Marco Mosconi

Modularisierung und Adaptierung von Komponenteninteraktionen mit Object Teams und dem CORBA Komponentenmodell

DIPLOMARBEIT

Berlin, 16. Mai 2003

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr. Ing. Stefan Jähnichen

Betreut von Dr. Stephan Herrmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

Um den Ansprüchen moderner Softwareentwicklung und Anforderungen immer komplexerer Software gerecht zu werden, haben sich in den vergangenen Jahren zwei Ansätze besonders hervorgetan, die Vorteile der objektorientierten Softwareentwicklung aufzugreifen und um neue Modularisierungskonzepte sowie mehr Flexibilität bei der Entwicklung zu erweitern. Dies ist auf der einen Seite der Ansatz der aspektorientierten Softwareentwicklung (*AOSD - Aspect Oriented Software Development*), der eher im Bereich der akademischen Forschung angesiedelt ist und dessen Technologien im kommerziellen Einsatz noch nicht weit verbreitet sind. Auf der anderen Seite finden sich Entwicklungen aus dem Bereich der komponentenorientierten Softwareentwicklung, wie *Enterprise Java Beans (EJB)* oder das *CORBA Component Model (CCM)*. Diese werden vornehmlich von größeren Firmen und Organisationen (in diesem Falle Sun und die OMG) in Form von Spezifikationen herausgegeben, die dann von diversen Anbietern umgesetzt und vertrieben werden.

Während der Fokus im Umfeld komponentenorientierter Softwareentwicklung meist auf der Vereinfachung der Entwicklung größerer und verteilter Systeme liegt, beschäftigt sich die aspektorientierte Seite mehr mit den Möglichkeiten der Flexibilisierung von Software, bei gleichzeitiger Modularisierung ihrer verschiedenen Aspekte. So unterschiedlich diese Ansätze auf den ersten Blick wirken, ähneln sich bei näherem Hinsehen zumindest die Ziele doch erheblich, so dass bereits mehrfach auf eine mögliche Zusammenführung der Anstrengungen hingewiesen wurde [HMO01, POM03]. In beiden Bereichen wurden Erfahrungen gesammelt und Lösungen erarbeitet, die dem jeweils anderen entweder fehlen oder in dessen Kontext bisher gar nicht für wichtig erachtet wurden.

Diese Arbeit stellt zwei der fortschrittlichsten Ansätze dieser beiden Welten vor, das aspektorientierte Programmiermodell Object Teams und das CORBA Component Model. Es wird eine mögliche Kombination der beiden Ansätze be-

schrieben und prototypisch implementiert, die die Modularisierungs- und Adaptierungskonzepte von Object Teams im Rahmen der Komponentenentwicklung mit dem CCM nutzbar macht.

1.2 Struktur

Diese Arbeit ist unterteilt in die folgenden Kapitel:

Kapitel 2 gibt zuerst einen Überblick über die komponentenorientierte Softwareentwicklung im Allgemeinen. Danach werden die wichtigsten Konzepte und Elemente des CORBA Component Model (CCM) vorgestellt, das die technologische Grundlage dieser Arbeit darstellt.

Kapitel 3 liefert eine Einführung in die Begriffe und Konzepte der aspektorientierten Softwareentwicklung und stellt insbesondere das in dieser Arbeit behandelte Modell *Object Teams* vor.

In Kapitel 4 werden die Anwendungsmöglichkeiten des Object Teams Modells im Rahmen komponentenorienter Entwicklung aufgezeigt und ein Modell für die Integration von Object Teams-Konzepten im CCM sowie die dafür nötigen Werkzeuge vorgestellt.

Kapitel 5 beinhaltet einen zusammenfassenden Überblick über die gewonnenen Erkenntnisse und einen Ausblick auf mögliche Verbesserungen des Modells und weiterführende Arbeiten in dieser Richtung.

Kapitel 2

Komponentenorientierte Softwareentwicklung mit dem CCM

Die komponentenorierte Softwareentwicklung als relativ neue Disziplin der Informatik bedient sich industrieller Vorbilder, die schon seit längerer Zeit komplexe Systeme (z.B. Autos) aus vorgefertigten und getesteten Teilen (von der Schraube bis zum kompletten Motor) zusammenbaut. Für solche Komponenten existieren Standards wie DIN oder ISO, die eine sichere Integration und hohe Wiederverwendbarkeit ermöglichen. Auf ähnliche Weise soll es in Zukunft möglich sein, Softwaresysteme aus in sich abgeschlossenenen "Bausteinen" mit spezifischer Funktionalität und standardisierten Schnittstellen zusammensetzen zu können. Diese Bausteine können sowohl Eigenentwicklungen als auch eingekaufte Komponenten von Drittanbietern, sogenannte *Commercial Off-The-Shelf* (COTS) Komponenten, sein. Man verspricht sich davon unter anderem eine verbesserte Produktivität und Entwicklungsgeschwindigkeit (Stichwort *time-to-market*) insbesondere bei der Erstellung umfangreicher Software, da nicht alle Teilaspekte neu implementiert werden müssen. Auch die Wiederverwendbarkeit und Austauschbarkeit von Teilen eines Systems soll sich durch den Einsatz streng gekapselter Komponenten mit ausschließlich expliziten Schnittstellen erhöhen. Durch die Einführung dieser weiteren Abstraktionsebene verspricht man sich bessere Modularisierungsmöglichkeiten als die reine Objektorientierung sie bietet, insbesondere bei komplexeren und verteilten Systemen. Dabei soll die Komponentenorientierung die Objektorientierung nicht ablösen, sondern sie ergänzen und auf deren Vorteilen aufbauen.

Diese Arbeit erhebt nicht den Anspruch, einen Konsens oder "die" Definition dafür zu finden, was komponentenorienterte Softwareentwicklung bedeutet und was nicht. Vielmehr soll das folgende Kapitel 2.1 umreißen, was im Rahmen dieser Arbeit unter Komponenten und verwandten Begriffen zu verstehen ist,

und die wichtigsten Konzepte aufzeigen. In Kapitel 2.2 wird das *CORBA Component Model* (CCM) vorgestellt, welches dem dieser Arbeit zugrundeliegenden Verständnis von komponentenorientierter Softwareentwicklung am nächsten kommt.

2.1 Begriffe und Konzepte

2.1.1 Der Komponentenbegriff

In den letzten Jahren hat sich der Begriff "Komponente" als fester Bestandteil im Bereich der Softwareentwicklung etabliert und ist kaum noch aus der Fachsprache wegzudenken. Mit etwas Skepsis betrachtet, kann man ihm nachsagen, ein Modebegriff zu sein, der gerne benutzt wird, um alte Konzepte "aufzupeppen". Tatsächlich wird der Komponentenbegriff aufgrund seiner Allgemeinheit auf völlig verschiedenen Ebenen und für grundverschiedene Dinge verwendet[1]. Um, zumindest für den Rahmen dieser Arbeit, ein einheitliches Bild von der Welt der Komponenten zu erhalten, sollen im folgenden die häufigsten Begriffe mit ihrer hier verwendeten Bedeutung vorgestellt werden:

(Software-)Komponenten stellen die zentralen Bausteine dar, aus denen komponentenbasierte Software zusammengesetzt wird. Sie werden als *first-class entities* betrachtet und entsprechen als solche den Objekten im objektorientierten Paradigma. Komponenten können auf allen Ebenen der Softwareentwicklung vorkommen, von der Architekturebene über das Implementierungsdesign (Modellierung) bis hin zur Implementierung und schließlich sollen sie idealerweise zur Laufzeit noch als solche identifizierbar sein. In dieser Arbeit wird ein bestimmter Typ von Komponenten betrachtet. Es handelt sich hierbei um nicht-visuelle, so genannte *serverseitige* Komponenten, die anwendungsspezifische Funktionalitäten implementieren. Eine ausführlichere Beschreibung der Merkmale solcher Komponenten findet sich im folgenden Kapitel 2.1.2.

Komponentensoftware (auch: komponentenbasierte Software oder *Componentware*) bezeichnet Software, die auf der Grundlage eines komponentenorientierten Ansatzes entwickelt wird bzw. wurde und deren technische Realisierung auf Komponenten basiert. Dabei können sowohl selbstentwickelte (*custom*) oder zugekaufte (*third-party, COTS*) Komponenten zum Einsatz kommen.

[1] Wenigstens besteht Einigkeit darüber, dass es sich bei Komponenten in diesem Kontext um irgendwie geartete Software-Artefakte handelt und nicht etwa um Dokumente, Modelle oder Methoden, die ja im natürlichsprachlichen Sinne auch Bestandteile (Komponenten) eines Softwareentwicklungsprozesses sind.

Komponentenmodelle Damit Komponenten miteinander kommunizieren und die oben beschriebenen Anforderungen bezüglich Wiederverwendbarkeit und Austauschbarkeit erfüllen können, muss ihnen eine gemeinsames Modell zugrundeliegen. Solche Komponentenmodelle definieren ein konkretes Verständnis des Komponentenbegriffs, also wie Komponenten in ihrem Kontext aussehen, und entsprechende Implementierungsvorschriften für die Entwicklung von und mit diesen Komponenten. Sie bieten damit den konzeptionellen und technologischen Rahmen, um Komponenten entwickeln und einsetzen zu können. Populäre Beispiele sind *Enterprise JavaBeans* (EJB) von Sun Microsystems, das *Component Object Model* (COM+) von Microsoft und das in Kapitel 2.2 auf Seite 12 ausführlich behandelte *CORBA Component Model* (CCM). Allen drei gemeinsam ist die Ausrichtung auf server-seitige Komponenten und die Bereitstellung von Infrastrukturdiensten über sogenannte Container, die die Laufzeitumgebung der Komponenten darstellen. Damit unterscheiden sie sich von Technologien wie *JavaBeans*, *.NET* und *CORBA*, die oftmals auch mit Komponentenmodellen in Verbindung gebracht oder als solche bezeichnet werden, jedoch jeweils eine andere Zielsetzung haben. Komponentenmodelle werden in der Literatur mitunter auch als Komponentenframeworks und Komponentenarchitekturen bezeichnet. Der Begriff Framework ist an dieser Stelle insofern passend, als dass die hier genannten Komponentenmodelle der einzusetzenden Fachlogik (den Komponenten) einen Ausführungsrahmen geben und die Framework-Metapher "Don’t call us, we call you." auch hier zutrifft. Trotzdem unterscheiden sich moderne Komponentenmodelle stark von klassischen Frameworks, unter anderem da erstere den Einsatz neuer Komponenten(typen) zur Laufzeit unterstützen und letztere in der Regel domänenspezifisch sind.

Komponentenarchitektur meint hier die grobe Struktur einer aus Komponenten zusammengesetzten Anwendung. Komponentenarchitekturen benutzen Komponenten als Strukturelemente, um Anwendungen auf einer hohen Abstraktionsstufe zu beschreiben. Eine Komponentenarchitektur beschreibt die grobe Struktur einer Anwendung mittels ihrer Komponenten und den Beziehungen zwischen ihnen und ordnet sie bestimmeten Ebenen (*layern*) der Architektur zu.

Komponentenplattformen stellen die technischen Realisierungen eines bestimmten Komponentenmodells dar. Sie stellen die Laufzeitumgebungen (Container) für Komponenten sowie Werkzeuge zu deren Erstellung und Einsatz zur Verfügung. Verschiedene Hersteller können das gleiche Komponentenmodell unterschiedlich implementieren, wobei die Kompatibilität untereinander durch das Komponentenmodell gesichert sein sollte. Im Idealfall lassen sich dann Komponenten, die auf der Plattform eines Anbieters

entwickelt wurden auch auf anderen Plattformen desselben Komponentenmodells einsetzen[2]. Auch hier wird als Synonym oft der Begriff Komponentenframework verwendet, wobei die Entsprechung an dieser Stelle eine Framework-Implementierung wäre, im Gegensatz zu konzeptionellen Frameworks in Form von Komponentenmodellen (siehe vorheriger Abschnitt).

Komponenten*basiert* oder komponenten*orientiert*? Während das Paradigma und der Softwareentwicklungsprozess in der Regel als *komponentenorientiert* bezeichnet werden, spricht man bei Software, die nach diesem Paradigma entwickelt wurde, eher von *komponentenbasierter* Software[3]. Auch wenn eine Differenzierung hier nicht notwendig erscheint, ist es doch verwirrend, eine derartige Vielzahl an Begriffen (und Abkürzungen) für die gleichen oder zumindest verwandten Dinge und Konzepte zu haben.

Des Weiteren kann man unterscheiden zwischen komponentenorientierter Softwareentwicklung (*Component-Based/Oriented Software Development* - CBSD/COSD) bzw. Softwaretechnik (*Component-Based/Oriented Software Engineering* - CBSE/COSE) und komponentenorientierter Programmierung (*Component-Based/Oriented Programming* - *CBP/COP*). Letztere soll das komponentenorienterte Erstellen von Software auf Programmiersprachenebene ermöglichen und unterstützen. Bisher gibt es jedoch nur wenige Ansätze, die solche Sprachfeatures explizit zur Verfügung stellen. Meist wird daher im Rahmen von COSD mit herkömmlichen objektorientierten Sprachen gearbeitet, die lediglich in einen komponentenorientierten Ansatz auf höherem Abstraktionsniveau eingebunden werden.

2.1.2 Merkmale von Komponenten

Dieser Abschnitt soll den Komponentenbegriff weiter verdeutlichen und besondere Merkmale von Komponenten hervorheben. Dabei geht es nicht um technische Details - diese sind ohnehin vom jeweiligen Komponentenmodell abhängig - sondern um generelle Aspekte, die Komponenten charakterisieren.

Definition

Es gibt viele, sehr unterschiedliche Definitionen dafür, was eine Komponente sein kann. Eine kurze und allgemein formulierte, aber dennoch klar umrissene und aussagekräftige Definition ist in [Szy99] zu finden:

[2] Das Beispiel EJB zeigt, dass das leider noch nicht der Stand der Dinge ist.

[3] Im Englischen wird allerdings anstelle von *Component-oriented Software Development (COSD)* häufiger der Begriff *Component-Based Software Development (CBSD)* verwendet.

> "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."

Damit sind die wichtigsten Eigenschaften einer Komponente zusammengefasst:

- *unit of composition:* Komponenten werden zusammengesetzt um komplexere Softwaresysteme zu erstellen. Jede Komponente erfüllt dabei ihre eigene, spezifische Aufgabe.
- *contractually specified interfaces:* Komponenten bieten ihre Dienste über (evtl. mehrere) Schnittstellen an, die ähnlich einem Vertrag spezifizieren, was die Komponente unter welchen Umständen leistet. In der Praxis beschränkt sich das jedoch meist auf die Definition von Signaturen und bestenfalls Vor- und Nachbedingungen in Form von Kommentaren.
- *explicit context dependencies:* Eine Komponente sollte alle ihre Kontextabhängigkeiten explizit machen (vgl. explizite Schnittstellen weiter unten), um die Austauschbarkeit, bzw. die Einsetzbarkeit in verschiedenen Umgebungen, zu gewährleisten.
- *independent deployment:* Komponenten in ihrer Auslieferform können unabhängig in verschiedenen Kontexten eingesetzt werden, solange alle expliziten Abhängigkeiten erfüllt sind.
- *composition by third party:* Das Kombinieren von Komponenten zu größeren Einheiten wird nicht vom Komponentenentwickler selbst durchgeführt, sondern vom "Benutzer".

Schnittstellen

Eine Komponente kommuniziert ausschließlich über ihre expliziten Schnittstellen mit anderen Komponenten oder Clients[4]. Daher kommt den Schnittstellen und ihrer Spezifikation hier eine besondere Bedeutung zu, die deutlich wird, wenn man sich vorstellt, dass eine Komponente in unvorhergesehenen Umgebungen eingesetzt werden soll. Da sich in einem solchen Szenario Anbieter und Benutzer einer Komponente zum Zeitpunkt der Entwicklung nicht "kennen", sind die Schnittstellen die verbindenden Einheiten, die das Zusammenspiel von Komponenten erst möglich machen und ihre exakte Spezifikation daher unerläßlich für deren Interoperabilität [Mz01]. Dieser erhöhten Anforderung könnte man

[4] Die Forderung nach ausschließlich expliziten Schnittstellen beinhaltet, dass die gesamte Kommunikation zwischen Komponenten und Clients über extern definierte und sichtbare Schnittstellen erfolgt. Eine implizite Schnittstellen würde entstehen, wenn Abhängigkeiten nur in der Implementierung auftauchen würden und somit "versteckt" wären.

mit *Kontrakten* gerecht werden, an die sich Anbieter und Benutzer der bereitgestellten Dienste binden, und die auf semi-formale Weise die funktionalen sowie nicht-funktionalen Aspekte der Schnittstelle spezifizieren. [Szy99] widmet dem Thema Schnittstellen und Kontrakte ein eigenes Kapitel, in dem er Probleme mit Schnittstellen (insbesondere bei Verwendung von *callbacks* und dem Auftreten von *re-entrance*) aufzeigt und Kontrakte als Lösungsansatz vorschlägt.

Kapselung

Komponenten sollen im Allgemeinen als *"black-box"* angesehen werden, was bedeutet, dass ihre Implementierung in ihnen gekapselt und nicht von außen zugänglich ist. Sie sollen ohne Kenntnis ihrer Implementierungsdetails und nur aufgrund ihrer Schnittstellen und Dokumentation integrierbar sein. Das erfordert gelegentlich Kompromisse, dafür erhält man eine *"plug and play"* Einsatzbereitschaft. Insbesondere bei Komponentenmodellen, deren Komponenten aus mehreren Klassen bestehen können (wie dem CCM), ist es entscheidend, dass die internen Klassen von außen nicht sichtbar sind und somit auch keine Abhängigkeiten zwischen Benutzer und Komponentenimplementierung bestehen können.

Granularität

Es stellt sich die Frage, wieviel Funktionalität eine Komponente beinhalten soll, also nach ihrer Granularität. Auch hier gehen die Meinungen auseinander. Das Spektrum reicht von Komponenten, die vom Umfang her einer Klasse bzw. einem Objekt entsprechen, bis hin zu Subsystemen, die ganze Klassenstrukturen in sich vereinen. Diese Frage lässt sich nicht pauschal beantworten, es muss immer ein Kompromiss zwischen dem Grad der Wiederverwendbarkeit und dem Grad der Spezialisierung einer Komponente gefunden werden. Eine hohe Wiederverwendbarkeit erreicht man bei kleinen Komponenten, die eine nicht zu spezielle Funktionalität erfüllen und somit in verschiedenen Kontexten einsetzbar sind. Allerdings weisen solche Komponenten in der Regel auch mehr Abhängigkeiten zu anderen Komponenten auf und führen so zu komplexeren Verbindungsgraphen in der Komponentenarchitektur. Das widerspricht dem Ziel, durch Komponenten die Komplexität auf Architekturebene zu reduzieren. Im Kontrast dazu stehen vergleichsweise "große" Komponenten, die stark spezialisiert sind und weniger Abhängigkeiten aufweisen, da sie benötigte Funktionalitäten selbst implementieren. Die Wiederverwendbarkeit solcher Komponenten kann jedoch durch diese Spezialisierung eingeschränkt sein und einen erhöhten Anpassungsbedarf zur Folge haben.

Konfigurierbarkeit

Um dem Anspruch der Wiederverwendbarkeit und der Einsetzbarkeit in verschiedenen, vorher nicht festgelegten Umgebungen gerecht zu werden, muss eine Komponente zu einem gewissen Grad konfigurierbar sein. Dies trifft besonders bei Komponenten zu, die aufgrund ihrer Größe und Spezialisierung ohne weitere Anpassung nicht flexibel einsetzbar sind. Beispielsweise sollten bei einer Komponente aus dem Finanzbereich die Währung und zu verwendende Umrechnungsfaktoren konfigurierbar sein. Als eine weitere Möglichkeit der Konfigurierbarkeit wäre die Auswahl verschiedener Implementierungen einer Schnittstellen-Funktion nach dem *Strategy* Design Pattern [GHJV95] über eine Konfigurationsschnittstelle denkbar. Eine Implementierung könnte für Geschwindigkeit, eine andere für Genauigkeit optimiert sein. Mit Hilfe solcher Konfigurationsoptionen lässt sich eine Komponente relativ flexibel an lokale Bedingungen anpassen.

Komponenten vs. Objekte

Die Frage, ob Komponentenorientierung einen Ersatz für Objektorientierung darstellt, wurde eingangs schon verneint. Es ist eher so, dass der komponentenorientierte Ansatz die bewährten Konzepte der Objektorientierung aufgreift und eine weitere Abstraktionsebene einführt. Die Basis für komponentenorientierte Softwareentwicklung ist im Allgemeinen, aber nicht notwendigerweise, das objektorientierte Paradigma. Oftmals werden auch die Begriffe "Objekt" und "Komponente" synonym verwendet, insbesondere im Umfeld verteiler objektorientierter Technologien wie CORBA. Komponenten sollten jedoch als eigenständiges Konzept verstanden werden, auch wenn sie einige gemeinsame Eigenschaften wie Kapselung und Schnittstellen mit Objekten teilen. In den meisten Fällen werden Komponenten gegenüber Objekten als gröbere Abstraktion und weniger technisch (implementierungsnah) angesehen. Ähnlich wie bei Objekten und deren Typen (durch Klassen bzw. Interfaces definiert) kann auch bei Komponenten unterschieden werden zwischen einem Komponententyp, seiner Implementierung und Instanzen dieses Typs. Komponenteninstanzen werden meist über *Factories* [5] erzeugt und sind, zumindest logisch, auch zur Laufzeit noch als solche identifizierbar (wenn auch nicht so atomar wie Objekte). Diese explizite Instanziierbarkeit unterscheidet Komponenten wesentlich von Modulen, Paketen und ähnlichen Konzepten zur Modularisierung auf Sourcecode- oder Namensraum-Ebene.

[5] In Anlehnung an das *Abstract Factory* Design Pattern aus [GHJV95]. Von den bewährten Design Patterns aus der Objektorientierung wird bei der komponentenorientierten Softwareentwicklung übrigens häufig Gebrauch gemacht.

Komponentenarten

Wie man an den vielen vagen Definitionen merkt, lassen sich Komponenten schlecht über einen Kamm scheren. Es macht daher Sinn, verschiedene Arten von Komponenten zu betrachten, die jeweils spezielle Ausprägungen der hier vorgestellten Merkmale aufweisen. Visuelle Komponenten wie Buttons oder Formulare, die man als "GUI-Komponenten" bezeichnen könnte, sollen hier nicht näher betrachtet werden. Bei den sogenannten "server-seitigen" Komponenten, die getrennt von graphischen Benutzerschnittstellen ihre Aufgaben erledigen, kann man grob folgende Arten unterscheiden:

- *Controller-Komponenten* (nach dem *Model-View-Controller* Prinzip) realisieren die Interaktion mit dem Benutzer bzw. technisch gesehen den GUI-Komponenten. Sie steuern den Ablauf von interaktiven Anwendungen.
- *Business-Komponenten* implementieren die tatsächliche Geschäftslogik der Anwendung und werden von den Controller-Komponenten aufgerufen und gesteuert. Man könnte hier weiter differenzieren in Geschäftsprozesse, -regeln und -objekte.
- *Infrastruktur-Komponenten* bieten schließlich Basis-Dienste wie Datenbank-Anbindung oder Authentifizierung an, die von den Business-Komponenten genutzt werden.

Diese Aufteilung entspricht der klassischen 3-Tier-Architektur, die man häufig bei verteilten Anwendungen vorfindet.

2.1.3 Ein komponentenorientierter Entwicklungsprozess

Der Übergang von eher monolithischer, objektorientierter Anwendungsentwicklung hin zu verteilter Komponentensoftware erfordert neben den notwendigen Technologien wie Middleware (vgl. Kapitel 2.2.1 auf Seite 12) und Komponentenmodellen auch ein angepasstes, komponentenorientiertes Vorgehensmodell für die Erstellung und den Einsatz solcher Systeme. Eine generelle Tendenz ist hier die möglichst deklarative Beschreibung der anwendungsspezifischen Komponentenarchitektur (Komponenten-Schnittstellen und deren Verbindungen) und der nicht-funktionalen Eigenschaften wie Persistenz, Transaktionen, Sicherheit oder Verteilung. Die Nutzung solcher, in der Regel von der Komponentenplattform bereitgestellten, Dienste erfolgt dann weitgehend durch generierten Code und wird gesteuert über die Konfiguration des Containers. In [Buc99] werden 5 Vorgehensmodelle für die komponentenorientierte Softwareentwicklung vorgestellt, darunter Catalysis und SELECT, sowie ein angepasstes Vorgehensmodell auf Basis von Catalysis. An dieser Stelle soll nur ein sehr grober Überblick über neue Aufgabenfelder und Rollen gegeben werden, die das komponentenorientierte Paradigma mit sich bringt.

Phasen

Neben den klassischen Phasen Analyse, Design, Implementierung und Testen werden bei der komponentenorientierten Softwareentwicklung erstmals auch *Packaging, Assembly* und *Deployment* explizit betrachtet.

Packaging ist der Begriff für das "Verpacken" von Komponenten zu auslieferbaren Einheiten, die in der Regel die Komponentenimplementierung in kompilierter Form sowie Konfigurationsdateien und Dokumentation enthalten. Assembly wird sowohl der Vorgang als auch das Produkt des Kombinierens von Komponenten zu einem (Teil-)System genannt. Assemblies können direkt vor ihrem Einsatz erstellt werden oder aber vorkonfiguriert und wiederum verpackt werden, um als Einheit weitergegeben werden zu können. Beispiele hierfür sind *Web Application Archives* für Servlet-Container [Sun01b] und *J2EE Application Assemblies* [Sun01a].

Das *Deployment* bezeichnet den Vorgang des Installierens einzelner Komponenten, Assemblies oder ganzer Anwendungen in einer entsprechenden Laufzeitumgebung (z.B. Container). Während der Deployment-Phase werden die verpackten Komponenten auf den Server geladen, entpackt, gegebenenfalls instanziiert und konfiguriert. Nach dem Deployment können die installierten Komponenten in der vorgesehenen Weise benutzt werden. Die zunehmende Bedeutung und das Interesse an dieser Phase des Lebenszyklus von Komponenten zeigt unter anderem die im Juni 2002 erstmalig in Berlin durchgeführte internationale Konferenz "Component Deployment" [Bis02, CD2002].

Rollen

Bei den beteiligten Rollen kann man grundsätzlich eine Unterscheidung zwischen *Komponentenentwicklern* und *Komponentennutzern (Anwendungsentwicklern)* treffen. Komponentenentwickler entwerfen einzelne Komponenten und implementieren diese anhand ihrer Spezifikation, Anwendungsentwickler integrieren sie später zu kompletten Systemen indem sie Assemblies aus ihnen bilden. Das Deployment der Anwendungen wird eine Art *Administrator* vornehmen, der für die Verwaltung des Applikationsservers zuständig ist. Der Applikationsserver wird schließlich vom *Container Provider* bereitgestellt, was auch in Form eines *Application Service Provider (ASP)* über das Internet denkbar wäre.

Die mögliche räumliche und auch zeitliche Trennung dieser Rollen und Phasen macht den Prozess sehr flexibel, erfordert aber auch ein hohes Maß an Kommunikation und Spezifikation, damit die Komponenten am Ende das tun, wofür sie anfangs gedacht waren. Dabei helfen Standards, Komponentenplattformen und intensive Werkzeugunterstützung, ohne die ein solches Vorgehensmodell nicht praktikabel wäre. Eine konkretere Beschreibung dieser Phasen und Rollen findet sich in Kapitel 2.2.6 auf Seite 31 am Beispiel des *CORBA Component Model*.

2.2 Das *CORBA Component Model*

Das *CORBA Component Model* (CCM) ist eine Spezifikation der Object Management Group (OMG) und definiert ein server-seitiges Komponentenmodell zum Erstellen von verteilten *"enterprise-class applications"*. Es basiert auf der ebenfalls von der OMG standardisierten *Common Object Request Broker Architecture (CORBA)* und ergänzt diese (objektbasierte) Middleware um ein vollwertiges und mächtiges Komponentenmodell. Damit soll es den Entwicklern komplexer Softwaresyteme möglich sein, Anwendungen als eine Menge von miteinander verbundenen Komponenten zu erstellen und diese in einer verteilten, heterogenen Umgebung einzusetzen.

2.2.1 Hintergrund

Vor dem Erscheinen server-seitiger Komponentenmodelle wie EJB und CCM wurden verteilte Anwendungen meist auf Basis von *Distributed Object Computing (DOC)* Middleware wie *CORBA* von der OMG, *DCOM/COM+* von Microsoft oder *Java Remote Method Invocation* (RMI) von Sun/JavaSoft erstellt. Diese Technologien bieten Entwicklern die Möglichkeit von konkreten Plattformen, Netzwerkprotokollen oder Programmiersprachen zu abstrahieren und bieten ein konsistentes, verteiltes Objektmodell sowie die Nutzung von Diensten wie Security, Transaktionssicherheit und Persistenz an. Die Entwicklung größerer Projekte mit dieser Technik ist jedoch sehr aufwendig und die entstehende Software komplex, da die Nutzung der Middleware und ihrer Dienste von Hand implementiert werden muss und eine Wiederverwendung der Implementierung nur eingeschränkt möglich ist. Ein weiterer Nachteil ist, dass Verbindungen zwischen Objekten in ihren Implementierungen erfolgen und nicht von außen konfigurierbar sind, dadurch wird die Sichtbarkeit der verteilten Anwendungs-Architektur erschwert.

Moderne Komponentenmodelle wie EJB und CCM sind mit dem Ziel entstanden, genau diese Probleme zu lösen. Sie erweitern ihre zugrundeliegenden Objektmodelle um ein Komponentenkonzept, dass eine bessere Reflektierung der verteilten Architektur durch explizite Verbindungsmechanismen ermöglicht und führen Container als Vermittler zwischen Komponenten und Middleware ein. Container stellen die Laufzeitumgebung für Komponenten dar, verwalten deren Lebenszyklus und bieten ihnen vereinfachten Zugriff auf Infrastrukturdienste. Darüber hinaus lassen sich diese Dienste und andere nicht-funktinonale Anforderungen mit Hilfe externer Konfigurationsdateien (*Deskriptoren*) konfigurieren, so dass die Komponenten idealerweise fast nur noch die tatsächliche Geschäftslogik implementieren müssen. Das *CORBA Component Model* ist aus den Erfahrungen entstanden, die bei der Entwicklung verteilter Systeme mit

CORBA und EJB gemacht wurden und integriert bewährte Design Patterns und Lösungsansätze zu einem homogenen Komponentenmodell.

CORBA

Da das CCM auf CORBA-Technologie basiert und diese durchgehend nutzt und erweitert, ist ein Verständnis der Grundkonzepte dieser Middleware von Vorteil. Um den Rahmen dieser Arbeit nicht zu sprengen, soll hier jedoch nur auf die essentiellen Konzepte, die für das CCM von Bedeutung sind, eingegangen werden.

Die *Common Object Request Broker Architecture* (CORBA) spezifiziert eine Standard-Architektur für verteilte, heterogene Objektsysteme. Dienste, die von einem Objekt angeboten werden, definiert man in der *Interface Definition Language* (IDL), die Implementierung dazu kann dann in einer beliebigen Programmiersprache erfolgen, für die es ein *language mapping* gibt[6]. Diese *language mappings* bilden die Elemente der IDL auf Konstrukte der Programmiersprache ab. Miteinander kommunizierende Objekte müssen nicht in der selben Sprache implementiert sein und nicht im gleichen Prozess oder auf dem selben Host laufen. Außerdem können sie auf unterschiedlichen Plattformen zum Einsatz kommen, solange ein entsprechender ORB (*Object Request Broker*) für diese vorhanden ist. Somit bietet CORBA Ortstransparenz über Prozess- und Rechnergrenzen, sowie Plattform- und Sprachunabhängigkeit für Anwendungs-Objekte auf Client- und Server-Seite.

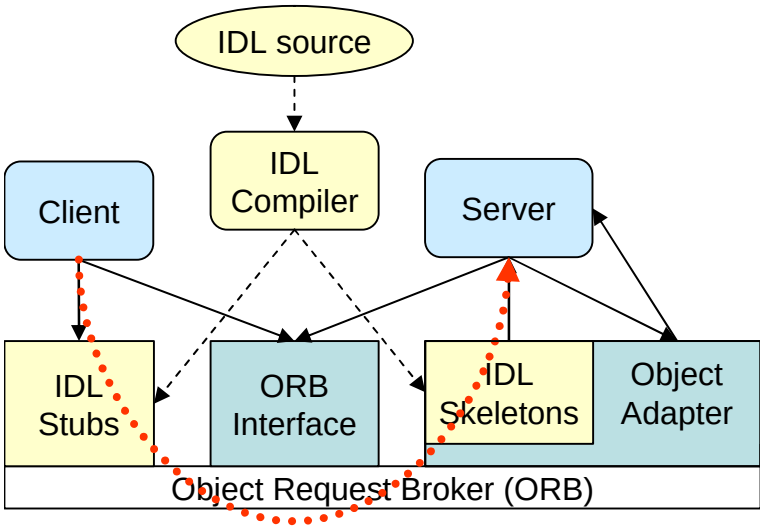

Abbildung 2.1: Architektur einer CORBA Anwendung

[6] Es gibt standardisierte mappings unter anderem für Java, C++, Smalltalk, Ada, Lisp und Python. Wie diese Auswahl zeigt, werden nicht nur objektorientierte und kompilierte Sprachen unterstützt.

Abbildung 2.1 zeigt die Architektur einer verteilten CORBA Anwendung. Ausgehend von den mittels IDL definierten Schnittstellen des Server-Objekts erzeugt ein IDL Compiler Stubs und Skeletons, das sind Proxies für die Client- bzw. Server-Seite. Ein Methodenaufruf des Client-Objekts an das Server-Objekt erfolgt immer über den lokalen Stub, so dass für den Client unerheblich ist, ob das aufzurufende Server-Objekt lokal ist oder in einem entfernten Prozess läuft. Der Stub wandelt den Methodenaufruf in eine transportfähige Nachricht (*marshalling*) und übergibt diese dann an den ORB, der das gewünschte Server-Objekt lokalisiert und ggf. über den Object Adapter aktiviert. Die Nachricht wird nun vom Skeleton wieder in einen lokalen Methodenaufruf umgewandelt (*demarshalling*) und dieser auf dem Server-Objekt ausgeführt.

Die Spezifikation

Das *CORBA Component Model* ist mittlerweile Bestandteil des aktuellen CORBA 3.0 Standards, der im letzten Jahr finalisiert und veröffentlicht wurde. Für das CCM sind in erster Linie die erweiterte Spezifikation des CORBA-Kerns [OMG02c] und die neu hinzugekommene Spezifikation des Komponentenmodells [OMG02a] von Bedeutung [7]. Die Spezifikation des Komponentenmodells umfasst mehrere Kapitel, die im einzelnen die folgenden Aspekte behandeln:

- Das *Abstract Component Model* beschreibt die äußeren Eigenschaften von CCM Komponenten und wie Komponententypen in IDL definiert werden. Siehe Kapitel 2.2.2 auf der nächsten Seite.

- Mit dem *Component Implementation Framework* wird das Programmiermodell zur Erstellung von Komponentenimplementierungen sowie deren Relationen zum generierten Code beschreiben. Siehe Kapitel 2.2.3 auf Seite 22.

- Das *Container Programming Model* enthält die Beschreibung der Container-Architektur und deren Schnittstellen zum ORB und zu den Komponenten. Siehe Kapitel 2.2.4 auf Seite 24.

- Im Kapitel *Packaging und Deployment* werden das Verpacken von Komponenten und Assemblies, die verschiedenen Arten von Deskriptoren sowie der Vorgang des Deployments beschrieben. Siehe Kapitel 2.2.5 auf Seite 27.

[7] Die im folgenden gemachten Aussagen beziehen sich auf die aktuelle, überarbeitete Version [OMG02b]. Da bestimmte Einzelheiten immer noch Änderungen unterworfen sind, kann in dieser Arbeit nur der aktuelle Stand zum Zeitpunkt des Schreibens wiedergegeben werden. Nicht eindeutige Aussagen unterliegen der Interpretation des Autors.

- Die *Integration mit EJBs* erlaubt eine EJB-Sicht auf CORBA Komponenten und eine CORBA-Sicht auf EJBs. Dazu werden Richtlinien festgelegt und Abbildungen zwischen beiden Modellen definiert.
- Des weiteren sind die *Metamodelle* für IDL und das CIF in Form von UML-Diagrammen und XMI DTDs enthalten.

Aufgrund des strengen Standardisierungsprozesses der OMG hat es mehrere Jahre gedauert, bis aus der ersten Version der Spezifikation [OMG99] ein offizieller Standard wurde. Seit dieser ersten Version aus dem Jahr 1999 hat sich eine Menge geändert. Auf Grund der gemachten Erfahrungen bei der Erstellung einer Referenzimplementierung wurden Details geändert und teilweise ganze Bereiche fallengelassen, die sich als nicht realisierbar erwiesen. Aufgrund dieser Verzögerung gibt es bisher kaum Plattformen, ORBs oder Applikationsserver, die das CCM unterstützen, während Produkte für J2EE/EJB und .NET bereits seit geraumer Zeit verfügbar und im Einsatz sind. Es wird sicher noch eine Weile dauern, bis die großen Hersteller Produkte für das CCM anbieten werden, so dass es schwierig sein dürfte, den Vorsprung der anderen Technologien wie EJB oder .NET aufzuholen und Marktanteile zurückzugewinnen.

Die folgenden Abschnitte sollen einen kurzen Überblick über die wesentlichen Bestandteile des CCM geben, die für das Verständnis der im praktischen Teil der Arbeit vorgestellten Lösung notwendig sind. Aufgrund des Umfangs der Spezifikation (> 1000 Seiten CORBA + > 400 Seiten CCM, das offizielle Tutorial hat über 200 Folien) muss hier auf viele Details verzichtet werden. Für eine tiefergehende Einführung sei der Leser auf die Spezifikationen sowie [MM02, Par03, Rui] verwiesen [8].

2.2.2 Abstract Component Model

Das *Abstract Component Model* beschreibt, welche extern sichtbaren Eigenschaften und Schnittstellen eine CORBA Komponente aufweisen kann. Dazu gehören die verschiedenen synchronen und asynchronen Schnittstellen[9] (*Ports*), die von einer Komponente angeboten und benötigt werden, sowie ihre öffentlichen Attribute. Auf dieser Architektur- oder Entwurfsebene wird also der Komponententyp festgelegt und auf welche Weise (mit welchen Protokollen) er interagieren kann. Die interne Struktur und die Implementierung einer Komponente sind gekapselt und nach außen nicht sichtbar. Die explizite Definition der logischen Sicht auf eine Komponente mittels IDL stellt eine strenge Entkopplung

[8] Dazu ein Zitat von der EJCCM Webseite [CP]: *"The short answer is to read the CCM specification several hundred times [...] and review the demo and test source code over and over again (you might even consider sleeping on hard copies of the source code)."*

[9] Als synchron bezeichnet man Schnittstellen, die über herkömmliche Methodenaufrufe benutzt werden. Asynchrone Schnittstellen benutzen dagegen Mechanismen der Ereignisbasierten Kommunikation.

von interagierenden Komponenten und ihren Benutzern dar, da deren Implementierungen nur indirekt über die definierten Schnittstellen kommunizieren. Abbildung 2.2 zeigt eine gängige Darstellung des abstrakten Modells einer CORBA Komponente. Die einzelnen Elemente werden in den folgenden Abschnitten erläutert.

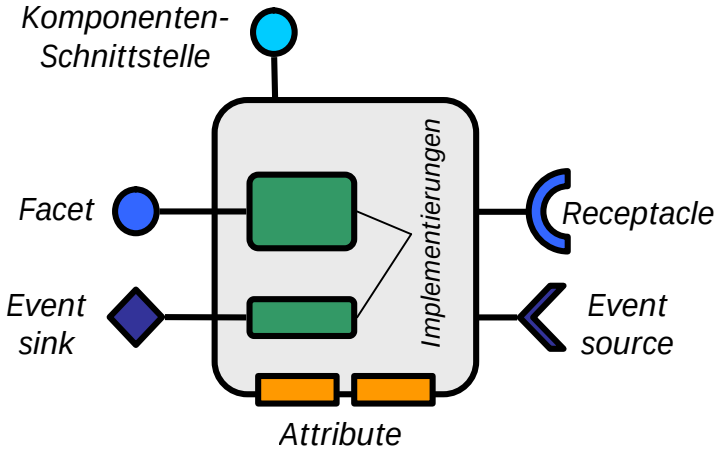

Abbildung 2.2: Logische Struktur einer CCM Komponente

Component Levels

Die Spezifikation unterscheidet zwei Stufen von Komponenten, namentlich *basic components* und *extended components*. Diese werden nicht explizit gekennzeichnet sondern unterscheiden sich nur in der Menge der unterstützten Merkmale des Komponentenmodells. *Basic components* besitzen nur ihre Komponenten-Schnittstelle sowie Attribute und können keine der erweiterten Port-Typen definieren. Sie dienen hauptsächlich der einfachen "Komponentisierung" von klassischen CORBA-Objekten und entsprechen im Fuktionsunfang den Enterprise Java Beans in der Version 1.1, was die Interoperabilität zwischen den beiden Modellen begünstigt. *Extended components* können alle Merkmale des Abstract Component Models enthalten. Ungeachtet dieser Klassifizierung können CORBA-clients, die nichts vom CCM wissen (*component-unaware clients*), immer auf die Funktionalitäten einzelner angebotener Schnittstellen zugreifen. Sie können lediglich nicht zwischen ihnen navigieren und keine erweiterten Aufgaben, wie das verbinden von Komponenten vornehmen.

Definition eines Komponententyps

Ein Komponententyp gruppiert Attribute und Ports, die als funktionale Schnittstellen einer Komponente gesehen werden können. Komponententypen werden

in der aus CORBA 2.x bekannten *Interface Definition Language* (IDL) definiert, die in der Version 3 um die nötigen Konstrukte für Komponenten erweitert wurde. So wird ein Komponententyp mit dem Schlüsselwort `component` definiert, welches eine Spezialisierung und Erweiterung des `interface` Konstrukts für CORBA Objekt-Schnittstellen darstellt. Hierbei ist zu beachten, dass der Komponententyp keine Operationen direkt auf seiner Komponentenschnittstelle definieren kann. Dies geht nur über die Unterstützung weiterer Schnittstellen (siehe folgender Abschnitt).

```
1 component Person {
2     // Attribute und Ports
3 };
```

Listing 2.1: Definition eines Komponententyps in IDL3

Komponenten-Schnittstelle und Vererbung Zur Laufzeit besitzt jede Komponenteninstanz eine eindeutige sogenannte Basis-Referenz. Diese Referenz ist technisch gesehen ein CORBA-Objekt vom Typ der Komponenten-Schnittstelle, die auch als *Äquivalente Schnittstelle* (*equivalent interface*) des zugehörigen Komponententyps bezeichnet wird. Die Komponenten-Schnittstelle eines Komponententyps enthält implizit generierte Operationen für die Navigation zwischen der Basis-Referenz und den angebotenen Schnittstellen sowie zum Verbinden von *Ports* (siehe folgende Unterabschnitte). Außerdem enthält sie implizite Zugriffsoperationen für die im Komponententyp definierten Attribute. Die Äquivalente Schnittstelle ist die Repräsentation des Komponententyps in IDL2. Die komponentenspezifischen Konstrukte werden dazu auf IDL2-Konstrukte abgebildet (siehe Beispiel). So ist die Kompatibilität zu CORBA 2.x Clients und ORBs gewährleistet.

Die Möglichkeiten der Vererbung ähneln denen in Java: ein Komponententyp kann nur von einem Basis-Komponententyp erben, aber mehrere Schnittstellen unterstützen (Schlüsselwort `supports`). Dabei sollte beachtet werden, dass es sich hier bei beiden Varianten ausschließlich um Schnittstellen-Vererbung auf Typ-Ebene handelt. Bei der Vererbung zwischen zwei Komponententypen erbt der Subtyp die Attribute und Ports des Basistyps. Unterstützt ein Komponententyp eine Schnittstelle via `supports`, so erbt seine Komponentenschnittstelle von dieser.

Im folgenden Beispiel erbt der Komponententyp `Student` vom Komponententyp `Person` und unterstützt zusätzlich die Schnittstelle ILearning, die unabhängig von beiden definiert ist.

```
1 interface ILearning { ...};
2 component Person { ...};
3 component Student : Person supports ILearning { ...};
```

Listing 2.2: Schnittstellen-Vererbung in IDL3

Die Äquivalente Schnittstelle der `Student` Komponente in IDL2 würde dann folgendermaßen aussehen:

```
1 interface Person { ...};
2 interface Student : Person, ILearning { ...};
```

Listing 2.3: Die Äquivalente Schnittstelle einer Komponente in IDL2

Damit ist eine Komponenteninstanz vom Typ `Student` polymorph zu `Person` und `ILearning`. Eine Implementierung des `Student` Komponententyps muss nun alle drei Schnittstellen implementieren. Welche Möglichkeiten zur Modularisierung der Implementierung das CCM bietet, wird in Kapitel 2.2.3 auf Seite 22 beschrieben.

Attribute Der Haupteinsatzzweck für komponenteneigene Attribute ist die Konfigurierbarkeit von Komponenten nach ihrer Instanziierung. So soll die Möglichkeit gegeben werden, die Komponenten während des Deployments an die Umgebung anzupassen. Beispiele hierfür wären die Auswahl eines Ausgabeformats für Logging-Informationen oder die Konfiguration eines länderspezifischen Profils. Die Konfiguration kann manuell durch Setzen der einzelnen Attributwerte geschehen, oder über Konfigurationsobjekte, die eine Menge von Attributwerten aufnehmen und dann in einem Vorgang auf die Komponente angewandt werden können. Die zweite Variante wird hier nicht näher betrachtet. Die Operationen zum Auslesen und Setzen der Attribute sind implizit in der Äquivalenten Schnittstelle des Komponententyps enthalten. Attribute können als nur lesbar definiert werden (modifier `readonly`) und beim Zugriff Exceptions werfen (`getraises` für Lesezugriffe und `setraises` für Schreibzugriffe). Im Äquivalenten Interface bedeutet das, dass für `readonly` Attribute eine *get*- aber keine *set*-Operation generiert wird und die Exceptions in den entsprechenden Operationsdefinitionen landen.

```
1 component Person {
2     readonly attribute string name;
3     attribute short age getraises(AgeNotSetException) \
4                         setraises(InvalidAgeException);
5 };
```

Listing 2.4: Attribute einer Komponente in IDL3

Ports

Wie bereits erwähnt, bezeichnet man die funktionalen Schnittstellen eines Komponententyps als *Ports*. Sie definieren die von einer Komponente bereitgestellten und benutzten, synchronen wie asynchronen Schnittstellentypen. Beim Deployment und zur Laufzeit können diese dann mit CORBA-Objekten des passenden Typs verbunden werden. Dies geschieht über die impliziten Operationen der Äquivalenten Schnittstellen. Die einzelnen Port-Typen sollen im folgenden kurz vorgestellt und erläutert werden:

Facets (*provided interfaces*) sind benannte Schnittstellen (im Sinne von CORBA 2.x), die von einer Komponente angeboten und implementiert werden und von Clients synchron genutzt werden können. Eine CORBA Komponente kann mehrere unabhängige Schnittstellen bereitstellen. Diese können unterschiedlichen Typs sein und müssen weder untereinander noch mit dem Komponententyp in einer Vererbungsbeziehung stehen. Es können auch mehrere Schnittstellen des gleichen Typs angeboten werden, die unter Umständen durch verschiedene Implementierungen realisiert werden. Das folgende Beispiel zeigt eine Komponente `Person`, die ein Facet vom Typ `IFamily` und zwei vom Typ `IEmployee` anbietet. Die beiden Facets `job1` und `job2` sind vom gleichen Typ und sehen daher für Clients identisch aus. Intern können jedoch jeweils unterschiedliche Implementierungen von `IEmployee` zum Einsatz kommen.

```
1 interface IFamily { ...};
2 interface IEmployee {
3     void do_my_job ();
4 };
5 component Person {
6     provides IFamily private;
7     provides IEmployee job1;
8     provides IEmployee job2;
9 };
```

Listing 2.5: Definition von Facets in IDL3

Facets sollen es ermöglichen, verschiedenen Clients spezifische Schnittstellen anzubieten, die nur zusammengehörige Funktionalität bieten. Sie stellen Sichten oder funktionale Aspekte einer Komponente dar, die auf diesem Weg modular definiert werden können. Ein Client kann zwischen den Facets und der Basisreferenz mit Hilfe generischer oder spezialisierter, implizit generierter Operationen

navigieren[10]. Die Implementierung der angebotenen Schnittstellen bleibt in der Komponente verborgen und ist an diese gebunden. Damit ist auch der Lebenszyklus eines Facets an den der umgebenden Komponenteninstanz gebunden. Welche Möglichkeiten das CCM zur Aufrechterhaltung dieser Modularisierung auch in der Implementierung bereitstellt, wird in Kapitel 2.2.3 auf Seite 22 beschrieben.

Receptacles (*used/expected interfaces*) sind benötigte und benannte externe Schnittstellen einer Komponente. Für die Ausführung und Erfüllung ihrer angebotenen Dienste kann es nötig sein, dass eine Komponente die Dienste anderer Komponenten oder Objekte benötigt. Das abstrakte Komponentenmodell des CCM bietet die Möglichkeit, diese Kontextabhängigkeiten explizit in die Komponentendefinition einzubeziehen. Auf Architekturebene wird der benötigte Schnittstellentyp festgelegt, die tatsächliche Verbindung (Lösung) zu (von) einer Objekt-Referenz erfolgt dynamisch zur Laufzeit mit Hilfe generierter Operationen auf dem Äquivalenten Interface (bzw. beim Deployment mittels entsprechendem Deskriptor). Damit ist diese Form der Komponentenkomposition von der Implementierung gelöst und kann zur Laufzeit rekonfiguriert werden. Ein Receptacle kann auch mehrere Verbindungen zu Objekten des spezifizierten Typs gleichzeitig aufnehmen (modifier: `multiple`). In diesem Fall erfolgt die Verwaltung und Zuordnung über Cookies.

Das Beispiel zeigt einen Komponententyp `Employer`, der Verbindungen zu mehreren Objekten des Typs `IEmployee` verwalten kann. Dieses Receptacle könnte auf Instanzebene mit verschiedenen `Person` Komponenten über deren Facets `job1` oder `job2` verbunden werden.

```
1 component Employer {
2     uses multiple IEmployee employees;
3 };
```

Listing 2.6: Definition von Receptacles in IDL3

Event sinks/sources Ein weitere Neuerung ist die Nutzung ereignisbasierter Kommunikation auf abstrakter Ebene. Das CCM bietet Komponenten asynchrone Kommunikation nach dem *publisher/subscriber* Modell auf Basis des CORBA Notification Service. Ereignistypen werden in IDL3 mit dem Konstrukt `eventtype` definiert und sind strukturell äquivalent zu IDL2 `valuetypes`. Ein

[10] Als Navigation bezeichnet man hier das Anfordern einer Objekt-Referenz auf eine bestimmte Schnittstelle der Komponente. Die Komponentenschnittstelle besitzt Operationen, um Referenzen auf die angebotenen Facets zu bekommen (z.B. `provide_job1()` und von jedem Facet erhält man über `get_component()` die zugehörige Basisreferenz der Komponente.

Komponententyp kann Schnittstellen für das Senden (*event sources*) und Empfangen (*event sinks*) von Ereignissen eines bestimmten Typs definieren. Ähnlich den Facets und Receptacles können auch diese Ports während des Deployments oder zur Laufzeit miteinander verbunden werden. In diesem Fall geschieht diese Verbindung nicht direkt, sondern über Ereignis-Kanäle, die von der Laufzeitumgebung bereitgestellt und verwaltet werden.

Im folgenden Beispiel werden drei Ereignistypen definiert, über die der Komponententyp `Person` kommunizieren kann. `emits` und `publishes` bezeichnen Ereignisquellen, wobei ersteres nur einen und letzteres mehrere Empfänger haben kann. `consumes` ist das Schlüsselwort für Ereignissenken, die Ereignisse von einem oder mehreren Sendern empfangen.

```
1 eventtype IllnessEvent { ...};
2 eventtype StateChangeEvent { ...};
3 eventtype NewsletterEvent { ...};
4 component Person {
5     emits IllnessEvent ill;
6     publishes StateChangeEvent change;
7     consumes NewsletterEvent news;
8 };
```

Listing 2.7: Eventtypen in IDL3

Homes

Die oben beschriebenen Konzepte dienen der Definition von Komponententypen. Mit dem ebenfalls neuen `home` Meta-Typ definert man Komponenten-Manager, die den Lebenszyklus von Komponenteninstanzen zur Laufzeit verwalten. Komponenteninstanzen können nur über die Operationen des zuständigen Homes erzeugt und wieder vernichtet werden. Ein Home ist für genau einen Komponententyp zuständig, wobei ein Komponententyp von verschiedenen Home-Typen verwaltet werden kann. Zur Laufzeit wird jedoch eine Komponenteninstanz immer von genau einer Home-Instanz verwaltet. Home-Typen werden zusammen mit den Komponententypen in IDL3 definiert, wie im folgenden Beispiel dargestellt.

```
1 component Person { ...};
2 home PersonHome manages Person {
3     factory create_person (in string name);
4     finder find_person_by_name (in string name);
5     // weitere Attribute und Operationen
6 };
```

Listing 2.8: Home-Definition in IDL3

Homes bieten dem Komponenten-Designer die Möglichkeit, optional eigene Operationen zum Erzeugen (`factory`) und Auffinden (`finder`) von Komponenteninstanzen zu definieren, die dann vom Entwickler implementiert werden müssen. Bei Verwendung eines Primärschlüssels (hier nicht gezeigt) wird die Identität der Komponenteninstanzen explizit und nach außen verfügbar gemacht. In diesem Fall werden Operationen zum Erzeugen, Finden und Entfernen anhand eines solchen Schlüssels generiert. Die Angabe eines Primärschlüssels (`primarykey`) ist ebenfalls optional und kann nur bei *Entity*-Komponenten verwendet werden. Die Komponentenkategorien *service, session, process* und *entity* werden in Kapitel 2.2.4 auf Seite 26 näher beschrieben.

2.2.3 Component Implementation Framework

Während das abstrakte Komponentenmodell die Entwurfssicht auf eine Komponente beschreibt, unterstützt ein Programmiermodell die Erstellung von Komponentenimplementierungen. Ein Hauptziel dabei ist es, Infrastruktur-bedingte und nicht-funktionale Aspekte deklarativ zu beschreiben und diesen Teil der Implementierung weitestgehend zu generieren. Der Komponentenentwickler soll möglichst nur noch die fachliche Logik der Komponenten, also ihre angebotenen Schnittstellen, implementieren müssen. Um diese beiden Teile sauber und möglichst einfach integrieren zu können, beschreibt das *Component Implementation Framework* (CIF) wie funktionale und nicht-funktionale Teile untereinander in Beziehung stehen.

Der nicht-funktionale Code wird in Form sogenannter *component skeletons* generiert. Diese sind gegenüber denen aus CORBA 2 stark erweitert und automatisieren grundlegende Aufgaben wie Port-Management, Navigation zwischen Facets und die Verwaltung des Komponenten-Lebenszyklus. Zur Generierung der Skeletons bedient sich das CIF der *Component Implementation Definition Language* (CIDL), die im folgenden näher beschrieben wird. Der funktionale Code einer Komponente bzw. eines Homes, der vom Komponentenentwickler zu implementieren ist, wird im CCM als *executor* bezeichnet.

CIDL und *executors*

Die *Component Implementation Definition Language* ermöglicht es dem Entwickler, die grobe Implementierungsstruktur sowie Persistenz-Eigenschaften von Komponententypen und Homes deklarativ zu beschreiben. Das zentrale Konstrukt hierbei heißt `composition` und verbindet *component executor* und *home executor*. Außerdem definiert eine `composition` die anzuwendende Komponentenkategorie (z.B. `entity` oder `session`, siehe Kapitel 2.2.4 auf Seite 26), was Auswirkungen auf die generierten Skeletons hat.

Das folgende CIDL Fragment zeigt eine Definition für den `Person` Komponententyp aus den vorangegangenen Beispielen. Durch die Zuordnung des

home executors zu `PersonHome` ergibt sich implizit die Verbindung zwischen dem Komponententyp `Person` aus der IDL und seinem *component executor* `Person_impl`.

```
1  composition entity PersonComposition {
2      home executor PersonHome_impl {
3          implements PersonHome;  // aus IDL
4          manages Person_impl {  // component executor
5              segment JobSegment {
6                provides (job1, job2);  //aus IDL
7              };
8              segment PrivateSegment {
9                provides (private);  // aus IDL
10             };
11         };
12     };
13 };
```

Listing 2.9: Implementierungsbeschreibung in CIDL

Optional besteht die Möglichkeit, den *component executor* in CIDL mit dem Schlüsselwort `segment` zu segmentieren. Ein Segment implementiert dann ein oder mehrere Facets und kann seinen eigenen persistenten Zustand definieren. Die Aufteilung der Implementierung in Segmente hat den Vorteil, dass diese vom Container einzeln aktiviert und passiviert werden können. Dadurch muss nicht für jede Benutzung die gesamte Komponente in den Speicher geladen werden, sondern nur das erforderliche Segment, was Vorteile in Sachen Performanz und Skalierbarkeit bringt. Auch eine deklarative Bindung von Komponenten an den Persistenz-Mechanismus kann in CIDL erfolgen, wird aber hier nicht näher beschrieben.

Implementierungsartefakte

Auf Basis der IDL und CIDL Beschreibungen generiert das CIF Stubs und Skeletons, die einen Großteil der infrastrukturellen Aufgaben übernehmen. Der Komponentenentwickler muss nur noch den *home executor* und den oder die *component executor(s)* implementieren, die zum Teil per Vererbung und zum Teil per Delegation mit den generierten Skeletons verbunden sind. Die Beziehungen zwischen generierten und eigenimplementierten Klassen zeigt ausschnittsweise Abbildung 2.3 am Beispiel der `Person` Komponente. Im *component executor* (hier: `PersonImpl`) müssen alle angebotenen Schnittstellen inklusive Teile der Komponenten-Schnittstelle, sowie callback-Schnittstellen für den Container implementiert werden. Die Klasse `Person_impl` ist das generierte Skeleton, welches einen Teil seiner Aufrufe an `PersonImpl` delegiert. Die Auswahl der zu

verwendenden Executor-Klasse findet im Software Deskriptor statt, der in Kapitel 2.2.5 auf Seite 27 beschrieben wird.

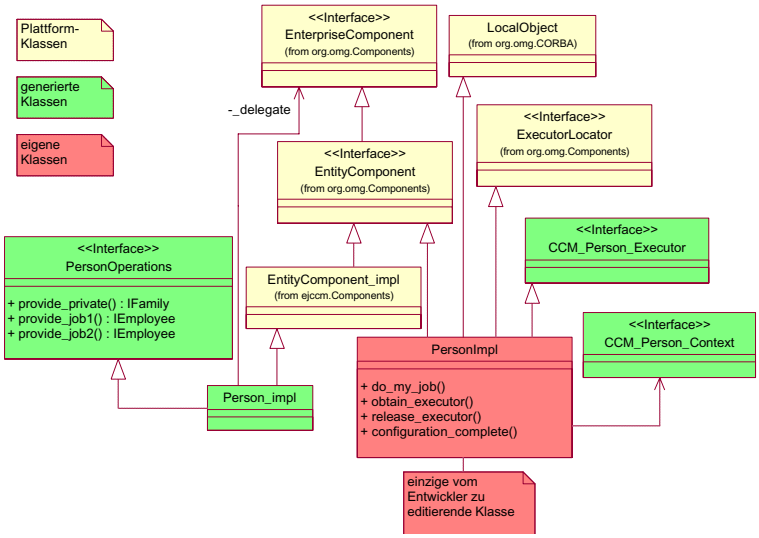

Abbildung 2.3: Implementierungsartefakte am Beispiel

monolithic vs. locator strategy

Es gibt zwei Strategien zur Implementierung eines *component executors*. Bei der *monolithic strategy* besteht der gesamte *component executor* aus einer einzigen Klasse, die alle oben genannten Schnittstellen implementiert. Im Gegensatz dazu kann bei der *locator strategy* für jeden benannten Port der Komponente eine eigene Klasse verwendet werden. Dazu implementiert man einen *main executor*, der über die callback-Methode `obtain_executor()` dem Container eine Instanz der implementierenden Klasse zu diesem Port liefert. Der Entwickler kann also die Implementierung eines Komponententyps weitestgehend modularisieren. Für den Container macht das jedoch in diesem Fall keinen Unterschied, für ihn ist eine solche Komponente trotzdem atomar und wird immer komplett aktiviert, was diesen Ansatz von den segmentierten Exekutoren unterscheidet.

2.2.4 Container Programming Model

Wie bereits erwähnt, bilden Container die Laufzeitumgebung für Komponenten und werden von einer Deployment Plattform, wie einem Applikationsserver oder einer IDE, bereitgestellt. Das *Container Programming Model* beschreibt die grundlegende Container-Architektur und damit das Zusammenspiel zwischen Client und Komponente (*external API types*), Komponente und Container (*container API type*), sowie zwischen Container, ORB und den an ihn angebundenen

Diensten (*CORBA Usage Model*). Diese Architektur wird in Abbildung 2.4 gezeigt.

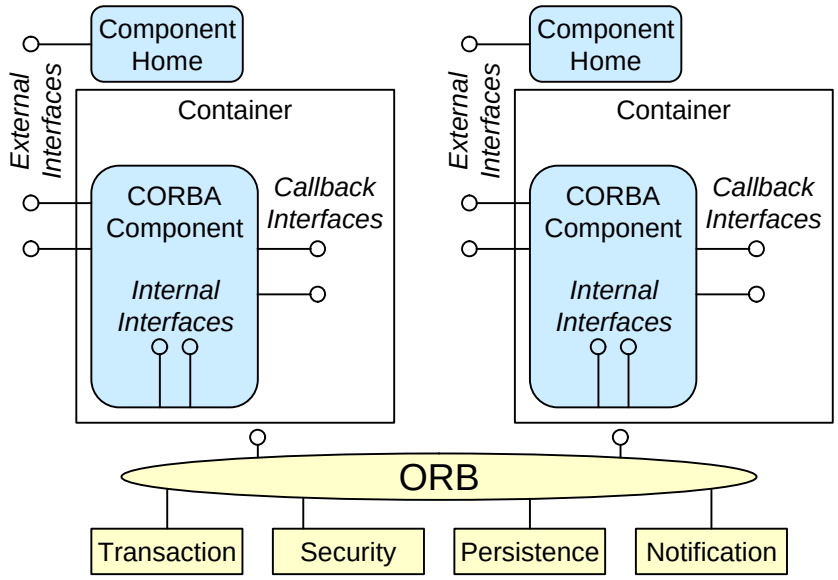

Abbildung 2.4: CCM Container Architektur

Die verschiedenen Schnittstellenarten des Container Programming Models werden in zwei Kategorien zusammengefasst:

- Die *external API types* bilden den Kontrakt zwischen Komponentenentwickler und Komponentennutzer. Man unterscheidet hier zwischen den Home- und den Anwendungs- bzw. Komponentenschnittstellen. Erstere enthalten die Operationen zum Erstellen und Auffinden von Komponenteninstanzen nach dem *factory*- bzw. *finder*- Design Pattern. Das *finder*-Pattern wird nur von Homes unterstützt, für die in der IDL ein Primärschlüssel definiert wurde. Dieser wird benutzt, um Komponenteninstanzen anhand ihrer (persistent gemachten) Identität wiederfinden zu können.

- Als *container API type* bezeichnet man das API Framework für den Komponentenentwickler, welches sich aus internen Container- und Callback-Schnittstellen zusammensetzt und so den Kontrakt zwischen Komponente und Container bildet. In IDL werden diese in Form von lokalen Schnittstellen definiert (neuer modifier `local`), was deutlich macht, dass sie nur lokal und nicht über den ORB erreichbar sind. Es gibt zwei grundlegende container API types: *session* für transiente und *entity* für persistente Objektreferenzen.

Des Weiteren werden drei *CORBA Usage Model* unterschieden, die über die Art der Interaktion zwischen Container, ORB und CORBA Services entscheiden. Diese sollen hier jedoch nicht näher betrachtet werden.

Kategorien von CCM Komponenten

Aus den gültigen Kombinationen von *external API type*, *container API type* und *CORBA Usage Model* leiten sich vier Kategorien von CORBA Komponenten ab, die jeweils unterschiedliche Charakteristika bezüglich Lebenszyklus, Aktivierung, Persistenz, Identität und Transaktionsfähigkeit haben. In Tabelle 2.1 sind diese Kombinationen mit ihren Entsprechungen im EJB Modell aufgelistet.

Service Components haben den kürzesten Lebenszyklus: sie werden nur für die Dauer einer einzelnen Operation aktiviert und danach gleich wieder passiviert. Sie besitzen keinen persistenten Zustand und keine Identität. Service Komponenten können beispielsweise einmalige Berechnungen durchführen, die aus nur einem Aufruf bestehen.

Session Components können eine Reihe von Aufrufen überdauern. Sie haben eine vorübergehende Identität und können eine Transaktion darstellen. Sie werden beim ersten Aufruf aktiviert und am Ende der Transaktion passiviert. Eine Beispielanwendung für Session Komponenten ist ein *Iterator*.

Process Components repräsentieren Geschäftsprozesse mit einem definierten Beginn und einem Ende, wie die Bestellungsabwicklung in einer E-Commerce Anwendung. Zustand und Identität einer Process Komponente sind persistent, aber für Clients nicht direkt sichtbar.

Entity Components sind geeignet, um die wirklich persistenten Elemente einer Anwendung, wie Kunden oder Konten, darzustellen. Ihr Zustand ist persistent und sie besitzen eine persistente Identität, die über einen Primärschlüssel auch für Clients explizit zugänglich ist.

CORBA Usage Model	Container API Type	Primary Key	Component Category	EJB Type
stateless	session	no	Service	-
conversational	session	no	Session	Session
durable	entity	no	Process	-
durable	entity	yes	Entity	Entity

Tabelle 2.1: Die verschiedenen Komponentenkategorien

Dadurch, dass Service und Session Komponenten keine persistenten Identitäten und Zustände haben, sind deren Referenzen nach einer Beendigung des

Server-Prozesses ungültig. Daher sollten Referenzen auf solche Komponenten nicht permanent gespeichert werden, etwa in einem Naming oder Trader Service. Sie sollten nur Funktionalität implementieren, die "self-contained", also in sich abgeschlossen und selbständig ist. Im Gegensatz dazu sind Process und Entity Komponenten auch nach einem Neustart des Servers noch verfügbar und ihre vorherigen Referenzen bleiben gültig.

2.2.5 Packaging und Deployment

Ein weiterer Teil der CCM Spezifikation ist das Packaging und Deployment Modell. Es beschreibt, in welcher Form Komponenten verpackt und weitergegeben werden und wie der Deployment-Prozess aussieht. Man unterscheidet beim *packaging* zwischen *component packages*, in denen jeweils ein einzelner Komponententyp untergebracht wird, und *assembly packages*, die mehrere solcher Komponentenarchive enthalten. Generell sind Packages ZIP bzw. JAR Archive und enthalten sowohl die Implementierungen von Komponenten und Homes[11] als auch *Deskriptoren*, die verschiedene Aspekte des Archivs bzw. der Komponenten beschreiben. Deskriptoren sind XML-Dateien, ihr Format ist in entsprechenden DTDs (*Document Type Descriptions*) festgelegt. Ihr Inhalt dient in erster Linie als Vorgabe für ein Deployment-Werkzeug, in dem die beschriebenen Eigenschaften dann vom Benutzer interaktiv angepasst werden können. Es gibt vier solcher Deskriptortypen, deren Inhalt und Verwendung in den folgenden Abschnitten deutlich werden.

Component Packaging

Ein Komponentenarchiv (*.car) enthält eine oder mehrere Implementierungen genau eines Komponententyps[12] sowie drei verschiedene Arten von Deskriptoren. Der *CORBA Software Descriptor* (*.csd) beschreibt den Inhalt des Archivs und enthält neben generellen Informationen, wie Autor, Beschreibung und Lizenzhinweis, Einzelheiten zu den enthaltenen Implementierungen. An dieser Stelle wird die implementierende Einheit (z.B. Klasse in Java), sowie der Einstiegspunkt zum Erstellen einer Home-Instanz für diesen Komponententyp festgelegt. Das folgende Listing zeigt ausschnittsweise einen solchen Software Deskriptor.

```
1 <softpkg>
2  <pkgtype>CORBA Component</pkgtype>
```

[11] Mit Implementierung ist im Bezug auf Packaging und Deployment immer die ausführbare, also in der Regel kompilierte, Form gemeint.

[12] So kann eine Komponente beispielsweise in verschiedenen Programmiersprachen implementiert ausgeliefert werden. Erst beim Deployment wird dann entschieden, welche zum Einsatz kommt.

```
  <title>Person</title>
  <author>Marco Mosconi</author>
  <description>Eine einfache Komponente...</description>
  <license/>
  <idl><link href="ftp://x.y.z/Person.idl"/></idl>
  <propertyfile><fileinarchive name="Person.cpf"/></propertyfile>
  <implementation>
    <programminglanguage/>
    <dependency/>
    <code type="Java class">
      <fileinarchive name="PersonHomeFactory.class"/>
      <entrypoint>PersonHomeFactory.create</entrypoint>
    </code>
  </implementation>
</softpkg>
```

Listing 2.10: Auszug eines *CORBA Software Descriptors*

Der *CORBA Component Descriptor* (*.ccd) beschreibt einen Komponententyp und hat zwei Aufgaben. Zum einen enthält er die Struktur des Komponententyps, analog zu den Definitionen in IDL, die zur Darstellung in einem Design-Tool verwendet werden könnte. Zum anderen werden in ihm Deployment-Informationen wie Security-,Transaktions- und Threading-Eigenschaften festgelegt, die zur Konfiguration des Containers dienen. Einige der möglichen Elemente werden im folgenden Listing beispielhaft gezeigt:

```
<corbacomponent>
  <componentkind>
    <entity><servant lifetime="container"/></entity>
  </componentkind>
  <threading policy="multithread"/>
  <componentfeatures name="Person" repid="IDL:Person:1.0">
    <ports>
      <provides providesname="job1" repid="IDL:IEmployee:1.0">
        <operationpolicies>
          <operation name="do_my_job">
            <requiredrights>...</requiredrights>
            <transaction use="required"/>
          </operation>
        </operationpolicies>
      </provides>
    </ports>
  </componentfeatures>
</corbacomponent>
```

Listing 2.11: Auszug eines *CORBA Component Descriptors*

In einem weiteren Deskriptor, einem *Component Property File* (*.cpf), können Default-Werte für die Attribute des Komponententyps angegeben werden. Dieser Deskriptortyp wird im nächsten Abschnitt gezeigt.

Assembly Packaging

Ein Assembly Archiv (*.aar) repräsentiert eine Menge miteinander verbundener Komponenten und enthält Komponentenarchive (s.o.), einen Assembly Deskriptor sowie Konfigurationsdateien. Im *Component Assembly Descriptor* (*.cad) sind die Informationen angesiedelt, die für die Instanziierung, Verbindung und Konfiguration der einzelnen Komponenten beim Deployment nötig sind[13]. Das folgende Listing zeigt die Struktur eines solchen Deskriptors anhand eines kleinen Ausschnitts der möglichen Elemente. Beispielsweise lässt sich die Instanziierung von Komponenten, sowie deren Konfiguration und die Registrierung bei einem Naming- oder Trading-Service definieren (Zeilen 5-8). Mit den Elementen `<processcollocation>` und `<hostcollocation>` kann die Verteilung der Homes und ihrer Komponenten bezüglich Prozessen und Hosts gesteuert werden. Im `<connections>` Element (Zeilen 13-16) werden die Verbindungen von Ports definiert, ein konkretes Beispiel dazu zeigt Abbildung 2.5 im nächsten Abschnitt zum Deployment.

```
1  <componentassembly>
2    <componentfiles/>
3    <partitioning>
4      <homeplacement>
5        <componentinstantiation>
6          <componentproperties/>
7          <registercomponent/>
8        </componentinstantiation>
9      </homeplacement>
10     <processcollocation/>
11     <hostcollocation/>
12   </partitioning>
13   <connections>
14     <connectinterface/>
15     <connectevent/>
16   </connections>
17 </componentassembly>
```

[13] Der Assembly Deskriptor entspricht somit dem, was man allgemein oft als Deployment Deskriptor bezeichnet.

Listing 2.12: Struktur eines *Component Assembly Descriptors*

Die bereits erwähnten *Component Property Files* (*.cpf) dienen hier der initialen und individuellen Konfiguration von Komponenteninstanzen über ihre Attribute. Das folgende Beispiel zeigt die möglichen Arten von Datentypen, die auf diese Weise konfiguriert werden können. Die Zeilen 2 bis 4 bewirken, dass das Komponenten-Attribut `age` vom Typ `short` mit dem Wert 35 belegt wird.

```
1 <properties>
2   <simple name="age" type="short">
3     <value>35</value>
4   </simple>
5   <sequence/>
6   <struct/>
7   <valuetype/>
8 </properties>
```

Listing 2.13: Beispiel eines *Component Property Files*

Der Umfang und die Komplexität insbesondere des Assembly Deskriptors legen es nahe, diese nicht von Hand editieren zu müssen. Dies sollte über eine graphische Benutzerschnittstelle, z.B. einer IDE oder eines Deployment-Tools geschehen.

Deployment

Beim Deployment werden die Komponenten auf den dafür vorgesehenen Hosts installiert, instanziiert, verbunden und konfiguriert. Es können sowohl einzelne Komponentenarchive deployed werden, als auch komplette Assemblies. Bei ersteren entfällt das Verbinden von Ports, letztere können über mehrere Hosts verteilt werden[14]. Die Tatsache, dass auf jedem Zielhost ein kompatibler Applikationsserver läuft und die benötigten Container bereitstellt, sei vorausgesetzt. Mit Hilfe eines Deployment-Tools kann der Anwender dann entscheiden, welche Komponenten auf welchem Host laufen sollen und die entsprechenden Archive auf die Zielhosts übertragen. Dort werden zunächst, falls notwendig, die Container erzeugt. Danach können Homes und Komponenten nach den Angaben des Benutzers und des Assembly Deskriptors instanziiert werden. Im nächsten Schritt werden diese Instanzen konfiguriert und ihre Ports miteinander verbunden, wie Abbildung 2.5 beispielhaft zeigt. Nachdem alle Instanzen erzeugt, konfiguriert und verbunden sind, wird auf jeder Komponenteninstanz die von ihr

[14] Im Gegensatz zu EJB-Anwendungen, die immer nur zusammenhängend auf einem Server installiert werden können.

zu implementierende callback-Methode `configuration_complete()` aufgerufen. Dieser Aufruf signalisiert das Ende der Konfigurationsphase und leitet die operationale Phase der Komponente ein. Man versucht damit, diese beiden Phasen klar voneinander zu trennen, wobei eine Forcierung seitens der Plattform nicht festgeschrieben ist.

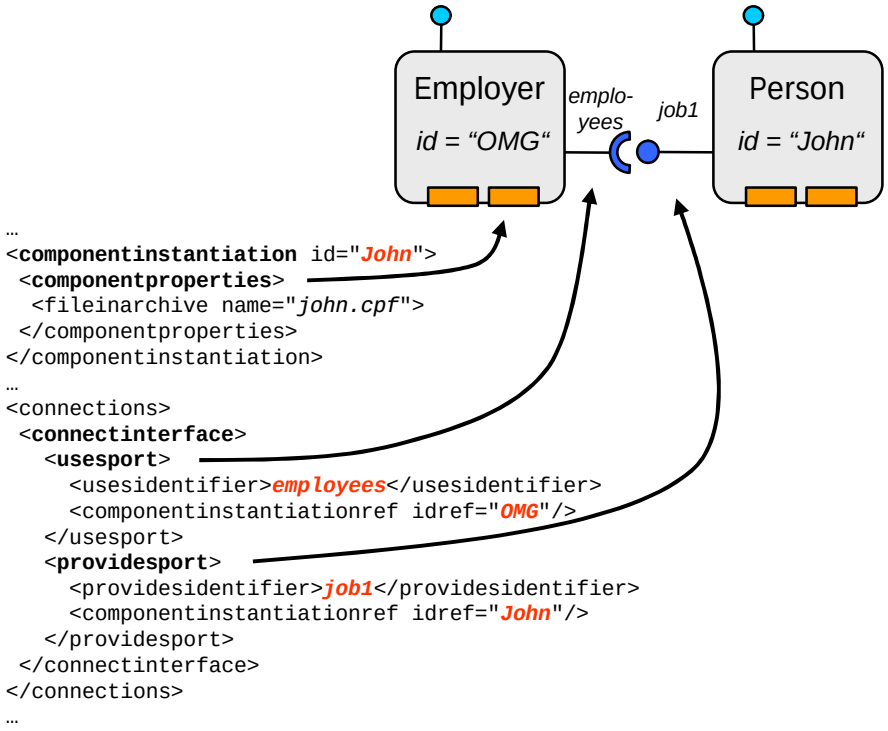

Abbildung 2.5: Konfiguration und Port-Verbindung per Assembly Deskriptor. Komponenteninstanzen werden innerhalb des Deskriptors über ihre ID referenziert, Ports über ihre Bezeichner. Die Konfiguration erfolgt durch Zuweisung eines Property Files.

2.2.6 Der Entwicklungsprozess

Nachdem nun die einzelnen Teile des CCM beschrieben wurden, soll dieser Abschnitt einen Überblick über die verschiedenen Phasen und die beteiligten Rollen bei der Erstellung einer CCM-Anwendung geben (vergleiche hierzu die allgemeinen Aussagen in Kapitel 2.1.3 auf Seite 10). Abbildung 2.6 zeigt den Ablauf vom Entwurf bis zum fertigen, deployment-fähigen Package anhand der jeweils beteiligten Artefakte und Werkzeuge. Sie ist grob unterteilt in die Phasen Entwurf/Entwicklung, Packaging und Assembly, deren einzelne Schritte im

folgenden kurz erklärt werden. Bei Entwicklung und Packaging wird dabei aus Gründen der Übersicht die Erstellung nur eines Komponententyps betrachtet[15].

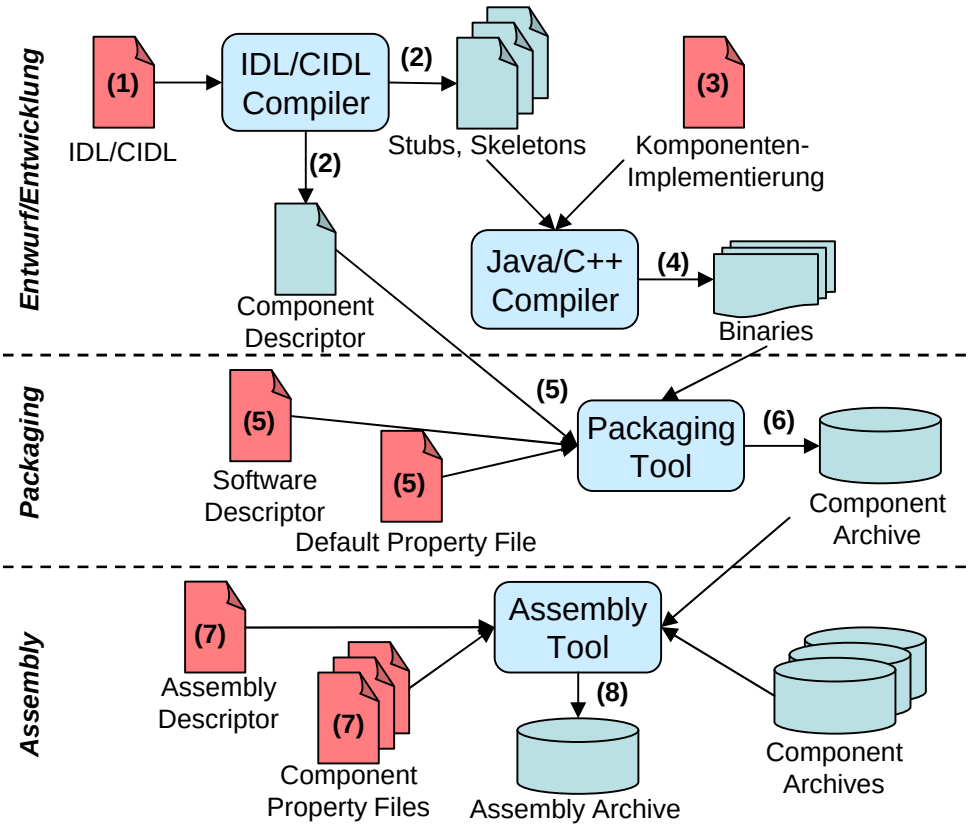

Abbildung 2.6: Überblick über den CCM Entwicklungsprozess und seine Artefakte

1. Zuerst werden der Komponententyp und das zugehörige Home vom *component designer* in IDL definiert. Dieser Schritt gehört zur Entwurfsphase, da hier nur die Schnittstellen und somit die Komponentenarchitektur der Anwendung festgelegt werden. Die deklarative Beschreibung der Implementierungsstruktur mittels CIDL kann auch noch dem implementierungsnahen Entwurf zugeordnet werden.

2. Über entsprechende Compiler und Code-Generatoren der Plattform werden aus den IDL- und CIDL-Beschreibungen die CORBA Stubs und Skeletons, sowie der vorläufige Component Deskriptor generiert.

[15] Normalerweise wird eine IDL-Datei mehrere Komponenten und diverse Schnittstellen beschreiben, sowie weitere importieren. Dies führt dann auch leicht zu einer drei- bis vierstelligen Anzahl generierter Stubs.

3. Unter Verwendung der generierten Stubs und Skeletons implementiert der *component developer* den Komponententyp und das Home in Form von *home* und *component executors*. Es können an dieser Stelle, wie bereits erwähnt, mehrere Implementierungen desselben Komponententyps erstellt werden.

4. Mit Hilfe von Werkzeugen der verwendeten Programmiersprache werden Stubs, Skeletons und Executors in ihre ausführbare Form gebracht[16]. Im Falle von Java wären dies *class files*, bei C++ unter Windows *dll*s. Diese sind im Diagramm der Einfachheit halber als "Binaries" bezeichnet.

5. In diesem Schritt wird der Component Deskriptor um die Informationen ergänzt, die nicht generiert werden konnten und der Software Deskriptor erstellt. Außerdem kann optional ein Property File mit einer Default-Konfiguration für den Komponententyp erstellt werden. Dieser Vorgang wird hier der Packaging-Phase zugeordnet und kann von einer weiteren Rolle, dem *component packager*, eventuell unter Zuhilfenahme graphischer Werkzeuge, ausgeführt werden.

6. Zusammen mit der ausführbaren Implementierung werden die drei Deskriptoren dann von einem *Packaging Tool* zu einem Komponentenarchiv verpackt, das als solches deployed oder in einem nächsten Schritt in einem Assembly verwendet werden kann.

7. Soll aus mehreren Komponenten ein Assembly gebildet werden, so erstellt der *component assembler* den Assembly Deskriptor sowie Property Files für die beim Deployment zu erzeugenden Komponenteninstanzen. Auch hier sollte eine entsprechendes Werkzeug (zum Beispiel eine integrierte Entwicklungsumgebung oder ein spezielles *Assembly Tool*) den Anwender unterstützen.

8. Schließlich werden die Komponentenarchive, der Assembly Deskriptor und die Property Files zu einem Assembly Package zusammengeführt, das alle Informationen enthält um auf einem oder mehreren Hosts deployed zu werden.

Die Frage, inwieweit die Phasen Entwurf/Entwicklung, Packaging und Assembly oder sogar einzelne Schritte tatsächlich auf verschiedene Rollen verteilt werden, hängt von der Größe des Projekts ab. Es ist auch denkbar, alle beschriebenen Schritte in eine Entwicklungsumgebung zu integrieren und von einer einzelnen Person durchführen zu lassen, was jedoch nur bei sehr kleinen Anwendungen funktionieren dürfte. Die Möglichkeit der strikten Aufteilung in räumlich

[16] Da prinzipiell auch interpretierte Programmiersprachen unterstützt werden, wurde hier bewusst auf den Begriff Compiler verzichtet. Im weiteren Verlauf dieser Arbeit wird jedoch von Java oder einer ähnlichen Implementierungssprache ausgegangen.

und zeitlich getrennte Rollen bietet in jedem Fall ein hohes Maß an Flexibilität und Skalierbarkeit in Bezug auf den Entwicklungsprozess.

Kapitel 3

Aspektorientierte Softwareentwicklung mit *Object Teams*

Die aspektorientierte Softwareentwicklung erwuchs im wesentlichen aus der Erkenntnis, dass die Objektorientierung allein nicht ausreicht, um alle Aspekte einer Software geeignet zu modularisieren. Daraus entstanden akademische wie industrielle Forschungsaktivitäten im Bereich des Programmiersprachenentwurfs, die sich zum Ziel gesetzt haben, dem Programmierer Möglichkeiten an die Hand zu geben, diese Aspekte programmiersprachlich zu erfassen. Zunächst entwickelte sich daraus die aspektorientierte Programmierung (AOP) [KLM+97], die hauptsächlich durch die Entwicklung und zunehmende Popularität von *AspectJ* [Asp] vorangetrieben wurde. Mittlerweile beschränkt sich die Forschung zur Aspektorientierung längst nicht mehr nur auf die Implementierungsebene, sondern betrachtet auch die Phasen der Analyse und Modellierung[1]. Die Aspektorientierung setzt, ähnlich der Komponentenorientierung, in der Regel auf objektorientierten Strukturen und Konzepten auf und wird diese eher erweitern als ablösen.

Um einen Überblick über den Bereich der aspektorientierten Softwareentwicklung zu bekommen, führt Kapitel 3.1.1 zunächst in die Problematik ein, die man durch *Aspekte* und zugehörige Konzepte, die in Kapitel 3.1.2 umrissen werden, in den Griff zu bekommen versucht. Eine Übersicht ausgewählter aspektorientierter Ansätze folgt in Kapitel 3.1.3. Schließlich führt Kapitel 3.2 in das *Object Teams* Modell ein, das in dieser Arbeit auf Komponententechnologien angewandt wurde.

[1] Die verschiedenen Aktivitäten werden unter dem Oberbegriff "Aspektorientierte Softwareentwicklung" (*Aspect-Oriented Software Development - AOSD*) zusammengefasst.

3.1 Begriffe und Konzepte

3.1.1 Die Problematik

Die Objektorientierung bietet sowohl auf Entwurfs- als auch auf Programmiersprachenebene Möglichkeiten zur Strukturierung und Modularisierung von Software. Das grundlegende Konzept zur Modularisierung ist hier das Objekt bzw. die Klasse. Diese kapseln Struktur und Funktionalität und stellen eine abstrakte Sicht auf Elemente der Anwendungsdomäne dar. Mit Hilfe von Vererbung und Objekt-Komposition werden sie miteinander in Beziehung gesetzt und bilden so die Anwendungsstruktur. Jedoch lassen sich auf diese Weise nicht alle Aspekte des Verhaltens eines Systems sauber voneinander getrennt modellieren und implementieren.

crosscutting concerns

Als *concerns* bezeichnet man bestimmte Belange oder Anforderungen an ein Programm, die in der Phase der Anforderungsermittlung und -analyse meist noch isoliert formuliert werden können, sich in der Implementierung jedoch in der Regel über mehrere Klassen verteilen. Beispiele dafür sind unter anderem nichtfunktionale Anforderungen wie Persistenz- und Transaktionseigenschaften, Verteilung oder Logging, sowie eher funktionale Anforderungen wie Druckfunktionalität oder Visualisierung. Die Tatsache, dass diese Belange oft quer zu einer objektorientierten Modularisierungsstruktur liegen und sie quasi "durchschneiden" führt zu der Bezeichnung *crosscutting concerns*.

scattering und *tangling*

Konkret haben *crosscutting concerns* in der Implementierung zur Folge, dass der Code zur Realisierung einzelner Concerns über mehrere Klassen verteilt ist. Man spricht hier von *scattering* (Verstreuung). Auf der anderen Seite werden in einer Klasse meist mehrere Concerns nebeneinander implementiert, was als *tangling* (Verflechtung) bezeichnet wird. Abbildung 3.1 zeigt dies am Beispiel eines Logging-Concerns. Während die Anforderung: "Alle Objekte sollen [ein bestimmtes Ereignis] in ein Log-File schreiben." noch getrennt formuliert werden kann, "durchschneidet" ihre Implementierung alle beteiligten Klassen.

separation of concerns

Zur Vermeidung von scattering und tangling bedarf es einer klaren Trennung der einzelnen Concerns über alle Phasen der Softwareentwicklung hinweg. Dafür hat sich der Begriff *separation of concerns* für dieses wichtige Grundprinzip

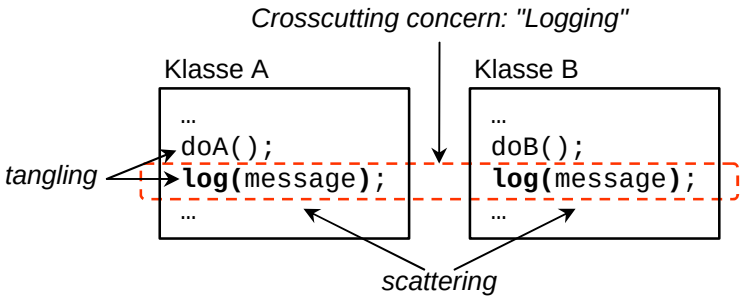

Abbildung 3.1: Scattering und Tangling eines Concerns

der Softwaretechnik etabliert. Das übergeordnete Ziel hierbei ist es, die verschiedenen Concerns, die sich auch gegenseitig überlappen können, voneinander getrennt definieren, modellieren, implementieren und einsetzen zu können.

Man kann verschiedene Typen von Concerns als Dimensionen auffassen, in denen Modularisiert werden kann. Zum Beispiel bietet die Objektorientierung die Modularisierung entlang einer Struktur-Dimension, Klassen kapseln dabei die Concerns dieser Dimension. Wie im nächsten Abschnitt deutlich wird, spannen *Aspekte* als Modularisierungskonzept für eher verhaltensbezogene Concerns eine weitere Dimension auf. Auch die verschiedenen Concern-Dimensionen sollen getrennt behandelt und gleichzeitig eingesetzt werden können, was den Begriff *multi-dimensional separation of concerns* geprägt hat.

3.1.2 Modularisierung von Concerns durch Aspekte

Wie schon erwähnt sollen *Aspekte* dabei helfen, bestimmte verhaltensbezogene Concerns zu modularisieren, die sich bei einer rein objektorientierten Struktur durch scattering und tangling bemerkbar machen würden. Die Aspektorientierte Programmierung führt dazu Aspekte als neues Programmierkonstrukt ein, die meist auf ähnlicher Ebene wie Klassen angesiedelt sind. Die Terminologie in diesem Abschnitt orientiert sich im Wesentlichen an AspectJ, einer der "klassischen" allgemeinen aspektorientierten Programmiersprachen.

Die Definition eines Aspektes unterteilt sich meist in die Angabe, an welchen Stellen der Aspekt "angewendet" werden soll und die Implementierung der Aspektfunktionalität. In einem Prozess, den man als "Weben" (*aspect weaving*) bezeichnet, wird die Aspektimplementierung an den definierten Stellen in die bestehenden Klassenimplementierung eingefügt.

joinpoints

Die Stellen in der Basisimplementierung, an denen per Aspekt Verhalten verändert oder hinzugefügt werden soll, werden *joinpoints* genannt. Sie bestimmen

die Art der Komposition von Aspekten mit den normalen Anwendungs-Klassen. Joinpoints können auf Klassen- und Methodennamen basieren, wobei auch Platzhalter und reguläre Ausdrücke denkbar sind, aber auch Stellen im Kontrollfluss einer Anwendung definieren, an denen ein Aspekt wirksam werden soll. Aktuelle Bestrebungen gehen in Richtung eigener Abfragesprachen für joinpoints (sogenannte *joinpoint query languages*), mit denen joinpoints nicht mehr explizit manuell aufgeschrieben werden müssen, sondern in einem separaten Vorgang dynamisch bestimmt werden können.

dynamisches und statisches Weben

Da die meisten aspektorientierten Techniken auf der Manipulation von Source-Code basieren, wurde hierfür der Begriff des "Webens" geprägt. Der Aspektcode wird dabei an den per joinpoints definierten Stellen in den Originalcode eingewoben. Da die kleinste zu ändernde Einheit in der Regel die Methode ist, ergeben sich drei Möglichkeiten zur Adaptierung: der Aspektcode kann vor, nach oder anstelle der ursprünglichen Methode eingefügt werden. In letzterem Fall sollte die Möglichkeit bestehen, die Originalmethode aus dem Aspektcode aufzurufen.

Aspekte müssen aber nicht unbedingt auf Sourcecode-Ebene eingewoben werden. Verschiedene Ansätze arbeiten mit einer Adaptierung zur Lade- oder sogar Laufzeit der Basisprogramme. Dies kann als dynamisches Weben bezeichnet werden, im Unterschied zum statischen Weben auf Sourcecode-Ebene. Einen weiteren Schritt stellt die dynamische Aktivierbarkeit von Aspekten nur für bestimmte Objekte zur Laufzeit dar. Die meisten Ansätze arbeiten jedoch auf Klassenebene, so dass in der Regel alle Objekte einer Klasse betroffen sind.

3.1.3 Verschiedene Ansätze

Im Folgenden sollen einige ausgewählte Beispiele aspektorientierter Ansätze kurz mit ihren besonderen Eigenschaften vorgestellt werden.

- *AspectJ* [Asp] ist eine aspektorientierte Erweiterung für Java. Aspekte werden hier ähnlich wie Java-Klassen notiert, die ein eigener Compiler mit den Anwendungs-Klassen verwebt und reguläre Java class files erzeugt. Es handelt sich also hierbei um statisches Weben auf Sourcecode-Ebene, wobei Aspektimplementierung (hier: *advices*) und joinpoints (hier: *pointcuts*) gemeinsam innerhalb eines Aspekts definiert werden.

- *Hyper/J* [OT00] ist die Umsetzung des *hyperspaces* genannten Ansatzes für *multi-dimensional separation of concerns* in Java. Concerns und ihre Dimensionen werden hier in Form einer *concern matrix* beschrieben und mit Hilfe sogenannter *hypermodules* komponiert. Die Implementierung der Concerns geschieht in Java, die Kompositionsregeln und Beziehungen zwischen ihnen werden davon getrennt in einer eigenen Sprache

definiert. Die Komposition findet zur Compile-Zeit statt, ist also auch hier statisch.

- Der *Composition Filters* [BA01] Ansatz verwendet sogenannte Filter zur Kapselung von crosscutting concerns. Filter werden deklarativ aufgeschrieben und an Objekte "angehangen". Ein- und ausgehende Nachrichten dieser Objekte werden dann durch einen oder mehrere Filter geleitet, die so Einfluss auf das Verhalten nehmen können. Durch ein *superimposition* Prinzip können auch joinpoints über mehrere Objekte hinweg definiert werden.

- Das *Darwin* Modell [Kni00] beschreibt die typsichere Erweiterung klassenbasierter objektorientierter Sprachen um statische und dynamische objektbasierte Vererbung (*Delegation*). Durch Delegation lässt sich das Verhalten von Objekten zur Laufzeit verändern, indem ihre parent-Objekte neu zugewiesen werden. Auf diese Wiese lassen sich auch gut Rollen-Objekte implementieren. Die zugehörige Java-Erweiterung heißt *Lava* und erzeugt ähnlich wie AspectJ normalen Java byte-code.

- *JMangler* [KCA01] ist ein Framework zur Adaption von Java-Programmen zur Ladezeit. Aspekte können hier in Form von *Transformator Komponenten* auf Klassen angewendet werden, die nur im byte-code vorliegen oder erst dynamisch erzeugt oder geladen werden. Der Sourcecode der zu adaptierenden Anwendung wird dazu nicht benötigt. Es können auch mehrere, unabhängig entwickelte Transformer zugleich angewendet werden. JMangler wird unter anderem im Rahmen der Java-Implementierung des *Object Teams* Modells eingesetzt [Hun03].

- *Lasagne* [TJJ00, TVJ+01b, TVJ+01a] ist ein Modell zur Herstellung anpassbarer Middleware und verteilter Dienste. Es ermöglicht die Komposition von Aspekten zur Laufzeit, also die dynamische Anpassung eines Systems, sowie eine Kontext-sensitive Auswahl dieser Aspekte und damit eine Client-spezifische Anpassung. Aspekte werden mit Wrappern auf Instanzebene realisiert, wodurch die Komposition von Erweiterungen zur Laufzeit möglich wird.

3.2 *Object Teams*

Das *Object Teams* Modell [Her02b] ist ein Programmiermodell, welches aus einer Reihe vorangegangener Entwicklungen im Bereich aspektorientierter Programmiersprachen und Modularisierungskonzepte hervorgegangen ist. Es vereint viele der Vorteile anderer Ansätze, kommt dabei aber mit relativ wenig neuen Konstrukten aus, was die Erlernung und Anwendung erleichtern sollte.

Die Interna des Modells hingegen sind durchaus komplex und auch noch nicht alle Details endgültig definiert. Das allgemeine Ziel ist die Erfassung und Implementierung von Objekt-Kollaborationen in wiederverwendbaren, unabhängig entwickelten Modulen, die nachträglich in bestehende Anwendungen integriert werden können. In diesem Abschnitt soll das *Object Teams* Modell mit seinen wichtigsten Features vorgestellt werden[2], wobei sich die Beschreibungen und Code-Beispiele hauptsächlich an der *Object Teams/Java* Implementierung [Bin02, Hun03] orientieren.

3.2.1 Kollaborationen und Rollen

In objektorientierten Anwendungen lässt sich neben der strukturellen Modularisierung durch Klassen meist auch eine zweite Dimension mit den verschiedenen Kollaborationen zwischen den Objekten dieser Klassen identifizieren. Eine Kollaboration umfasst immer mehrere Objekte, die ihrerseits in verschiedenen Kollaborationen beteiligt sein können und dort jeweils unterschiedliche Rollen spielen. In einer normalen objektorientierten Implementierung werden alle Rollen eines Objekts in seiner Klasse implementiert. Es gibt keine Möglichkeit, sowohl die Struktur als auch die Kollaborationen zu modularisieren. Die Kollaborationen bilden so crosscutting concerns, wie in Abbildung 3.2 dargestellt.

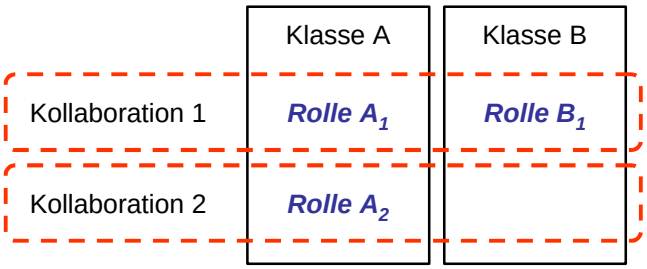

Abbildung 3.2: Kollaborationen als crosscutting concerns

Ansätze zum *collaboration-based design* haben zum Ziel, Kollaborationen in voneinander getrennten Modulen implementieren zu können. Die bereits vorgestellten aspektorientierten Ansätze bieten zwar prinzipiell die Möglichkeit der Implementierung einzelner Concerns, doch fehlt es ihnen an einem Modul-Konzept für die Kapselung ganzer Kollaborationen. Des Weiteren gibt es das Konzept von

[2] Auf die Beschreibung diverser Details wie der Vererbungssemantik von Rollen und Teams sowie *externalized roles* wird hier aus Platzgründen verzichtet. Diese sind für das Verständnis dieser Arbeit nicht von Bedeutung und können bei Interesse in [Her02b] und [Her03] nachgelesen werden.

Rollen-Objekten, die das Verhalten von Objekten verändern, indem sie verschiedene "Sichten" darauf implementieren und quasi zwischen Objekt und Client geschaltet werden. *Object Teams* vereint Ideen und Techniken dieser Ansätze zu einem konsistenten Modell, das in den folgenden Abschnitten beschrieben wird.

3.2.2 *Teams* kapseln Kollaborationen

Als Konzept zur Kapselung von Objekt-Kollaborationen wird ein *Team* als neues Modul eingeführt. Es dient als Container für eine Menge von Rollen einer Kollaboration, die getrennt von ihrer strukturellen Basisanwendung aufgeschrieben und dann auf diese angewendet werden kann. Ein Team ist eine instanziierbare Einheit, die mehrere Rollenklassen sowie eigene Methoden implementieren kann und zur Laufzeit einen Kontext für die enthaltenen Rollenobjekte bildet. Teams werden in Object Teams/Java in Form von Klassen, versehen mit dem neuen Schlüsselwort `team`, und ihre Rollen als innere Klassen definiert. Als Beispiel ist in Listing 3.1 das *Observer Pattern* [GHJV95] als abstraktes Team mit den Rollen `Observer` und `Subject` implementiert. Auf einem Subject können Observer an- und abgemeldet werden (`attach()` und `detach()`), die von ihm benachrichtigt werden, wenn sich sein Zustand ändert (`notify()`).

```
1  team class ObserverPattern {
2    abstract class Observer { // Rollenklasse
3      abstract void update(Subject s);
4    }
5    class Subject { // Rollenklasse
6      List observers;
7      void attach (Observer o) { observers.add(o); }
8      void detach (Observer o) { observers.remove(o); }
9      void notify () {
10       // für alle o in observers: o.update(this)
11     }
12   }
13 }
```

Listing 3.1: Das Observer Pattern als abstraktes Team

Die Rollen sind hier völlig unabhängig von der Basisanwendung implementiert, die sie adaptieren sollen. Um deklarativ vollständig zu sein, werden Methoden der Basisanwendung, die von den Rollen benötigt werden (*expected interfaces*), aber noch nicht bekannt sind, als abstrakte Methoden definiert und sind somit innerhalb des Teams benutzbar. Später werden diese abstrakten Methoden dann an Methoden der Basisklassen gebunden. In Listing 3.1 ist zum Beispiel die Methode `update` abstrakt, da an dieser Stelle nicht bekannt ist, was im Observer passieren soll, wenn sich ein Subject geändert hat. Rollen können dabei

auch neue Kollaborationen zwischen vorher einander unbekannten Klassen aufspannen, wie im Beispiel zwischen den Basisklassen von Subject und Observer.

3.2.3 Bindung von Rollen an Basisklassen

Um eine Basisanwendung zu adaptieren, müssen die Joinpoints festgelegt werden, an denen eine Adaption des Originals vorgenommen werden soll. Dies geschieht bei Object Teams durch die Bindung von Rollenklassen an Basisklassen und die Definition von sogenannten *callin* und *callout bindings*. Die Bindung von Rollen eines Teams an die entsprechenden Klassen der Basisanwendung erfolgt innerhalb eines sogenannten *Konnektors*. Ein Konnektor ist ein Team, das mindestens eine der genannten Bindungsarten enthält. Im Idealfall implementiert ein abstraktes Team die Rollen und ein weiteres fungiert als Konnektor, der alle Bindungen an eine bestimmte Basis enthält. Damit kann ein Team unabhängig von seinem Einsatz implementiert werden und ist somit wiederverwendbar. Konnektor und Team stehen in einer Vererbungsbeziehung, bei der der Konnektor die Rollen des Teams implizit erbt und ihre Bindungen definieren kann. Welche Basisklasse welche Rolle spielt, wird mit dem Schlüsselwort `playedBy` in der Rollendefinition angegeben.

Listing 3.2 zeigt einen Konnektor, der vom Team aus Listing 3.1 erbt und für die Rollen `Observer` und `Subject` Bindungen definiert. Die Basisklasse `Model` sei eine beliebige Klasse, deren Zustand sich durch Aufruf ihrer Methode `edit()` ändert und die nichts von Observern oder der Klasse `View` weiß. Die Klasse `View` wiederum kann Objekte vom Typ `Model` mit `show(Model m)` visualisieren, weiß aber nicht, wann sich deren Zustand ändert. Dazu wird das `ObserverPattern` Team auf diese beiden Klassen angewandt, um nachträglich zu erwirken, das jede Änderung eines `Model`-Objekts eine Aktualisierung der `View` zur Folge hat.

```
1  team class ObserveModel extends ObserverPattern {
2    class Observer playedBy View {
3      // callout binding:
4      update(Subject s) -> show(Model m) with {s->m};
5    }
6    class Subject playedBy Model {
7      // callin binding:
8      notify <- after edit;
9    }
10 }
```

Listing 3.2: Anwendung des Observer Patterns auf Model und View durch einen Konnektor

In Zeile 2 und 6 werden die Rollenklassen den Basisklassen zugeordnet, die

unterschiedlichen Bindungen der Methoden in Zeile 4 und 8 werden im folgenden erläutert.

callout bindings

Mit Hilfe der *callout bindings* werden Abhängigkeiten der Rollen von Funktionalitäten der Basis aufgelöst. Dazu werden abstrakte Methoden einer Rollenklasse durch Delegation an Methoden ihrer Basisklasse gebunden. Ein Callout wird durch einen Pfeil von der Rollenmethode zur Basismethode (->) dargestellt, wie in Listing 3.2, Zeile 4, zu sehen [3]. Dort ist auch die Möglichkeit des Abbildens von Parametern gebundener Methoden dargestellt. Mit dem Schlüsselwort `with` können im anschließenden Block die Parameter von Rollen- und Basismethode einander zugewiesen werden, wobei auch Umrechnungen sowie eine Anpassung des Rückgabewertes möglich sind. Wird keine expilizite Zuordnung vorgenommen, werden die Parameter nach Möglichkeit eins zu eins übergeben.

callin bindings

Ein *callin* adaptiert das Verhalten einer Basisklasse und entspricht dem Einweben von Aspektcode in anderen aspektorientierten Sprachen. Allerdings geschieht das Weben bei Object Teams/Java nicht auf Sourcecode-Ebene, sondern zur Ladezeit, so dass die Basisklassen nicht einmal im Sourcecode vorliegen müssen. Eine Callin Bindung definiert, bei welchen Methoden der Basisklasse eine bestimmte Rollenmethode ausgeführt werden soll. Dafür gibt es drei Möglichkeiten:

- `before`: vor dem Aufruf der Basismethode
- `after`: nach dem Aufruf der Basismethode
- `replace`: anstelle des Aufrufs der Basismethode

Bei den ersten beiden Varianten wird die Originalmethode immer aufgerufen, bei der dritten wird der Aufruf ersetzt durch einen Aufruf der Callin-Methode der Rolle. Die Implementierung einer solchen replace-Callin Methode kann explizit die Original-Methode der Basis aufrufen. Dies wird *base-call* genannt und geschieht über einen Aufruf von `base.<name>`, ähnlich dem Aufruf von `super` bei normaler Vererbung. `<name>` ist hierbei der Name der Rollenmethode.

In Listing 3.2, Zeile 8, wird zum Beispiel definiert, dass nach der Ausführung von `edit` auf einem Objekt der Basisklasse `Model` ein Aufruf an die Methode `notify` der gebundenen Rollenklasse `Subject` erfolgen soll. Auch Callin Bindungen können die oben beschriebenen Parameter-Abbildungen enthalten.

[3] Es gilt die Regel, dass bei allen Bindungen die Rollen bzw. ihre Elemente auf der linken Seite und Basen bzw. ihre Elemente auf der rechten Seite stehen.

Teamaktivierung

Die Adaptierungen durch Callins sind nur bei aktiviertem Team wirksam. Wurde ein Team mit dem Modifier `static` als statisch deklariert, so ist dieses immer aktiv. Andernfalls kann eine Team-Instanz dynamisch zur Laufzeit mit den Methoden `activate()` und `deactivate()` aktiviert bzw. deaktiviert werden. Alternativ kann man mit einem `within` Block erreichen, dass ein Team nur für eine festgelegte Folge von Statements aktiv ist. Das folgende Listing zeigt die verschiedenen Möglichkeiten.

```
1 ObserveModel o = new(ObserveModel);
2 o.activate();
3   // das Team o ist nun global aktiv
4 o.deactivate();
5   // das Team o ist nun nicht mehr aktiv
6 within (o) do{
7   // nur innerhalb dieses Blocks ist das Team aktiv
8 }
```

Listing 3.3: Aktivierung einer Team-Instanz

Durch die explizite Team-Aktivierung ist es möglich, zur Laufzeit zu entscheiden wann eine Verhaltensänderung der Basisobjekte erfolgt und wann nicht. Bei deaktiviertem Team verhält sich die Basisanwendung so, als wäre keine Adaptierung vorgenommen worden. Unter bestimmten Bedingungen werden Team-Instanzen auch automatisch aktiviert, beispielsweise beim Aufruf einer Team-Methode.

3.2.4 Laufzeitmodell

Zur Laufzeit kann es mehrere Instanzen eines Teams geben[4], deren Rollenobjekte die selben Basisobjekte adaptieren. Dabei existiert innerhalb eines Teams immer exakt ein Rollenobjekt zu jedem Basisobjekt, für dessen Klasse eine Bindung definiert ist. Rollenobjekte sind in ihrer jeweiligen Team-Instanz gekapselt und außerhalb dieser normalerweise nicht sichtbar [5]. Das Team verwaltet auch die Verknüpfungen zwischen Rollen- und Basisobjekten.

Abbildung 3.3 zeigt eine Konnektor-Instanz mit ihren Rollenobjekten und den zugehörigen Basisobjekten zur Laufzeit, sowie den Kontrollfluss für das Beispiel aus Listing 3.2. Entsprechend der Definition im Konnektor wird nach dem Aufruf von `m.edit()` der Kontrollfluss geändert und die Callin-Methode `s.notify()` auf dem verbundenen Rollenobjekt aufgerufen. Die `Subject`

[4] Um instanziierbar zu sein, muss ein Team alle Bindungen, insbesondere alle Callout Bindungen, enthalten, also ein vollständiger Konnektor sein.

[5] Ausnahmen sind sogenannte *externalized roles*, die hier aber nicht weiter betrachtet werden.

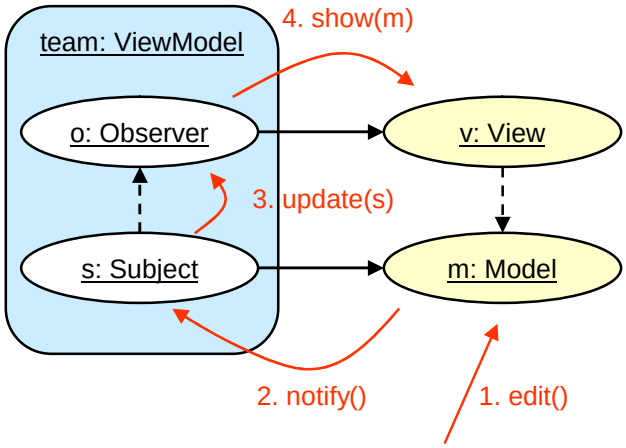

Abbildung 3.3: Das Observer-Team zur Laufzeit

Rolle informiert nun alle ihre `Observer` durch `update()`, wobei sie sich selbst als Parameter mit übergibt. Da es sich hierbei um eine per Callout gebundene Methode handelt, wird in diesem Fall der Aufruf an `v.show()` weitergeleitet.

Translationspolymorphismus

Da Rollen nur innerhalb ihrer umschließenden Teams auftreten können und die Basis sie nicht kennt, und weil Rollen in der Regel auch keine Kenntnis von ihren Basen haben, da sie unabhängig von ihnen implementiert wurden, muss an der Schnittstelle zwischen Team und Basis eine Überführung von Rolle nach Basis und umgekehrt stattfinden. Die Tatsache, dass die Substituierbarkeit von Rollen- und Basisobjekten nicht auf einer direkten Subtyp-Beziehung beruht, sondern auf einer impliziten Überführung, führt zu der Bezeichnung Translationspolymorphismus. Es gibt zwei Arten der Überführung:

- *lowering* ist der Übergang von einem Rollenobjekt zu einem Basisobjekt und geschieht beispielsweise bei einem Callout, wie dem in Listing 3.2. Dort wird der Callout-Methode `Observer.update()` ein Parameter vom (Rollen-)Typ `Subject` übergeben, die gebundene Methode `View.show()` erwartet aber den (Basis-)Typ `Model`. Diese Umwandlung wird automatisch von der Laufzeitumgebung übernommen und kann nicht explizit ausgeführt werden.

- *lifting* wird benötigt, wenn ein Basisobjekt in ein Team "hineingereicht" wird, zum Beispiel durch ein replace-Callin. Es wird dann implizit mit seinem Rollenobjekt "dekoriert" und ist innerhalb des Teams nur noch als Rolle sichtbar. Ein Basisobjekt wird innerhalb eines Teams immer durch

dasselbe eindeutige Rollenobjekt dargestellt, dazu verwaltet jede Team-Instanz einen eigenen Rollencache.

3.2.5 Aspektorientierte Modellierung mit UFA

Für die aspektorientierte Modellierung von *Object Teams* Anwendungen wurde eine Erweiterung der UML namens UFA (*UML for Aspects*) entworfen, die in [Her02a] beschrieben wird. Im Rahmen einer Diplomarbeit wurde dazu ein graphischer Editor mit Anbindung an die Softwareentwicklungsumgebung PIROL entworfen [Hac02]. Abbildung 3.4 zeigt eine Übersicht dieser Notation anhand des Observer Beispiels.

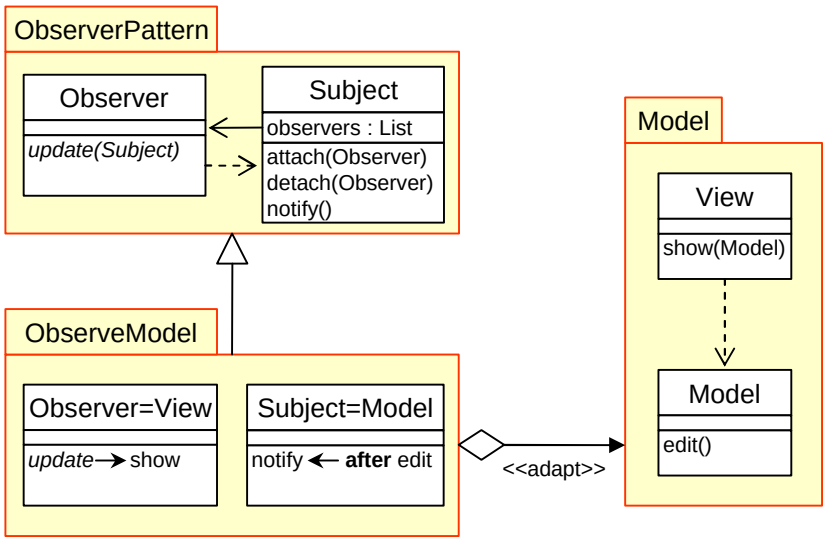

Abbildung 3.4: Adaption einer Basisanwendung in UFA

Teams werden in UFA durch Pakete dargestellt, die zusätzlich zur Standard UML-Notation neben den (Rollen-)Klassen auch Attribute und Methoden enthalten können (hier nicht gezeigt). Des Weiteren wird Generalisierung zwischen Paketen eingeführt, um Team-Vererbung zu modellieren. Die Adaption einer Basisanwendung durch einen Konnektor wird als Aggregation mit dem stereotype `adapt` notiert. Die Rollenbindungen innerhalb eines Konnektors werden durch Klassen dargestellt, die nach der Form `Rolle=Basis` benannt werden. Die Notation von Callin und Callout Bindungen entspricht der auf Sourcecode-Ebene.

3.2.6 Entwicklung mit *Object Teams/Java*

Neben einer Implementierung in Ruby (*Ruby Object Teams*, [Vei02]) und einer in Arbeit befindlichen Variante für C++, ist *Object Teams/Java* die am weitesten fortgeschrittene Implementierung des Object Teams Modells. Sie basiert auf einem Compiler und einer Laufzeitumgebung. Der modifizierte `javac` Compiler übersetzt Code in der erweiterten Java-Syntax in regulären Java Bytecode, der mit Attributen für die Laufzeitumgebung angereichert ist. Die Laufzeitumgebung selbst nutzt das JMangler Framework, um anhand der Informationen aus den Attributen eine Adaptierung der Basisklassen zur Ladezeit vorzunehmen. Somit unterstützt Object Teams/Java sowohl die getrennte Implementierung von Basis-Anwendung und Team, als auch die Adaptierung von Basis-Anwendungen, die nur im Bytecode vorliegen. Der Compiler und die Laufzeitumgebung von *Object Teams/Java* werden in den Diplomarbeiten [Bin02] respektive [Hun03] näher beschrieben.

The problem is that components are not normally the most variable elements of a software architecture - the interactions between components are.

– [AKB99]

Kapitel 4

Die Kombination von Object Teams und CCM

Nachdem in den beiden vorangegangenen Kapiteln zwei Ansätze aus dem komponenten- bzw. aspektorientierten Umfeld vorgestellt wurden, soll nun die im Rahmen dieser Arbeit vorgenommene Integration der beiden beschrieben werden. Die Motivation für eine Kombination beider Ansätze liegt zum Einen darin, herauszufinden inwieweit sich das Object Teams Modell auch im Bereich komponentenorientierter Softwareentwicklung anwenden lässt. Zum Anderen soll eine Möglichkeit gezeigt werden, wie das CORBA Component Model um einen Adaptierungsmechanismus erweitert werden kann, der Komponenten und die Interaktionen zwischen ihnen modularisierbarer und dynamisch anpassbar macht.

Kapitel 4.1 soll zunächst den aktuellen Stand sowie weitere Forschungsvorhaben, die in ähnliche Richtungen gehen, aufzeigen. Das konkrete Modell der Anwendung von Object Teams auf das CORBA Component Model wird in Kapitel 4.2 vorgestellt, die dafür nötigen Werkzeuge und der erweiterte Entwicklungsprozess werden in Kapitel 4.3 beschrieben.

4.1 Der Status Quo

Der Forderung nach Austauschbarkeit von Komponentenimplementierungen wird das CCM im Grunde gerecht, da die Auswahl der verwendeten Komponentenimplementierung für den Nutzer transparent bleibt. Somit können neue Implementierungen des gleichen Komponententyps deployed werden, ohne dass die verbundenen Komponenten es merken. Allerdings fehlt hier bislang technisch die Möglichkeit, ein solches *Re-Deployment* dynamisch, also zur Laufzeit durchzuführen, so dass die Auswahl der Implementierung in der Regel beim Deployment vorgenommen werden muss. Diese Form der Austauschbarkeit setzt somit voraus, dass der Komponententyp re-implementiert wird, also zwangsläufig im

Sourcecode vorliegen muss. Änderungen müssten invasiv erfolgen. Eine nachträgliche Adaptierung des Verhaltens von "verpackten" oder bereits deployten Komponenten ist nicht vorgesehen.

Die Adaptierung von Komponenteninteraktionen wird vom CCM nur teilweise unterstützt. Zwar lassen sich die Verbindungen von konkreten Komponenten-Instanzen über ihre Ports auch zur Laufzeit (re-)konfigurieren, doch müssen die verbundenen Referenzen den jeweiligen, vorher statisch in IDL definierten, Schnittstellentypen entsprechen. Das bedeutet, dass nur vorhergesehene Interaktionen mit festgelegten Schnittstellen möglich sind.

4.1.1 Interaktionsmuster von Komponenten

Wenn man sich die bestehenden Konzepte zur Behandlung von Komponenteninteraktionen im CCM anschaut, so stellt man fest, dass dort bereits eine breite Unterstützung für gängige Interaktionsmuster vorhanden ist. Als Beispiel seien hier die in [Esk99] beschriebenen *Component Interaction Patterns* in ihrer Kurzfassung erwähnt und deren Entsprechungen und Realisierungen im CCM aufgezeigt.

Abstract Interactions

> "Reduce a component's dependence on its environment by defining interaction protocols between components separately from the components themselves. Specify these interactions in terms of abstract interfaces and implement components to communicate with each other through them."

Komponenten kommunizieren im CCM ausschließlich über ihre extern in IDL definierten Schnittstellen. Die Trennung wird noch verstärkt durch den Einsatz von Stubs und Skeletons, die zur Laufzeit als Proxies fungieren.

Component Bus

> "Bind components to an information bus that manages the routing of information between communicating components to remove explicit dependencies from the components themselves. Define interaction protocols that not only specify interfaces required for components to participate, but also the nature of interactions occuring between them."

Dieses Pattern wird in sofern erfüllt, als das Komponenten, durch den Container gekapselt, an eine Middleware (den ORB) gebunden sind, die die tatsächliche Kommunikation zwischen den, möglicherweise über ein Netzwerk verteilten,

Komponenten übernimmt. Weiterhin werden durch den Einsatz von Homes sowie Naming und Trading Services die direkten Abhängigkeiten zwischen Komponenten reduziert.

Component Glue

> "Create glue code to act as an adapter for incompatible components, or as a mediator between peers. Only build full-flegded components when glue doesn't meet all of your requirements."

Der ORB mit den Stubs und Skeletons stellt bereits einen Mediator zwischen Komponenten dar, der unter anderem das Umwandeln von Datentypen übernimmt, falls zwei kommunizierende Objekte in unterschiedlichen Sprachen implementiert sind. Der zweite Punkt entspricht der Tatsache, dass nicht alle Teile einer CCM-Anwendung als Komponenten implementiert werden müssen. Es können nach wie vor auch "normale" CORBA-Objekte, valuetypes oder structs verwendet werden.

Third-Party Binding

> "Remove connections established in the implementation of a component by having a third component bind two interacting components together."

Das Verbinden von CCM-Komponenten wird primär über ihre Ports vorgenommen und ist somit von ihrer Implementierung weitgehend gelöst. Die entsprechenden Operationen auf der äquivalenten Schnittstelle einer Komponente erlauben es externen CORBA-Clients, CCM-Komponenten oder einem Deployment-Tool, Verbindungen herzustellen oder aufzulösen.

Consumer-Producer

> "Provide components a unified interface to multiple, seamless connectivity to heterogeneous service providers."

Ein Container bietet den Komponenten einheitlichen Zugriff auf Dienste wie Naming, Security oder Transaktionen, deren Implementierungen transparent austauschbar sind (zum Beispiel durch Wechsel der Plattform). Die Konfiguration dieser Dienste erfolgt einheitlich über externe Deskriptoren. Mit dem Trading Service ist es außerdem möglich, Komponenten aufzufinden, die bestimmten Kriterien entsprechen.

4.1.2 Verwandte Arbeiten

An verschiedenen Stellen wurde bereits auf die Gemeinsamkeiten und Unterschiede von aspektorientierter Programmierung und komponentenorientierter Entwicklung auf Basis von Container-Architekturen hingewiesen. [HMO01] beschäftigt sich mit der Frage, inwieweit eine stärkere Integration beider Ansätze die Umsetzung von *Separation of Concerns* fördern könnte und gibt Hinweise auf Anknüpfungspunkte zwischen beiden Welten. In [POM03] untersuchen die Autoren die Modularisierung von nicht-funktionalen Anforderungen durch Container im EJB-Modell und inwieweit diese mit AspectJ umgesetzt werden könnten. Vor- und Nachteile beider Varianten werden am Beispiel des Security-Dienstes aufgezeigt.

Weitere Forschungsaktivitäten zu adaptierbarer Middleware zielen hauptsächlich auf die Unterstützung von *Quality of Service* (QoS) Eigenschaften, die dynamische Anbindung von Diensten sowie allgemein die Rekonfigurierbarkeit von objekt- bzw- komponentenbasierter Middleware. Beispiele hierfür sind:

- CIAO (*Component-Integrated ACE ORB*) [WSa03], eine CCM-Implementierung basierend auf dem echtzeitfähigen ORB TAO
- Qedo (*QoS enabled distributed objects*) [Que], eine weitere CCM-Implementierung mit Schwerpunkt auf QoS
- Lasagne [TJJ00, TVJ+01b, TVJ+01a], ein Modell zur dynamischen Anpassung von Komponentensoftware, das aspektorientierte Techniken verwendet (vgl. Kapitel 3.1.3 auf Seite 38)

Die Adaptierung und Modularisierung funktionaler Aspekte und Kollaborationen einer Anwendung wird hingegen selten betrachtet.

4.1.3 Anwendungsmöglichkeiten

Object Teams bietet Separation of Concerns durch Modularisierung von Kollaborationen und deren nachträgliche Integration in bestehende Anwendungen. Auf der Suche nach möglichen Einsatzzwecken von Object Teams im Rahmen des CCM lassen sich folgende Bereiche identifizieren:

- Entwicklung *von* Komponenten: Hiermit ist die interne Implementierung von Komponenten, also ihrer *executors* gemeint. An dieser Stelle könnte beispielsweise Object Teams/Java zum Einsatz kommen. Allerdings würde sich der Nutzen dabei auf die Interna der Komponente beschränken und somit keine Adaption von Komponenteninteraktionen erlauben. Schwierigkeiten könnten hier die diversen generierten Klassen und Abhängigkeiten des CIF darstellen. Des weiteren wäre zu untersuchen, inwieweit der Einsatz von JMangler in einem verteilten Szenario möglich ist.

- Entwicklung *mit* Komponenten: Bei der Erstellung von Anwendungen aus Komponenten werden diese als "black-boxes" gesehen, die nur über ihre per IDL defineirten Schnittstellen miteinander kommunizieren. Auf dieser Architekturebene ließen sich Kollaborationen von Komponenten mit Hilfe eines angepassten Object Teams Modells modularisieren und adaptieren. Dazu müssten sowohl Schnittstellendefinitionen als auch Implementierung und Deskriptoren der Komponenten angepasst werden.

- Container-Entwicklung: Bei der Entwicklung von Containern bzw. ganzer CCM-Plattformen könnte Object Teams oder allgemein AOP zum Einsatz kommen, um die Anbindung von Diensten flexibler zu gestalten, QoS-Eigenschaften zu ermöglichen und das CIF eleganter und transparenter umzusetzen.

- Unterstützung für *Unanticipated Software Evolution* (USE): Wie bereits erwähnt, unterstützt das CCM zwar in gewissem Maße eine "plug and play"-Austauschbarkeit von Komponenten, jedoch müssen diese dazu in der Regel re-implementiert werden.

- Migration / Integration von (inkompatiblen) Komponenten: Unter dem Stichwort *Enterprise Application Integration* (EAI) könnte Object Teams dabei helfen, bestehende Software zu Komponentensoftware zu migrieren und möglicherweise inkompatible Komponenten zu integrieren.

Diese Arbeit richtet ihren Fokus auf die Entwicklung *mit* Komponenten. Was bei Object Teams mit Objekten und Objekt-Kollaborationen möglich ist, soll auch für CCM Komponenten und deren Kollaborationen ermöglicht werden. Damit wird auch die Unterstützung für USE und EAI gefördert.

4.2 *Object Teams/CCM*

Die Umsetzung von Object Teams für das CCM, im Folgenden *Object Teams/CCM* genannt, besteht aus einem konzeptionellen Modell, welches die Eigenschaften von Object Teams mit Hilfe von CCM-Konzepten nachbildet und in diesem Kapitel beschrieben wird. Die Hilfsmittel zum Entwickeln einer Anwendung mit Object Teams/CCM werden in Kapitel 4.3 vorgestellt.

4.2.1 Alternativen

Grundsätzlich gab es zwei Alternativen, Object Teams/CCM zu realisieren:

1. Teams und Rollen werden vom Container bzw. der Plattform zur Laufzeit implizit erzeugt und verwaltet. Das bedeutet, Teams und Rollen tauchen

nicht in der Komponentenarchitektur auf und sind für Clients nicht sichtbar. Die Veränderung des Kontrollflusses müsste hier durch das Abfangen von Methodenaufrufen (*method interception*) im Container erfolgen. Dazu könnte das *Interceptor Framework* von CORBA zum Einsatz kommen oder es müsste auf Container-Ebene mit AOP gearbeitet werden. Für diese Variante müsste ein eigener Container entwickelt oder ein bestehender angepasst werden. Dadurch ist der Aufwand recht hoch und man ist an eine Containerimplementierung gebunden, was Portabilität und Einsatzmöglichkeiten stark einschränkt. Vorteile hingegen sind die Transparenz gegenüber dem Entwickler sowie die Möglichkeit einer optimierten Laufzeitumgebung.

2. Teams und Rollen erscheinen expilizit in der Architektur und sind reguläre CORBA Objekte oder Komponenten. Diese können zum Teil generiert sein und sind unabhängig vom verwendeten Container. Die Veränderung des Kontrollflusses könnte dann durch *method interception* oder erneute Zuweisung der Client-Referenzen auf die entsprechenden Rollenobjekte realisiert werden. Die Vorteile dieser Alternative sind die Unabhängigkeit von der eingesetzten Plattform und eine einfachere Implementierung. Allerdings wird hier die sichtbare Komplexität der Anwendungen erhöht, was sich negativ auf Performanz und Skalierbarkeit auswirken dürfte.

Für diese Arbeit wurde der zweite Ansatz gewählt. Die Begründung dafür liegt zum Einen darin, dass zum Zeitpunkt des Schreibens die vorhandenen Open Source Implementierungen für das CCM noch nicht weit genug entwickelt waren, um gravierende Änderungen am Container vorzunehmen. Außerdem erschien eine prototypische Umsetzung nach dem zweiten Ansatz realistischer und anschaulicher für die Evaluierung der Konzepte.

4.2.2 Der Ansatz

Da der gewählte Ansatz eine Adaptierung auf Architekturebene vorsieht, stellt sich als erstes die Frage nach den Entsprechungen von Elementen des Object Teams Modells und denen des CCM. Soll ein Team durch eine Komponente repräsentiert werden und deren Facets entsprächen den Rollen? Damit würden Basis- und Rollenobjekte des Object Teams Modells im CCM CORBA-Objekte sein und Komponenten an sich gar nicht Gegenstand der Adaptierung. Da es aber gerade um die Modularisierung und Adaptierung von Komponenteninteraktionen geht, wurde die Granularität von Basen und Rollen so gewählt, dass diese in Object Teams/CCM Komponenten entsprechen. Im Gegensatz zum Object Teams Modell kann daher eine Basis neben Methoden (auf der Komponentenschnittstelle) auch Facets, Receptacles und Event-Ports enthalten. Was die Frage aufwirft, welche der Oberflächenfeatures einer Komponente denn adaptiert

werden können. In der hier vorgestellten Version ermöglicht Object Teams/CCM lediglich die Adaptierung von Komponentenschnittstelle und Facets einer Basis-Komponente. Teams werden zunächst auch als Komponenten modelliert, da das CCM keine Möglichkeit zur Komposition von Komponenten in einem umschließenden Komponententyp vorsieht, wie beispielsweise das *Composite* Pattern [GHJV95].

Die Anforderungen lassen sich folgendermaßen zusammenfassen: Object Teams/CCM soll es ermöglichen, Teams zu definieren, deren Rollen-Komponenten die Interaktionen und das Verhalten von Basis-Komponenten verändern können. Dabei ist davon auszugehen, dass von den Basis-Komponenten nur IDL und Komponentenarchive vorliegen und diese auch nicht zur Ladezeit adaptiert werden können.[1] Der entscheidende Unterschied zu Object Teams/Java ist daher, dass im hier vorgestellten Modell nicht die Basisimplementierung an sich modifiziert wird, sondern Wrapper vorgeschaltet werden, wie im folgenden Abschnitt beschrieben. Dies erschwert die Sicherstellung, dass Aufrufe nur an die Rollen-Komponente und nicht direkt an die Basis gerichtet werden. Nur wenn das sichergestellt ist, können Probleme aufgrund von *Objekt Schizophrenie* [2] vermieden werden. Die Behandlung von Objekt Schizophrenie im Object Teams Modell wird in [Her03] näher beleuchtet.

Teams und Rollen sollen in Object Teams/CCM als reguläre CCM Komponenten entwickelt werden können, ein externer Konnektor enthält die Informationen zur Adaptierung. Auf die Einführung einer neuen Sprache und die Erweiterung der IDL oder der Implementierungssprache wird bewusst verzichtet, um Portabilität zu gewährleisten und bestehende Werkzeuge wie IDL Compiler und Code-Generatoren benutzen zu können.

4.2.3 Wrapper-Komponenten

Für die dynamische und nicht-invasive Anpassung von Komponenten bietet sich die Verwendung von *Wrappern* an. Eine Wrapper-Komponente funktioniert nach dem *Decorator* Pattern [GHJV95] und "dekoriert" eine Basis-Komponente, indem sie dieselben Schnittstellen anbietet und anstelle der Basis-Komponente benutzt wird. Für den Client ist diese Dekorierung transparent. Ein Wrapper hat immer Zugriff auf seine Basis, um dessen ursprüngliche Operationen aufrufen zu können (vgl. *base-call* und Callout bei Object Teams, Kapitel 3.2.3). Die Implementierung der Basis-Komponente wird auf diese Weise weder zur La-

[1] Die Adaptierung der implementierenden Klassen zur Ladezeit wie in Object Teams/Java ist hier zunächst nicht möglich, da diese Klassen in der internen Struktur des Executors verborgen und somit nicht bekannt sind. Eventuell könnte aber eine Adaptierung der Stubs und Skeletons zur Ladezeit erfolgen. Diese Möglichkeit bleibt noch zu überprüfen.

[2] Man spricht von Objekt Schizophrenie, wenn ein Objekt zur Laufzeit von mehreren atomaren Objekten repräsentiert wird, die jeweils eine eigene Identität besitzen. Dies ist ein generelles Problem bei Rollen- und Wrapperbasierten Ansätzen.

dezeit noch zur Laufzeit direkt verändert, es wird lediglich ein Wrapper "dazwischengeschaltet". Wrapper-Komponenten übernehmen in diesem Modell also die Funktion der Rollen und implementieren gleichzeitig die Adaptierung der Basis-Komponenten. Entsprechend den Basis- und Rollenobjekten in Object Teams besteht auch hier eine 1:1-Beziehung zwischen Instanzen der Basis- und Rollen-Komponenten, die von der zuständigen Team-Komponente verwaltet wird.

Da die Benutzung der Rolle anstelle der Basis für Clients transparent sein soll und die nachträgliche Definition eines gemeinsamen Obertyps nicht in Frage kommt, werden die Wrapper-Komponenten in IDL als Subtypen der Basis-Komponenten definiert. Damit ist gewährleistet, dass sie dieselben Attribute und Ports anbieten wie ihre Basis-Komponenten und diese dank Polymorphie durch ihre Rollen substituierbar sind. Das grobe Schema dieser Rollen in Form von Wrapper-Komponenten wird im Folgenden an einem simplen Beispiel erläutert:

Listing 4.1 zeigt die vollständige IDL3 eines Basis-Moduls [3], in dem die Schnittstelle `IBase` und der Komponententyp `BaseComp` definiert sind. `BaseComp` bietet ein Facet vom Typ `IBase` mit dem Identifier `baseFacet` an.

```
1  module base {
2    interface IBase {
3      void baseMethod();
4      void baseMethodCallin(); // für später
5    };
6    component BaseComp {
7      provides IBase baseFacet;
8    };
9    home BaseCompHome manages BaseComp {};
10 };
```

Listing 4.1: Basis-Komponente in IDL3

Eine Instanz dieses Komponententyps kann beim Deployment beispielsweise bei einem Naming Service registriert werden und ist somit für CORBA-Clients erreichbar. Listing 4.2 zeigt einen Ausschnitt aus einem CORBA-Client, der die oben definierte Komponente benutzt. Dazu besorgt er sich zunächst vom Naming Service eine Referenz auf die dort registrierte Komponente vom Typ `BaseComp`. Auf dessen äquivalenter Schnittstelle befindet sich die von der Plattform implizit generierte Methode `provide_baseFacet()`, die ihm die entsprechende Facet-Referenz vom Typ `IBase` liefert, auf der er die Methode `baseMethod()` aufruft.

```
1  ...
```

[3] Das `module`-Konstrukt in IDL entspricht dem `package`-Konstrukt in Java und definiert lediglich einen Namensraum für die enthaltenen Elemente. Hierarchien von Namensräumen lassen sich duch Schachtelung von Modulen erreichen.

```
2 BaseComp bc = ... // vom Naming Service
3 IBase facetRef = bc.provide_baseFacet();
4 facetRef.baseMethod();
5 ...
```

Listing 4.2: Ausschnitt CORBA-Client

Das Verhalten der Methode `baseMethod` soll nun durch eine separat entwickelte Wrapper-Komponente adaptiert werden. Dazu wird in Listing 4.3 ein Modul `team` definiert[4], das den (Rollen-)Komponententyp `RoleComp` enthält. Dieser ist von `BaseComp` abgeleitet und bietet somit implizit auch ein Facet vom Typ `IBase` an, dessen Operationen im Executor von `RoleComp` implementiert werden müssen. Zusätzlich definiert `RoleComp` einen Receptacle-Port `baseReceptacle`, der beim Deployment mit dem entsprechenden Facet der Basis-Komponente verbunden wird. Über diese Verbindung kann die Rollen-Komponente auf die Implementierung der Basis-Komponente zugreifen und basecalls ausführen.

```
1 import base; // Basis-Modul importieren
2 module team {
3   // Rollen-Komponente als Subtyp der Basis-Komponente:
4   component RoleComp : base::BaseComp {
5     // Verbindung zum Original-Facet:
6     uses base::IBase baseReceptacle;
7   };
8   home RoleCompHome manages RoleComp {};
9 };
```

Listing 4.3: Einfache Wrapper-Komponente in IDL3

Der implementierende Executor von `RoleComp` kann, wie in Listing 4.4 gezeigt, die Methode `baseMethod()` neu implementieren und dabei das Original über das Receptacle `baseReceptacle` aufrufen. Die Referenz auf das verbundene Facet vom Typ `IBase` erhält er durch den Aufruf von `get_connection_baseReceptacle(` einer vom CIF generierten Methode auf seinem Kontext-Objekt, das vom Container per Callback-Methode beim Deployment gesetzt wird.

```
1 ...
2 CCM_RoleComp_Context ctx;
3 ... // Kontext wird vom Container gesetzt
4 base.IBase baseRef = ctx.get_connection_baseReceptacle();
5 public void baseMethod() {
6   ... // verändertes Verhalten
```

[4] Der Name `team` ist hier willkürlich gewählt und hat keine Funktion als Schlüsselwort.

```
7     baseRef.baseMethod(); // base-call
8     ... // verändertes Verhalten
9   }
10  ...
```

Listing 4.4: Ausschnitt aus dem Executor der Wrapper-Komponente

Beim Deployment der adaptierten Anwendung wird anstelle der Basis-Komponente eine Instanz von `RoleComp` unter demselben Namen beim Naming Service registriert. Dadurch, dass `RoleComp` ein Subtyp von `BaseComp` und somit polymorph zu ihm ist, greift der Client bei erneuter Ausführung unbewusst auf die Wrapper-Komponente und deren Implementierung von IBase.baseMethod() zu. Prinzipiell wäre es auch möglich, die Wrapper-Komponente auf eine bereits deployte Basis-Komponente anzuwenden, indem deren Eintrag im Naming Service durch den Wrapper ersetzt würde. Sobald ein Client erneut die Referenz vom Naming Service erfragt, arbeitet er automatisch mit dem Wrapper. Weitere Aspekte des Deployments werden in Kapitel 4.2.4 auf Seite 63 beschrieben.

4.2.4 Ebenen der Adaptierung

Wie bereits das einfache Beispiel aus dem vorigen Abschnitt zeigt, erfordert die Adaptierung durch Rollen-Komponenten Änderungen auf den drei Ebenen Schnittstelle, Implementierung und Deployment. Die Einzelheiten dazu werden im Folgenden in den jeweiligen Unterabschnitten erklärt.

Schnittstellen

Ein gebundener Rollen-Komponententyp bietet neben den explizit für ihn in IDL definierten Attributen und Ports durch die Subtyp-Beziehung implizit alle Attribute und Ports seines Basis-Komponententyps an. Für jedes Facet der Basis-Komponente muss außerdem ein Receptacle definiert werden, das dazu dient Aufrufe an die Original-Methoden weiterleiten zu können. Ein weiteres Receptacle dient der Verknüpfung mit der zuständigen Team-Komponente. Die Schnittstelle `ITeam`, die von jedem Team angeboten werden muss, enthält Methoden zum Registrieren von Rollen- mit ihren Basisinstanzen (`addRoleComp()`), zum *Liften* von Basisinstanzen (`lift2RoleComp()`) und zur Abfrage, ob das Team gerade aktiv ist (`isActive()`). Der Aufbau einer Team-Komponente wird in Kapitel 4.2.5 auf Seite 64 beschrieben. Auf der Komponentenschnittstelle der Rolle wird zusätzlich das Attribut `active` definiert, über das jede Rolleninstanz einzeln aktiviert bzw. deaktiviert werden kann. Abbildung 4.1 zeigt die relevanten Schnittstellen einer gebundenen Rollen-Komponente in einem UML-Diagramm. IDL-Konstrukte wie `uses` und `provides` sind hier als Assoziationen dargestellt und mit den entsprechenden Stereotypes gekennzeichnet.

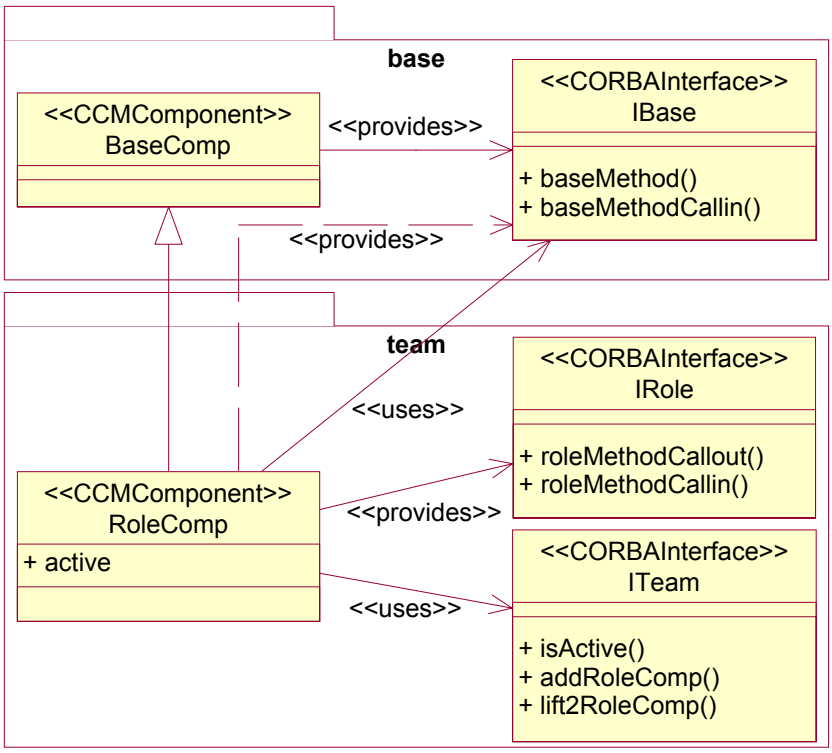

Abbildung 4.1: Schnittstellen einer gebundenen Rollen-Komponente

Durch die Benutzung von Subtyping für das Binden von Rollen-Komponenten ergeben sich möglicherweise folgende Nachteile bzw. Einschränkungen:

- Die Vererbung in IDL ist sehr statischer Natur und wirkt sich auf viele generierte Elemente wie Stubs und Skeletons aus. Ein einigermaßen dynamisches Zuweisen von Rollen zu Basen ist somit ausgeschlossen, die Zuweisung erfolgt an allererster Stelle des Entwicklungsprozesses.

- Da für jeden Rollentyp ein Subtyp der zugehörigen Basis erzeugt wird, entstehen zusätzliche Vererbungshierarchien, die nicht anwendungsspezifisch sind und in größeren Anwendungen die Komplexität unerwünscht erhöhen könnten.

- Zusätzlich ergeben sich Einschränkungen für die Entwicklung der Rollen-Komponenten, da in IDL nur Einfachvererbung möglich ist und diese bereits durch die Bindung an die Basis belegt wird. Das Erben weiterer Schnittstellen über das `supports`-Schlüsselwort ist laut Spezifikation für abgeleitete Komponententypen auch nicht erlaubt [5].

[5] Hierüber scheint noch Uneinigkeit zu bestehen, da diese Möglichkeit zumindest in der verwendeten Plattform gegeben ist und kein Grund für eine solche Einschränkung erkennbar ist.

Implementierung

Nachdem der vorige Abschnitt die externen Schnittstellen einer gebundenen Rollen-Komponente beschrieben hat, soll nun deren Implementierung näher betrachtet werden. Diese besteht aus generiertem und vom Entwickler ergänzten Code, die Unterscheidung wird jeweils deutlich gemacht. Das folgende fortlaufende Listing zeigt ausschnittweise die Implementierung eines Executors für den Komponententyp `RoleComp`. Dieser ist monolithisch implementiert (vgl. Kapitel 2.2.3 auf Seite 24), das heißt die Schnittstellen von Basis- und Rollen-Komponente sind gemeinsam in der Klasse `RoleCompImpl` implementiert. In Listing 4.5 sind zunächst die wichtigsten internen Attribute dargestellt, deren Deklarationen und spätere Wertzuweisungen generiert werden können. [6]

```
1  package team;
2  public class RoleCompImpl
3  ...
4  // entscheidet über Adaptierung der Basis:
5  private boolean _ot_active;
6  // Referenz auf Komponentenschnittstelle der gebundenen Basis:
7  private base.BaseComp _ot_baseComp;
8  // Referenzen auf alle Facets der Basis-Komponente:
9  private base.IBase _ot_baseFacet;
10 ...
11 // Referenz auf Team-Facet für Rollen:
12 private ITeam _ot_team;
13 // Komponentenkontext vom Container:
14 private CCM_RoleComp_Context ctx;
15 ...
```

Listing 4.5: Implementierung einer Rollen-Komponente: interne Attribute

Initialisierung der Rollen-Komponente

Die Methode `configuration_complete()` in Listing 4.6 ist eine Callback-Methode, die vom Container am Ende der Deployment-Phase aufgerufen wird und vom Executor der Komponente implementiert werden muss. Sie markiert den Übergang von der Konfigurations- zur Nutzungsphase einer Komponenteninstanz. An dieser Stelle werden die Referenzen auf die Basis- und Team-Schnittstellen gesetzt und die Rolle bei ihrem Team registriert. Der Code für diese Methode wird generiert, der Entwickler kann ihn bei Bedarf erweitern.

[6] In der Darstellung abstrakter Signaturen werden hier Platzhalter für konkrete Elemente in spitzen Klammern notiert, zum Beispiel `<rtype>` für den beliebigen Typ eines Rückgabewerts oder `<param-list>` für eine nicht näher spezifizierte Parameterliste. Der Prefix `_ot_` kennzeichnet Attribute und Methoden, die von OT/CCM generiert werden.

```
16 ...
17 /* Operation CCMObject::configuration_complete */
18 public void configuration_complete()
19     throws org.omg.Components.InvalidConfiguration {
20   ...
21   // get connection to base component:
22   _ot_baseFacet = ctx.get_connection_baseReceptacle();
23   _ot_baseComp =
24       base.BaseCompHelper.narrow(_ot_baseFacet.get_component());
25   // get connection to team component:
26   _ot_team = ctx.get_connection_teamReceptacle();
27   // register myself at the team:
28   RoleComp myself = RoleCompHelper.narrow(this.get_component());
29   _ot_team.addRoleComp(myself, _ot_baseComp);
30   ...
31 }
32 ...
```

Listing 4.6: Implementierung einer Rollen-Komponente: Initialisierung

Basismethoden, für die kein Callin definiert ist

Alle Operationen auf den Facets oder der Komponentenschnittstelle der Basis, für die kein Callin definiert ist, werden einfach an die entsprechende Schnittstelle der Basis weitergeleitet. Diese Delegations-Methoden können vollständig generiert werden. Listing 4.7 zeigt eine solche Basismethode, die auf einem Facet vom Typ `base::IBase` in IDL definiert wurde und nicht adaptiert werden soll.

```
33 ...
34 /* Operation base::IBase::baseMethod */
35 public <rtype> baseMethod(<param-list>) {
36   return _ot_baseFacet.baseMethod(<param-list>);
37 }
38 ...
```

Listing 4.7: Implementierung einer Rollen-Komponente: nicht-adaptierte Basismethoden

Basismethoden, für die ein Callin definiert ist

Soll eine Operation auf den Schnittstellen der Basis-Komponente durch ein Callin adaptiert werden, so muss in ihr die gebundene Rollenmethode aufgerufen werden, wie in Listing 4.8 gezeigt. Vorher muss gegebenenfalls noch die

Parameterliste angepasst werden. Bei einem `after`- oder `before`-Callin wird zusätzlich vor bzw. nach der Rollenmethode die Originalmethode auf der Basis aufgerufen. Solche Callin-Adaptierungen können ebenfalls generiert werden.

```
39 ...
40 /* Operation base::IBase::baseMethodCallin */
41 public <rtype> baseMethodCallin(<param-list>) {
42   // bei after: _ot_baseFacet.baseMethodCallin(<param-list>);
43   // ggf. lifting und Anpassung von Parametern
44   <rtype2> <rvalue2> = this.roleMethodCallin(<param-list2>);
45   // bei before: _ot_baseFacet.baseMethodCallin(<param-list>);
46   // ggf. lowering und Anpassung des Rückgabewerts
47   return <rvalue>;
48 }
49 ...
```

Listing 4.8: Implementierung einer Rollen-Komponente: Basismethode mit Callin

Rollenmethoden, für die ein Callout definiert ist

Ist eine Rollenmethode als Callout definiert, so muss sie die gebundene Basismethode aufrufen, wie in Listing 4.9 dargestellt. Dazu wird, falls nötig, zunächst die Parameterliste und nach dem Aufruf der Rückgabewert und -typ angepasst. Wie das `this` verdeutlichen soll, wird die Basismethode `baseMethod()` nicht direkt auf der Basis-Komponente, sondern wiederum auf der Rollen-Komponente aufgerufen, so dass eine weitere Adaptierung durch Callins möglich ist. Auch Callouts können vollständig generiert werden.

```
50 ...
51 /* Operation team::IRole::roleMethodCallout */
52 public <rtype> roleMethodCallout(<param-list>) {
53   // ggf. lowering und Anpassung von Parametern
54   <rtype2> <rvalue2> = this.baseMethod(<param-list2>);
55   // ggf. lifting und Anpassung des Rückgabewerts
56   return <rvalue>;
57 }
58 ...
```

Listing 4.9: Implementierung einer Rollen-Komponente: Callout

Rollenmethoden, die als replace-Callin fungieren

Rollenmethoden, die nicht per Callout gebunden sind, müssen generell vom Entwickler ausimplementiert werden. Eine Besonderheit sind hier Rollenmethoden, die als replace-Callin gebunden sind und somit in der Lage sein müssen,

einen base-call durchführen zu können. Da die Implementierung der Rollenmethode die gebundene Basismethode aber noch nicht kennen muss, kann sie diese nicht direkt aufrufen. Der tatsächliche base-call wird daher von einer generierten Methode durchgeführt, die der Namenskonvention `_ot_<name>()` folgt, wobei `<name>` der Name der Callin-Rollenmethode ist und die Parameterlisten der beiden identisch sind. Ein Beispiel dafür ist in Listing 4.10 zu sehen.

```
59 ...
60 /* Operation team::IRole::roleMethodCallin */
61 public <rtype> roleMethodCallin(<param-list>) {
62   ...
63   this._ot_roleMethodCallin(<param-list>); // base-call
64   ...
65 }
66 /* generierte base-call Methode */
67 private <rtype> _ot_roleMethodCallin(<param-list>) {
68   // ggf. lowering und Anpassung von Parametern
69   <rtype2> <rvalue2> = _ot_baseFacet.baseMethodCallin(<param-list>)
70   // ggf. lifting und Anpassung des Rückgabewerts
71   return <rvalue>;
72 }
73 ...
```

Listing 4.10: Implementierung einer Rollen-Komponente: replace-Callin und base-call

Deployment

Beim Deployment ergeben sich einige Schwierigkeiten, insbesondere wenn man davon ausgeht, dass die Basisanwendung bereit im Einsatz ist. Dann sind nämlich mit hoher Wahrscheinlichkeit bereits Referenzen im Umlauf, die von Clients und anderen Komponenten genutzt werden. Alle existierenden Referenzen nachträglich auf die Rollen-Komponenten "umzubiegen" erscheint unrealistisch, da sie unter anderem im Netzwerk verteilt, beim Naming oder Trading Service registriert oder in Form von IORs [7] gespeichert sein können. Für diesen Einsatzzweck, also die Adaptierung bereits laufender Systeme, bedarf es einer Lösung die mit *method interception* arbeitet, so dass jeder Aufruf an eine adaptierte Basis-Komponente umgelenkt werden kann. Eine solche Alternative wurde bereits in Kapitel 4.2.1 auf Seite 53 erwähnt. Eine weitere Möglichkeit besteht darin, dass Rollen-Komponenten theoretisch auch auf anderen Hosts deployed werden können als ihre Basis-Komponenten. Damit wäre eine Art "Fern-

[7] Eine *Interoperable Object Reference* (IOR) ist eine als Zeichenkette serialisierte CORBA Objekt-Referenz. In einer IOR sind unter anderem eine eindeutige ObjectID und der Hostname kodiert.

Adaptierung" möglich. Die Anwendbarkeit und Realisierung eines solchen Szenarios bleibt aber noch zu untersuchen.

Für diese Arbeit wurde also zunächst davon ausgegangen, dass Basisanwendung und Team gemeinsam deployed werden. Das entspricht im Prinzip der Adaptierung zur Ladezeit, die Object Teams/Java bietet. Außerdem beschränkt sich die Adaptierung auf per Assembly-Deskriptor erzeugte Basisinstanzen, da die Verknüpfung von Rollen- und Basis-Komponenten noch manuell beim Deployment erfolgen muss. Später sollte automatisch beim Erzeugen einer Basisinstanz die entsprechende Rolleninstanz erzeugt, verbunden und anstelle der Basis verwendet werden. Dazu müsste die Home-Implementierung der Basis beim Deployment ausgetauscht werden, da Komponenteninstanzen immer über ihre Homes erzeugt werden. Dieser Schritt ist noch nicht vollzogen, so dass sich der Einsatz momentan auf "statische" Kollaborationen, also eine fixe Menge von Basis- und Rolleninstanzen, beschränkt, die im Assembly Deskriptor festgelegt werden. Die Referenzen der Rollen-Komponenten wurden in den entwickelten Beispielen beim Naming Service registriert und ausschließlich auf diesem Weg von den Clients genutzt.

4.2.5 Die Team-Komponente

Ein Team wird in Object Teams/CCM ebenfalls durch eine Komponente realisiert. Eine solche Team-Komponente kann unabhängig von ihren Rollen eigene, vom Entwickler definierte Schnittstellen anbieten, um Funktionalitäten auf Team-Ebene bereitzustellen. Jede Team-Komponente bietet automatisch ein Facet `ITeam` an, das von den Rollen-Komponenten benutzt wird. Darin enthalten sind Operationen für das lifting und lowering von Komponenten, zum Registrieren von Rollen-Instanzen sowie zur Feststellung, ob das Team aktiv ist. Intern verwaltet eine Team-Komponente alle Verbindungen zwischen Rollen- und Basis-Komponenten. Dazu wird im Executor des Teams je gebundenem Rollentyp eine Hashtable angelegt, die die Rollen- und Basisinstanzen aufnimmt, sowie die Methoden `void add<roleType>(<roleType> role, <baseType> base)` und `<roleType> lift2<roleType>(<baseType> base)` generiert.

Ein Team lässt sich über das Attribut `active` aktivieren bzw. deaktivieren. Eine Änderung seines Zustandes wird an alle Rollen dieses Teams weitergegeben. Rollen-Komponenten können jederzeit einzeln deaktiviert werden, eine Re-Aktivierung ist nur möglich, wenn das zugehörige Team auch aktiv ist. Wird eine Team-eigene Methode aufgerufen, ohne dass das Team aktiv ist, so wird es automatisch aktiviert.

4.2.6 Manuelle Implementierung am Beispiel

Das oben beschriebene Adaptierungsmodell wurde zunächst anhand eines konkreten Beispiels manuell implementiert, um die Realisierbarkeit der Konzepte zu überprüfen. Als Beispiel diente hier das bereits in [Her02b] verwendete Flugbuchungssystem, das nachträglich um ein Bonus-System erweitert wurde. Dabei konnte auch die verwendete Plattform EJCCM [CP] ausgiebig getestet und deren Status bezüglich der Realisierung der Spezifikation ausgelotet werden. So wurden während der Entwicklung der Beispielanwendung diverse Fehler in der Plattform und den Compilern und Code-Generatoren aufgedeckt, sowie generelle Fragen aufgeworfen, wie Details der Vererbung zwischen Komponenten, die bis dato scheinbar nicht eindeutig geklärt waren. Erst nachdem die Beispielanwendung auf "normalem" Wege implementiert werden konnte und lauffähig war, wurde der Entwicklungsprozess wie im nächsten Kapitel beschrieben erweitert.

4.3 Erweiterter Entwicklungsprozess

Um Anwendungen mit Object Teams/CCM entwickeln zu können, ohne die Details der Adaptierungen von Hand codieren zu müssen, wurde der CCM-Entwicklungsprozess (siehe Kapitel 2.2.6 auf Seite 31) um die nötigen Artefakte und Transformationen erweitert, die eine weitgehend automatisierte Generierung des Object Teams/CCM-spezifischen Codes erlauben. Der Entwickler soll lediglich das Team und die Rollen in IDL definieren und die Funktionalität der Rollenmethoden implementieren müssen. Dazu wurden ein Konnektor im XML-Format, mehrere XSLT-Transformationen für die IDL- und Code-Adaptierung, sowie ein Build-Prozess mit Ant entwickelt. Da sich die verwendeten Compiler und Code-Generatoren der CCM-Plattform selbst noch in der Entwicklung befinden, wurden die Erweiterungen nur als Aufsatz bzw. Zwischenschritte implementiert, um eine parallele und weitgehend unabhängige Weiterentwicklung von EJCCM und Object Teams/CCM zu ermöglichen.

4.3.1 Code weaving auf Basis von XML und XSLT

Die zu adaptierenden Artefakte umfassen momentan IDL- und Java-Quellen, später können auch CIDL-Beschreibungen und XML-Deskriptoren hinzukommen. Für jeden dieser Artefakt-Typen existieren bereits Parser, die eine XML-Repräsentation dieser Quellen erzeugen können. Da die Werkzeuge der Plattform nicht modifiziert werden sollten und die Neu-Entwicklung eigener Parser und Code-Generatoren im Rahmen dieser Arbeit nicht angemessen gewesen wäre, bot sich der Umweg über ein Zwischenformat in XML an. In [SPS02] wurde bereits ein Ansatz zum Weben von Aspektcode mit Hilfe von XML-Repräsentationen von Abstrakten Syntaxbäumen vorgestellt. XML bietet an die-

ser Stelle den Vorteil einer generischen Syntax für die Definition eigener Sprachen und die Verfügbarkeit vieler Werkzeuge, die die Verarbeitung von XML unterstützen. Dadurch wird der infrastrukturelle Aufwand so gering wie möglich gehalten.

Für die Durchführung der Generierung und Adaptierung von IDL- und Java-Code fiel die Wahl auf XSLT (*Extensible Stylesheet Language Transformations*), das die Transformation von XML mit Hilfe sogenannter Stylesheets ermöglicht. XSLT-Stylesheets werden wiederum in einem XML-Format aufgeschrieben und enthalten Schablonen für die Elemente des Eingangsdokuments. Ein XSLT-Prozessor verarbeitet ein zu transformierendes XML-Dokument, indem er dessen Baumstruktur traversiert und jedes gefundene Element durch die Schablone im Stylesheet ersetzt. Die XSLT-Syntax bietet diverse Möglichkeiten, auf den Daten des Eingangsdokuments zu arbeiten, unter anderem den Zugriff auf beliebige Knoten des Baums über XPath, das Einbinden weiterer Dokumente, bedingte Ausführung von Transformationen sowie Operationen auf Zeichenketten.

4.3.2 Der Konnektor

Da die Transformationen auf Basis von XML stattfinden sollten, war es naheliegend, auch für den Konnektor ein XML-Format zu definieren. In Listing 4.11 ist ein solcher Konnektor für das Beispiel aus Abbildung 4.1 auf Seite 59 zu sehen. Ein Object Teams/CCM-Konnektor enthält alle Informationen die nötig sind, um eine Team-IDL, sowie die dazu von der Plattform generierten Executors um die Object Teams/CCM-spezifischen Anteile zu erweitern. Er entspricht im Prinzip einem vollständigen Konnektor in Object Teams/Java, bis auf die Tatsache, dass hier keine Implementierung im Konnektor erfolgt. Alle Bindungen werden innerhalb des Konnektors und außerhalb der Implementierung von Team und Rolle definiert. Da die gebundenen Basis- und Rollenmethoden auf mehrere Facets und die äquivalente Schnittstelle verteilt sein können, muss in einer Callin- bzw. Callout-Definition festgelegt werden, welcher Port gemeint ist (Attribute `port` und `porttype`).

```
1  <connector name="sample">
2   <playedBy>
3    <role name="RoleComp" full_scope="::team::RoleComp"/>
4    <base name="BaseComp" full_scope="::base::BaseComp"/>
5    <callin strategy="replace">
6     <rolemethod name="roleMethodCallin" port="roleFacet"
7                 porttype="IRole"/>
8     <basemethod name="baseMethodCallin" port="baseFacet"
9                 porttype="IBase"/>
10    <!-- <parametermapping/> -->
```

```
11    </callin>
12    <callout>
13     <rolemethod name="roleMethodCallout" port="roleFacet"
14                 porttype="IRole"/>
15     <basemethod name="baseMethod" port="baseFacet"
16                 porttype="IBase"/>
17     <!-- <parametermapping/> -->
18    </callout>
19   </playedBy>
20 </connector>
```

Listing 4.11: Beispiel eines Object Teams/CCM-Konnektors

Die im Listing angedeuteten Parametermappings sind noch nicht implementiert, da noch unklar ist, wie komplexere Berechnungen notiert werden sollen. Da im Konnektor keine Implementierung erfolgen soll, könnten Berechnungen extern implementiert und im Konnektor eingebunden werden.

4.3.3 Erzeugung der gebundenen Team-IDL

Soll eine Basis-Anwendung wie in Abbildung 4.1 auf Seite 59 gezeigt adaptiert werden, so definiert der Entwickler zunächst unabhängig von der Basis die Team-Anwendung in IDL, wie in Listing 4.12 dargestellt.

```
1  module team {
2    interface IRole {
3      void roleMethodCallout();
4      void roleMethodCallin();
5    };
6    component RoleComp {
7      provides IRole roleFacet;
8    };
9    home RoleCompHome manages RoleComp {};
10 };
```

Listing 4.12: IDL des ungebundenen Teams

Dem CCM-Entwicklungsprozess folgend, muss die vollständige IDL einer zu implementierenden Anwendung vorliegen, damit daraus Stubs und Skeletons, sowie gegebenenfalls Vorlagen für die Executors generiert werden können. In diesem Falle ist das die um Bindungsinformationen angereicherte Team-IDL aus obigem Listing. Die vollständige IDL für das gebundene Team wird auf Grundlage der vom Entwickler erstellten, ungebundenen IDL des Teams, der IDL der Basis-Anwendung und den Informationen des Konnektors erzeugt. Die im Konnektor definierten Bindungen von Rollen- an Basis-Komponenten sowie

die Team-Schnittstelle `ITeam` werden dabei in die Team-IDL eingewebt. Abbildung 4.2 zeigt den Teilprozess der Generierung der gebundenen Team-IDL mit den beteiligten Werkzeugen und Zwischenformaten, die im Folgenden näher beschrieben werden:

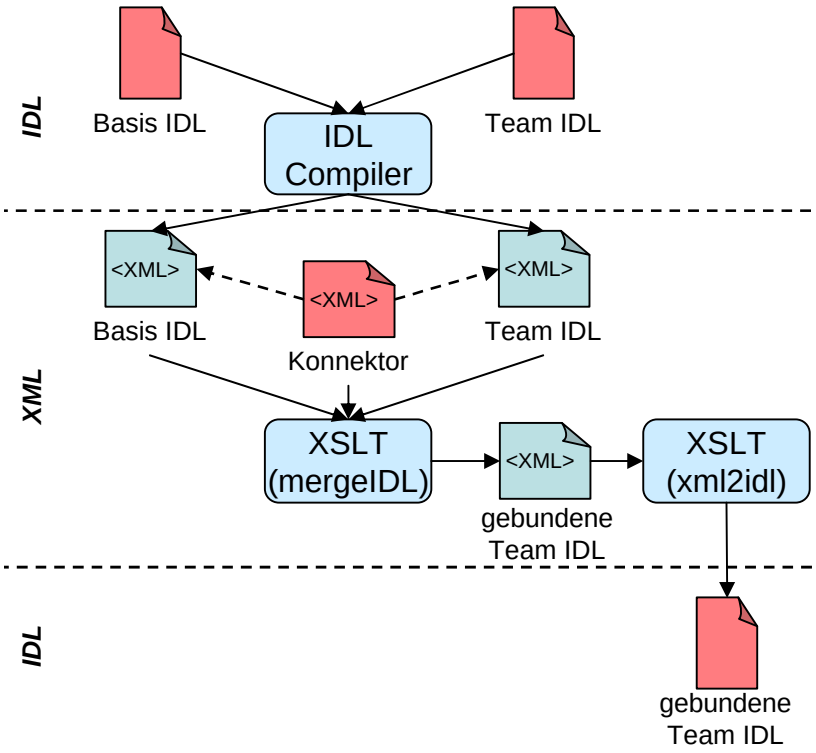

Abbildung 4.2: Erzeugung der IDL des gebundenen Teams

Der IDL3 Compiler von EJCCM

Aus der ungebundenen Team-IDL und der in Listing 4.1 auf Seite 56 gezeigten IDL der Basisanwendung erzeugt zunächst der IDL Compiler von EJCCM jeweils eine XML-Repräsentation. Die wesentlichen Elemente dieser Ausgabe für die Team-IDL aus Listing 4.12 sind in Listing 4.13 abgebildet.

```
1 ...
2 <module name="team">
3  <interface name="IRole" full_scope="::team::IRole"
4             abstract="false" local="false">
5   <operation name="roleMethodCallout" type="void"/>
6   <operation name="roleMethodCallin" type="void"/>
7  </interface>
```

```
8   <component name="RoleComp" full_scope="::team::RoleComp">
9    <provides name="roleFacet" type="::team::IRole"/>
10  </component>
11  <home name="RoleCompHome" full_scope="::team::RoleCompHome"
12        manages="::team::RoleComp"/>
13 </module>
14 ...
```

Listing 4.13: XML-Repräsentation der Team-IDL

IDLXML DTD und *mergeIDL* XSLT Stylesheet

Für das vom EJCCM IDL Compiler generierte XML wurde zunächst durch reverse-engineering eine DTD erstellt, die dazu dienen soll, die erzeugten XML-Dokumente auf Korrektheit zu überprüfen. Dies war angebracht, da zum Einen die XML-Ausgaben des Compilers teilweise fehlerhaft waren, zum Anderen um sicherzugehen, dass die transformierte IDL keine Fehler enthält.

Die Team-IDL in XML-Form wird vom *mergeIDL*-Stylesheet transformiert und um die Bindungsinformationen ergänzt. Dazu greift das Stylesheet per XPath auf die Elemente von Basis-IDL und Konnektor zu. Die Regeln für die Transformation der Team-IDL lauten wie folgt:

1. Importieren des Basis-Moduls
2. Definition der Schnittstelle `ITeam` mit den `add*()`- und `lift2*()`-Methoden für jeden Rollentyp
3. Hinzufügen des Facets `ITeam` und des Attributs `active` zum Team-Komponententyp.
4. Erweiterung jedes Rollen-Komponententyps um:
   - Ableitung vom gebundenen Basis-Komponententyp
   - Attribut `active`
   - Receptacles für jedes Facet der Basis-Komponente
   - Receptacle vom Typ ITeam

Das Ergebnis der Transformation ist die XML-Repräsentation der vollständigen, gebundenen Team-IDL und kann wiederum anhand der *IDLXML* DTD validiert werden.

xml2idl XSLT Stylesheet

Die so erzeugte XML-Repräsentation der gebundenen Team-IDL muss nun wieder in normale IDL-Form überführt werden, um im Rahmen des weiteren Entwicklungsprozesses vom IDL-Compiler regulär weiterverarbeitet werden zu können. Da es hierzu bisher keine Möglichkeit seitens der Plattform gab und eine direkte Verwendung der XML-Repräsentation nicht vorgesehen ist, wurde ein weiteres XSLT-Stylesheet entwickelt, dass die Rück-Konvertierung von XML nach IDL vornimmt. Hierbei kommt zum Tragen, dass das Resultat von XSLT-Transformationen nicht zwangsläufig XML sein muss, sondern wie in diesem Fall einfaches ASCII sein kann.

Die *IDLXML* DTD und das *xml2idl* XSLT Stylesheet werden nach sorgfältiger Überprüfung auf Vollständigkeit dem EJCCM Projekt zur weiteren Verwendung zur Verfügung gestellt werden.

4.3.4 Generierung der Implementierungsvorlagen

Die in den vorigen Schritten erzeugte gebundene Team-IDL muss im weiteren Verlauf vollständig implementiert werden. Da der Entwickler aber nur die Rollenfunktionalität implementieren soll, die im ungebundenen Team definiert wurde, wird die Implementierung der implizit erzeugten Schnittstellen und Abhängigkeiten des gebundenen Teams generiert. Dazu werden die von der Plattform optional generierten Vorlagen für die Executors um die Object Teams/CCM-spezifischen Anteile ergänzt. Der Ablauf ist in Abbildung 4.3 dargestellt.

EJCCM Code Generator

Der Code Generator der EJCCM Plattform generiert optional aus den Informationen von IDL und CIDL Beschreibungen einen *main executor* nach der *monolithic* oder *locator strategy* (siehe Kapitel 2.2.3 auf Seite 24) für jeden definierten Komponententyp. Diese dienen als Vorlage für den Entwickler und beinhalten bereits alle Methoden, die aufgrund der Schnittstellen implementiert werden müssen - teilweise mit leeren Rümpfen - und sind somit zumindest compilierbar.

BeautyJ

Um auch die Generierung der Implementierungsanteile mit Hilfe von XSLT durchführen zu können, wird an dieser Stelle ein weiteres Open-Source Tool integriert. *BeautyJ* [Gul] ist ein Werkzeug zur Transformation und automatischen Re-Formatierung von Java Source-Code. Außerdem bietet es die Konvertierung in das *XJava* XML-Format und zurück. Das XJava-Format ist eine zu BeautyJ gehörende XML DTD, die eine 1:1-Abbildung der Elemente des Java Source-Codes darstellt. Methodenrümpfe werden nicht weiter analysiert, sie werden

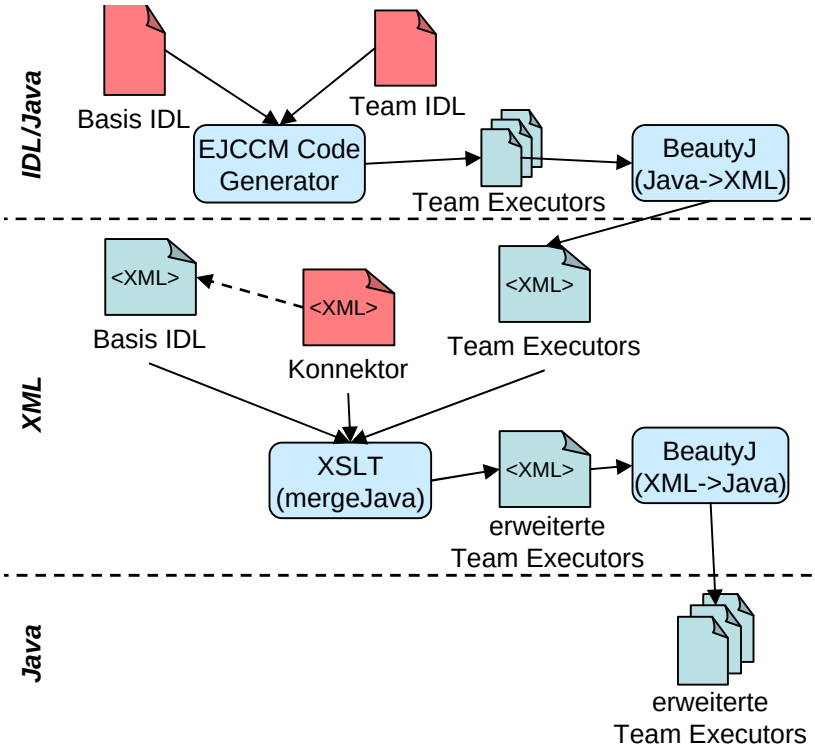

Abbildung 4.3: Erweiterung der generierten Executors

als zusammenhängendes Element repräsentiert. Das verhindert eine komfortable Bearbeitung der Rümpfe in XSLT, was aber hier zunächst auch nicht nötig ist, da Methodenrümpfe bei Bedarf immer komplett generiert werden. [8]

mergeJava XSLT Stylesheet

Die mit Hilfe von BeautyJ erstellten XML-Repräsentationen der Executor-Vorlagen werden im nächsten Adaptierungsschritt per XSLT transformiert und um die Implementierung von Object Teams/CCM-spezifischen Methoden erweitert. Dafür wurde das *mergeJava* Stylesheet entwickelt, das folgende Erweiterungen an den Executor-Vorlagen vornimmt:

Regeln für die Team-Komponente:

1. Importieren benötigter Pakete wie `java.util.*`
2. Attribut `active`: Initialisierung mit `false`, Rumpf der setter-Methode und der Methode `isActive()` ergänzen

[8] Eine Alternative zum XJava-Format könnte *JavaML* darstellen, eine XML DTD für Java Source-Code, die auch Methodenrümpfe in XML-Elemente splittet und somit einen kompletten AST darstellt.

3. Rollen/Basis-Cache für jeden gebundenen Rollentyp anlegen:
   `Hashtable <roleType>_cache`

4. Rumpf der folgenden Methoden für jeden gebundenen Rollentyp ergänzen:

   - `void add<roleType>(<roleType> role, <baseType> base)`
   - `<roleType> lift2<roleType>(<baseType> base)`

Regeln für die Rollen-Komponenten:

1. private Attribute für Referenzen auf: (vgl. Listing 4.5 auf Seite 60)

   - Komponenten-Schnittstelle der Basis-Komponente
   - alle Facets der Basis-Komponente
   - Team-Facet

2. Ergänzen der Methode `configuration_complete()` um das Setzen der o.g. Attribute durch die verbundenen Receptacles und Registrierung beim Team durch `add<roleType>(myself, _ot_baseComp)` (vgl. Listing 4.6 auf Seite 61)

3. Ergänzen der Methodenrümpfe für alle Operationen der Basis-Schnittstellen (vgl. Listing 4.7 auf Seite 61 und Listing 4.8 auf Seite 62)

4. Ergänzen der Methodenrümpfe aller per Callout gebundenen Operationen der Rollen-Schnittstellen (vgl. Listing 4.9 auf Seite 62)

5. Anlegen privater base-call Methoden für jedes replace-Callin (vgl. Listing 4.10 auf Seite 63)

Die so erweiterten Executor-Vorlagen werden anschließend durch BeautyJ wieder in regulären Java Source-Code umgewandelt. Bis auf die nicht per Callout gebundenen Rollenmethoden sind nun alle Methoden implementiert, so dass der Entwickler nur noch für diese die Rümpfe einfügen muss. Dann kann die Implementierung im Rahmen des weiteren Build-Prozesses zusammen mit den Stubs und Skeletons kompiliert werden.

4.3.5 Steuerung des Build-Prozesses

Wie Kapitel 2.2.6 auf Seite 31 bereits gezeigt hat, ist die Entwicklung von Anwendungen mit dem CCM ein relativ komplexer Prozess, in dem diverse Artefakte beteiligt sind oder generiert werden und in dem eine Reihe von Compilern, Code-Generatoren und weiterer Hilfsmittel zum Einsatz kommen. Daher wurde

zunächst eine auf Ant[9] basierende Build-Umgebung für EJCCM erstellt, die seit der Version 0.1.4 Bestandteil der Distribution ist. Sie reflektiert den Entwicklungsprozess des CCM wesentlich besser als die bis dahin zum Einsatz gekommene Variante mit make und wurde stetig weiterentwickelt.

Ein Ant Build-Skript für die Entwicklung einer CCM-Anwendung umfasst targets für die Verarbeitung von IDL- und CIDL-Dateien, die Generierung von Stubs und Skeletons sowie deren Kompilierung, die optionale Generierung von Executor-Vorlagen, die Kompilierung der Komponentenimplementierung, sowie das Packaging, Assembly und Deployment der Anwendung. Über Konfigurationsdateien kann der Entwickler unter anderem die Generierung von Executor-Vorlagen und diverse Compiler-Optionen steuern.

[9] Ant ist ein Java-basiertes Build Tool, das durch eigene Klassen erweiterbar ist. Der Build-Prozess wird dabei über XML-Dateien gesteuert, die sogenannte *targets* definieren. Innerhalb eines targets können diverse *tasks* ausgeführt werden, die verschiedene Aufgaben wie Compileraufrufe kapseln.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Das CORBA Component Model bietet durch seine Container-Architektur und die damit verbundene relativ lose Anbindung von Diensten bereits eine Form von Separation of Concerns. Allerdings ist die Menge dieser Dienste hier vorgegeben und lässt sich auch nur begrenzt anpassen. Durch die Strukturierung durch Komponenten ist auch eine Modularisierung der Anwendungen in funktionale, austauschbare Einheiten möglich. Dabei wird der Entwickler durch die explizite Definition angebotener und erwarteter Schnittstellen durch die verschiedenen Ports unterstützt, die eine vergleichsweise lose Koppelung zwischen Komponenten bieten. Jedoch gibt es kein Konzept zur nachträglichen Adaptierung des Verhaltens einzelner Komponenten oder gar der Interaktionen zwischen ihnen.

Genau das sind wiederum Features, die AOP und im Speziellen Object Teams bereits im Rahmen objektorientierter Entwicklung erfolgreich lösen. Hier ist die Menge der Aspekte nicht begrenzt, neue Aspekte können unabhängig entwickelt und angewendet werden. Mit Object Teams lassen sich zur Zeit Java- und Ruby-Anwendungen nachträglich um Objekt-Kollaborationen erweitern und somit in ihrem Verhalten verändern.

Um diese Möglichkeiten der Modularisierung und nicht-invasiven Adaptierung von Interaktionen auf die Ebene von Komponenten zu übertragen, wurde das Object Teams-Modell im CCM nachgebildet. Mit den entwickelten prototypischen Werkzeugen lassen sich Teams im Sinne von Object Teams als CCM-Anwendungen implementieren und an Basis-Anwendungen binden, von denen nur die Beschreibung der Schnittstellen in IDL verfügbar ist. Dies kann sowohl zur nachträglichen Integration inkompatibler Komponenten (Stichwort *Enterprise Application Integration*) oder zur Anpassung bestehender Komponenten an veränderte Anforderungen (Stichwort *Unanticipated Software Evolution*) genutzt werden. Bei Anwendungen, die von vornherein mit Object Teams/CCM entworfen werden, lassen sich Concerns besser modularisieren und flexibler zu-

sammenführen, als dies bisher der Fall ist. Ein Vorteil, den das CCM als Plattform bietet, ist dass sich mit Object Teams/CCM prinzipiell Anwendungen adaptieren lassen, die in anderen Programmiersprachen implementiert sind und eventuell sogar auf anderen Hosts laufen als die Team-Anwendung.

5.2 Ausblick

Nächste mögliche Entwicklungsschritte für Object Teams/CCM wären:

- Dynamische Erzeugung und Verknüpfung von Rollen mit Basen beim Deployment
- method interception anstelle des manuellen "Umbiegens" der Client-Referenzen auf Wrapper
- Adaptierung einer Basis-Klasse durch mehrere Teams/Rollen
- Verstärkte Unterstützung durch graphische Werkzeuge

Außerdem sollte das entworfene Modell weiter auf die Einsetzbarkeit im Hinblick auf größere und komplexere Anwendungen, insbesondere bei Ausnutzung von Plattformdiensten wie Persistenz und Transaktionen überprüft werden.

Literaturverzeichnis

[AKB99] ATKINSON, COLIN, THOMAS KÜHNE und CHRISTIAN BUNSE: *Dimensions of Component-based Development*. In: *4th International Workshop on Component-Oriented Programming*. University of Karlskrona/Ronneby, 1999. 49

[Asp] ASPECTJ: *Homepage von AspectJ: http://www.aspectj.org*. 35, 38

[BA01] BERGMANS, L. und M. AKSIT: *Composing Multiple Concerns Using Composition Filters*, 2001. 39

[Bin02] BINDER, CHRISTOF: *Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspektorientierte Techniken*. Diplomarbeit, Technische Universität Berlin, 2002. 40, 47

[Bis02] BISHOP, JUDITH (Herausgeber): *Proceedings of the 1st Int. Working Conference on Component Deployment, Berlin 2002*. Springer-Verlag, 2002. 11

[Buc99] BUCHHOLZ, JOACHIM: *Component-Based Development: Methodik und Toolunterstützung*. Diplomarbeit, Fachhochschule Furtwangen, 1999. 10

[CP] COMPUTATIONAL PHYSICS, INC.: *Enterprise Java CORBA Component Model Homepage: http://www.ejccm.org*. 15, 65

[Esk99] ESKELIN, P.: *Component Interaction Patterns*. In: *Proceedings of PLoP 1999*, 1999. 50

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995. 9, 41, 55

[Gul] GULDEN, JENS: *BeautyJ: Customizable Java Source Code Transformation Tool: http://beautyj.berlios.de*. 70

[Hac02] HACKER, FLORIAN: *Aspektorientiertes Entwerfen mit 'Aspectual Collaborations' - Entwicklung eines grafischen Editors für die repository-basierte Entwicklungsumgebung PIROL*. Diplomarbeit, Technische Universität Berlin, 2002. 46

[Her02a] HERRMANN, STEPHAN: *Composable designs with UFA*. In: *Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*, 2002. 46

[Her02b] HERRMANN, STEPHAN: *Object Teams: Improving Modularity for Crosscutting Collaborations*. In: *Net Object Days 2002*, 2002. 39, 40, 65

[Her03] HERRMANN, STEPHAN: *Object Confinement in Object Teams - Reconciling Encapsulation and Flexible Integration*. In: *3. AOSD Workshop der GI, Essen 2003*, 2003. 40, 55

[HMO01] HERRMANN, STEPHAN, MIRA MEZINI und KLAUS OSTERMANN: *Joint efforts to dispel an approaching modularity crisis - Divide et impera, quo vadis?* In: *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001. 1, 52

[Hun03] HUNDT, CHRISTINE: *Bytecode-Transformation zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/Java*. Diplomarbeit, Technische Universität Berlin, 2003. 39, 40, 47

[KCA01] KNIESEL, G., P. COSTANZA und M. AUSTERMANN: *JMangler - A Framework for Load-Time Transformation of Java Class Files*, 2001. 39

[KLM+97] KICZALES, GREGOR, JOHN LAMPING, ANURAG MENHDHEKAR, CHRIS MAEDA, CRISTINA LOPES, JEAN-MARC LOINGTIER und JOHN IRWIN: *Aspect-Oriented Programming*. In: *Proceedings European Conference on Object-Oriented Programming*. 1997. 35

[Kni00] KNIESEL, GÜNTER: *Dynamic Object-Based Inheritance with Subtyping*. Doktorarbeit, Universität Bonn, 2000. 39

[MM02] MARVIE, RAPHAËL und PHILIPPE MERLE: *CORBA Component Model: Discussion and Use with OpenCCM*. Technischer Bericht, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2002. 15

[Mz01] MOSCONI, MARCO und MESUT ÖZHAN: *Schnittstellen und Kontrakte*. TU Berlin, 2001. 7

[OMG99] OMG: *CORBA Components - Volume I: Joint Revised Submission*. Object Management Group. TC Document orbos/99-07-01, 1999. 15

[OMG02a] OMG: *CORBA Components: Specification, Version 3.0*. Object Management Group. TC Document formal/02-06-65, 2002. 14

[OMG02b] OMG: *CORBA Components: Specification, Version 3.0 (Updated)*. Object Management Group. TC Document ptc/02-08-03, 2002. 14

[OMG02c] OMG: *CORBA: Core Specification, Version 3.0*. Object Management Group. TC Document formal/02-12-06, 2002. 14

[OT00] OSSHER, H. und P. TARR: *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. In: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000. 38

[Par03] PARKER, SEAN: *EJCCM User Guide*. 2003. 15

[POM03] PICHLER, ROMAN, KLAUS OSTERMANN und MIRA MEZINI: *On Aspectualizing Component Models*. Software Practice and Experience, Wiley Publishers, 2003. 1, 52

[Que] QUEDO: *Homepage von Quedo (QoS enabled distributed objects): http://qedo.berlios.de*. 52

[Rui] RUIZ, DIEGO SEVILLA: *The CORBA and CORBA Component Model (CCM) Page: http://ditec.um.es/ dsevilla/ccm/*. 15

[SPS02] SCHONGER, STEFAN, ELKE PULVERMÜLLER und STEFAN SARSTEDT: *Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees*. In: *2. AOSD Workshop der GI, Bonn 2002*, 2002. 65

[Sun01a] SUN: *Java 2 Platform Enterprise Edition Specification, v1.4*. Sun Microsystems, 2001. 11

[Sun01b] SUN: *Java Servlet Specification, Version 2.3*. Sun Microsystems, 2001. 11

[Szy99] SZYPERSKI, CLEMENS: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999. 6, 8

[TJJ00] TRUYEN, E., B. JRGENSEN und W. JOOSEN: *Customization of Component-Based Object Request Brokers through Dynamic Configuration*. In: *Proceedings of TOOLS Europe'2000*, 2000. 39, 52

[TVJ+01a] TRUYEN, EDDY, BART VANHAUTE, WOUTER JOOSEN, PIERRE VERBAETEN und BO NORREGAARD JORGENSEN: *Dynamic and Selective Combination of Extensions in Component-Based Applications*. In: *International Conference on Software Engineering*, 2001. 39, 52

[TVJ+01b] TRUYEN, EDDY, BART VANHAUTE, WOUTER JOOSEN, PIERRE VERBAETEN und BO NØRREGAARD JØRGENSEN: *Customization of On-line Services with Simultaneous Client-Specific Views*, 2001. 39, 52

[Vei02] VEIT, MATTHIAS: *Evaluierung modularer Softwareentwicklung mit 'Object Teams' am Beispiel eines Projektmanagementsystems*. Diplomarbeit, Technische Universität Berlin, 2002. 47

[WSa03] WANG, NANBOR, DOUGLAS C. SCHMIDT und ANDERE: *Total Quality of Service Provisioning in Middleware and Applications*, 2003. 52