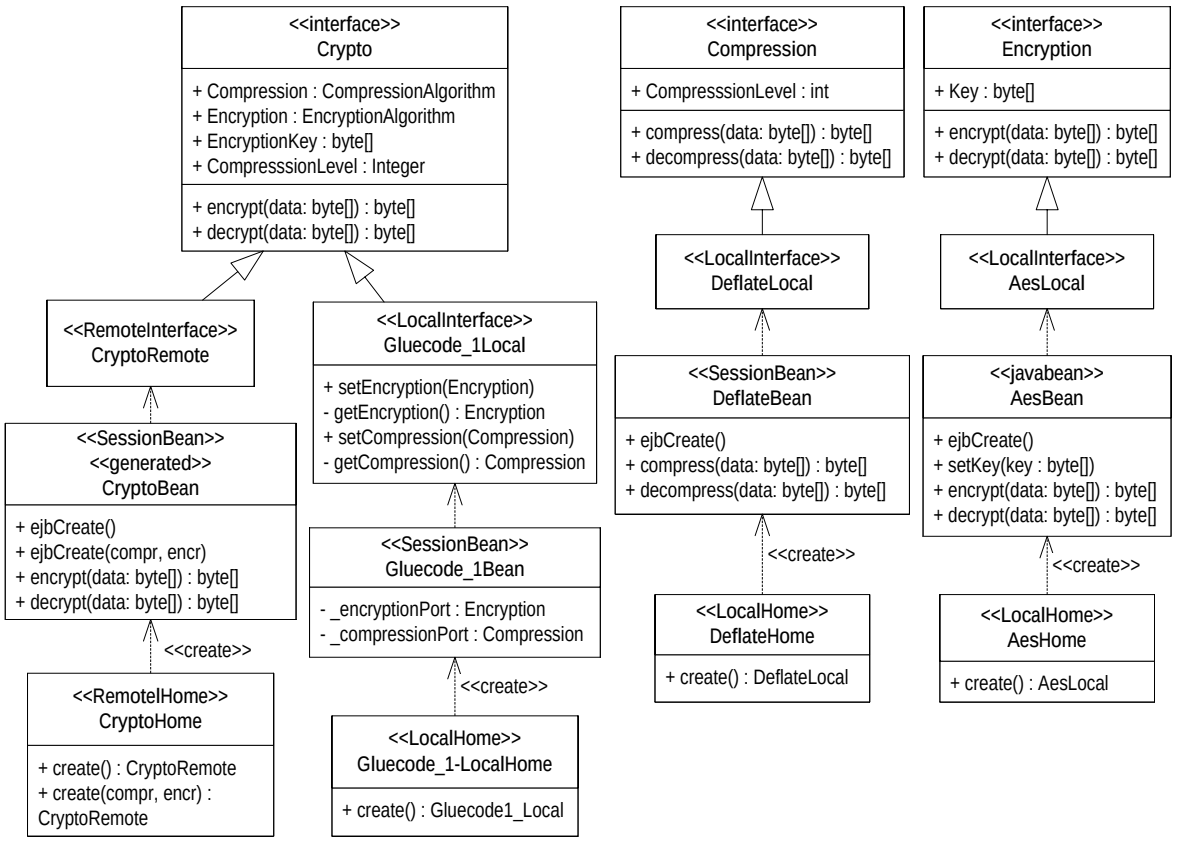

Abbildung 5.1: Implementierung der Kryptographiekomponente mit EJB

eingesetzt werden. Aufgrund der weiten Verbreitung wurde JavaBeans als zweite Plattform für die Unterstützung von Adaptierbaren Komponenten ausgewählt.

5.2.1 Analyse der Eigenschaften der Komponentenplattform JavaBeans

Verglichen mit anderen Komponentenplattformen wie EJB ist JavaBeans relativ leichtgewichtig. Komponenten, ebenfalls als JavaBeans bezeichnet, benötigen keine spezielle Laufzeitumgebung und müssen sich nur an einige wenige Konventionen halten. Das Komponentenmodell ist sehr eng an die objektorientierte Programmierung mit Java geknüpft. Im einfachsten Fall erfüllt jede Java-Klasse, die einen Default-Konstruktor[5] implementiert, die Anforderungen an eine JavaBean-Komponente. Mit Hilfe von optionalen Klassen aus dem `java.beans`-Paket der Java-Klassenbibliothek können weitere Metainformationen speziell für die Anwendungsentwicklung mit JavaBeans in Entwicklungsumgebungen (*Bean Editor*) definiert werden.

Die Eigenschaften von JavaBeans entsprechend der Kriterien aus Unterabschnitt 4.2.1 sind in Tabelle 5.2 dargestellt. Im Folgenden werden einige ausgewählte Eigenschaften zusammen mit ihnen Auswirkungen auf Implementierungsentscheidungen näher erläutert.

[5]Der Default-Konstruktor erzeugt mit Hilfe des `new`-Operators ein Objekt der jeweiligen Java-Klasse ohne die Übergabe von Parametern.

Merkmal		Unterstützung
E–1	Zusammengesetzte Komponenten	nein
E–2	Variable Port-Verknüpfung	—
E–3	Trennung Implementierung und Schnittstelle	nein
E–4	Komponentenparameter	ja
E–7	*required*-Ports	nein
E–8	Mehrfach *required*-Ports	nein
E–9	Mehrere *provided*-Ports	nein
E–10	Explizite Verbindungen	nein
E–11	Änderbare explizite Verbindungen	nein
E–12	Nur explizite Verbindungen	nein
E–13	Synchrone Kommunikation	ja
E–14	Zustandsbehaftete Komponenten	ja
E–15	MOP für Komponenteneigenschaften	ja, eingeschränkt
E–16	Verhinderung gleichzeitiger Methodenaufrufe	nein
E–17	Dynamische Proxies und Reflection	ja
E–18	Bytecode-Modifikation	ja

Tabelle 5.2: Ermittlung der Eigenschaften von JavaBeans

***Required*-Ports** JavaBeans unterstützen keine *required*-Ports in der Art, wie sie bei Adaptierbaren Komponenten vorgesehen sind. Es gibt allerdings ein Verfahren zum synchronen Versenden von Events zwischen JavaBeans (siehe Kapitel 6 in [Ham97]), das durch eine Kombination von Schnittstellen, Namenskonventionen für Methoden und die Verwendung von normalen Java-Methodenaufrufen realisiert wird. Eine JavaBean kann prinzipiell verschiedene Events erzeugen und wird dann als Event-Quelle bezeichnet. Mit Hilfe von bestimmten Methoden in der Komponentenschnittstelle der Event-Quelle (`add`*Eventname*`Listener`, `remove`*Eventname*`Listener`) können sich andere JavaBeans für diese Events registrieren. Dazu müssen sie ein entsprechendes *Listener*-Interface implementieren. Wenn ein Event erzeugt wird, werden Methoden dieses *Listener*-Interfaces von der Event-Quelle aufgerufen. Auf diese Weise wird der Event an alle registrierten JavaBeans verteilt.

Für Verbindungen, die zur Verteilung von Events verwendet werden, werden Event-Quellen als optionaler Mehrfach-*required*-Port (siehe Unterunterabschnitt 3.3.1.1) und Listener als *provided*-Port aufgefasst und direkt auf die entsprechenden JavaBeans-Mechanismen abgebildet.

Für alle anderen Verbindungen wird ein *required*-Port wie in Unterunterabschnitt 4.2.2.4 beschrieben durch eine spezielle Methode in der Komponentenschnittstelle emuliert. Über diese Methode kann eine Referenz auf einen *provided*-Port gesetzt werden, um damit eine Verbindung zwischen zwei Ports zu realisieren. Innerhalb der Komponente wird die Referenz des *required*-Ports in einer Membervariable gespeichert.

***Provided*-Ports** JavaBeans unterstützen im Allgemeinen nur einen *provided*-Port, der durch die Implementierungsklasse selbst definiert wird und nicht durch eine extra Schnittstelle (Ei-

genschaft E-3 nicht erfüllt). Es ist allerdings möglich, dass eine JavaBean verschiedene Java-Interfaces implementiert bzw. andere JavaBeans durch eine Vererbungsbeziehung erweitert. Mit Hilfe eines expliziten Typ-Casts kann dann eine andere Sicht auf die JavaBean ausgewählt werden, die einem anderen *provided*-Port entspricht. Die JavaBean-Spezifikation schreibt allerdings vor (siehe Abschnitt 2.10 in [Ham97]), dass die Typ-Casts nicht mit dem Java-Typ-Cast-Operator durchgeführt werden dürfen, sondern mit der Methode `Beans.getInstanceOf`. Diese Vorschrift entstand, da ursprünglich geplant war, dass auch mehrere Java-Klassen zu einer JavaBean zusammengefasst werden können. Für die Umsetzung von mehreren *provided*-Ports pro Komponente können dennoch die Verfahren aus Unterunterabschnitt 4.2.2.4 angewendet werden.

Komponentenparameter Parameter werden bei JavaBeans in Form von *Properties* durch ein Paar von Zugriffsmethoden (*getter* und *setter*) unterstützt. Die Kennzeichnung dieser *Properties* erfolgt entweder durch Namenskonventionen in der JavaBean (`get`*PropertyName*`()` und `set`*PropertyName*`(...)`, siehe Kapitel 8 in [Ham97]) oder durch zusätzliche Deskriptor-Klassen (`BeanInfo` und `PropertyDescriptor`). Beide Verfahren eignen sich zur direkten Umsetzung des Parameterkonzeptes von Adaptierbaren Komponenten.

JavaBeans unterstützen drei Arten von *Properties* (siehe Kapitel 7 in [Ham97]):

- **Beliebige Properties** können alle Java-Klassen inklusive der primitiven Datentypen als Typ verwenden.
- **Indexed Properties** unterstützen einen endlichen Bereich von Werten, die über einen Index vom Typ `int` festgelegt werden. Sie entsprechen damit Enumeration-Parametern bei Adaptierbaren Komponenten.
- **Bound and Constrained Properties** können andere JavaBeans mit Hilfe eines `PropertyChangeEvent` über Änderungen einer Property informieren bzw. anderen JavaBeans die Gelegenheit bieten, die Änderung abzulehnen (`PropertyVetoException`). Um diese Events zu erhalten, müssen die daran interessierten JavaBeans die Schnittstellen `PropertyChangeListener` bzw. `VetoableChangeListener` implementieren und sich für die Events registrieren.

Trennung von Implementierung und Schnittstellen JavaBeans erzwingt keine Trennung von Implementierung und Schnittstellen. Stattdessen wird die Schnittstelle einer JavaBean implizit aus allen öffentlichen Methoden (`public`) in der Implementierungsklasse definiert. Da es aber nicht verboten ist, zusätzliche Java-Interfaces in der Implementierungsklasse zu implementieren, kann die Trennung von Schnittstelle und Implementierung durch eine Programmierrichtlinie erzwungen werden.

Rekonfiguration Da JavaBeans keine Mechanismen zur Verhinderung von gleichzeitigen (*reentrant*) Methodenaufrufen unterstützt (Eigenschaft E-16 nicht erfüllt), müssen Interceptoren zusammen mit einem Adaptionsmanager den Rekonfigurationsprozess im Laufzeit-PSM steuern.

Verbotene Operationen Die JavaBeans-Spezifikation enthält keine Einschränkungen für die Verwendung von bestimmten Operationen.

MOP JavaBeans unterstützen nur lesenden Zugriff auf Komponenteneigenschaften (bezeichnet als *Introspection*, siehe Kapitel 8 in [Ham97]), Änderungen können jedoch nicht vorgenommen werden. Daher sind die Voraussetzungen für ein MOP nicht erfüllt.

Komponenten-Lebenszyklus Die JavaBean-Spezifikation definiert keinen speziellen Lebenszyklus für JavaBeans. Der Lebenszyklus entspricht daher dem von beliebigen Java-Objekten: JavaBeans werden mit dem Java-`new`-Operator erzeugt und initialisiert. Wenn keine anderen Objekte mehr Referenzen auf JavaBeans-Objekte besitzen, werden sie vom Java-Garbage-Collector automatisch entfernt.

Komponentenarchiv Ein oder mehrere (`Class`-Dateien von) JavaBeans werden in einem `jar`-Archiv zusammengepackt. Dabei muss für jedes enthaltene JavaBean die Implementierungsklasse in der so genannten Manifest-Datei des Archivs (`META-INF/MANIFEST.MF`, siehe Abschnitt 11.5 in [Ham97]) angegeben werden. Zusätzlich können auch Abhängigkeiten zu anderen JavaBeans bzw. Bibliotheken als weitere Metainformationen spezifiziert werden.

Als Konsequenz sollten alle JavaBeans, welche die Adaptierbare Komponente und die enthaltenen Subkomponenten repräsentieren, in einem `jar`-Archiv gebündelt werden. Die Abhängigkeit zur Bibliothek der Laufzeitunterstützung und alle Implementierungsklasse der JavaBeans müssen in der Manifest-Datei des Archivs definiert werden.

5.2.2 Laufzeitunterstützung für JavaBeans

In diesem Abschnitt wird ein Gesamtüberblick über die Implementierungsunterstützung von Adaptierbaren Komponenten auf die Komponentenplattform JavaBeans gegeben, nachdem im vorhergehenden Abschnitt bereits verschiedene Details der Umsetzung beschrieben wurden.

Aufgrund der ermittelten Eigenschaften von JavaBeans und der in Unterabschnitt 4.2.2 beschriebenen Abbildung von Modellkonzepten ergibt sich die folgende Umsetzung:

- Zusammengesetzte Komponenten: Eine JavaBean wird als Platzhalter für eine Adaptierbare Komponente benutzt und verwaltet eine Liste mit allen JavaBeans, die als Subkomponenten definiert wurden. Auf die Subkomponenteninstanzen darf von JavaBeans außerhalb der Adaptierbaren Komponente nicht zugegriffen werden (Programmierrichtlinie).

- Subkomponenten: Jede Subkomponente und jeder Glue-Code wird als JavaBean implementiert.

- Komponentenparameter: Alle Komponentenparameter außer vom Typ Enumeration werden auf Properties von JavaBeans abgebildet. Enumeration-Parameter werden auf Indexed-Properties von JavaBeans abgebildet.

- Komponentenports: Die Schnittstelle einer JavaBean ist gleichzeitig ihr *provided*-Port. Mehrfach-*provided*-Ports werden durch mehrere implementierte Schnittstellen und explizite Typ-Casts emuliert. *Required*-Ports werden durch spezielle Zugriffsmethoden in der Schnittstelle der JavaBean emuliert, ähnlich wie das Event-Konzept von JavaBeans.

- Verbindungen: Alle nicht explizit definierten Verbindungen zwischen Komponenten werden durch eine Programmierrichtlinie verboten. Verbindungen zwischen Subkomponenten werden von der JavaBean, die als Platzhalter für eine Adaptierbare Komponente fungiert, hergestellt. Dazu wird eine Referenz auf die Schnittstelle des *provided*-Ports als Parameter beim Methodenaufruf der speziellen Zugriffsmethode des *required*-Ports übergeben. Die Referenz berücksichtigt auch eventuell vorhandene Interceptoren bzw. Adaptions- und Aspektoperatoren.

- Adaptions- und Aspektoperatoren: Alle Implementierungsvarianten (siehe Unterabschnitt 4.3.5) können angewendet werden.

- Rekonfiguration: Alle JavaBeans müssen mindestens ein Java-Interface implementieren, dass die Schnittstelle (*provided*-Port) definiert, um eine Trennung von Schnittstelle und Implementierung zu erreichen und Interceptoren einsetzen zu können. Der Adaptionsmanager initialisiert und ändert Verbindungen im Rahmen einer Rekonfiguration.

Programmierrichtlinien Jede JavaBean, die als Subkomponente innerhalb einer Adaptierbaren Komponente verwendet werden soll, muss die folgenden Voraussetzungen erfüllen:

- Jede JavaBean-Komponente implementiert mindestens ein Java-Interface, dass die funktionale Schnittstelle definiert. Mehrere unterschiedliche implementierte Interfaces werden als verschiedene *provided*-Ports aufgefasst. Die Auswahl eines bestimmten *provided*-Ports erfolgt mit Hilfe eines Typ-Casts durch `Beans.getInstanceOf`.

- Im Programmcode einer JavaBean-Komponente dürfen keine neuen Komponenten erzeugt und keine Referenzen auf andere Komponenten über andere Mechanismen (z. B. Namensdienst) außer *required*-Ports erlangt werden. Ausgenommen davon bleibt die Erzeugung von Hilfsobjekten wie z. B. Strings oder Maps.

- Für jeden *required*-Port existiert eine typkompatible Membervariable sowie zugehörige *getter*- und *setter*-Methoden in der JavaBean. Für jeden Zugriff auf einen *required*-Port muss entweder die Membervariable oder die zugehörige *getter*-Methode verwendet werden. Die Variable darf innerhalb der Komponente, außer in der *setter*-Methode, nicht geändert werden.

- Im Programmcode einer JavaBean-Komponente dürfen keine neuen Threads erzeugt werden. Ebenso dürfen keine Funktionen aufgerufen werden, die die Ausführung von Methoden auf unbestimmte Zeit blockieren (z. B. `wait`).

- Jede JavaBeans-Klasse kann eine `init`-Methode implementieren. Diese Methode wird dann im Rahmen des Initialisierungsprozesses nach dem Setzen aller Komponentenparameter aufgerufen.

- Bei der Verwendung einer Adaptierbaren Komponente muss nach dem Setzen aller Komponentenparameter zwingend die `init`-Methode aufgerufen werden, bevor ein anderer Methodenaufruf erfolgt.

5.2.3 Implementierung der Kryptographiekomponente mit JavaBeans

Die Laufzeitunterstützung für JavaBeans wurde durch die Erweiterung des Java-Frameworks (siehe Abschnitt 4.3) realisiert. Abbildung 5.1 zeigt die Implementierung der Kryptographiekomponente aus Abschnitt 3.2 in Form eines UML-Klassendiagramms[6]. Entsprechend der Programmierrichtlinie implementieren alle JavaBeans eine Schnittstelle, die alle public-Methoden der Bean-Klasse enthält. Auch der Glue-Code wird durch eine JavaBean realisiert.

Die JavaBean-Komponente „Crypto“ repräsentiert die Adaptierbare Komponente. Eine Instanz wird mit dem `new`-Operator erzeugt. Anschießend können mit den setter-Methoden die Parameterwerte zur Auswahl des Verschlüsselungs- und Kompressionsalgorithmus definiert werden, bevor die Initialisierung entsprechend der Programmierrichtlinie mit dem Aufruf der `init`-Methode abgeschlossen wird. Mit den setter-Methoden kann die Auswahl zur Laufzeit auch wieder geändert werden, sofern das Laufzeit-PSM verwendet wurde.

Alle JavaBean-Komponenten der Kryptographiekomponente werden in einer `jar`-Datei gebündelt, die auch einen gemeinsamen Komponentendeskriptor `MANIFEST.MF` enthält, der alle JavaBeans auflistet. Die Bibliotheken der Laufzeitbibliothek werden ebenfalls zur `jar`-Datei hinzugefügt.

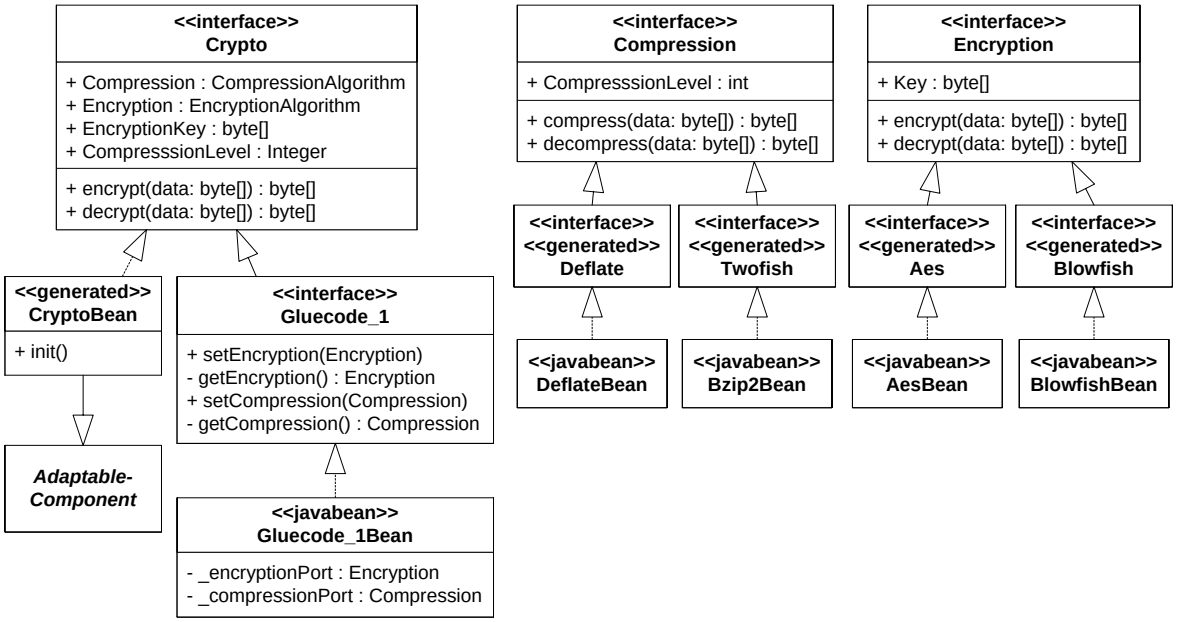

Abbildung 5.2: Implementierung der Kryptographiekomponente mit JavaBeans

5.3 Unterstützung von Microsoft COM

Die Komponentenplattform COM (Component Object Model [Rog97]) wurde von Microsoft speziell für die Entwicklung von Anwendungen auf den Windows-Betriebssystemen konzipiert und definiert einen Industriestandard für binäre Software-Komponenten. Weitere Microsoft

[6]Die Subkomponente Twofish zusammen mit dem Adaptionsoperator wurden aus Platzgründen weggelassen.

Merkmal		Unterstützung
E–1	Zusammengesetzte Komponenten	ja, eingeschränkt
E–2	Variable Port-Verknüpfung	ja
E–3	Trennung Implementierung und Schnittstelle	ja
E–4	Komponentenparameter	nein
E–7	*required*-Ports	nein
E–8	Mehrfach *required*-Ports	—
E–9	Mehrere *provided*-Ports	ja
E–10	Explizite Verbindungen	nein
E–11	Änderbare explizite Verbindungen	nein
E–12	Nur explizite Verbindungen	nein
E–13	Synchrone Kommunikation	ja
E–14	Zustandsbehaftete Komponenten	ja
E–15	MOP für Komponenteneigenschaften	nein
E–16	Verhinderung gleichzeitiger Methodenaufrufe	ja
E–17	Dynamische Proxies und Reflection	nein
E–18	Bytecode-Modifikation	nein

Tabelle 5.3: Ermittlung der Eigenschaften von COM

Technologien wie OLE, ActiveX, COM+ und DCOM bauen darauf auf. COM wird außerdem in vielen Windows-Anwendungen eingesetzt. COM orientiert sich stark an C++, kann aber prinzipiell auch mit anderen Programmiersprachen wie z. B. Visual Basic verwendet werden. Die Komponentenentwicklung erfolgt aber vorwiegend mit C++. Viele der Konzepte von COM wurden auch in die neue Komponentenplattform .NET [Mic05] übernommen.

COM wurde aufgrund der weiten Verbreitung und als ein Vertreter einer Komponentenplattform ausgewählt, die auf der etwas älteren objektorientierten Programmiersprache C++ basiert. C++ unterstützt im Gegensatz zu Java oder C# weder Reflection noch automatische Garbage-Collection.

5.3.1 Analyse der Eigenschaften von COM

Die Eigenschaften von COM entsprechend der Kriterien aus Unterabschnitt 4.2.1 sind in Tabelle 5.3 dargestellt. Im Folgenden werden einige ausgewählte Eigenschaften zusammen mit ihnen Auswirkungen auf Implementierungsentscheidungen näher erläutert.

***Provided*-Ports** Eine COM-Komponente stellt ihre Funktionalität über ein oder mehrere Schnittstellen zur Verfügung. Jede Schnittstelle ist durch einen weltweit eindeutigen Bezeichner – der so genannten *Interface ID* (IID) – gekennzeichnet. Jede COM-Komponente implementiert die Schnittstelle `IUnknown`, die Methoden zur Navigation zu allen weiteren Schnittstellen der COM-Komponente mit Hilfe ihrer IIDs (`QueryInterface`) und zur Unterstützung des Lebenszyklus durch Referenzzählung (`AddRef` und `Release`) enthält. Alle anderen Schnittstellen müssen direkt oder indirekt von `IUnknown` abgeleitet sein.

Durch diese Eigenschaften können mehrere *provided*-Ports direkt auf unterschiedliche Schnittstellen einer COM-Komponente abgebildet werden. Die IID einer COM-Schnittstelle sollte als Name für *provided*-Ports verwendet werden.

Alle Methoden von COM-Schnittstellen geben immer einen Wert vom Typ `HRESULT` zurück, mit dem Fehler beim Methodenaufruf an den Client gemeldet werden, als ein Ersatz für fehlende Exceptions. Die eigentlichen Rückgabeparameter einer Methode müssen daher über Zeiger realisiert werden, z. B. `int*` für einen `int`-Rückgabeparameter.

Zusammengesetzte Komponenten COM bietet eine konzeptionelle Unterstützung für zusammengesetzte Komponenten in Form von *Containment* und *Aggregation*[7] an. Bei diesen Techniken wird aber nicht die Bündelung von mehreren Komponenten in einer zusammengesetzten Komponente unterstützt, sondern es handelt sich um Techniken zur Wiederverwendung bzw. Erweiterung von existierenden Komponenten. Eine zusammengesetzte COM-Komponente (oft als *outer component* bezeichnet) greift dabei zur Implementierung der Funktionalität auf eine oder mehrere andere COM-Komponenten (*inner components*) zurück und ist für die Erzeugung bzw. Entfernung der Komponenteninstanzen verantwortlich. Die Komponentenplattform bietet aber keine direkte Unterstützung zur Implementierung dieser Konzepte.

Eine Adaptierbare Komponente kann damit als eine COM-Komponente umgesetzt werden, die eine Menge von Subkomponenten ebenfalls in Form von COM-Komponenten als *Containment* enthält.

COM-Komponenten und Windows-Registry Durch die enge Verzahnung der Komponentenplattform COM mit den Windows-Betriebssystemen spielt die Windows-Registry eine wesentliche Rolle. Alle Informationen zu COM-Komponenten und COM-Schnittstellen werden dort zentral gespeichert. Daher müssen COM-Komponenten vor der ersten Benutzung installiert werden, um diese Informationen in der Registry abzuspeichern.

Komponententypen werden durch Klassen-Bezeichner (*class ID* oder *CLSID*) identifiziert, die eine Form von weltweit eindeutigen Bezeichnern (*Globally Unique Identifiers, GUID*) darstellen. In ähnlicher Weise erhalten auch alle Schnittstellen einen eindeutigen Bezeichner (*IID*). Die COM-Bibliotheken verwenden die Informationen aus der Windows-Registry zur Erzeugung von COM-Komponenteninstanzen zur Laufzeit.

Reflection Durch die Verwendung von C++ als Programmiersprache stehen Reflection und dynamische Proxies nicht zur Verfügung. Es gibt zwar auch für C++ Ansätze zur Unterstützung von Reflection (z. B. OpenC++ [Chi95]), allerdings sind dazu Änderungen zur Übersetzungszeit erforderlich (Compile-time MOP), die nicht mit dem Blackbox Ansatz von binären COM-Komponenten kompatibel sind. Aus diesem Grund muss auf die Verwendung von Reflection verzichtet werden.

Trennung von Implementierung und Schnittstellen COM schreibt eine Trennung von Implementierung und Schnittstellen von Komponenten vor (Eigenschaft E-3 erfüllt). Schnittstel-

[7] *Aggregation* ist ein Spezialfall von *Containment*, bei dem eine Schnittstelle einer inneren Komponente von der äußeren Komponente implementiert wird und Methodenaufrufe an die innere Komponente delegiert werden.

len sind dabei sogar binär-kompatibel. Sie werden optional mit MIDL (*Microsoft Interface Definition Language* [Rog97]) definiert.

Komponentenarten und Komponentenarchiv COM unterstützt nur eine Art von Komponenten, die allerdings auf unterschiedliche Arten implementiert und ausgeführt werden kann. Dabei wird nach zwei orthogonalen Kriterien klassifiziert: der Art des Komponentenarchivs und der Art des so genannten *Apartment*-Modells.

Ein oder mehrere COM-Komponenten können entweder innerhalb einer EXE- oder einer DLL-Datei[8] gebündelt werden. Die EXE- oder DLL-Dateien sind damit das Komponentenarchiv von COM. Komponenten innerhalb einer EXE-Datei werden immer in einem eigenständigen Prozess ausgeführt (*Out-of-Process Server*). Client-Aufrufe bei solchen Komponenten werden automatisch vom COM-Laufzeitsystem durch Proxy-Objekte, Marshalling und Unmarshalling von Aufrufdaten und Interprozess-Kommunikation abgewickelt, allerdings mit den damit verbundenen negativen Auswirkungen auf die Performance. Komponenten innerhalb von DLL-Dateien können dagegen direkt in einen bestehenden Prozess geladen und ausgeführt werden (*In-Process Server*). Damit sind auch direkte Methodenaufrufe ohne die Zwischenschaltung von Proxy-Objekten möglich[9].

Unabhängig von der Art des Komponentenarchivs läuft jede COM-Komponenteninstanz und auch jeder Client-Thread immer in genau einem von drei möglichen COM-*Apartments* ab, dass bei der Initialisierung festgelegt wird und die Eigenschaften in Bezug auf Anwendungen mit mehreren Threads definiert:

- Im *Single Threaded Apartment* (STA) werden alle Methodenaufrufe bei darin ablaufenden COM-Komponenten von genau einem Thread durchgeführt. Gleichzeitige Methodenaufrufe können damit nicht auftreten und müssen bei der Programmierung auch nicht berücksichtigt werden. Methodenaufrufe von Clients oder anderen COM-Komponenten, die im gleichen Prozess aber in einem anderen Thread ablaufen, werden vom COM-Laufzeitsystem serialisiert[10].

- Im *Multiple Threaded Apartment* (MTA) können beliebige Threads auf COM-Komponenten zugreifen. Gleichzeitige Methodenaufrufe und damit verbundene Synchronisationsmaßnahmen müssen dadurch bei der Programmierung berücksichtigt werden.

- Das *Thread Neutral Apartment* (TNA), dass erst mit COM+ eingeführt wurde, verbindet die positiven Eigenschaften von STA und MTA miteinander. Wie beim MTA können beliebige Threads auf COM-Komponenten zugreifen. Allerdings kann zu jeder Zeit immer nur ein Thread das TNA betreten und Methoden aufrufen. Gleichzeitige Methodenaufrufe sind damit wie beim STA nicht möglich, aber das zeitaufwändige Marshalling durch Proxies entfällt.

Innerhalb eines Prozesses kann es beliebig viele STA aber nur maximal ein MTA bzw. TNA geben. Nur Methodenaufrufe von Threads innerhalb des gleichen Apartments werden direkt

[8] EXE-Dateien sind ausführbare Programme (*executables*) und DLL-Dateien (*Dynamic Link Library*) Code-Bibliotheken unter Microsoft Windows.

[9] Sofern die COM-Komponente und der Client-Thread innerhalb des gleichen *Apartments* ablaufen.

[10] Methodenaufrufe werden mit Hilfe von Nachrichten an ein unsichtbares Fenster innerhalb des STA gesendet und dort vom Thread des STA durch die zugehörige Nachrichtenschlage (*message loop*) der Fenster-Prozedur empfangen und schließlich an die COM-Komponente weitergeleitet.

ausgeführt. Alle Methodenaufrufe von Threads aus anderen Apartments werden dagegen vom COM-Laufzeitsystem über Proxies ausgeführt, was mit entsprechenden Leistungseinbußen verbunden ist.

Für Adaptierbare Komponenten und ihre Subkomponenten können prinzipiell alle verschiedenen Komponentenarchive und Apartments verwendetet werden. Zur Steigerung der Performance ist es allerdings sehr empfehlenswert, wenn ausschließlich DLL-Dateien und das gleiche Apartment für alle Komponenten verwendet werden.

Verhinderung gleichzeitiger Methodenaufrufe Das COM-Laufzeitsystem verhindert für COM-Komponenten innerhalb des STA und TNA gleichzeitige Methodenaufrufe durch unterschiedliche Threads. Im MTA sind sie dagegen möglich und müssen berücksichtigt werden.

Lebenszyklus Jeder Thread einer Anwendung, der COM-Komponenten benutzen will, muss zuerst mit Hilfe der Funktion `CoInitialize` bzw. `CoInitializeEx` das COM-Laufzeitsystem initialisieren. Dabei wird auch festgelegt, zu welchem Apartment (STA, MTA oder TNA) dieser Thread gehört.

COM-Komponenteninstanzen werden immer von einer *ClassFactory* durch Angabe der CLSID erzeugt. Komponentenarchive in Form von EXE- bzw. DLL-Dateien enthalten mindestens eine *ClassFactory*, die das Interface `IClassFactory` implementiert und beim Laden registriert wird[11]. Da COM kein automatisches Garbage Collection von nicht mehr benötigten Komponenteninstanzen unterstützt, muss jeder Client die `IUnknown`-Methoden `AddRef` und `Release` aufrufen, um damit eine Referenzzählung zu ermöglichen. Alle reservierten und verwendeten Ressourcen, insbesondere Speicher, müssen beim Entfernen einer Komponenteninstanz explizit freigeben werden. Dies ist der Fall, wenn der Referenzzähler den Wert Null erreicht.

Die COM-Komponente, die als Stellvertreter für die Adaptierbare Komponente fungiert, erzeugt alle COM-Komponenten, die Subkomponenten repräsentieren.

Rekonfiguration Die Implementierung der Unterstützung von Rekonfigurationen hängt vom *Apartment*-Modell der COM-Komponenten ab, die für die Adaptierbare Komponente eingesetzt werden. Wenn alle COM-Komponenten das STA oder TNA verwenden, kann auf die Verwendung von Interceptoren verzichtet werden. Da Rekonfigurationen durch eine Parameteränderung ausgelöst werden, kann der entsprechende Thread auch sofort die Rekonfiguration ausführen, ohne andere Threads, wie in Abschnitt 3.5 beschrieben, mit Hilfe von Interceptoren berücksichtigen zu müssen.

Bei der Verwendung des MTA müssen Interceptoren zur Synchronisation der Rekonfiguration eingesetzt werden. Für die Implementierung der Interceptoren können nur generierte Adapter verwendet werden, die auf das Verfahren zur Verwaltung aktiver Threads (siehe Unterabschnitt 4.3.6) zurückgreifen, da C++ Reflection, die Modifikation von Bytecode und den direkten Zugriff auf den Stack nicht erlaubt.

[11] Bei DLLs wird die Methode `DllGetClassObject` und bei EXE-Dateien die Methode `CoRegisterClassObject` verwendet.

5.3.2 Laufzeitunterstützung für COM

In diesem Abschnitt wird ein Gesamtüberblick über die Implementierungsunterstützung von Adaptierbaren Komponenten auf die Komponentenplattform COM gegeben, nachdem im vorhergehenden Abschnitt bereits verschiedene Details der Umsetzung beschrieben wurden.

Aufgrund der ermittelten Eigenschaften von COM und der in Unterabschnitt 4.2.2 beschriebenen Abbildung von Modellkonzepten ergibt sich die folgende Umsetzung:

- Zusammengesetzte Komponenten: Eine COM-Komponente wird als Platzhalter für eine Adaptierbare Komponente benutzt und verwaltet eine Liste mit allen COM-Komponenten, die als Subkomponenten definiert wurden. Auf die Subkomponenteninstanzen kann von COM-Komponenten außerhalb der Adaptierbaren Komponente nicht zugegriffen werden (*Containment*).
- Subkomponenten: Jede Subkomponente und jeder Glue-Code wird als COM-Komponente implementiert.
- Komponentenparameter: Alle Komponentenparameter werden durch spezielle Zugriffsmethoden in der Komponentenschnittstelle realisiert.
- Komponentenports: Jede COM-Komponente unterstützt das Interface `IUnknown`, über das mit Hilfe der Methode `QueryInterface` auf alle anderen Schnittstellen der *provided*-Ports zugegriffen werden kann. *Required*-Ports werden durch spezielle Zugriffsmethoden in der Komponentenschnittstelle realisiert.
- Verbindungen: Alle nicht explizit definierten Verbindungen zwischen Komponenten werden durch eine Programmierrichtlinie verboten. Verbindungen zwischen Subkomponenten werden von der COM-Komponente, die als Platzhalter für eine Adaptierbare Komponente fungiert, hergestellt. Dazu wird eine Referenz auf die Schnittstelle des *provided*-Ports als Parameter beim Methodenaufruf der speziellen Zugriffsmethode des *required*-Ports übergeben. Die Referenz berücksichtigt auch eventuell vorhandene Interceptoren bzw. Adaptions- und Aspektoperatoren.
- Adaptions- und Aspektoperatoren: Nur die Erzeugung von speziellen Adaptern kann angewendet werden. Die Erzeugung bzw. Änderung von Quelltext ist nicht praktikabel, da ein C++-Compiler auf vielen Rechnern nicht verfügbar ist. Für die direkte Erzeugung und Änderung des Binärcodes von COM-Komponenten sind dem Autor keine existierenden Verfahren bekannt.
- Rekonfiguration: Der Adaptionsmanager initialisiert und ändert Verbindungen im Rahmen einer Rekonfiguration. Nur bei Verwendung der MTA müssen Interceptoren zur Blockierung von Methodenaufrufen während der Rekonfiguration verwendet werden.

Programmierrichtlinien Jede COM-Komponente, die als Subkomponente innerhalb einer Adaptierbaren Komponente verwendet werden soll, muss die folgenden Voraussetzungen erfüllen:

- Die Schnittstellen aller *provided*-Ports sind von `IUnknown` abgeleitet. Jeder *provided*-Port wird als Schnittstelle von der COM-Komponente implementiert und kann über den Standardmechanismus mittels `QueryInterface` von `IUnknown` ausgewählt werden.

- Im Programmcode einer COM-Komponente dürfen keine neuen Komponenten erzeugt und keine Referenzen auf andere Komponenten über andere Mechanismen außer *required*-Ports erlangt werden. Ausgenommen davon bleibt die Erzeugung von Hilfsobjekten.
- Alle Methoden von COM-Schnittstellen verwenden `HRESULT` als Rückgabetyp zur Meldung von Fehlern während der Methodenabarbeitung.
- Für jeden *required*-Port existiert eine typkompatible Membervariable sowie zugehörige *getter*- und *setter*-Methoden in der COM-Komponente. Für jeden Zugriff auf einen *required*-Port muss entweder die Membervariable oder die zugehörige *getter*-Methode verwendet werden. Die Variable darf innerhalb der Komponente, außer in der *setter*-Methode, nicht geändert werden.
- Im Programmcode einer COM-Komponente dürfen keine Thread-Operationen (Erzeugen, Stoppen, Beenden) ausgeführt werden.
- Die Schnittstelle des Komponentenports der Adaptierbaren Komponente enthält eine `init`-Methode, die nach dem Setzen aller Komponentenparameter aufgerufen werden muss, bevor ein anderer Methodenaufruf erfolgt.
- Alle COM-Komponenten mit allen ihren Schnittstellen müssen vor der Verwendung in der Windows-Registry registriert werden.

5.3.3 Implementierung der Kryptographiekomponente mit COM

Die Implementierung der Laufzeitunterstützung für COM basiert nicht auf dem im Abschnitt 4.3 beschriebenen Framework, da C++ als Programmiersprache verwendet musste. Daher wurden für die Implementierung der Kryptographiekomponente nur die absolut notwendigen Funktionen des Frameworks nach C++ portiert. Alle Komponenten verwenden das STA, so dass auf die Verwendung von Interceptoren verzichtet werden konnten.

Abbildung 5.3 zeigt die Implementierung der Kryptographiekomponente aus Abschnitt 3.2 in Form eines UML-Klassendiagramms[12]. Die COM-Komponente `CryptoClass` repräsentiert die Adaptierbare Komponente. Eine Instanz wird mit durch die *ClassFactory* (`CryptoFactory`) erzeugt. Anschießend können mit den setter-Methoden die Parameterwerte zur Auswahl des Verschlüsselungs- und Kompressionsalgorithmus definiert werden, bevor die Initialisierung entsprechend der Programmierrichtlinie mit dem Aufruf der `init`-Methode abgeschlossen wird. Mit den setter-Methoden kann die Auswahl zur Laufzeit auch wieder geändert werden, sofern das Laufzeit-PSM verwendet wurde.

Alle Schnittstellen und Komponentenklassen erhalten eine GUID und werden in der Windows-Registry registriert. Sie werden zusammen mit einer *ClassFactory* (`CryptoFactory`) in einer DDL-Datei gebündelt. Diese DLL-Datei kann als *In-Process Server* von einem Client in den jeweiligen Prozess geladen werden.

[12] Die Subkomponente Twofish zusammen mit dem Adaptionsoperator wurden aus Platzgründen weggelassen. Alle Methoden verwenden eigentlich `HRESULT` als Rückgabetyp.

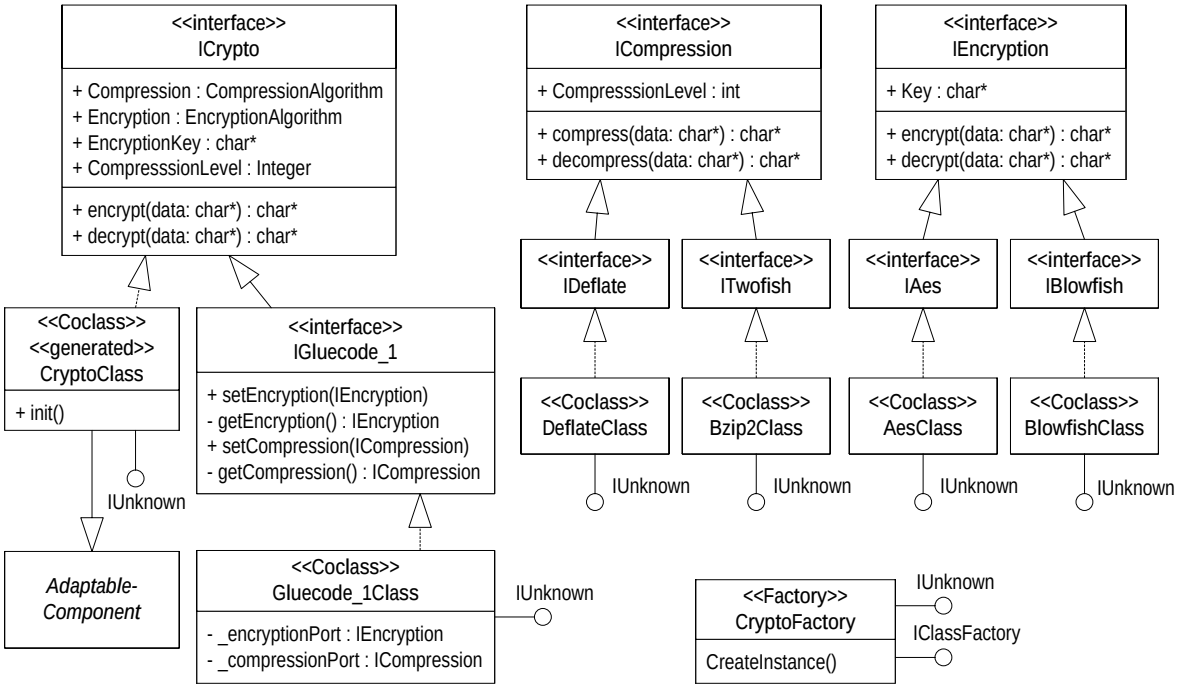

Abbildung 5.3: Implementierung der Kryptographiekomponente mit COM

5.4 Unterstützung von Web-Services

Web-Services sind im Gegensatz zu EJB, JavaBeans und COM keine Komponentenplattform, da die eigentliche Implementierung von Komponenten überhaupt nicht beschrieben wird. Sie definieren vielmehr eine Reihe von Standards zur Kommunikation und zur Beschreibung von Diensten (*Services*). Komponenten sind aber in vielen Fällen ein Weg zur Implementierung von Web-Services. Das bedeutet, dass Web-Services zur Definition von Komponentenschnittstellen und zur Durchführung der Kommunikation zwischen Komponenten verwendet werden. Aus diesen Gründen und wegen der wachsenden Bedeutung von Web-Services für die Realisierung von verteilten Anwendungen im Internet werden Web-Services in dieser Arbeit unter dem Gesichtspunkt einer Komponentenplattform betrachtet.

Ein Web-Service ist durch eine URI eindeutig identifizierbar und wird im Allgemeinen mit Hilfe von SOAP [Mit03], einem XML-basierten Kommunikationsprotokoll und Kodierungsformat für die Kommunikation zwischen verteilten Anwendungen, angesprochen. Web-Services sind unabhängig von bestimmten Programmiersprachen und Implementierungstechnologien definiert. In den folgenden Abschnitten wird daher die Umsetzung von Adaptierbaren Komponenten mit Hilfe von Web-Services ebenfalls möglichst unabhängig von Implementierungstechnologien beschrieben. Wenn das nicht möglich ist, wird ausdrücklich darauf hingewiesen.

5.4.1 Analyse der Eigenschaften von Web-Services

Die Eigenschaften von Web-Services entsprechend der Kriterien aus Unterabschnitt 4.2.1 sind in Tabelle 5.4 dargestellt. Im Folgenden werden einige ausgewählte Eigenschaften zusammen mit ihnen Auswirkungen auf Implementierungsentscheidungen näher erläutert.

Merkmal		Unterstützung
E–1	Zusammengesetzte Komponenten	nein
E–2	Variable Port-Verknüpfung	—
E–3	Trennung Implementierung und Schnittstelle	ja
E–4	Komponentenparameter	nein
E–7	*required*-Ports	nein
E–8	Mehrfach *required*-Ports	—
E–9	Mehrere *provided*-Ports	nein
E–10	Explizite Verbindungen	nein
E–11	Änderbare explizite Verbindungen	nein
E–12	Nur explizite Verbindungen	nein
E–13	Synchrone Kommunikation	ja
E–14	Zustandsbehaftete Komponenten	nein, nur mit Hilfsmitteln
E–15	MOP für Komponenteneigenschaften	nein
E–16	Verhinderung gleichzeitiger Methodenaufrufe	implementierungsspezifisch
E–17	Dynamische Proxies und Reflection	implementierungsspezifisch
E–18	Bytecode-Modifikation	implementierungsspezifisch

Tabelle 5.4: Ermittlung der Eigenschaften von Web-Services

Sitzungsunterstützung und Zustandsbehaftete Web-Services Web-Services [Web05] besitzen per Definition keinen Zustand und auch demzufolge auch keine Unterstützung für Sitzungen (*Sessions*). Alle benötigten Informationen sollen beim Aufruf eines Web-Services mitgeliefert werden. Es gibt aber dennoch viele Szenarien, bei denen diese Annahme nicht erfüllt werden kann und mehrere Web-Services-Aufrufe logisch zusammen gehören. In diesem Fall müssen logisch zusammengehörende Aufrufe entsprechend gekennzeichnet und Zustandsinformation auf Seite der Web-Services gespeichert werden. Zurzeit (2005) laufen verschiedene Bestrebungen, einen Standard für zustandsbehaftete Web-Services zu etablieren. Zwei mögliche Kandidaten für einen solchen Standard – WS-Context und WS-Coordination – werden im Folgenden kurz vorgestellt. Beiden Ansätzen ist gemein, dass sie eine Sitzung erzeugen können und diese Sitzung mit späteren Web-Service-Aufrufen in Beziehung setzen.

WS-Context [LN03b] ist ein Bestandteil des *Web-Service Composite Application Framework* (WS-CAF [LN03a]). Die beiden Spezifikationen wurden gemeinschaftlich von Arjuna, Oracle und Iona entwickelt und sollen bei OASIS (*Organization for the Advancement of Structured Information Standards*) standardisiert werden. Mit WS-Context ist es möglich, mehrere Web-Service-Nachrichten mit einander in Beziehung zu setzen. Die zusammengehörenden Nachrichten werden dabei zu einer Aktivität (*Activity*) zusammengefasst. Die verschiedenen Aktivitäten werden mit Hilfe eines *Context Services* verwaltet. Dieser spezielle Web-Service wird zum Erzeugen und Beenden von Aktivitäten verwendet.

WS-Coordination [Fei05] ist Bestandteil von WS-Transactions [WST05] und wurde in Zusammenarbeit von Arjuna, BEA, Hitachi, IBM, IONA und Microsoft entwickelt. Die Spezifikation beschreibt ein Framework für einen Koordinationsdienst, mit dem mehrere Web-Service-Aufrufe zu einer Aktivität zusammengefasst werden können, um darauf aufbauend

z. B. verteilte Transaktionen zu koordinieren. Mit Hilfe eines *Coordination-Service* wird ein *CoordinationContext* erzeugt, der als Identifikator einer Aktivität dient. Darüber hinaus wird WS-Coordination zur Koordinierung mehrerer Teilnehmer verwendet, etwa im Rahmen von Transaktionen. Mehrere Web-Services können als Teilnehmer bei einem Koordinations-Web-Service registriert werden, der ein bestimmtes Koordinationsprotokoll steuert (z. B. 2-Phase-Commit).

Da für WS-Coordination und WS-Context noch keine Open-Source-Implementierungen verfügbar sind und die Standardisierung noch nicht abgeschlossen ist (2005), wurde zur Implementierung die Sitzungsunterstützung von Apache Axis [Apa05] verwendet, die entweder HTTP Cookies oder Transportprotokoll-unabhängige SOAP-Header verwendet. Die Grundprinzipien der verschiedenen Ansätze zur Sitzungsunterstützung sind relativ ähnlich, wodurch das Verfahren später einfach ausgetauscht werden kann.

Standardmäßig wird bei Apache Axis für jeden eintreffenden Aufruf bei einem Web-Service ein Java-Objekt erzeugt, das die Abarbeitung übernimmt. Mit Hilfe der Sitzungsunterstützung (*session scope*) können mehrere Web-Service-Aufrufe zum selben Java-Objekt weitergeleitet werden. Das entsprechende Java-Objekt kann damit Zustandsinformationen über die Grenze eines Web-Service-Aufrufs hinaus zwischenspeichern. Eine Sitzung wird vom Client aus erzeugt, indem eine Sitzungs-ID als Cookie innerhalb eines HTTP-Headers oder als Eintrag im SOAP-Header zum Web-Service-Aufruf hinzugefügt wird. Bei Client-Anwendungen auf Basis der Bibliotheken von Axis steht eine entsprechende Methode (`Stub.setMaintainSession(true)`) im generierten Stub-Objekt eines Web-Services zur Verfügung, die diese Aufgabe übernimmt.

Kapselung von Subkomponenten Das Prinzip der Kapselung (siehe Unterabschnitt 3.3.1) wird bei der Umsetzung von Subkomponenten durch Web-Services nicht mehr eingehalten, da Web-Services über eine URI in der Regel immer erreichbar sind. Ausgenommen sind lediglich Zugriffsbeschränkungen innerhalb von Firmen und Organisationen durch Firewalls.

Referenzen auf Web-Services Zum Herstellen von Verbindungen zwischen Web-Services, die Verbindungen zwischen Subkomponenten bei Adaptierbaren Komponenten entsprechen, müssen Referenzen ausgetauscht werden. WS-Addressing [GH05] definiert ein Format zum Austausch von Referenzen, dass zu diesem Zweck verwendet werden kann.

***Provided*-Ports** Ein Web-Service definiert immer genau eine Zugriffsschnittstelle, die genau einem *provided*-Port entspricht. Wenn möglich sollte daher im Modell der Adaptierbaren Komponente auf die Verwendung von Komponenten mit mehreren *provided*-Ports verzichtet werden. Anderenfalls müssen die in Unterunterabschnitt 4.2.2.4 beschriebenen Verfahren zur Emulation angewendet werden.

***Required*-Ports** Web-Services unterstützen kein Konzept für *required*-Ports. Für zustandsbehaftete Web-Services können die in Unterunterabschnitt 4.2.2.4 vorgestellte Verfahren angewendet werden. Ein Verfahren für zustandslose Web-Services wird im nächsten Abschnitt beschrieben.

Komponentenparameter Web-Services unterstützen kein spezielles Konzept für Komponentenparameter. Beliebige Parameter können allerdings sowohl im SOAP-Body als auch im SOAP-Header bei einem Web-Service-Aufruf übermittelt werden. Ähnlich wie bei *required*-Ports setzt die Verwendung von Komponentenparametern aber voraus, dass Zustandsinformationen in der Komponenteninstanz bzw. dem entsprechenden Web-Service gespeichert werden können. Ein Verfahren für zustandslose Web-Services wird im nächsten Abschnitt beschrieben.

Trennung von Implementierung und Schnittstelle Web-Services erfordern per Definition immer eine Trennung von Implementierung und Schnittstelle. Die Schnittstelle wird mit Hilfe der *Web Service Description Language* (WSDL [CCMW01]) beschrieben.

Verbotene Operationen Web-Services definieren keinerlei Einschränkungen für Implementierungen.

Komponentenarten Web-Services setzen keine bestimmte Implementierungstechnik voraus und definieren daher auch keine Komponentenarten.

Lebenszyklus Web-Services definieren keinen Lebenszyklus. Ein Web-Service ist über eine bestimmte URI erreichbar und kann zu jeder Zeit aufgerufen werden. Alles weitere sind Implementierungsdetails und daher nicht Gegenstand von Web-Services.

Komponentenarchiv Web-Services definieren keinen Standard für Komponentenarchive.

Keine Unterstützung von Threads Mehrere Web-Services werden gewöhnlich auf unterschiedlichen Rechnern oder zumindest in unterschiedlichen Prozessen ausgeführt. Daher können Information über Threads nicht zur Synchronisierung der Rekonfiguration verwendet werden (siehe Abschnitt 3.5). Ähnlich wie bei der Sitzungsunterstützung kann aber auch für Threads eine eindeutige ID zusammen mit Web-Service-Aufrufen übergeben werden, z. B. im SOAP-Header, die anstelle eines Threads zur Synchronisation benutzt werden kann.

Implementierungsspezifische Eigenschaften Die Merkmale E-16, E-17 und E-18 wurden in Tabelle 5.4 als implementierungsspezifisch gekennzeichnet. Das gleiche gilt auch für alle Erläuterungseigenschaften aus Unterabschnitt 4.2.1.

5.4.2 Laufzeitunterstützung für Web-Services

Für die Umsetzung von Adaptierbaren Komponenten mit Web-Services werden zwei verschiedene Verfahren vorgestellt. Das erste Verfahren setzt eine Sitzungsunterstützung bzw. zustandsbehaftete Web-Services voraus. Das zweite Verfahren verzichtet auf diese Voraussetzung und arbeitet mit zustandslosen Web-Services.

Die beiden Verfahren besitzen jedoch eine Reihe von Gemeinsamkeiten für die Implementierung, die sich aufgrund der ermittelten Eigenschaften von Web-Services und der in Unterabschnitt 4.2.2 beschriebenen Abbildung von Modellkonzepten ergeben:

- Zusammengesetzte Komponenten: Ein Web-Service wird als Platzhalter für eine Adaptierbare Komponente benutzt und verwaltet intern eine Liste mit allen Web-Services, die als Subkomponenten definiert wurden.
- Subkomponenten: Jede Subkomponente wird als Web-Service implementiert. Glue-Code wird dagegen nicht als eigenständiger Web-Service, sondern innerhalb des Web-Services der Adaptierbaren Komponente implementiert, da er nur in diesem Kontext definiert ist.
- Adaptions- und Aspektoperatoren: Die möglichen Implementierungsvarianten (siehe Unterabschnitt 4.3.5) hängen von der verwendeten Implementierungstechnologie für die Web-Services ab.

5.4.2.1 Umsetzung ohne Unterstützung von Sitzungen

Ohne Unterstützung von Sitzungen können zwei Aufrufe eines Clients bei einem Web-Service nicht logisch zusammengefasst werden. Dadurch kann der Web-Service auch keine Zustandsinformationen wie etwa aktuelle Parameterwerte oder die aktuelle Konfiguration speichern, außer die Informationen gelten anschließend für alle weiteren Aufrufer des Web-Services. Um die Konzepte von Adaptierbaren Komponenten dennoch nutzen zu können, wird ein geändertes Modell für Web-Service-Aufrufe benötigt. Dieses Modell muss die folgenden Anforderungen erfüllen:

- Ein Web-Service-Aufruf enthält alle notwendigen Informationen für die Ausführung.
- Web-Services speichern keine Zustandsinformationen zwischen zwei Web-Service-Aufrufen.

Diese Forderungen können nur erfüllt werden, wenn nicht mehr zwischen der Erzeugung, der Definition von Parametern und der Nutzung von Adaptierbaren Komponenten unterschieden wird. Stattdessen werden diese drei Schritte zu einem Schritt zusammengefasst. Das bedeutet, dass erst beim Aufruf eines Web-Services, der eine Adaptierbare Komponente repräsentiert, die zur Abarbeitung notwendige Konfiguration aus Subkomponenten erzeugt wird. Dazu ist es erforderlich, dass bei jedem Web-Service-Aufruf auch die Werte für Komponentenparameter mit übertragen werden. Dies kann innerhalb des SOAP-Headers erfolgen.

Ein Web-Service-Aufruf wird nach diesem Verfahren in folgenden Schritten durchgeführt:

- Ein Web-Service-Client ruft einen Web-Service auf und überträgt dabei Komponentenparameter im SOAP-Header.
- Der Web-Service, als Stellvertreter für die Adaptierbare Komponente, wertet die empfangenen Parameter aus und ermittelt die Konfiguration zur Abarbeitung dieses Aufrufs.
- Der Aufruf wird entsprechend der ermittelten Konfiguration an einen bestimmten Web-Service weitergeleitet, der eine Subkomponente repräsentiert. Im SOAP-Header werden dabei wiederum Werte für Parameter bzw. *required*-Ports übermittelt.

Der Nachteil des Verfahrens besteht darin, dass bei jedem Web-Service-Aufruf die Konfiguration entsprechend der definierten Parameterabbildung (siehe Unterabschnitt 3.4.6) ermittelt

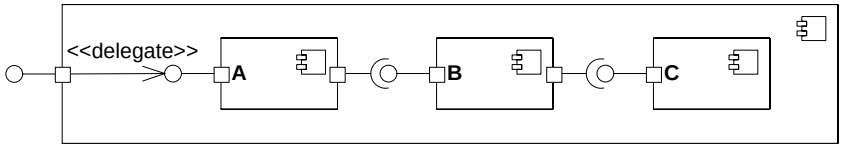

Abbildung 5.4: Problem: Verkettung von mehreren Subkomponenten mit *required*-Ports

werden muss. Die dafür benötigte Zeit hängt linear von der Anzahl der Regel in der Definition der Parameterabbildung ab. Verglichen mit den Zeiten, die für das Erstellen, die Übertragung und dem Parsen von SOAP-Nachrichten bei einem Web-Service-Aufruf benötigt werden, kann die Zeit für die Bestimmung der aktiven Konfiguration vernachlässigt werden.

Ein weiteres Problem ergibt sich, wenn man eine Konfiguration wie in Abbildung 5.4 betrachtet. Dabei stellt sich die Frage, wie die Subkomponente B erfährt, mit welchem Port ihr *required*-Port momentan verbunden ist, wenn alle Komponenten durch zustandslose Web-Services umgesetzt werden sollen. Für Subkomponente A kann der *required*-Port durch die Übertragung von Informationen im SOAP-Header verknüpft werden, da der Aufruf direkt von der Adaptierbaren Komponente erfolgt. Subkomponente B wird dagegen nur indirekt über Subkomponente A aufgerufen. Es ist nicht möglich, dass die Adaptierbare Komponente bzw. ihr stellvertretender Web-Service die Verbindungen bereits im Vorfeld eines Methodenaufrufes herstellt. Folgende Umsetzungsstrategien sind aber möglich:

- Bei einem Web-Service-Aufruf bei einer Subkomponente werden nicht nur die für sie zutreffenden Parameter und *required*-Ports im SOAP-Header übertragen, sondern auch für alle Subkomponenten, die aufgrund der aktuellen Konfiguration indirekt aufgerufen werden. Angewandt auf das Beispiel aus Abbildung 5.4, müssen beim Methodenaufruf bei Subkomponente A auch Definitionen für Parameter und *required*-Ports von Subkomponente B mit übertragen werden.

- Bei allen Subkomponenten wird auf die Verwendung von Komponentenparametern und *required*-Ports verzichtet. Dadurch werden allerdings die möglichen Modelle Adaptierbarer Komponenten eingeschränkt.

Damit ergeben sich bei der fehlenden Unterstützung von Sitzungen die folgenden Umsetzungen der Implementierung, zusätzlich zu den bereits beschriebenen generellen Umsetzungen bei Web-Services:

- Komponentenparameter: Alle Komponentenparameter, denen ein Wert zugewiesen werden soll, werden im SOAP-Header von allen Web-Service-Aufrufen übertragen.

- Komponentenports: Die Schnittstelle eines Web-Services ist gleichzeitig sein *provided*-Port. Mehrfach-*provided*-Ports werden nicht unterstützt. *Required*-Ports werden als spezielle Komponentenparameter aufgefasst. Referenzen auf andere Web-Services[13] werden zusammen mit anderen Komponentenparametern im SOAP-Header übertragen, wie im vorhergehenden Punkt beschrieben.

[13] in Form von URIs bzw. *Service Endpoint References* unter Nutzung von WS-Addressing [GH05]

- Verbindungen: Verbindungen werden nur temporär im Rahmen eines Web-Service-Aufrufs durch die Übertragung von Referenzen im SOAP-Header hergestellt. Die beteiligten Web-Services bleiben sonst nur lose gekoppelt (*loose coupling*).
- Rekonfiguration: Eine Unterstützung für Rekonfiguration ist nicht notwendig, da sowieso potentiell jeder Web-Service-Aufruf durch eine andere Konfiguration abgearbeitet wird.

5.4.2.2 Umsetzung mit Unterstützung von Sitzungen

Wenn Sitzungen von Web-Services unterstützt werden, erfolgt die Umsetzung von Adaptierbaren Komponenten ähnlich wie bei anderen Komponentenplattformen. Unabhängig vom verwendeten PSM (siehe Abschnitt 4.1), werden die folgenden Schritte durchgeführt:

1. Ein Web-Service-Client erzeugt eine Sitzung mit einem Web-Service, der eine Adaptierbare Komponente repräsentiert. Die Details für die Erzeugung dieser Sitzung hängen von dem verwendeten Verfahren ab und wurden im vorhergehenden Abschnitt diskutiert. Als Ergebnis erhält man auf jeden Fall eine Sitzungs-ID, die bei weiteren Web-Service-Aufrufen verwendet werden muss.
2. Der Client setzt alle Parameter beim Web-Service durch den Aufruf einer entsprechenden Methode[14]. Der Web-Service ermittelt die aktuelle Konfiguration und erzeugt Sitzungen mit allen Web-Services, die Subkomponenten der Adaptierbaren Komponente repräsentieren.

Damit ergeben sich bei der Unterstützung von Sitzungen die folgenden Umsetzungen der Implementierung, zusätzlich zu den bereits beschriebenen generellen Umsetzungen bei Web-Services:

- Komponentenparameter: Alle Komponentenparameter werden durch eine spezielle Methode in der Schnittstelle des Web-Service definiert. Diese Methode akzeptiert als einzigen Parameter eine Struktur, die Parameterwerte für alle möglichen Parameter des Web-Services enthält. Auf diese Weise können mit einem Web-Service-Aufruf alle Parameter gesetzt werden, was die Anzahl der Aufrufe minimiert und eher dem grobgranularen Prinzip von Web-Services entspricht.
- Komponentenports: Die Schnittstelle eines Web-Services ist gleichzeitig der *provided*-Port. Mehrfach-*provided*-Ports werden nicht unterstützt. *Required*-Ports werden als spezielle Komponentenparameter aufgefasst. Referenzen auf andere Web-Services werden zusammen mit anderen Komponentenparametern durch eine spezielle Zugriffsmethode definiert, wie im vorhergehenden Punkt beschrieben.
- Verbindungen: Alle nicht explizit definierten Verbindungen zwischen Komponenten werden durch eine Programmierrichtlinie verboten. Verbindungen zwischen Subkomponenten werden von der Implementierung des Web-Services, die als Platzhalter für eine Adaptierbare Komponente fungiert, hergestellt.

[14] Dieser Schritt entfällt beim Installationszeit-PSM, da dort bereits alle Parameter bei der Installation definiert wurden.

- Rekonfigurationen: Der Adaptionsmanager als Teil der Implementierung des Web-Services der Adaptierbaren Komponente initialisiert und ändert Verbindungen im Rahmen einer Rekonfiguration.

5.4.3 Implementierung der Kryptographiekomponente mit Web-Services

Die Laufzeitunterstützung für Web-Services wurde durch die Erweiterung des Java-Frameworks (siehe Abschnitt 4.3) und der Verwendung von Apache Axis [Apa05] realisiert. Abbildung 5.1 zeigt die Implementierung der Kryptographiekomponente aus Abschnitt 3.2 in Form eines UML-Klassendiagramms[15]. Für alle Web-Services wurde die Sitzungsunterstützung von Axis verwendet und dementsprechend die Umsetzung aus dem vorhergehenden Abschnitt realisiert. Alle Komponentenschnittstellen wurden mit WSDL definiert, das mit dem Werkzeug `java2wsdl`[16] aus entsprechenden Java-Schnittstellen erzeugt wurden.

Der Web-Service `Crypto` repräsentiert die Adaptierbare Komponente. Eine Instanz wird implizit durch einen Web-Service-Aufruf erzeugt. Anschießend können mit den setter-Methoden die Parameterwerte zur Auswahl des Verschlüsselungs- und Kompressionsalgorithmus definiert werden, bevor die Initialisierung entsprechend der Programmierrichtlinie mit dem Aufruf der `init`-Methode abgeschlossen wird. Mit den setter-Methoden kann die Auswahl zur Laufzeit auch wieder geändert werden, sofern das Laufzeit-PSM verwendet wurde.

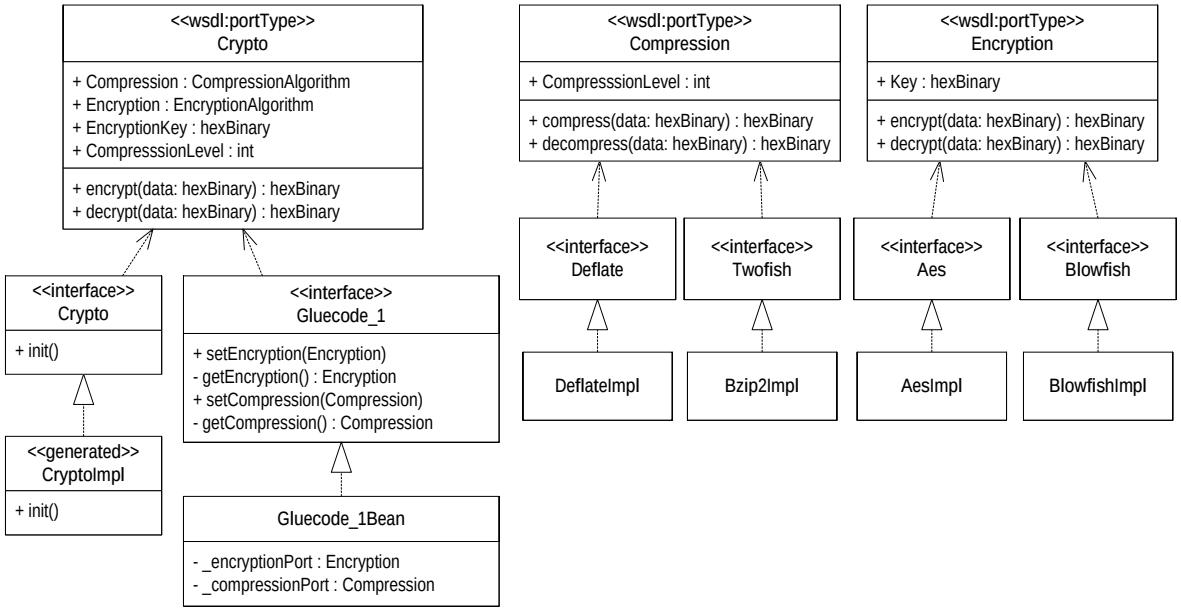

Abbildung 5.5: Implementierung der Kryptographiekomponente mit Web-Services

[15] Die Subkomponente Twofish zusammen mit dem Adaptionsoperator wurden aus Platzgründen weggelassen.
[16] `java2wsdl` ist in Apache Axis enthalten.

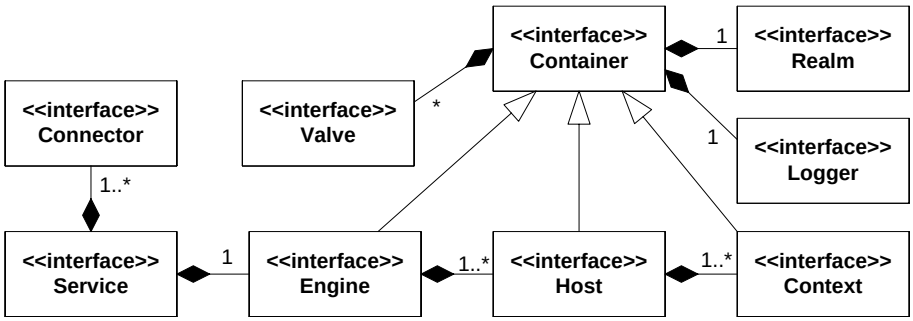

Abbildung 5.6: UML-Klassendiagramm der Schnittstellen von Tomcat

5.5 Fallstudie: Servlet-Container Apache Tomcat

Apache Tomcat 5.0 [Tom05] ist ein Servlet-Container, der die Servlet-Spezifikation in der Version 2.4 [Ser03] und die *JavaServer Pages*-Spezifikation (JSP [JSP03]) in der Version 2.0 unterstützt. Sun verwendet Tomcat als Referenzimplementierung für die Servlet- und JSP-Spezifikationen und verdeutlicht damit seine Bedeutung. Mit Hilfe einer umfangreichen Konfigurationsdatei können zahlreiche Anpassungen und Einstellungen bei Tomcat vorgenommen werden. Viele Einstellungen wirken sich auch auf die interne Struktur aus. Aus diesem Grund wurde Tomcat als Fallstudie ausgewählt, um die Beschreibungsmöglichkeiten von Adaptierbaren Komponenten zu überprüfen, wenn Tomcat selbst als eine Adaptierbare Komponente betrachtet wird.

Im Folgenden werden zuerst die Konfigurationsmöglichkeiten und die interne Architektur von Tomcat analysiert. Im Anschluss daran wird Tomcat als Adaptierbare Komponente beschrieben. Zum Abschluss wird dieser Ansatz mit der bestehenden Implementierung von Tomcat verglichen.

5.5.1 Analyse von Tomcat

Tomcat ist intern komponentenorientiert aufgebaut. Die Komponenten werden durch eine kleine Anzahl von Java-Interfaces beschrieben, wie in Abbildung 5.6 illustriert. Für eine bestimmte Schnittstelle existieren teilweise unterschiedliche Implementierungen mit speziellen Eigenschaften. Die Auswahl der Implementierungen und die Kombination der Komponenten legen die Laufzeiteigenschaften von Tomcat fest.

Die wesentlichen Komponenten-Schnittstellen von Tomcat werden im Folgenden kurz vorgestellt:

- Ein `Server` repräsentiert den kompletten Servlet-Container, der mehrere `Service`s enthält. Neben der Funktionalität zum Starten und Beenden des Containers speichert er auch globale Eigenschaften (*Global Naming Resources*).
- Ein `Service` koppelt ein oder mehrere `Connector`s mit genau einer `Engine`, besitzt sonst aber keine Funktionalität. Er definiert damit, auf welchen Wegen HTTP-Requests empfangen und wie sie anschließend verarbeitet werden.
- Eine `Engine` implementiert die Logik zur Abarbeitung von Anfragen (*Requests*), die

über einen `Connector` empfangen wurden und nach der Verarbeitung durch einen `Host` auch wieder darüber an den Client zurückgeschickt werden.

- Ein `Connector` implementiert die Logik zur Kommunikation mit Clients. Tomcat unterstützt wahlweise den Empfang von HTTP-Requests über einen integrierten Web-Server oder die Integration in einen eigenständigen Web-Server wie Apache oder den Microsoft Internet Information Server, insbesondere zur Steigerung der Performance.

- Ein `Host` stellt eine Verbindung zu einem bestimmten Netzwerk-Namen her. Eine `Engine` kann durch die Definition mehrerer `Host`s Anfragen für unterschiedliche Host-Namen verarbeiten. Dieses Konzept der *virtual Hosts* wird im Internet häufig eingesetzt, um mehrere unterschiedliche Web-Adressen (`www.firma1.com`, `www.firma2.com`) von einem physischen Web-Server bereitzustellen.

- Ein `Context` repräsentiert eine bestimmte Web-Anwendung, die innerhalb eines `Host`s abläuft. Eine Web-Anwendung wird entsprechend der Servlet-Spezifikation in Form einer `war`-Datei gebündelt und enthält Deskriptoren (z. B. `web.xml`), die zur Konfiguration des `Context`es verwendet werden. Web-Anwendungen selbst sind nicht mehr Teil von Tomcat, sondern werden nur von ihm ausgeführt.

- Ein `Logger` ist für die Verarbeitung von Status-, Fehler- und Debugging-Meldungen verantwortlich, die innerhalb einer zugeordneten `Engine`, `Host` oder `Context` auftreten. Der `Logger` entscheidet, in welcher Form und ob diese Meldungen gespeichert oder ausgegeben werden.

- Ein `Realm` stellt Authentifizierungsinformationen in Form einer Datenbank mit Nutzernamen, zugeordneten Rollen und Passwörtern bereit, die einer `Engine`, einem `Host` oder einem `Context` zugeordnet werden. Zur Laufzeit werden die Informationen zur Authentifizierung von Client-Anfragen verwendet (*Container Managed Security*), so dass nur bestimmte Nutzer auf entsprechende Web-Anwendungen zugreifen dürfen.

- Ein `Valve` repräsentiert eine Komponente innerhalb der Pipeline zur Verarbeitung von Anfragen, die einer `Engine`, einem `Host` oder einem `Context` zugeordnet ist. Damit wird das *Chain-of-Responsibility*-Muster [GHJV95] umgesetzt. `Valve`s können auf alle Informationen einer Anfrage (HTTP-Request) zugreifen und auch Änderungen daran vornehmen (z. B. Protokollierung oder Filterung).

Für die Schnittstellen `Server`, `Service`, `Engine`, `Host` und `Context` bringt Tomcat nur jeweils genau eine Standardimplementierung (z. B. `StandardServer` und `StandardService`) mit. Für alle anderen Schnittstellen enthält Tabelle 5.5 eine Liste von Implementierungen jeweils mit einer kurzen Beschreibung der Funktionalität.

Tomcat verwendet eine umfangreiche XML-Konfigurationsdatei (`server.xml`, siehe Listing 5.1), die vor dem Start definiert werden muss. Der Aufbau der Konfigurationsdatei orientiert sich eng an der Software-Architektur von Tomcat. Das äußert sich darin, dass für alle bereits vorgestellten Java-Interfaces ein gleichnamiges XML-Element existiert. Diese XML-Elemente verwenden das Attribut `classname`[17] zur Auswahl der Implementierungsklasse der

[17] Das Attribut ist außer bei `Realm`, `Valve` und `Logger` optional. Die Standard-Implementierungsklasse wird ausgewählt, wenn es nicht definiert wurde.

Schnittstelle	Implementierung	Eigenschaft
Connector	CoyoteConnector	Verbindet `Engine` mit integriertem Web-Server
	Ajp13Connector	Verbindet `Engine` mit Web-Server über AJP-Protokoll (*Apache Jserv Protocol* [Mil00])
Logger	FileLogger	Schreibt Meldungen in Datei
	SystemErrLogger	Schreibt Meldungen in Standard-Ausgabe-Stream
	SystemOutLogger	Schreibt Meldungen in Standard-Error-Stream
Realm	DataSourceRealm	Lädt Nutzer, Rollen und Passwörter aus relationaler Datenbank mittels JNDI-Ressource
	JDBCRealm	Lädt Nutzer, Rollen und Passwörter aus relationaler Datenbank mittels JDBC-Verbindung
	JNDIRealm	Lädt Nutzer, Rollen und Passwörter aus LDAP-Verzeichnis
	MemoryRealm	Lädt Nutzer, Rollen und Passwörter aus XML-Datei und speichert die Daten als Java-Objekte
Valve	AccessLogValve	Protokolliert Informationen zu HTTP-Requests in einer Datei
	RequestDumperValve	Protokolliert vollständige HTTP-Requests (Debugging)
	RemoteAddrValve	Filtert bestimme IP-Adressen bei HTTP-Requests
	RemoteHostValve	Filtert bestimmte Host-Namen bei HTTP-Requests
	SingleSingOn	Verwendet gemeinsame Authentifizierung für alle Web-Anwendungen eines `Host`s

Tabelle 5.5: Implementierungsklassen von Tomcat-Schnittstellen

Komponente. Tomcat wertet dieses Attribut zur Laufzeit aus und instanziiert mit Hilfe von Java-Reflection Objekte der entsprechenden Klassen.

Weitere mögliche Attribute hängen zum einen vom Namen des XML-Elements (z. B. `Host`) und zum anderen von der durch das `classname`-Attribut ausgewählten Implementierungsklasse ab. Generell werden mit diesen Attributen Parameter der jeweiligen Komponenten festgelegt, z. B. der verwendete TCP-Port eines `Connector`s oder der Name eines `Host`s (siehe Tabelle 5.6). Für ein bestimmtes XML-Element ist dabei jeweils eine Menge von Attributen definiert, die von allen Implementierungsklassen unterstützt werden. Eine bestimmte Implementierungsklasse kann zusätzlich weitere Attribute auswerten. Alle durch Attribute definierten Parameter werden nach dem Prinzip von Java-*Properties* direkt von den zugeordneten Implementierungsklassen bzw. Komponenten unterstützt. Diese Vorgehensweise untermauert die enge Kopplung von Konfigurationsdatei und Implementierungsdetails.

Zusätzlich zu den XML-Elementen, die sich direkt auf die Architektur von Tomcat aus-

Listing 5.1: Beispiel für die Tomcat-Konfigurationsdatei server.xml

```
<Server port="8005" shutdown="SHUTDOWN">
  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
      description="User database that can be updated and saved">
    </Resource>
    <ResourceParams name="UserDatabase">
      <parameter>
        <name>factory</name>
        <value>org.apache.catalina.users.MemoryUserDatabaseFactory</value>
      </parameter>
    </ResourceParams>
  </GlobalNamingResources>

  <Service name="Catalina">
    <Connector port="8080" />

    <Engine name="Catalina" defaultHost="localhost">
      <Logger className="org.apache.catalina.logger.FileLogger" />

      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase" />

      <Host name="localhost" appBase="webapps" />
    </Engine>
  </Service>
</Server>
```

wirken, gibt es noch XML-Elemente zur Definition von komplexen Parametern[18] für einzelne Komponenten. Mit dem `GlobalNamingResources`-Element und darin enthaltenen `Resource`, `Environment` und `ResourceParams`-Elementen können verschiedene globale JNDI-Ressourcen definiert werden. Mit dem `DefaultContext`-Element können Einstellungen für alle Web-Anwendungen zugeordneter `Engine`s und `Host`s vorgenommen werden, z. B. ob Cookies für die Sitzungsverwaltung verwendet werden oder ob Web-Anwendungen bei Änderungen automatisch neu geladen werden.

Zusätzlich zur Konfigurationsdatei `server.xml` unterstützt Tomcat zur Laufzeit auch Konfigurationsänderungen mit Hilfe von JMX (*Java Management Extensions* [JMX02]). Die einzelnen Implementierungsklassen der Tomcat-Schnittstellen stellen dazu bestimmte Attribute und Operationen zur Verfügung. JMX ermöglicht lesenden und schreibenden Zugriff über verschiedene Protokolle, z. B. über eine integrierte Management-Web-Anwendung von Tomcat.

5.5.2 Tomcat als Adaptierbare Komponente

Tomcat besitzt eine Reihe von Eigenschaften, um ihn als eine komplexe Komponente zu betrachten und im Speziellen als eine Adaptierbare Komponente:

- Tomcat kann in Anwendungen integriert werden, um die Funktionalität eines Servlet-Containers zur Verfügung zu stellen. Beispielsweise basiert die Web-Service-Implementierung Axis [Apa05] auf Tomcat. Anwendungen, die Schnittstellen in Form von Web-

[18] Damit sind Parameter gemeint, die nicht als ein einzelnes Attribut eines XML-Elementes definiert werden können.

Komponente	Parameter	Beschreibung
Server	port	TCP-Port für Kommando zum Beenden des Servers
	shutdown	Kommando-String zum Beenden des Servers
Service	name	Logischer Name des Services
	debug	Debug-Level für Meldungen an den `Logger`
Engine	defaultHost	Name des Standard-Hosts zur Bearbeitung von Anfragen
	name	Logischer Name der Engine
	debug	Debug-Level für Meldungen an den `Logger`
Host	appBase	Verzeichnis für Web-Anwendungen dieses Hosts
	name	Logischer Name des Hosts
	autodeploy	Automatische Installation von Web-Anwendungen
	debug	Debug-Level für Meldungen an den `Logger`
Coyote-Connector	port	TCP-Port für HTTP-Requests
	compression	GZip-Kompression von Daten ein-/ausschalten
AjpConnector	port	TCP-Port zur Kommunikation mit dem Web-Server
FileLogger	directory	Verzeichnis zur Speicherung von Log-Dateien
	prefix	Prefix für alle Log-Dateien
Remote-AddrValve	allow	Liste mit erlaubten Hostnamen für HTTP-Requests
	deny	Liste mit verbotenen Hostnamen für HTTP-Requests
Remote-HostValve	allow	Liste mit erlaubten IP-Adressen für HTTP-Requests
	deny	Liste mit verbotenen IP-Adressen für HTTP-Requests

Tabelle 5.6: Ausgewählte Parameter von Tomcat-Komponenten

Services anbieten wollen, können somit die Tomcat-Komponente zusammen mit Axis integrieren.

- Die Implementierungsdetails von Tomcat sind für die meisten Anwender nicht von Interesse. Eine Blackbox-Sicht ist im Allgemeinen ausreichend. Zum Verständnis der Konfigurationsdatei `server.xml` ist gegenwärtig aber zumindest ein Grundwissen über die Architektur von Tomcat erforderlich.
- Parameter, die gegenwärtig in der Konfigurationsdatei `server.xml` zusammengefasst sind, können in einer ähnlichen Weise auch als Komponentenparameter zur Konfiguration verwendet werden.

Für die Umwandlung von Tomcat in eine Adaptierbare Komponente müssen die möglichen Konfigurationen mit den Modellierungsverfahren aus Abschnitt 3.4 beschrieben werden. Dazu soll möglichst die bisherige Architektur übernommen werden. Anschließend müssen Komponentenparameter als Ersatz für die Konfigurationsdatei `server.xml` festgelegt werden, die sich möglichst an der damit erreichten Funktionalität orientieren. Im letzten Schritt müssen Parameterwerte auf die eingangs definierten Konfigurationen abgebildet werden.

5.5.2.1 Beschreibung der Konfigurationen von Tomcat

Die interne Architektur von Tomcat (siehe Abbildung 5.6) zusammen mit den bereitgestellten Komponentenimplementierungen (siehe Tabelle 5.5) legt den Rahmen für die möglichen Konfigurationen fest. Wenn Tomcat als Adaptierbare Komponente betrachtet wird, besteht das Ziel nicht darin, alle theoretisch möglichen Konfigurationen zu unterstützen, sondern lediglich alle *sinnvollen* und funktionsfähigen. Jede dieser Konfigurationen zeichnet sich dabei durch bestimmte Eigenschaften hinsichtlich der Funktionalität aus.

Zur Definition von Konfigurationen mit den Modellierungstechniken aus Abschnitt 3.4 müssen zuerst die verfügbaren Implementierungsklassen der Tomcat-Schnittstellen aus Abbildung 5.6 als Subkomponenten modelliert werden. Die Schnittstelle einer Implementierungsklasse wird dabei zum Typ des *provided*-Ports. *Required*-Ports werden eingeführt, um die Assoziationen im Klassendiagramm in Verbindungen zwischen Subkomponenten umzuwandeln.

Abbildung 5.7 definiert eine explizite Konfiguration (siehe Unterabschnitt 3.4.3), die eine minimale Konfiguration von Tomcat darstellt[19]. Die darin enthaltenen Komponenten StandardService und StandardEngine werden in allen funktionsfähigen Konfigurationen benötigt, d. h. alle anderen Konfigurationen sind lediglich Erweiterungen[20] der Basis-Konfiguration. Diese Erweiterungen werden durch eine Reihe von Template-Variationen definiert, die gemeinsam in Abbildung 5.8 dargestellt sind. Die Template-Variationen *Host*, *Engine_ Valve* und *Host_ Valve* sind Mehrfach-Variationen (siehe Unterabschnitt 3.4.4) und können demzufolge beliebig oft angewendet werden. *Host_ Logger*, *Host_ Realm* und *Host_ Valve* erweitern die *Host*-Variation, während alle anderen Variationen die explizite Konfiguration *Basis* aus Abbildung 5.7 erweitern. Zum besseren Verständnis sind die Auswirkungen der einzelnen Variationen in Tabelle 5.7 kurz erläutert.

Die beiden `Connector`-Komponenten (CoyoteConnector und AjpConnector) verwalten einen oder mehrere Threads zur Abarbeitung von Anfragen, die sie über TCP-Sockets von Clients erhalten. Sie verletzen damit eine Voraussetzung für die Durchführung von Rekonfigurationen (siehe Abschnitt 3.5), da sie durch die Abarbeitung der eigenen Threads in endlicher Zeit niemals inaktiv werden, auch wenn alle externen Methodenaufrufe blockiert werden. Deshalb wird für die `Connector`-Komponenten ein spezieller Aspektoperator eingeführt, der mit Hilfe der Joinpoints *preReconfiguration* und *postReconfiguration* die Threads vor einer Rekonfiguration anhält und danach wieder startet.

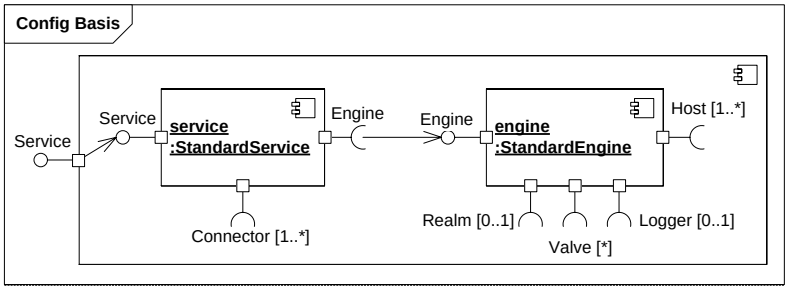

Abbildung 5.7: Basis-Konfiguration von Tomcat

[19]Diese Konfiguration ist noch nicht funktionsfähig, da mindestens ein `Host` und ein `Connector` fehlt.
[20]D.h. zusätzliche Komponenten und Verbindungen sind enthalten.

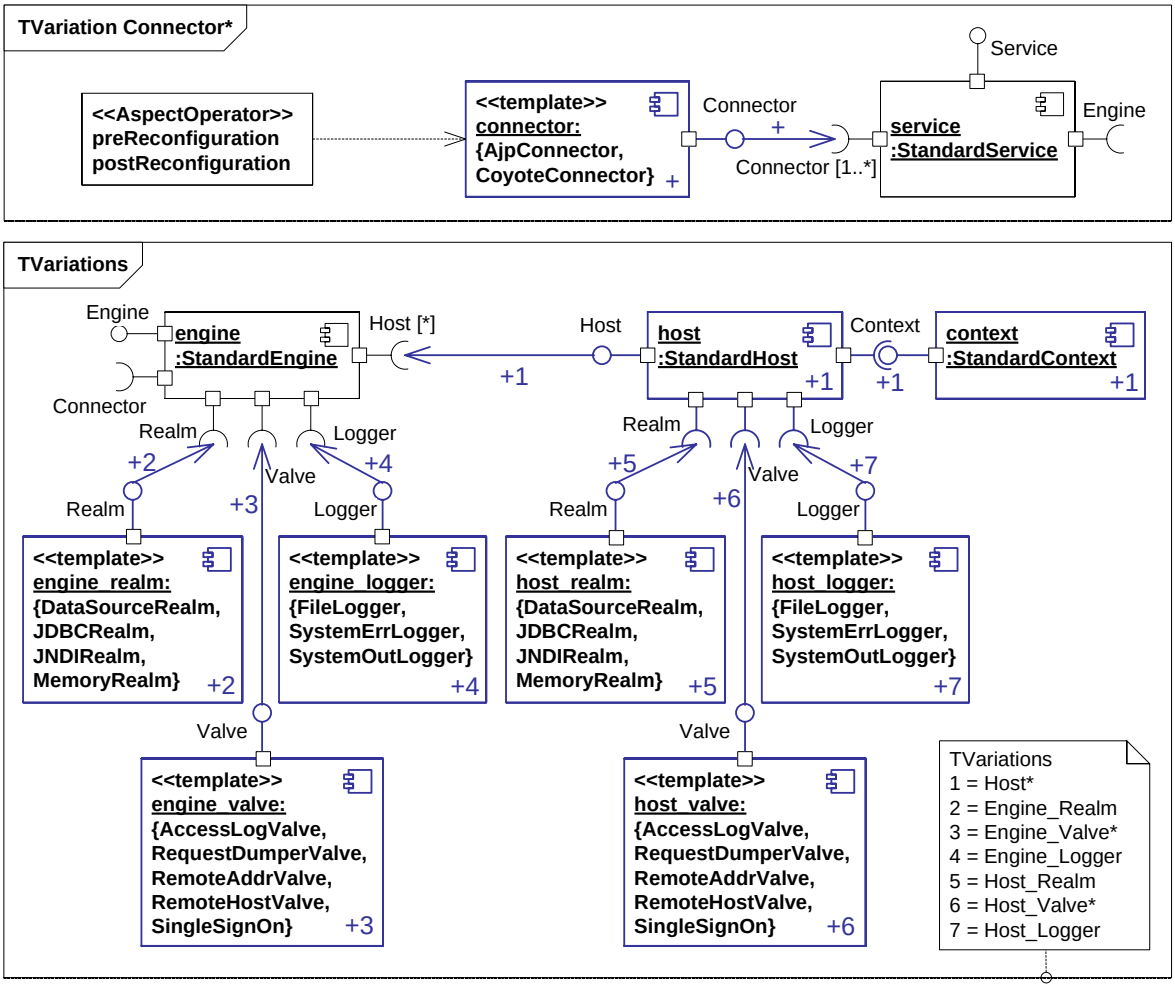

Abbildung 5.8: Template-Variationen für die Basis-Konfiguration von Tomcat

5.5.2.2 Komponentenparameter und Parameter-Abbildung

Die Parameter der Tomcat-Komponente werden so gewählt, dass sie die verschiedenen denkbaren Anwendungsfälle widerspiegeln, ohne damit unnötige Implementierungsdetails offen zu legen (siehe Tabelle 5.8). Die Abstraktion von Implementierungsdetails wird an den folgenden beiden Beispielen erläutert:

- In der Konfigurationsdatei `server.xml` wird durch die Auswahl von `Connector`s festgelegt, ob Tomcat einen eigenen oder einen externen HTTP-Server verwenden soll. Der gleiche Effekt kann durch einen einfachen Komponentenparameter *serverMode* realisiert werden, ohne dabei die interne Implementierung mit verschiedenen `Connector`s offen zu legen.
- Die Protokollierung von HTTP-Requests wird durch die Komponente `RequestDumperValve` realisiert. Ein Boolean-Parameter *requestDumper* genügt zur Auswahl. Anwender

Variationsname	Aufgabe
Connector	Hinzufügen eines Connectors zum Service
Host	Hinzufügen eines virtuellen Hosts
Engine_Realm	Festlegung des Realm (vier Implementierungen) der Engine
Engine_Valve	Hinzufügen einer Valve (fünf Implementierungen) zur Engine
Engine_Logger	Festlegung des Loggers (drei Implementierungen) der Engine
Host_Realm	Festlegung des Realm (vier Implementierungen) eines Hosts
Host_Valve	Hinzufügen einer Valve (fünf Implementierungen) zu einem Host
Host_Logger	Festlegung des Loggers (drei Implementierungen) eines Hosts

Tabelle 5.7: Aufgabe der verschiedenen Variationen aus Abbildung 5.8

der Tomcat-Komponente müssen dann nicht wissen, welche Implementierungsklasse dazu an welche Stelle in die Architektur eingefügt werden muss.

In ähnlicher Weise wirken sich auch viele weitere Komponentenparameter aus Tabelle 5.8 bzw. Tabelle 5.9 auf die interne Architektur von Tomcat aus.

Die Parameter für den Default-Host und die die zusätzlichen virtuellen Hosts sind in Tabelle 5.9 kurz beschrieben. Die Parameter des Default-Hosts könnten anstelle des komplexen Map-Parameters *defaultHost* auch direkt als Parameter der Tomcat-Komponente realisiert werden. Da der Default-Host und alle virtuellen Hosts jedoch die gleichen Parameter verwenden, wurde aus Übersichtlichkeitsgründen darauf verzichtet.

Eine funktionsfähige Konfiguration von Tomcat erfordert je mindestens eine Subkomponente für die folgenden Tomcat-Schnittstellen: `Service`, `Engine`, `Connector` und `Host`. Um diese Voraussetzung zu erfüllen, müssen unabhängig der festgelegten Parameterwerte der Tomcat-Komponente die Konfiguration *Basis* sowie je mindestens einmal die Template-Konfigurationen *Connector* und *Host* ausgewählt werden. Die folgende Parameterabbildung (siehe Unterabschnitt 3.4.6) ordnet den verschiedenen Parameterwerten die entsprechenden Konfiguration bzw. Variationen zu.

```
serverMode = EXTERN:     Basis, Host, Connector<AjpConnector>
serverMode = HTTP:       Basis, Host, Connector<CoyoteConnector>
serverMode = HTTPS:      Basis, Host, Connector<CoyoteConnector>
serverMode = HTTP_HTTPS: Basis, Host, Connector<CoyoteConnector>,
                         Connector<CoyoteConnector>

loggerMode = STDOUT:     Engine_Logger<SystemOutLogger>
loggerMode = STDERR:     Engine_Logger<SystemErrLogger>
loggerMode = FILE:       Engine_Logger<FileLogger>

realm = JDBC:            Engine_Realm<JDBCRealm>
realm = JNDI:            Engine_Realm<JNDIRealm>
realm = MEMORY:          Engine_Realm<MemoryRealm>
realm = DATA_SOURCE:     Engine_Realm<DataSourceRealm>
```

Parameter	Typ	Beschreibung
serverMode*	Enum	Legt fest, ob Tomcat einen internen oder externen HTTP-Server verwendet (EXTERN, HTTP, HTTPS, HTTP_HTTPS)
defaultHost*	Map	Legt Parameter des Default-Hosts fest
hosts*	Array	Definiert zusätzliche virtuelle Hosts mit angegebenen Parametern
loggerMode*	Enum	Legt fest, ob und wie Logging-Meldungen der Engine ausgegeben werden (None, STDOUT, STDERR, FILE)
httpPort	int	Definiert den TCP-Port für den internen HTTPS-Server, falls er verwendet wird.
httpsPort	int	Definiert den TCP-Port für den internen HTTPS-Server, falls er verwendet wird.
realm*	Enum	Legt die verwendete Datenbank für die Nutzer-Authentifizierung fest (JDBC, JNDI, MEMORY, DATA_SOURCE)
logDir	String	Verzeichnis für Log-Dateien der Engine
logPrefix	String	Prefix für alle Log-Dateien der Engine
requestDump*	Boolean	Protokollierung aller HTTP-Requests der Engine (ja/nein)
allowedHosts*	String	Liste mit erlaubten Hostnamen für HTTP-Requests
deniedHosts*	String	Liste mit verbotenen Hostnamen für HTTP-Requests
allowedIPs*	String	Liste mit erlaubten IP-Adressen für HTTP-Requests
deniedIPs*	String	Liste mit verbotenen IP-Adressen für HTTP-Requests

Tabelle 5.8: Parameter der Tomcat-Komponente (mit * gekennzeichnete Parameter wirken sich auf die interne Konfiguration von Tomcat aus)

```
requestDump:              Engine_Valve<RequestDumperValve>
singleSignOn:             Engine_Valve<SingleSignOn>
allowedHost:              Engine_Valve<RemoteHostValve>
deniedHost:               Engine_Valve<RemoteHostValve>
allowedIPs:               Engine_Valve<RemoteAddrValve>
deniedIPs:                Engine_Valve<RemoteAddrValve>
defaultHost:              Host
hosts:                    Host+
```

Die folgenden Parameter des Map-Parameter *defaultHost* bzw. des Array-Parameters *hosts* wählen weitere Variationen auf Basis der Variation *Host* aus:

```
realm = JDBC:         Host_Realm<JDBCRealm>
realm = JNDI:         Host_Realm<JNDIRealm>
realm = MEMORY:       Host_Realm<MemoryRealm>
realm = DATA_SOURCE: Host_Realm<DataSourceRealm>

logger = STDOUT:      Host_Logger<SystemOutLogger>
```

Parameter	Typ	Beschreibung
realm*	Enum	Legt die verwendete Datenbank für die Nutzer-Authentifizierung fest (JDBC, JNDI, MEMORY, DATA_SOURCE, NONE)
appBase	String	Verzeichnis für Web-Anwendungen des Hosts
loggerMode*	Enum	Legt fest, ob und wie Logging-Meldungen des Hosts ausgegeben werden (NONE, STDOUT, STDERR, FILE)
logDir	String	Verzeichnis für Log-Dateien des Hosts
logPrefix	String	Prefix für alle Log-Dateien des Hosts
autodeploy	Boolean	Automatische Installation von Web-Anwendungen (ja/nein)
requestDump*	Boolean	Protokollierung aller HTTP-Requests des Hosts (ja/nein)
singeSignOn*	Boolean	Unterstützung von Single-Sign-On (ja/nein)
allowedHosts*	String	Liste mit erlaubten Hostnamen für HTTP-Requests
deniedHosts*	String	Liste mit verbotenen Hostnamen für HTTP-Requests
allowedIPs*	String	Liste mit erlaubten IP-Adressen für HTTP-Requests
deniedIPs*	String	Liste mit verbotenen IP-Adressen für HTTP-Requests

Tabelle 5.9: Parameter für Default-Host und virtuelle Hosts (mit * gekennzeichnete Parameter wirken sich auf die interne Konfiguration von Tomcat aus)

```
logger = STDERR:     Host_Logger<SystemErrLogger>
logger = FILE:       Host_Logger<FileLogger>

requestDump:         Host_Valve<RequestDumperValve>
singleSignOn:        Host_Valve<SingleSignOn>
allowedHost:         Host_Valve<RemoteHostValve>
deniedHost:          Host_Valve<RemoteHostValve>
allowedIPs:          Host_Valve<RemoteAddrValve>
deniedIPs:           Host_Valve<RemoteAddrValve>
```

5.5.3 Vergleich des existierenden Tomcats mit Tomcat als Adaptierbare Komponente

Die Konfiguration von Tomcat mit Hilfe der Datei `server.xml` hat mehrere Nachteile:

- Anwender benötigen detaillierte Kenntnisse über die internen Aufbau von Tomcat, um alle Einstellungsmöglichkeiten der Konfigurationsdatei zu verstehen. Als Konsequenz davon enthält die bei Tomcat mitgelieferte Standard-Konfigurationsdatei eine Menge von alternativen Einstellungen innerhalb von Kommentaren, die bei Bedarf auskommentiert werden können.
- Zur Laufzeit kann Tomcat nicht mit Hilfe der Konfigurationsdatei rekonfiguriert werden. Ein direkter Zugriff auf die einzelnen Komponenten zur Laufzeit mit JMX wird jedoch unterstützt. In diesem Fall sind jedoch noch detailliertere Kenntnisse der Arbeitsweise von Tomcat erforderlich, um zu wissen, welche Komponentenmethoden zu welchem Zweck aufgerufen werden müssen.

- Die Programmlogik zur Auswertung der Konfigurationsdatei und zur entsprechenden Initialisierung der Komponenten wurde speziell für Tomcat entwickelt. Bei der Realisierung als Adaptierbare Komponente kann für diesen Zweck auf Werkzeuge und Bibliotheken (siehe Kapitel 4) zurückgegriffen werden, was die Entwicklung wesentlich vereinfacht.

Die Konfigurationsdatei von Tomcat hat aber nicht nur Nachteile. Die relativ komplexe Konfigurationsdatei ermöglicht andererseits eine hohe Flexibilität und die einfache Erweiterung. Neue Implementierungen für Tomcat-Schnittstellen, z. B. eine spezielle `Valve`-Komponente, können über das `classname`-Attribut und die Angabe des Klassennamen einfach in Tomcat integriert werden.

Die Umsetzung von Tomcat als Adaptierbare Komponente erzielt eine Reihe von Vorteilen:

- Die Implementierungsdetails bleiben für den Anwender transparent. Mit den Parametern der Tomcat-Komponente wird lediglich die gewünschte Funktionalität eingestellt.
- Durch die getrennte Beschreibung von Konfigurationen und Parameterabbildungen können vereinfachte Tomcat-Komponenten mit nur noch wenigen Parametern entwickelt werden, ohne Implementierungen oder Konfigurationsbeschreibungen ändern zu müssen. Auf diese Weise kann beispielsweise eine minimale Tomcat-Komponente definiert und erzeugt werden, die viele Merkmale unveränderbar festlegt und damit nicht mehr alle Konfigurationsmöglichkeiten bietet, aber dafür mit wenigen Komponentenparametern auskommt und die Anwendung vereinfacht.
- Die unterschiedlichen möglichen PSMs (siehe Abschnitt 4.1) ermöglichen für den jeweiligen Anwendungsfall zugeschnittene Flexibilität und Optimierungen. Beim Startzeit-PSM können beispielsweise nicht verwendete Subkomponenten weggelassen werden, um dadurch die Größe der Tomcat-Komponente zu reduzieren.
- Durch die Umsetzung als Komponente kann Tomcat einfacher in andere Anwendungen integriert werden, um dort die Funktion eines Servlet-Containers anzubieten.
- Die relativ aufwändige manuelle Entwicklung der Programmlogik zur Auswertung der Konfigurationsdatei und zur Unterstützung der Rekonfiguration mit JMX entfällt. Die Entwickler können sich dadurch auf die eigentliche Programmlogik konzentrieren.

5.6 Fallstudie: Komponenten mit mehreren QoS-Profilen in Comquad

Nichtfunktionale Eigenschaften und insbesondere *Quality of Service* (QoS) spielen in einigen Anwendungsbereichen (z. B. Echtzeit- oder Multimedia-Anwendungen) eine wichtige Rolle. Im Rahmen des COMQUAD-Projektes wurden diese Eigenschaften bei der komponentenorientierten Softwareentwicklung berücksichtigt. Das Konzept von Adaptierbaren Komponenten wird in diesem Zusammenhang zur Realisierung von Komponenten mit mehreren QoS-Profilen verwendet. Diese Idee und erste Ergebnisse wurden bereits in [GPRZ04], [GPA+04b] und [Göb04] vorgestellt.

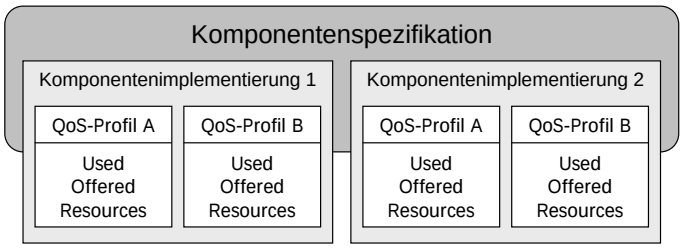

Abbildung 5.9: Beziehung zwischen Komponentenspezifikation, Implementierungen und QoS-Profilen

In diesem Abschnitt werden zunächst einige Grundlagen von COMQUAD beschrieben, um anschließend die Anwendung von Adaptierbaren Komponenten anhand einer Videoplayer-Anwendung zu veranschaulichen.

5.6.1 Überblick zu Comquad

COMQUAD (***COM**ponents with **QU**antitative properties and **AD**aptivity* [ABF+03, AFG+03, GPA+04c]) war von 2001-2004 eine von der Deutschen Forschungsgemeinschaft (DFG) geförderte Forschergruppe an der TU Dresden, an der sich sechs Lehrstühle beteiligten. Zielstellung war die Entwicklung von Komponenten, die Zusagen über nichtfunktionalen Eigenschaften – im Speziellen *Quality-of-Service* und Sicherheitseigenschaften – einhalten können. Diese Eigenschaften werden zur Entwurfszeit spezifiziert und zur Entwicklungszeit weiter verfeinert. Der Komponentencontainer, der die Laufzeitumgebung für COMQUAD-Komponenten darstellt, verwendet diese Informationen, um Verträge (*Contracts*) über nichtfunktionale Eigenschaften zwischen verschiedenen Komponenten auszuhandeln. Eine Menge von Komponenten bildet dabei ein Komponentennetz, das durch spezielle Deskriptoren definiert wird [GPRZ04]. Zur Laufzeit übernimmt der Komponentencontainer die Überwachung und Durchsetzung der Verträge.

Das COMQUAD-Komponentenmodell folgt einer Klassifikation von Cheesman und Daniels [CD01]: Eine Komponenten-Spezifikation kann mehrere Implementierungen besitzen und jede Implementierung kann mehrere QoS-Profile unterstützen (siehe Abbildung 5.9). Der Komponentencontainer wählt zur Laufzeit während der *Vertragsaushandlung* entsprechend den Anforderungen eines Clients geeignete Implementierungen und QoS-Profile aus. Ein QoS-Profil einer COMQUAD-Komponente repräsentiert dabei einen bestimmten Arbeitsbereich, der durch benötigte und zur Verfügung gestellte nichtfunktionale Eigenschaften sowie einen bestimmten Ressourcenbedarf (z. B. Speicher) gekennzeichnet ist. Die nichtfunktionalen Eigenschaften von Komponenten werden zur Entwicklungszeit ermittelt und mit der entwickelten Beschreibungssprache CQML+ [RZ03] beschrieben.

Obwohl COMQUAD Ressourcenreservierungen einsetzt, um Verträge über nichtfunktionale Eigenschaften einzuhalten, wird die Möglichkeit von unvorhergesehenen Vertragsverletzungen nicht ausgeschlossen, sondern durch Adaptionsmechanismen explizit berücksichtigt. Auf diese Weise schließt sich die gleichzeitige Verwendung von Mechanismen zur Ressourcenreservierung und zur Adaption nicht gegenseitig auch, sondern kann sich sogar gegenseitig ergänzen.

Das Wechseln der aktiven QoS-Profile von Komponenten zur Laufzeit wird als eine Adaptionsstrategie zur Behandlung von wechselnden Umgebungsbedingungen (unterschiedliche

Verfügbarkeit von Ressourcen) eingesetzt. Das aktive QoS-Profil wird immer vom Komponentencontainer festgelegt und kann bei Bedarf gewechselt werden, um sich den neuen Bedingungen anzupassen. Dabei stellt sich jedoch die Frage, wie Komponenten entwickelt werden können, die mehrere QoS-Profile unterstützen. Adaptierbare Komponenten sind eine Lösung für dieses Problem, wenn die verschiedenen QoS-Profile als spezielle Komponentenparameter betrachtet werden. Ein QoS-Profil kann auf diese Weise auf eine bestimmte interne Konfiguration einer Adaptierbaren Komponente abgebildet werden. Der Wechsel des QoS-Profils zur Laufzeit löst eine Rekonfiguration aus und führt damit zu veränderten Komponenteneigenschaften.

Zur Reservierung von Ressourcen und zur Durchsetzung von Verträgen werden Mechanismen des Echtzeitbetriebssystems DROPS (*Dresden Realtime Operating System* [HBB+98]) verwendet. Der COMQUAD-Komponentencontainer [GPA+04b, GPA+04a] wird dazu in einen relativ kleinen echtzeitfähigen (*Non-Realtime Container* – NRTC) und einen komplexeren nicht echtzeitfähigen Teil (*Realtime Container* – RTC) aufgeteilt. Der NRTC wurde in Java entwickelt und läuft unter L4Linux [HHW98], einer Portierung von Linux auf die L4-Microkernel API, die als User-Prozess unter DROPS ausgeführt werden kann. Der RTC läuft direkt unter der Kontrolle von DROPS. Über eine Schnittstelle kommunizieren die beiden Teile des Komponentencontainers und der NRTC steuert den RTC zur Laufzeit. Der NRTC übernimmt alle Aufgaben, für die keine Echtzeitanforderungen eingehalten werden müssen, insbesondere die Installation und Initialisierung von Komponenten einschließlich der Vertragsaushandlung. Der RTC ist dagegen nur für die Ausführung von komponentenbasierten Anwendungen und die Überwachung von Verträgen verantwortlich. Für die kontinuierliche Datenübertragung zwischen Komponenten, die insbesondere für Multimedia-Anwendungen benötigt werden, unterstützen der RTC und DROPS spezielle Stream-Verbindungen (*DROPS Streaming Interface* [LHR01]).

5.6.2 Beispielanwendung: Videoplayer

Zur Validierung der entwickelten Konzepte von COMQUAD wurde ein Videoplayer als Beispielanwendung implementiert (vorgestellt in [GPRZ04]). Der Videoplayer spielt Videos mit einer garantierten Framerate von 25 Frames pro Sekunde ab, unabhängig von sonstigen laufenden Anwendungen auf demselben Rechner. Insgesamt fünf Komponenten arbeiten dabei zusammen (siehe Abbildung 5.10), wobei zur Datenübertragung Stream-Verbindungen verwendet werden:

- Eine *Demultiplexer*-Komponente erhält über einen *required*-Port einen komprimierten Videostrom, der z. B. aus einer Datei gelesen oder über das Netzwerk übertragen wurde, und zerlegt ihn in die reinen Audio- und Videoanteile. Die dabei entstehenden komprimierten Audio- und Videoströme werden über *provided*-Ports zur Weiterverarbeitung angeboten.

- Eine *VideoDecoder*-Komponente nimmt über einen *required*-Port einen komprimierten Videostrom entgegen und dekomprimiert die einzelnen enthaltenen Frames. Als Ergebnis wird ein unkomprimierter Videostrom über einen *provided*-Port zur Weiterverarbeitung angeboten.

- Eine *AudioDecoder*-Komponente nimmt über einen *required*-Port einen komprimierten

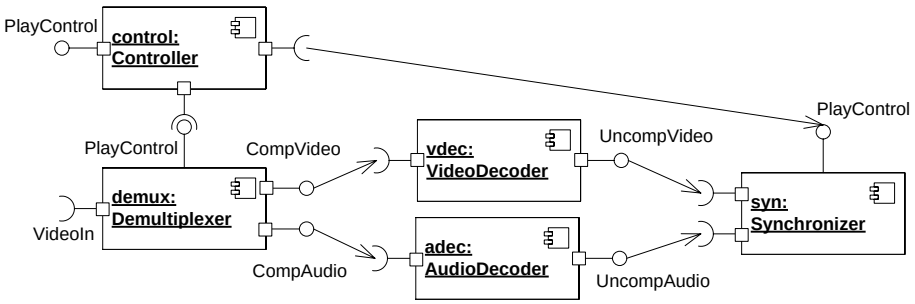

Abbildung 5.10: Videoplayer-Anwendung

Audiostrom entgegen und dekomprimiert die darin enthaltenen Audiodaten. Als Ergebnis wird ein unkomprimierter Audiostrom über einen *provided*-Port zur Weiterverarbeitung angeboten.

- Eine *Synchronizer*-Komponente nimmt über zwei *required*-Ports je einen unkomprimierten Audio- und Videostrom entgegen und synchronisiert dabei Bild und Ton. Durch die unabhängige Verarbeitung von Bild- und Toninformationen könnten sich andernfalls die Daten zeitlich verschieben und der Ton würde nicht mehr zum gezeigten Bild passen (Lippensynchronität).
- Eine *Controller*-Komponente steuert über Schnittstellen die *Demultiplexer*- und *Synchronizer*-Komponente und kontrolliert damit das Starten und Stoppen der Videodekodierung.

Von diesen Komponenten unterstützt nur die *VideoDecoder*-Komponente mehrere QoS-Profile. In Abhängigkeit der vorhandenen Ressourcen (insbesondere Rechenzeit) können über die QoS-Profile unterschiedliche Verfahren zur Nachbearbeitung (*Postprocessing*) von dekomprimierten Frames ausgewählt werden. Bei ausreichender Verfügbarkeit von Ressourcen kann eine aufwändige Nachbearbeitung der Frames durchgeführt werden und damit eine bessere Qualität der angezeigten Videos erreicht werden. In [Rie03] werden vier verschiedene Verfahren der Nachbearbeitung beschrieben: keine Nachbearbeitung, schnelle horizontale und vertikale *Deblocking*-Filterung[21], *Deringing*-Filterung[22] sowie aufwändige *Deblocking*-Filterung in Kombination mit *Deringing*-Filterung.

Adaptierbare Komponenten werden zur Beschreibung der unterschiedlichen QoS-Profile der *VideoDecoder*-Komponente verwendet. Abbildung 5.11 zeigt dazu die unterschiedlichen Konfigurationen für die vier QoS-Profile. Wenn die QoS-Profile von null (geringste Qualität) bis drei (höchste Qualität) durchnummeriert werden, ergibt sich folgende einfache Parameterabbildung:

```
QoS-Profil = 0: Videodecoder
QoS-Profil = 1: Videodecoder_Deblock
QoS-Profil = 2: Videodecoder_Dering<FastDeblockingFilter>
QoS-Profil = 3: Videodecoder_Dering<DeblockingFilter>
```

[21] Deblocking-Filter reduzieren Block-Artefakte in Frames, die durch blockweise Arbeitsweise der Videokompression entstehen.

[22] Deringing-Filter reduzieren Ring-Artefakte in Frames, die durch die Videokompression entstehen.

Für jede Konfiguration bzw. jedes QoS-Profil müssen durch Messung die genauen QoS-Eigenschaften ermittelt (siehe [MN04]) und mit CQML$^+$ beschrieben werden. Der COMQUAD-Komponentencontainer kann damit zur Laufzeit das jeweils passende QoS-Profil auswählen.

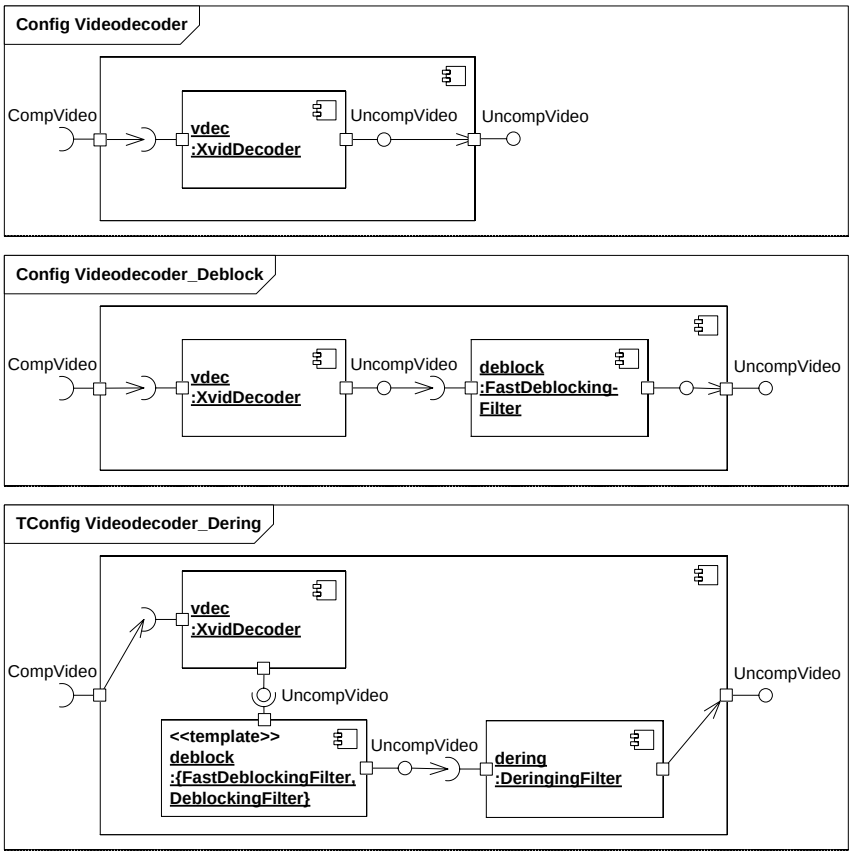

Abbildung 5.11: Konfigurationen der Videodecoder-Komponente

5.6.3 Selbstadaptive Videodecoder-Komponente

Die Videodecoder-Komponente kann außerhalb der COMQUAD-Umgebung auch als eine selbstadaptive Komponente umgesetzt werden. In Abhängigkeit der Rechengeschwindigkeit und der CPU-Auslastung wird dabei das jeweils passende QoS-Profil bzw. die zugehörige Konfiguration ausgewählt. Zur Ermittlung dieser Werte wird ein Kontextmodell und die zugehörigen Implementierungen der Kontextquellen benötigt (siehe Unterabschnitt 3.4.7 und Unterabschnitt 4.3.8).

Die Rechengeschwindigkeit und CPU-Auslastung werden indirekt durch die Messung der Zeit zum Dekodieren eines Frames durch die Videodecoder-Komponente ermittelt. Dazu wird ein Aspektoperator für die XvidDecoder-Komponente definiert, der über die Joinpoints *preMethod* und *postMethod* die Zeit zum Dekodieren eines Frames ermittelt (in ms). Der Wert wird an eine Kontextquelle weitergeleitet, die jeweils den gleitenden Durchschnitt aus den letzten 10 Frames als Kontextwert („DecodeTime“) zur Verfügung stellt.

Zum flüssigen Abspielen eines Videos müssen 25 Frames pro Sekunde von der Videodecoder-Komponente bearbeitet werden, d. h. für ein einzelnes Frame stehen weniger als 40 ms zur Verfügung. Innerhalb dieser Zeit muss auch die Nachbearbeitung der Frames durchgeführt werden, wenn ein entsprechendes QoS-Profil ausgewählt wurde. Das bedeutet, je weniger Zeit zum Dekodieren eines Frames benötigt wird, desto mehr Zeit kann für die Nachbearbeitung verwendet werden. Damit können die folgenden Regeln zur Selbstadaption definiert werden:

Prämisse	**Konklusion**
$QoS\text{-}Profile \geq 1 \wedge DecodeTime \geq 28$	$QoS\text{-}Profile := 0$
$QoS\text{-}Profile = 0 \wedge 15 \leq DecodeTime < 25$	$QoS\text{-}Profile := 1$
$QoS\text{-}Profile \geq 2 \wedge 18 \leq DecodeTime < 28$	$QoS\text{-}Profile := 1$
$QoS\text{-}Profile \leq 1 \wedge 5 \leq DecodeTime < 15$	$QoS\text{-}Profile := 2$
$QoS\text{-}Profile = 3 \wedge 8 \leq DecodeTime < 18$	$QoS\text{-}Profile := 2$
$QoS\text{-}Profile \leq 2 \wedge DecodeTime < 5$	$QoS\text{-}Profile := 3$

Die Werte für Zeitangaben müssen durch Messungen der Zeiten für die verschiedenen Nachbearbeitungsverfahren bestimmt werden. Durch die Berücksichtigung des jeweils gerade aktiven QoS-Profiles in den Regeln wird eine Hysterese von 3 ms erreicht, damit ständige Rekonfigurationen bei geringen Schwankungen der *DecodeTime* verhindert werden. Bei der Initialisierung der Videodecoder-Komponente wird das QoS-Profile null ausgewählt.

5.6.4 Bewertung

Die Videodecoder-Komponente kann prinzipiell auch ohne Verwendung von Adaptierbaren Komponenten implementiert werden. Die Programmlogik zur Rekonfiguration muss dann manuell entwickelt werden. Typischerweise wird dazu an einer Stelle in der Komponente eine bedingte Auswahl (z. B. mit `switch-case` oder `if...else`-Anweisungen) implementiert, die in Abhängigkeit des ausgewählten QoS-Profils den Verarbeitungsablauf steuert.

Bei der Verwendung von Adaptierbaren Komponenten wird klar zwischen der Beschreibung der Konfigurationen und der Implementierung der Komponenten getrennt (*Separation of Concerns*). Jedes QoS-Profile wird einer bestimmten Konfiguration zugeordnet. Tests und Messungen der nichtfunktionalen Eigenschaften können dadurch einfacher durchgeführt werden. Die Programmlogik zur Rekonfiguration beim Wechsel des QoS-Profils wird automatisch erzeugt bzw. durch die Laufzeitbibliothek bereitgestellt. Wenn zur Laufzeit nur ein bestimmtes QoS-Profil benötigt wird, kann mit Hilfe des Installationszeit- bzw. Startzeit-PSM ohne zusätzlichen Aufwand die Komplexität der Komponente weiter reduziert werden.

5.7 Fazit der Validierung

Durch die realisierten Unterstützungen verschiedener Komponentenplattformen wurde die Plattformunabhängigkeit der Konzepte von Adaptierbaren Komponenten demonstriert. Dabei wurde auch gleichzeitig das im Abschnitt 4.2 beschriebene Verfahren zur Entwicklung plattformspezifischer Modelle überprüft. Die Laufzeitunterstützungen für verschiedene Komponentenplattformen (außer COM) basierten auf dem im Abschnitt 4.3 erläuterten Framework und erweiterten es um plattformspezifische Funktionen. Entsprechend einem wesentlichen Ziel der Arbeit (siehe Abschnitt 1.2) können damit Adaptierbare Komponenten auf

vorhandenen Komponentenplattformen eingesetzt werden und in existierende Anwendungen integriert werden, ohne die Komponentenplattformen selbst ändern zu müssen.

Durch die zwei Fallstudien konnte weiterhin gezeigt werden, dass die entwickelten Modellierungstechniken (siehe Kapitel 3) auch für komplexe Anwendungsbeispiele geeignet sind. Es wurde außerdem gezeigt, dass alle Modellierungskonzepte im praktischen Einsatz angewendet und benötigt werden.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

In der vorliegenden Arbeit wurde ein neues Adaptionskonzept für Komponenten entwickelt. Im Folgenden werden noch einmal die wesentlichen Ergebnisse mit den Schwerpunkten Modellierung, Modelltransformation und Plattformunterstützung sowie Validierung zusammengefasst.

Adaptionskonzepte für Komponenten und deren Modellierung Adaptierbare Komponenten verwenden ein hierarchisches Komponentenmodell, d. h. sie setzen sich aus einer Menge von Subkomponenten zusammen. Die wesentliche Idee zur Umsetzung der Adaptivität besteht darin, bestimmte Parameterwerte einer Adaptierbaren Komponente auf unterschiedliche interne Konfigurationen abzubilden. Eine Konfiguration wird dabei durch eine Menge von Subkomponenten und Verbindungen zwischen Komponentenports definiert. Mit einer Parameterabbildung wird festgelegt, welche Konfiguration bei bestimmten Werten von Komponentenparametern der Adaptierbaren Komponente ausgewählt wird. Parameteränderungen können damit intern Rekonfigurationen auslösen, die jedoch nach außen gekapselt werden.

Zusätzlich zu dieser Grundfunktionalität unterstützen Adaptierbare Komponenten mit Adaptions- und Aspektoperatoren sowie einem Kontextmodell noch eine Reihe von optionalen Konzepten. Adaptionsoperatoren können zur Integration von eigentlich inkompatiblen Subkomponenten in eine Adaptierbare Komponente verwendet werden. Aspektoperatoren ergänzen *cross-cutting concerns*, wie beispielsweise Zugriffskontrollen oder Fehlerbehandlung. Sie übernehmen dazu das Konzept von Joinpoints aus der aspektorientierten Programmierung und erlauben das Einfügen von zusätzlicher Programmlogik an diesen Stellen. Mit der Definition eines Kontextmodells können Adaptierbare Komponenten auf Umgebungsinformationen (Kontext) zugreifen, um damit eigenständig auf Änderungen zu reagieren und Anpassungen vorzunehmen. Durch eine Menge von Regeln werden unter bestimmten Bedingungen Parameterwerte geändert, die dann indirekt die Rekonfiguration der Adaptierbaren Komponente veranlassen.

Zur Beschreibung der möglichen internen Konfigurationen von Adaptierbaren Komponenten wurden vier verschiedene graphische Modellierungstechniken entwickelt, die alle auf der graphischen Notation von UML-Komponentendiagrammen basieren und lediglich einige Erweiterungen in Form von Stereotypen hinzufügen. *Explizite Konfigurationen* beschreiben einen Teil oder eine vollständige Konfiguration einer Adaptierbaren Komponente. *Template-Konfigurationen* enthalten zusätzlich mindestens eine so genannte *Template-Komponente*, die als Stellvertreter für eine Menge von möglichen konkreten Komponenten fungiert. *Variationen* beschreiben lediglich Änderungen (Hinzufügen/Entfernen von Subkomponenten bzw. Verbin-

dungen) an einer Ausgangskonfiguration. *Template-Variationen* integrieren das Konzept von *Template-Komponenten* in Beschreibung von Variationen, um damit eine Menge von Variationen zu modellieren.

Alle Ergebnisse der Modellierungsphase inklusive Kontextmodell und Parameterabbildung werden in einem MOF-basierten Metamodell für Adaptierbare Komponenten gespeichert und sind damit die Basis für die spätere Modelltransformation und Laufzeitunterstützung.

Modelltransformation und Laufzeitunterstützung Adaptierbarer Komponenten Die Konzepte Adaptierbarer Komponenten sollen auf möglichst vielen Komponentenplattformen angewendet werden können. Aus diesem Grund wurde ein generisches Verfahren zur Entwicklung von Modelltransformationen und Laufzeitunterstützungen für verschiedene Komponentenplattformen entwickelt. Dazu wurden zunächst Eigenschaften zur Klassifikation von Komponentenplattformen ermittelt, die sich auf die Modelltransformation von Adaptierbaren Komponenten auswirken. Die Eigenschaften charakterisieren den Aufbau, Kommunikationsbeziehungen, die Verwaltung und das Laufzeitverhalten und den Lebenszyklus von Komponenten.

In Abhängigkeit der ermittelten Eigenschaften einer Komponentenplattform wurden Regeln zur Umsetzung von verschiedenen Konzepten Adaptierbarer Komponenten aufgestellt. Anhand dieser Regeln kann später die Modelltransformation und Unterstützung für eine bestimmte Komponentenplattform entwickelt werden. Die Konzepte von Adaptierbaren Komponenten müssen entweder direkt auf vorhandene Konzepte einer bestimmten Komponentenplattform abgebildet werden, oder es muss eine geeignete Strategie zur Emulation entwickelt werden. Zur Emulation wird eine Kombination von Programmcode-Erzeugung, Hilfsfunktionen einer Laufzeitbibliothek und Programmierrichtlinien verwendet.

Zur Erhöhung der Flexibilität werden für eine bestimmte Komponentenplattform nicht nur eine, sondern drei verschiedene Modelltransformationen für plattformspezifische Modelle (PSM) mit jeweils unterschiedlichen Eigenschaften unterstützt. Der wesentliche Unterschied zwischen diesen Modellen besteht in den jeweils möglichen Zeitpunkten für die Festlegung bzw. Änderung von Komponentenparametern. Je nach Anforderungen einer Anwendung können Entwickler das jeweils passende plattformspezifische Modell auswählen. Auch alle nicht automatisch erzeugten Bestandteile von Komponentenimplementierungen, insbesondere die Quelltexte, bleiben für alle drei Modelle gleich. Durch den Verzicht an Flexibilität können verschiedene Optimierungen durchgeführt werden, um beispielsweise die Ausführungsgeschwindigkeit durch den Wegfall der komplexen Steuerungslogik für die Rekonfiguration zu steigern oder die Komponentengröße durch das Weglassen von nicht benötigten Komponenten zu reduzieren.

Als Ergänzung zu den Transformationsregeln wurde ein Framework zur Unterstützung Adaptierbarer Komponenten entwickelt. Dieses Framework enthält verschiedene Pakete für die wesentlichen Funktionen einer Werkzeug- und Laufzeitunterstützung, jedoch immer noch unabhängig von einer bestimmten Komponentenplattform. Der Philosophie eines Frameworks folgend, können plattformspezifische Anpassungen und Erweiterungen durch die Implementierung von Subklassen und das Definieren von abstrakten Methoden realisiert werden.

Für eine Reihe von Funktionen des Frameworks wurden zwei Implementierungsvarianten vorgesehen: Mit der direkten Implementierung wird die jeweilige Funktion beim Aufruf direkt ausgeführt. Bei der indirekten Implementierung wird dagegen Quelltext generiert, der erst später ausgeführt wird. Mit indirekten Implementierungen können in vielen Fällen hö-

here Geschwindigkeiten erzielt werden, beispielsweise wenn Interceptoren im Gegensatz zur direkten Implementierung ohne die Verwendung von Reflection realisiert werden können.

Validierung: Unterstützung verschiedener Komponentenplattformen und Fallstudien Das Verfahren zur Entwicklung von Modelltransformationen wurde zur Implementierung von Laufzeitunterstützungen für drei verschiedene Komponentenplattformen angewendet: EJB, JavaBeans und Microsoft COM. Als Spezialfall wurde auch eine Unterstützung auf Basis von Web-Services realisiert. Auch wenn es sich dabei nicht um eine Komponentenplattform handelt, besitzen Web-Services dennoch eine Reihe von Gemeinsamkeiten mit Komponentenplattformen.

Für jede einzelne Komponentenplattform und für Web-Services wurden zunächst die Eigenschaften entsprechend der entwickelten Klassifikation ermittelt. Darauf aufbauend und unter Berücksichtigung der Implementierungsentscheidungen wurde das generische Framework zur Laufzeitunterstützung um plattformspezifische Funktionen erweitert. Alle Implementierungen wurden mit einer einfachen Beispielanwendung in Form einer Kryptographiekomponente getestet. Damit wurde die praktische Anwendbarkeit von Adaptierbaren Komponenten demonstriert.

Anhand von zwei Fallstudien wurde außerdem untersucht, ob sich die Modellierungskonzepte von Adaptierbaren Komponenten auch für komplexe Beispiele anwenden lassen. In der ersten Fallstudie wurde der Open-Source Servlet-Container Apache Tomcat analysiert und als Adaptierbare Komponente modelliert. Dabei wurde die bisherige Konfigurationsbeschreibung in Form einer XML-Datei durch Parameter der Tomcat-Komponente ersetzt, womit Implementierungsdetails transparent für den Anwender bleiben. Für den Entwickler entfällt die manuelle Entwicklung der Programmlogik zur Auswertung der Konfigurationsdatei und zur Unterstützung von Rekonfigurationen zur Laufzeit.

In der zweiten Fallstudie wurde eine komponentenbasierte Videoplayer-Anwendung untersucht, die bestimmte QoS-Anforderungen erfüllt. Dieses Anwendungsbeispiel und die grundsätzliche Betrachtung von Komponenten mit nichtfunktionalen Eigenschaften waren Gegenstand der Forschungsgruppe ComQuad an der TU Dresden. In der Fallstudie konnte anhand einer Videodecoder-Komponente gezeigt werden, dass sich Adaptierbare Komponenten zur Modellierung und Entwicklung von Komponenten mit mehreren QoS-Profilen eignen. Dieses Konzept wurde als Adaptionsstrategie für Schwankungen in der Verfügbarkeit von Systemressourcen im Rahmen von ComQuad entwickelt.

6.2 Ausblick

Die in der vorliegenden Arbeit entwickelten Adaptierbaren Komponenten bieten eine Reihe von Anknüpfungspunkten für weiterführende Arbeiten. Verschiedene Ideen für Erweiterungen und Verbesserungen wurden bereits bei der Bearbeitung identifiziert, konnten jedoch aufgrund der Komplexität nicht mehr behandelt werden. Zu unterscheiden sind dabei Detailverbesserungen an einzelnen Konzepten und die Integration völlig neuer Aspekte, die bisher bei der Entwicklung nicht berücksichtigt wurden.

In einem nächsten Schritt können zusätzliche Komponentenplattformen wie Microsoft .NET und OSGi unterstützt werden, um den Anwendungsbereich Adaptierbarer Komponenten weiter zu vergrößern. Es wäre weiterhin wünschenswert, zusätzliche Erfahrungen durch den prak-

tischen Einsatz in verschiedenen Anwendungsbereichen zu gewinnen. Insbesondere für OSGi wäre zu prüfen, ob sich Adaptierbare Komponenten auch im Bereich von Embedded Systems einsetzen lassen.

Wie bereits in Unterabschnitt 3.4.7 angedeutet, können verbesserte Mechanismen für die Beschreibung und Steuerung der Selbstadaptivität entwickelt werden. Hierzu können insbesondere Forschungsergebnisse aus dem Bereich der Self-Managed Systeme (siehe Abschnitt 2.3) integriert werden.

Größerer Forschungsbedarf besteht für die Berücksichtigung von Verhaltensspezifikationen bei der Entwicklung von Adaptierbaren Komponenten. Anknüpfungspunkte bieten hierfür Forschungsergebnisse aus verschiedenen Bereichen: ADLs wie Wright [All97] und Rapide [Luc96]), Komponentenplattformen wie SOFA [Pla05] und formalen Methoden wie Zustandsautomaten [BS03], die Z-Notation [WD96], Prozess-Algebren (z. B. CSP [Hoa78], π-Calculus [Mil89]) und temporale Logik [Lam94, MP92]. Entscheidend ist die richtige Wahl des Abstraktionsniveaus und Umfangs der Verhaltensspezifikationen, um einerseits die Komponentenentwicklung nicht unnötig zu erschweren, andererseits aber den Nutzen durch zusätzliche Überprüfungen zu maximieren.

Es ist vorstellbar, Verhaltensspezifikationen in einigen Bereichen der Entwicklung von Adaptierbaren Komponenten einzusetzen: Bei der Definition von Konfigurationen und Variationen könnte überprüft werden, ob die entsprechenden Komponenten aufgrund ihres Verhaltens zueinander passen. Momentan werden nur Methodensignaturen für die Bestimmung der Kompatibilität von Komponenten bzw. Komponentenports verwendet. Beim Austausch von zustandsbehafteten Komponenten im Rahmen von Rekonfigurationen könnte überprüft werden, ob die übertragenen Zustandsinformationen semantisch kompatibel zueinander sind. Schließlich könnte die Entwicklung von Adaptionsoperatoren automatisiert oder teilweise auf Korrektheit überprüft werden.

Ein letzter Bereich für weiterführende Arbeiten, der in diesem Ausblick erwähnt werden soll, ist die Weiterentwicklung der Werkzeugunterstützung für Adaptierbare Komponenten. Die verschiedenen benötigten Werkzeuge sollten dabei möglichst in eine vorhandene Entwicklungsumgebung wie z. B. Eclipse integriert werden, um damit die Entwicklung weiter zu vereinfachen.

Anhang A

Performance-Messungen

Alle Messungen wurden auf einem Intel Pentium M mit 1,6 GHz, 1 GB Hauptspeicher, Windows XP SP1 und Java Version 1.5.0_04-b05 durchgeführt.

A.1 Vergleich des Zeitbedarfs für verschiedene Verfahren des Methodenaufrufs

Im Framework zur Laufzeitunterstützung (siehe Abschnitt 4.3) wurden mehrere Verfahren für Methodenaufrufe implementiert. Mit den Messungen soll belegt werden, dass sich durch einen Verzicht auf Flexibilität und durch zusätzlichen Implementierungsaufwand für die Codeerzeugung eine höhere Leistung erreicht werden kann, die den Aufwand damit rechtfertigt.

Für die Messungen wurden 1000 Objekte einer Testklasse erzeugt, die verschiedenen Methoden unterschiedlichen Signaturen enthält. Anschließend wurde die Zeit für insgesamt 1000 Methodenaufrufe gemessen, je einer bei einem bestimmten Objekt. Die Methodenaufrufe wurden auf drei unterschiedlichen Wegen durchgeführt:

- Direkter Methodenaufruf (`object.method()`)
- Indirekter Methodenaufruf über einen generierten Adapter, der die gleiche Schnittstelle wie die Testklasse implementiert
- Direkter Methodenaufruf mit Reflection (`Method.invoke()`), wobei die Zeit über das Ermitteln des `Method`-Objektes aus der Testklasse nicht berücksichtigt wird
- Indirekter Methodenaufruf unter Nutzung eines Dynamisches Proxies, dessen `InvocationHandler` die Methode der Testklasse aufruft

Die folgende Tabelle enthält die Zeiten (Maximalwert, Minimalwert) in ms für jeweils 1000 Methodenaufrufe. Durch die Hotspot-Optimierungen der Java Virtual Machine nimmt der Zeitbedarf ab, wenn die Messungen mehrmals wiederholt werden. Ab der dritten Wiederholung stabilisiert sich die gemessene Zeit auf dem Minimalwert.

Verfahren	void m()	void m(int)	void m(String)	int m(int)	String m(String)
Direkt	115 – 18	110 – 15	130 – 18	125 – 18	145 – 18
Adapter	325 – 27	310 – 25	335 – 28	350 – 28	350 – 28
Reflection	2100 – 145	3200 – 200	2000 – 155	2400 – 230	4000 – 160
Dyn. Proxy	5400 – 150	8200 – 220	5500 – 180	7400 – 240	3600 – 185

Unabhängig von den absoluten Werten werden direkte Methodenaufrufe 2–3 mal schneller als Methodenaufrufe über einen generierten Adapter, 8–30 mal schneller als Methodenaufrufe mit Reflection und 8-75 mal schneller als Methodenaufrufe über dynamische Proxies durchgeführt. Die Methodensignatur hat dagegen nur einen geringen Einfluss auf die Ausführungszeiten von Methodenaufrufen und kann daher vernachlässigt werden.

A.2 Vergleich verschiedener Implementierungsverfahren für Interceptoren

Mit den Messungen wird der Zeitbedarf für die verschiedenen Implementierungsverfahren aus Unterabschnitt 4.3.6 anhand der der Kryptographiekomponente ermittelt (jeweils 100 Methodenaufrufe). Dabei wird zwischen dem Zeitbedarf im Normalfall ohne eine aktive Blockierung und im Falle einer Rekonfiguration unterschieden. Entscheidend für die Geschwindigkeit von Anwendungen ist der Zeitbedarf für Methodenaufrufe bei Interceptoren ohne aktive Blockierung, da Rekonfiguration im Vergleich zu den sonstigen Zeiten sehr selten auftreten.

Verfahren	**Nicht Blockiert**	**Blockiert**
Adapter & Stack-Trace-Analyse	430	1050
Adapter & Threadverwaltung	590	710
Dyn. Proxy & Stack-Trace-Analyse	1280	1830
Dyn. Proxy & Threadverwaltung	1440	1560

Man erkennt, dass die Stack-Trace-Analyse im Normalfall wesentlich schneller, als die Threadverwaltung arbeitet. Dafür wird im Blockierungsfall mehr Zeit für das Erzeugen des Stack-Trace und dessen Auswertung benötigt. Durch die zusätzliche Programmlogik sind die Unterschiede zwischen dynamic Proxy und dem generierten Adapter nicht mehr gravierent wie bei den Messungen in Abschnitt A.1.

Anhang B

Metamodell für Adaptierbare Komponenten

In diesem Anhang werden die Klassen des Metamodells für Adaptierbare Komponenten vorgestellt. Das Metamodell dient zur Speicherung der Modellierungsergebnisse aus Kapitel 3 und wurde auch zur Erzeugung der Java-Klassen des Metamodell-Paketes (siehe Unterabschnitt 4.3.1) mit Hilfe von EMF [emf05] verwendet.

Für jede Klasse wird im Folgenden kurz die Bedeutung sowie die enthaltenen Membervariablen und Assoziationen beschrieben. Die meisten Namen sind dabei selbsterklärend.

AbstractComponentInstance

Abstrakte Klasse. Die Klasse beschreibt eine Komponenteninstanz, die innerhalb von (Template-)Konfigurationen und (Template-)Variationen verwendet wird. Sie besitzt einen eindeutigen Namen im Kontext der Adaptierbaren Komponente.

Membervariablen
`id : String`

AdaptableComponent

Subtyp von `Component`. Die Klasse ist der Einstiegspunkt in das Metamodell, d. h. alle anderen Klassen sind von hier direkt oder indirekt durch Assoziationen erreichbar. Die Klasse beschreibt eine Instanz einer Adaptierbaren Komponente, die durch das Tupel *AdKomponente* (siehe Definition 3.1) definiert ist. Sie enthält in Form von Kompositionsbeziehungen alle Konfigurationen, Variationen, Komponententypen, Komponenteninstanzen und Regeln zur Parameterabbildung und Selbstadaption, die in der Adaptierbaren Komponente verwendet werden.

Kompositionen
`configurations : Configuration[1..*]`
`variations : Variation[*]`
`componentInstances : ComponentInstance[1..*]`
`parameterMappings : ParameterMapping[*]`
`components : Component[1..*]`
`variations : Variation[*]`
`contextModel : ContextModel[0..1]`
`contextRules : ContextRule[*]`
Assoziationen
`defaultConfiguration : Configuration[1]`

`defaultVariations : Variation[*]`

AdaptationOperator

Subtyp von `Component`. Die Klasse beschreibt einen Adaptionsoperator, der durch die Funktion *AdOp* (siehe Definition 3.13) definiert ist.

Membervariablen
`implementation : Class`
Assoziationen
`target : BasicComponent[1]`

AspectOperator

Subtyp von `Component`. Die Klasse beschreibt einen Aspektoperator, der durch die Funktion *AsOp* (siehe Definition 3.14) definiert ist.

Membervariablen
`implementation : Class`
Assoziationen
`target : BasicComponent[1]`

BasicComponent

Subtyp von `Component`. Die Klasse beschreibt einen atomaren Komponententyp, der als Subkomponente innerhalb einer Adaptierbaren Komponente verwendet wird.

Membervariablen
`implementation : Class`

Component

Abstrakte Basisklasse für alle Komponententypen. Jeder Komponententyp besitzt einen eindeutigen Namen innerhalb der Adaptierbaren Komponente sowie Mengen von Komponentenparametern und Ports.

Membervariablen
`id : String`
Kompositionen
`parameters : Parameter[*]`
`providedPorts : Port[1..*]`
`requiredPorts : Port[*]`

ComponentInstance

Subtyp von `AbstractComponentInstance`. Die Klasse beschreibt eine Komponenteninstanz eines bestimmten Komponententyps innerhalb einer Adaptierbaren Komponente.

Membervariablen
`reference : Object`
`interceptorReference : Object`
Assoziationen
`component : Component`

Configuration

Die Klasse beschreibt eine explizite Konfiguration, die durch das Tupel *VK* (siehe Definition 3.19) definiert ist. Sie besitzt einen eindeutigen Namen und enthält eine Menge von Komponenteninstanzen, die durch Verbindungen verknüpft werden.

Membervariablen
`id : String`
Kompositionen
`connections : Connection[*]`
Assoziationen
`componentInstances : ComponentInstance[1..*]`

ConfigurationMapping

Subtyp von `ParameterMapping`. Diese Klasse beschreibt die Auswahl einer bestimmten Konfiguration bei einem bestimmten Parameterwert der Adaptierbaren Komponente.

Membervariablen
`parameterValue : Object`
Assoziationen
`configuration : Configuration[1]`
`parameter : Parameter[1]`

Connection

Abstrakte Klasse. Die Klasse beschreibt eine Verbindung zwischen Subkomponenten oder zwischen internen und externen Ports.

Membervariablen
`id : String`

ContextInformation

Die Klasse beschreibt eine einzelne Kontext-Information des Kontextmodells, die durch ein Tupel *Kontext* (siehe Definition 3.15) definiert ist.

Membervariablen
`id : String`
`type : Class`
`contextSource : Class`

ContextModel

Die Klasse beschreibt das Kontextmodell der Adaptierbaren Komponente, bestehend aus einer Menge von Kontextinformationen.

Kompositionen
`information : ContextInformation[*]`

ContextRule

Die Klasse beschreibt eine Regel zur Selbstadaption. Wenn eine bestimmte Bedingung eines Kontextwertes erfüllt ist, wird der Wert eines Parameters der Adaptierbaren Komponente geändert.

Membervariablen
`kontextValue : Object`
`parameterValue : Object`
Assoziationen
`kontext : ContextInformation[1]`
`parameter : Parameter[1]`

GlueCode

Subtyp von `Component`. Die Klasse beschreibt die Implementierung von Glue-Code (siehe Unterabschnitt 3.3.2).

Membervariablen
`implementation : Class`

MethodMapping

Subtyp von `Connection`. Die Klasse beschreibt eine Methoden-Verknüpfung zwischen einer Methoden eines externen *provided*-Ports mit einer Methode eines internen *provided*-Ports, die durch das Tupel *Verknuepfung* (siehe Definition 3.12) definiert ist.

Membervariablen
`externSignature : String`
`internSignature : String`
Assoziationen
`externPort : Port[1]`
`internInstance : ComponentInstance[1]`
`internPort : Port[1]`

Parameter

Die Klasse beschreibt einen Komponentenparameter, der durch das Tupel *Parameter* (siehe Definition 3.7) definiert ist. Jeder Parameter besitzt einen eindeutigen Namen, einen Typ und optional einen Standardwert.

Membervariablen
`name : String`
`type : Class`
`default : Object`

ParameterConnection

Subtyp von `Connection`. Die Klasse beschreibt eine Verbindung zwischen einem Parameter der Adaptierbaren Komponente und einem Parameter einer Subkomponenteninstanz.

Assoziationen
`sourceParameter : Parameter[1]`
`targetInstance : ComponentInstance[1]`
`targetParameter : Parameter[1]`

ParameterMapping

Abstrakte Klasse. Die Klasse beschreibt die Parameterabbildung, die durch das Tupel *Mapping* (siehe Definition 3.26) definiert ist.

Assoziationen
`parameter : Parameter[1]`

Port

Die Klasse beschreibt einen Komponentenport, der durch das Tupel *Port* (siehe Definition 3.4) definiert ist.

Membervariablen
`name : String`
`kind : PortKind`

`type : Class`

PortConnection

Subtyp von `Connection`. Die Klasse beschreibt eine Verbindung zwischen zwei Ports von Subkomponenten, die durch das Tupel *Verbindung* (siehe Definition 3.10) definiert ist.

Assoziationen
`providedInstance : ComponentInstance[1]`
`providedPort : Port[1]`
`requiredInstance : ComponentInstance[1]`
`requiredPort : Port[1]`

PortKind

Diese Enumeration beschreibt die möglichen Arten von Ports.

PortMapping

Subtyp von `Connection`. Die Klasse beschreibt eine Port-Verknüpfung zwischen einem internen und einem externen Port, die durch das Tupel *Verknuepfung* (siehe Definition 3.11) definiert ist.

Assoziationen
`externPort : Port[1]`
`internInstance : ComponentInstance[1]`
`internPort : Port[1]`

TemplateComponent

Subtyp von `AbstractComponentInstance`. Die Klasse beschreibt eine Komponenteninstanz innerhalb einer Template-Konfiguration oder Template-Variation, die eine Menge von unterschiedlichen Komponententypen repräsentieren kann.

Assoziationen
`components : Component[1..*]`

TemplateConfiguration

Subtyp von `Configuration`. Die Klasse beschreibt eine Konfiguration, die durch das Tupel *TK* (siehe Definition 3.21) definiert ist. Sie enthält mindestens eine Template-Komponente.

Assoziationen
`templates : TemplateComponent[1..*]`

TemplateVariation

Subtyp von `Variation`. Die Klasse beschreibt eine Template-Variation, die durch das Tupel *TVariation* (siehe Definition 3.25) definiert ist.

Assoziationen
`templatesAdded : TemplateComponent[*]`
`templatesRemoved : TemplateComponent[*]`

Variation

Die Klasse beschreibt eine Variation, die durch das Tupel *Variation* (siehe Definition 3.23) definiert ist. Sie enthält eine Menge von Verbindungen und Komponenteninstanzen, die durch die Variation in angegebenen Konfigurationen oder Variationen hinzugefügt bzw. entfernt werden.

Membervariablen
`id : String`
`parameterValues : Object[*]`
Assoziationen
`targetConfiguration : Configuration[*]`
`targetVariation : Variation[*]`
`removedConnection : Connection[*]`
`removedComponents : ComponentInstance[*]`
`addedConnections : Connection[*]`
`addedComponents : ComponentInstance[*]`
`parameters : Parameter[*]`

VariationMapping

Subtyp von `ParameterMapping`. Die Klasse beschreibt die Auswahl von bestimmten Variationen bei einem bestimmten Parameterwert der Adaptierbaren Komponente.

Membervariablen
`parameterValue : Object`
Assoziationen
`variations : Variation[*]`
`parameter : Parameter[1]`

VariationCountMapping

Subtyp von `ParameterMapping`. Die Klasse beschreibt, wie oft eine bestimmte Variation entsprechend des Wertes eines Integer-Komponentenparameters angewendet werden soll.

Assoziationen
`variation : Variation[1]`
`parameter : Parameter[1]`

Literaturverzeichnis

[ABF+03] Aigner, Ronald, Henrike Berthold, Elke Franz, Steffen Göbel, Herrmann Härtig, Heinrich Hussmann, Klaus Meissner, Klaus Mayer-Wegener, Marcus Meyerhöfer, Andreas Pfitzmann, Simone Röttger, Alexander Schill, Thomas Springer und Frank Wehner: *COMQUAD – Komponentenbasierte Softwaresysteme mit zusagbaren quantitativen Eigenschaften und Adapptionsfähigkeit.* Informatik Forschung und Entwicklung, Seiten 39–40, 2003.

[ABV92] Aksit, Mehmet, Lodewijk Bergmans und Sinan Vural: *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach.* In: *6th European Conference on Object-Oriented Programming (ECOOP'92)*, Band 615 der Reihe *LNCS*, Seiten 372–395. Springer-Verlag, 1992.

[ADG98] Allen, Robert, Remi Douence und David Garlan: *Specifying and Analyzing Dynamic Software Architectures.* In: *Conference on Fundamental Approaches to Software Engineering (FASE'98)*, 1998.

[AFG+03] Aigner, Ronald, Elke Franz, Steffen Göbel, Herrmann Härtig, Heinrich Hussmann, Klaus Meissner, Klaus Mayer-Wegener, Marcus Meyerhöfer, Andreas Pfitzmann, Christoph Pohl, Martin Pohlack, Simone Röttger, Alexander Schill, Frank Wehner und Steffen Zschaler: *Zwischenbericht der DFG-Forschergruppe 428 Components with Quentitative Properties and Adaptivity (COMQUAD).* Technischer Bericht TUD-FI03-10, Technische Universität Dresden, Fakultät Informatik, August 2003.

[AHP94] Agnew, Brent, Christine Hofmeister und James Purtilo: *Planning for change: a reconfiguration language for distributed systems.* In: *Proceedings of the Second International Workshop on Configurable Distributed Systems*, Seiten 313–322. IEEE Computer Society Press, 1994.

[Ald03] Aldrich, Jonathan: *Using Types to Enforce Architectural Structure.* Doktorarbeit, University of Washington, August 2003.

[All97] Allen, Robert: *A Formal Approach to Software Architecture.* Doktorarbeit, Carnegie Mellon, School of Computer Science, January 1997.

[AM03a] Aguirre, Nazareno und Tom Maibaum: *A Logical Basis for the Specification of Reconfigurable Component-Based Systems.* In: *FASE 2003*, Band 2621 der Reihe *LNCS*, Seiten 37–51. Springer-Verlag, 2003.

[Aßm03b] Assmann, Uwe: *Invasive Software Composition.* Springer-Verlag, 2003.

[Apa05] Apache Software Foundation: *Apache Web Services Project: Axis*, 2005. http://ws.apache.org/axis.

[BA01] Bergmans, Lodewijk und Mehmet Aksit: *Composing Crosscutting Concerns using Composition Filters*. Communications of the ACM, 44(10):51–57, Oct. 2001.

[BC90] Bracha, Gilad und William Cook: *Mixin-Based Inheritance*. In: Meyrowitz, Norman (Herausgeber): *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, Seiten 303–311. ACM Press, 1990.

[BCDW04] Bradbury, Jeremy S., James R. Cordy, Juergen Dingel und Michel Wermelinger: *A survey of self-management in dynamic software architecture specifications*. In: *ACM SIGSOFT workshop on Self-managed systems (WOSS'04)*, Seiten 28–33. ACM Press, 2004.

[bce05] Apache Software Foundation: *Byte Code Engineering Library*, 2005. http://jakarta.apache.org/bcel/.

[BCRP98] Blair, Gordon S., G. Coulson, P. Robin und M. Papathomas: *An Architecture for Next Generation Middleware*. In: *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[BCS02] Bruneton, Eric, Thierry Coupaye und Jean-Bernard Stefani: *Recursive and Dynamic Software Composition with Sharing*. In: *Workshop on Component-Oriented Programming (WCOP'2002)*, 2002.

[BHH+05] Baumeister, Hubert, Florian Hacklinger, Rolf Hennicker, Alexander Knapp und Martin Wirsing: *A Component Model for Architectural Programming*. In: *Proceedings of the 2nd International Workshop Formal Aspects of Component Software (FACS'05)*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 2005.

[Bos99] Bosch, Jan: *Superimposition: A Component Adaptation Technique*. Information and Software Technology, 41(5):257–273, March 1999.

[BS03] Börger, Egon und Robert Stärk: *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[CBCP02] Coulson, Geoff, Gordon S. Blair, Michael Clarke und Nikos Parlavantzas: *The design of a configurable and reconfigurable middleware platform*. Distributed Computing, 15(2):109–126, 2002.

[CCK98] Cohen, Avron, Jeffrey S. Chase und David L. Kaminsky: *Automatic Program Transformation with JOIE*. In: *1998 USENIX Annual Technical Symposium*, Seiten 167–178, 1998.

[CCMW01] Christensen, Erik, Francisco Curbera, Greg Meredith und Sanjiva Weerawarana: *Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001.* W3C Consortium, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[CD01] Cheesman, John und John Daniels: *UML Components: A Simple Process for Specifying Component-Based Software.* Addison Wesley Longman, Inc., 2001.

[CE00] Czarnecki, Krzysztof und Ulrich Eisenecker: *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000.

[CES71] Coffman, E. G., M. Elphick und A. Shoshani: *System Deadlocks.* ACM Computing Surveys, 3(2):67–78, 1971.

[CHE04] Czarnecki, Simon Helsen und Ulrich Eisenecker: *Staged Configuration Using Feature Models.* In: *Proceedings of Software Product Line Conference 2004*, Band 3154 der Reihe *LNCS*, Seiten 266–283. Springer-Verlag, 2004.

[Chi95] Chiba, Shigeru: *A metaobject protocol for C++.* In: *Proceedings of the tenth annual conference on Object-oriented programming systems languages, and applications (OOPSLA'95)*, Seiten 285–299, New York, NY, USA, 1995. ACM Press.

[Chi00] Chiba, Shigeru: *Load-time Structural Reflection in Java.* In: *European Conference on Object-Oriented Programming (ECOOP'2000)*, Band 1850 der Reihe *LNCS*, Seiten 313–336. Springer-Verlag, 2000.

[Cox90] Cox, Brad J.: *Planning the software industrial revolution.* IEEE Software, 7(6):25–33, Nov. 1990.

[CST03] Chiba, Shigeru, Yoshiki Sato und Michiaki Tatsubori: *Using HotSwap for Implementing Dynamic AOP Systems.* In: *ECOOP'03 Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI)*, July 2003.

[Dah01] Dahm, Markus: *Byte Code Engineering with the BCEL API.* Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.

[DeM03] DeMichiel, Linda G.: *Enterprise JavaBeans Specification, Version 2.1.* Sun Microsystems, Inc., 2003.

[Deu96] Deutsch, Peter: *DEFLATE Compressed Data Format Specification version 1.3 (RFC 1951).* IETF, 1996. http://www.ietf.org/rfc/rfc1951.txt.

[Dij76] Dijkstra, Edsger Wybe: *A Discipline of Programming.* Prentice Hall, 1976.

[DR02] Daemen, Joan und Vincent Rijmen: *The design of Rijndael - AES - the Advanced Encryption Standard ; with 17 tables.* Information security and cryptography. Springer-Verlag, 2002.

[DvdHT02a] Dashofy, Eric M., André van der Hoek und Richard N. Taylor: *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages.* In: *24th International Conference on Software Engineering (ICSE'2002)*, 2002.

[DvdHT02b] Dashofy, Eric M., André van der Hoek und Richard N. Taylor: *Towards Architecture-based Self-Healing Systems.* In: *Workshop on Self-Healing Systems (WOSS '02)*, Seiten 21–26, 2002.

[dyn99] Sun Microsystems: *Dynamic Proxy Classes*, 1999. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[emf05] Eclipse Foundation: *Eclipse Modeling Framework (EMF)*, 2005. http://eclipse.org/emf/.

[End94] Endler, Markus: *A Language for Implementing Generic Dynamic Reconfigurations of Distributed Programs.* In: *12th Brazilian Symposium on Computer Networks (SBRC 12)*, 1994.

[Fei05] Feingold, Max: *Web Services Coordination (WS-Cordination), Version 1.0*, 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[FR03] Fleury, Marc und Francisco Reverbel: *The JBoss Extensible Server.* In: Endler, Markus und Douglas Schmidt (Herausgeber): *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, Band 2672 der Reihe *LNCS*, Seiten 344–373. Springer-Verlag, 2003.

[Fra03] Frankel, J.: *Model Driven Architecture – Applying MDA to Enterprise Computing.* John Wiley & Sons, 2003.

[GH05] Gudgin, Martin und Marc Hadley: *Web Services Addressing 1.0 - Core, W3C Candidate Recommendation.* W3C Consortium, August 2005. http://www.w3.org/TR/2005/CR-ws-addr-core-20050817.

[GHJV95] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[GMK02] Georgiadis, Ioannis, Jeff Magee und Jeff Kramer: *Self-Organising Software Architectures for Distributed Systems.* In: *Workshop on Self-Healing Systems (WOSS '02)*, Seiten 33–38, 2002.

[GMW97] Garlan, David, Robert Monroe und Dave Wile: *ACME: An Architecture Description Interchange Language.* In: *Proceedings of CASCON'97*, Seiten 169–183, 1997.

[GN04] Göbel, Steffen und Michael Nestler: *Composite Component Support for EJB.* In: *Winter International Symposium on Information and Communication Technologies (WISICT'04).* Trinity College Dublin, 2004.

[Göb04] Göbel, Steffen: *Encapsulation of Structural Adaptation by Composite Components.* In: *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04).* ACM Press, Oct 2004.

[Gom04] Gomaa, Hassan: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Addison-Wesley Professional, 2004.

[GPA+04a] Göbel, Steffen, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger und Steffen Zschaler: *The COMQUAD component container architecture and contract negotiation.* Technical Report TUD-FI04-04, TU Dresden, Fakultät Informatik, April 2004.

[GPA+04b] Göbel, S., C. Pohl, R. Aigner, M. Pohlack, S. Röttger und S. Zschaler: *The COMQUAD component container architecture.* In: Magee, Jeff, Clemens Szyperski und Jan Bosch (Herausgeber): *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, Seiten 315–318. IEEE, Juni 2004.

[GPA+04c] Göbel, Steffen, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger und Steffen Zschaler: *The COMQUAD Component Container Architecture.* In: *4th IEEE/IFIP Working Conf. on Software Architecture (WICSA-4).* IEEE Computer Society, Juni 2004.

[GPRZ04] Göbel, Steffen, Christoph Pohl, Simone Röttger und Steffen Zschaler: *The COMQUAD Component Model—Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects.* In: *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Seiten 74–82. ACM Press, 22–26 März 2004.

[GR91] Gorlick, Michael M. und Rami R. Razouk: *Using Weaves for Software Construction and Analysis.* In: *Proceedings of the 13th International Conference on Software Engineering (ICSE'96)*, Seiten 23–34, 1991.

[GS04] Greenfield, Jack und Keith Short: *Software Factories: Assembling Aplications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[Ham97] Hamilton, Graham: *JavaBeans, Version 1.01.* Sun Microsystems, Inc., 1997.

[HBB+98] Härtig, H., R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg und J. Wolter: *DROPS: OS Support for Distributed Multimedia Applications.* In: *8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, September 1998.

[HHW98] Härtig, Hermann, Michael Hohmuth und Jean Wolter: *Taming Linux.* In: *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, September 1998.

[HIM99] Hirsch, Dan, Paola Inverardi und Ugo Montanari: *Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving.* In: *First Working IFIP Conference on Software Architecture*, Seiten 127–142. Kluwer Academic Publishers, Feb. 1999.

[Hir03] Hirsch, Dan: *Graph Transformation Models for Software Architecture Styles*. Doktorarbeit, Dept. of Computer Science, Universidad de Buenos Aires, May 2003. http://www.di.unipi.it/~dhirsch/thesis.pdf.

[HNS00] Hofmeister, Christine, Robert Nord und Dilip Soni: *Applied Software Architecture*. Addison-Wesley, 2000.

[Hoa78] Hoare, C.A.R.: *Communicating Sequential Processes*. Communications of the ACM, 21(8):666–677, 1978.

[HR83] Haerder, Theo und Andreas Reuter: *Principles of transaction-oriented database recovery*. ACM Computing Surveys, 15(4):287–317, Dec. 1983.

[IJ97] Ivar Jacobson, Martin Griss, Patrik Jonsson: *Software Reuse : Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997.

[IW95] Inverardi, Paola und Alexander L. Wolf: *Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model*. IEEE Transactions on Software Engineering, 21(4):373–386, 1995.

[JAN05] Jonathan Aldrich, Craig Chambers und David Notkin: *ArchJava: Connecting Software Architecture to Implementation*. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE'2002)*, Seiten 187–197, New York, NY, USA, 2005. ACM Press.

[JDL02] Jarir, Zahi, Pierre-Charles David und Thomas Ledoux: *Dynamic Adaptability of Services in Enterprise JavaBeans Architecture*. In: *Workshop on Component-Oriented Programming (WCOP 2002)*, Juni 2002.

[JMX02] Sun Microsystems: *Java Management Extensions Specification (JMX) v1.2*, 2002. http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html.

[JSP03] Sun Microsystems: *JavaServer Pages 2.0 Specification*, 2003. http://jcp.org/aboutJava/communityprocess/final/jsr152/.

[JT03] Jørgensen, Bo Nørregaard und Eddy Truyen: *Evolution of collective object behavior in presence of simultaneous client-specific views*. In: *9th International Conference on Object-Oriented Information Systems (OOIS 2003)*, Band 2817 der Reihe *LNCS*, Seiten 18–32. Springer-Verlag, 2003.

[KCA01] Kniesel, Günter, Pascal Constanza und Michael Austermann: *JMangler - A Framework for Load-Time Transformation of Java Class Files*. In: *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2001.

[KCH+90] Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak und A.Spencer Peterson: *Byte Code Engineering with the BCEL API*. Technischer Bericht CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute, 1990.

[KH98] Keller, Ralph und Urs Hölzle: *Binary Component Adaptation.* In: *European Conference on Object-Oriented Programming (ECOOP'98)*, Band 1445 der Reihe *LNCS*. Springer, 1998.

[KHH+01] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold: *An Overview of AspectJ.* In: *European Conference on Object-Oriented Programming (ECOOP'01)*, Band 2072 der Reihe *LNCS*, Seiten 327–355. Springer-Verlag, 2001.

[KLM+97] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier und John Irwin: *Aspect-Oriented Programming.* In: *11th European Conference on Object-Oriented Programming (ECOOP'97)*, Band 1241 der Reihe *LNCS*, Seiten 220–242. Springer-Verlag, 1997.

[KM90] Kramer, Jeff und Jeff Magee: *The Evolving Philosophers Problem: Dynamic Change Management.* IEEE Transactions on Software Engineering, 16(11):1293–1306, 1990.

[KRB91] Kiczales, Gregor, Jim des Rivieres und Daniel G. Bobrow: *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KT02] Kalibera, T. und P. Tuma: *Distributed Component System Based On Architecture Description: The SOFA Experience.* In: *International Symposium on Distributed Objects and Applications (DOA 2002)*, Band 2519 der Reihe *LNCS*, Seiten 981–994. Springer-Verlag, Oct 2002.

[Lam94] Lamport, Leslie: *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.

[LB98] Liang, Sheng und Gilad Bracha: *Dynamic class loading in the Java virtual machine.* In: *Conference on Object-oriented programming systems languages, and applications (OOPSLA'98)*, Seiten 36–44, 1998.

[LHR01] Löser, Jork, Hermann Härtig und Lars Reuther: *A Streaming Interface for Real-Time Interprocess Communication.* Technical Report TUD-FI01-09, TU Dresden, Fakultät Informatik, August 2001.

[Lie96] Lieberherr, Karl J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[LLM99] Lieberherr, Karl, David Lorenz und Mira Mezini: *Programming with Aspectual Components.* Technischer Bericht NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, 1999.

[LLO03] Lieberherr, Karl, David H. Lorenz und Johan Ovlinger: *Aspectual Collaborations: Combining Modules and Aspects.* The Computer Journal, 46(5):542–565, Sep 2003.

[LN03a] Little, Mark und Eric Newcomer: *Web Services Coordination Framework (WS-CAF), Version 1.0*, 2003. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf.

[LN03b] Little, Mark und Eric Newcomer: *WS-Context*, 2003. http://developers.sun.com/techtopics/webservices/wscaf/wsctx.pdf.

[LSNA97] Lumpe, Markus, Jean-Guy Schneider, Oscar Nierstrasz und Franz Achermann: *Towards a formal composition language*. In: *Workshop on Foundations of Component-Based Systems (ESEC'97)*, Seiten 178–187, 1997.

[Luc96] Luckham, David C.: *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events*. In: *DIMACS Partial Order Methods Workshop IV*. Princeton University, 1996.

[LW93] Liskov, Barbara und Jeannette M. Wing: *Family Values: A Behavioral Notion of Subtyping*. Technischer Bericht MIT/LCS/TR-562b, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[Mar02] Marinescu, Floyd: *EJB Design Patterns. Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, 2002.

[MB04] Mellor, Stephen und Marc Balcer: *MDA Distilled, Principles of Model Driven Architecture*. Addison-Wesley Professional, 2004.

[McI68] McIlroy, M. Douglas: *Mass produced softare components*. In: *Report on the NATO Software Engineering Conference*, Seiten 138–150, 1968.

[mda03] Object Management Group(OMG): *MDA Guide Version 1.0.1*, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[MDEK95] Magee, Jeff, Naranker Dulay, Susan Eisenbach und Jeff Krammer: *Specifying Distributed Software Architectures*. In: *5th European Software Engineering Conference (ESEC'95)*, Band 989 der Reihe *LNCS*, Seiten 137–153. Springer-Verlag, 1995.

[Med96] Medvidovic, Nenad: *ADLs and dynamic architecture changes*. In: *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, Seiten 24–27, New York, NY, USA, 1996. ACM Press.

[MHS05] Mernik, Marjan, Jan Heering und Anthony M. Sloane: *When and how to develop domain-specific languages*. ACM Computing Surveys, 37(4):316–344, 2005.

[Mic05] Microsoft Corporation: *Microsoft .NET*, 2005. http://www.microsoft.com/net.

[Mil80] Milner, Robin: *A Calculus of Communicating Systems*, Band 92 der Reihe *LNCS*. Springer-Verlag, 1980.

[Mil89] Milner, Robin: *A Calculus of mobile processes, part I+II.* Technischer Bericht ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, 1989.

[Mil00] Milstein, Dan: *Apache JServ Protocol version 1.3*, December 2000. http://tomcat.apache.org/tomcat-3.3-doc/AJPv13.html.

[Mit03] Mitra, Nilo: *SOAP Version 1.2 Part 0: Primer, W3C Recommendation.* W3C Consortium, 2003. http://www.w3.org/TR/soap12-part0/.

[MJ02] Mellor, Stephen J. und Marc J.Balcer: *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, 2002.

[MK96] Magee, Jeff und Jeff Kramer: *Dynamic structure in software architectures.* In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'96)*, Seiten 3–14, New York, NY, USA, 1996. ACM Press.

[MKS89] Magee, Jeff, Jeff Kramer und Morris Slomann: *Constructing Distributed Systems in CONIC.* IEEE Transactions on Software Engineering, 15(6):663–675, 1989.

[ML98] Mezini, Mira und Karl Lieberherr: *Adaptive Plug-and-Play Components for Evolutionary Software Development.* In: *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, Seiten 97–116. ACM Press, 1998.

[MN04] Meyerhöfer, Marcus und Christoph Neumann: *TESTEJB – A Measurement Framework for EJBs.* In: Crnkovic, Ivica, Judith A. Stafford, Heinz W. Schmidt und Kurt Wallnau (Herausgeber): *Proc. 7th Intl. Symposium on Component-Based Software Engineering (CBSE'04)*, Nummer 3054 in *LNCS*, Seiten 294–301. Springer, 2004.

[MOF06] Open Management Group (OMG): *Meta Object Facility Core Specification version 2.0*, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[MORT96] Medvidovic, Nenad, Peyman Oreizy, Jason E. Robbins und Richard N. Taylor: *Using Object-Oriented Typing to Support Architectural Design in the C2 Style.* In: *Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, Seiten 24–32, 1996.

[MP92] Manna, Zohar und Amir Pnueli: *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[Mét96] Métayer, Daniel Le: *Software architecture styles as graph grammars.* In: *ACM SIGSOFT Symposium on the Foundation of Software Engineering.* ACM Press, 1996.

[Mét98] Métayer, Daniel Le: *Describing Software Architecture Styles Using Graph Grammars.* IEEE Transactions on Software Engineering, 24(7):521–533, 1998.

[MT00] Medvidovic, Nenad und Richard N. Taylor: *A Classification and Comparison Framework for Software Architecture Description Languages*. Transactions on Software Engineering, 26(1):70–93, 2000.

[Nes03] Nestler, Michael: *Design and Evaluation of Application Server Support for Composite Components*. Diplomarbeit, TU Dresden, 2003.

[NJ04] Nørregaard Jørgensen, Bo: *Enhancing Java with Support for Simultaneous Independent Extensibility of Collaborating Objects*. In: *IASTED International Conference Software Engeneering*, 2004.

[OCL05] Open Management Group (OMG): *OCL 2.0 Specification, Version 2.0*, 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-06-06.

[OGT+99] Oreizy, Peyman, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum und Alexander L. Wolf: *An Architecture-Based Approach to Self-Adaptive Software*. IEEE Intelligent Systems, 14(3):54–62, 1999.

[OLKM00] Ommering, Rob van, Frank van der Linden, Jeff Kramer und Jeff Magee: *The Koala Component Model for Consumer Electronics Software*. IEEE Computer, Seiten 78–85, 2000.

[Par76] Parnas, David L.: *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, SE-2(1):1–9, 1976.

[PBJ98] Plasil, Frantisek, Dusan Balek und Radovan Janecek: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. In: *International Conference on Configurable Distributed Systems (ICCDS'98)*, May 1998.

[Pla05] Plasil, Frantisek: *Enhancing Component Specification by Behavior Description - the SOFA Experience*. In: *4th International Symposium on Information and Communication Technologies (WISICT 2005)*, Band 92 der Reihe *ACM international conference proceedings series*, Seiten 185–190. Trinity College Dublin, Jan 2005.

[qvt05] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Final Adopted Specification*, November 2005.

[RAC+02] Rutherford, Matthew J., Kenneth Anderson, Antonio Carzaniga, Dennis Heimbigner und Alexander L. Wolf: *Reconfiguration in the Enterprise JavaBean Component Model*. In: *Component Deployment (CD'2002)*, Band 2370 der Reihe *LNCS*, Seiten 67–81. Springer-Verlag, 2002.

[rec06] *RECODER*, 2006. http://recoder.sf.net/.

[Ric04] Richter, Mirko: *Adaption zustandsbehafteter Komponenten*. Großer Beleg, TU Dresden, 2004.

[Rie03] Rietzschel, Carsten: *VERNER - ein Video EnkodeR uNd playER für DROPS*. Diplomarbeit, Technische Universität Dresden, 2003.

[Rog97] Rogerson, Dale: *Inside COM*. Microsoft Press, 1997.

[RZ03] Röttger, Simone und Steffen Zschaler: *CQML+: Enhancements to CQML*. In: Bruel, Jean-Michel (Herausgeber): *1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, Seiten 43–56, Toulouse, France, Juni 2003. Cépaduès-Éditions.

[Sch94] Schneier, Bruce: *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*. In: *Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993)*. Springer-Verlag, 1994.

[Ser03] Sun Microsystems: *Java Servlet 2.4 Specification*, 2003. http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[Sew05] Seward, Julian: *bzip2 and libbzip2, version 1.0.3, A program and library for data compression*, 2005. http://www.bzip.org.

[SG96] Shaw, Mary und David Garlan: *Software Architecture – Perspectives for an Emerging Discipline*. Prentice Hall, 1996.

[SG02] Schmerl, Bradley und David Garlan: *Exploiting architectural design knowledge to support self-repairing systems*. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Seiten 241–248, New York, NY, USA, 2002. ACM Press.

[SKW+99] Schneier, Bruce, John Kelsey, Doug Whiting, David Wagner, Chris Hall und Niels Ferguson: *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley Sons, Inc., New York, NY, USA, 1999.

[SRL96] Savetz, Kevin, Neil Randall und Yves Lepage: *MBONE: Multicasting Tomorrow's Internet*. John Wiley & Sons Inc., 1996.

[SV05] Stahl, Thomas und Markus Völter: *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, 2005.

[Szy98] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TOHS99] Tarr, Peri, Harold Ossher, W. Herrison und S. M. Sutton: *N degrees of separation: Multi-dimensional separation of concerns*. In: *Proceedings of International Conference on Software Engineering (ICSE'99)*, Seiten 107–119. ACM Press, 1999.

[Tom05] Apache Software Foundation: *Apache Tomcat*, 2005. http://tomcat.apache.org/.

[TVJ+01] Truyen, Eddy, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten und Bo Nørregaard Jørgensen: *Dynamic and Selective Combination of Extensions in Component-Based Applications*. In: *23rd International Conference on Software Engineering (ICSE'2001)*, Seiten 233–242. IEEE Computer Society, 2001.

[uml04] Object Management Group(OMG): *Metamodel and UML Profile for Java and EJB, v1.0*, Feb. 2004. http://www.omg.org/cgi-bin/doc?formal/2004-02-02.

[uml05a] Object Management Group (OMG): *Unified Modeling Language: Infrastructure Specification, Version 2.0*, July 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-05.

[uml05b] Object Management Group (OMG): *Unified Modeling Language: Superstructure Specification, Version 2.0*, July 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[UML05c] Open Management Group (OMG): *UML Profile for Schedulability, Performance and Time, Version 1.1*, 2005. http://www.omg.org/cgi-bin/doc?formal/2005-01-02.

[WD96] Woodcock, Jim und Jim David: *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[Web05] World Wide Web Consortium (W3C): *Web Services Transactions Specifications*, 2005. http://www.w3.org/2002/ws/.

[Wer98] Wermelinger, Michel: *Towards a Chemical Model for Software Architecture Reconfiguration*. IEE Proceedings – Software, 145(5):130–136, 1998.

[Wil04] Wile, David S.: *Patterns of Self-Management*. In: *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04)*. ACM Press, Oct 2004.

[Wir86] Wirsing, Martin: *Structured algebraic specifications: A kernel language*. Theoretical Computer Science, 42(2):123–244, 1986.

[WST05] *Web Services Transactions Specifications*, 2005. http://www-128.ibm.com/developerworks/library/specification/ws-tx/.

[xdo05] *XDoclet*, 2005. http://xdoclet.sourceforge.net.

[XMI05] Open Management Group (OMG): *MOF 2.0 / XMI Mapping Specification, Version 2.1*, 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[YS01] Yuan, Ruixi und W. Timothy Strayer: *Virtual Private Networks: Technologies and Solutions*. Addison-Wesley Professional, 2001.