Nicht–invasive Komponentenadaption

Übertragung des Programmiermodells “Object Teams” auf verteilte Systeme

Lehrstuhl von Prof. Dr. S. Jähnichen
Institut für Softwaretechnik – Fakultät IV
Technische Universität Berlin

Diplomarbeit
von Timmo Gierke

April, 2003

Kontakt: Timmo Gierke <timmo@cs.tu-berlin.de>
Matrikelnr.: 169366

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befaßt sich mit nicht–invasiven Adaptionstechniken in Bezug auf verteilte, komponentenbasierte Systeme.

Der Begriff "nicht–invasiv" bezieht sich in der Medizin auf die verwendeten Fomulierungen nicht–invasive Behandlung und im Gegensatz dazu invasive Therapie. Mit nicht–invasiver Medizin werden äußerliche Anwendungen bezeichnet, eine invasive Behandlung von Krankheiten erfolgt durch einen Eingriff / Operation. In beiden Fällen soll natürlich als Ergebnis eine Genesung des Patienten eintreten.

Nicht–invasive Adaption wird daher in dieser Arbeit als ein Nicht–Eingreifen in den Aufbau und die Struktur einer Entität definiert, obwohl das von außen erkennbare Verhalten derselben sich sehr wohl ändern kann.

Was ist eine Komponente?

Der Begriff "Komponente" wird in den unterschiedlichsten Bereichen verschieden benutzt. Ein Flugzeugbauer wird mit "Komponente" vermutlich ein Subsystem der gesamten Flugzeugtechnik bezeichnen, Elektrotechniker verstehen unter "Komponente" i.d.R. ein einzelnes Bauteil, aber möglicherweise auch einen Teilbereich innerhalb eines komplexeren Systems.

Die Bezeichnung "Software–Komponente" allein sagt noch nicht besonders viel aus, denn der Begriff "Komponente" bezieht auch im Bereich der Informatik eine Vielzahl von Dingen (meist Modulkonstrukte) mit ein, wobei jede Bezeichnung sicherlich ihre eigene, gleichwertige Gültigkeit hat, aber keine Präzision beinhaltet. Was eine Komponente ist, kann eher daran erkannt werden, welche Eigenschaften eine Komponente hat und wie sich die Sicht auf eine Komponente während des Entwicklungsprozesses ändert (Komponentenformen). Eine Einordnung von Software–Komponenten wird anhand dieser Kriterien vorgenommen.

Unter "Adaption" versteht man generell eine Umarbeitung, Bearbeitung bzw. Anpassung einer Entität an die (sich ändernde) Umwelt. Das Thema dieser Arbeit ist es, unter anderem, darzustellen, was Adaption von Komponenten leisten und wie eine technische Realisierung erfolgen kann.

Adaptionstechniken sind gerade auch für komponentenbasierte Systeme von entscheidener Bedeutung und teilweise unabdingbar, im Hinblick auf kommerzielle Komponenten. In dieser Arbeit wird aufgezeigt, wie die Konzepte der Komponententechnologie mit den Ansätzen des aspektorientierten Paradigmas verknüpft werden können, um damit die Adaption von Komponenten, neben der "klassischen" Form der Evolution von Komponenten, zu etablieren.

2 Komponententechnologie

Dieses Kapitel soll den Begriff *Komponente*, wie er in dieser Arbeit verwendet wird, einordnen und verständlich machen. Es werden Notations- und Spezifikationstechniken eingeführt, die als Grundlage für die weiteren Kapitel aufgefasst werden können[1].

Das Thema *Evolution* von Komponenten wird erörtert und die sich daraus ergebenen Probleme vorgestellt.

2.1 Eigenschaften von Software–Komponenten

Der Grundgedanke der Komponententechnologie folgt dem uralten Prinzip: *dividie et impera*! Das mit Komponenten angestrebte Ziel ist es, Herr über die Komplexität des Systems zu werden und diese zu verwalten.

Es ist sicher ein hehres Ziel, etwas einmal zu entwickeln und dann immer und immer wieder verwenden zu können. Tatsache ist jedoch: Die Welt ändert sich, heutzutage schneller denn je!

Gerade im Bereich der Informatik sind Anforderungsänderungen häufig anzutreffen. Daher steht v.a. die Abhängigkeiten zwischen Komponenten (und deren Verwaltung) im Vordergrund der Entwicklung eines Systems. Einzelne Komponente müssen leicht austauschbar sein, entweder durch eine komplett andere Implementierung oder eine verbesserte Version. Komponententechnologie fokussiert daher die Architektur eines Systems, bzw. die Organisation des Gesamtsystemes.

Viele Ideen der Komponententechnologie folgen dem objektorientierten Paradigma: Komponenten fassen Daten und Funktionalität zur Bearbeitung dieser Daten zu einer Einheit zusammen. Dabei wird das Prinzip der *Kapselung* angewandt: Es bleibt "Klienten"[2] einer Komponente verborgen, in welcher Form die Daten gespeichert sind und wie die Funktionalität erbracht wird. Die Abhängigkeit zwischen einem Klienten und einer Komponente wird dadurch auf die (von außen sichtbare) Spezifikation der Komponente reduziert. Es besteht also *keine* Abhängigkeit zu einer spezifischen Implementierung ('*seperation of concerns*'). Ebenfalls aus der objektorientierten Welt übernommen wurde das Konzept der *Identität*. Jedes Software–Objekt hat eine eindeutige Identität, unabhängig von dessen Zustand.

[1] Vorgestellte Konzepte in Kap. 2.1 u. 2.3 basierend auf 'UML Components', Cheeseman & Daniel, [8].

[2] Die Bedeutung eines Klienten wird im folgenden definiert als eine benutzende (Software–) Entität.

Komponenten erweitern die Konzepte des objektorientierten Paradigmas um eine explizite Repräsentation spezifizierter Abhängigkeiten. Diese wird mit *Schnittstelle* bezeichnet ('*interface*', Symbol: ○——). Klienten eines Objektes nutzen dadurch die Fähigkeiten des Objektes auf Umwegen, d.h., indirekt über dessen Schnittstelle. Die Menge aller Fähigkeiten eines Objektes kann dabei über mehrer Schnittstellen verteilt sein.

Komponenten zeichnen sich also v.a. durch die Trennung der Komponentenspezifikation von der Komponentenimplementierung aus sowie der Aufteilung einer Komponentenspezifikation in mehrere Schnittstellen. Abhängigkeiten zwischen Komponenten können dadurch auf einzelne Schnittstellen begrenzt werden. Die Menge der notwendigen Änderungen ist damit deutlich reduzierbar: Eine Komponente kann durch eine andere (mit vielleicht anderer Spezifikation) ausgetauscht werden, solange die von einem "Klienten" benutzte Schnittstelle angeboten wird.

2.2 Komponentenformen

Die zweite wichtige Frage ist, wie ändert sich die Sicht auf Komponenten während des Entwicklungsprozesses, also von der Anforderungsanalyse ('*requirements analysis*') zur Spezifikation, vom Entwurf ('*design*') zur Realisierung ('*provisioning*'), vom Zusammenbau ('*assembly*') zur Auslieferung ('*deployement*') bis hin zur Laufzeit ('*runtime*')?

Anwendungen bestehen aus einer oder mehreren Komponenten. Damit ein Zusammenbau reibungslos erfolgen kann, müssen Komponenten gewissen Standards folgen, meist bestimmt durch die Umgebung, in welche Komponenten ausgeliefert werden sollen (*component environment standards*). Bekannte Umgebungen sind bsp. EJB[22] oder CCM[15]. Große Softwarefirmen definieren i.d.R. ebenfalls eigene, spezifische Standards. Diese sind notwendig, damit eine Komponente unproblematisch eingefügt werden kann (*plug & play*). Komponenten folgen also einem *Komponentenstandard*.

Komponentenstandards sind sicherlich eine Grundvoraussetzung für einen reibunglosen Einbau einer Komponente. Einem Standard zu folgen allein reicht nicht aus, es muß v.a. spezifiziert sein, welche Funktionalität eine Komponente erbringt. Es bedarf also einer klaren *Komponentenspezifikation*, wobei die Hauptaufgabe während der Spezifikation darin besteht, die *Komponentenschnittstellen* zu definieren. Eine Komponentenspezifikation umfaßt alle angebotenen Schnittstellen ('*offered interfaces*') sowie alle erwarteten Schnittstellen einer Komponente ('*used interfaces* or *expected interfaces*').

Abbildung 2.1 zeigt die graphische Darstellung einer Komponentenspezifikation in UML, wie sie von Cheeseman und Daniels[8] vorgeschlagen wird. Eine Komponente auf der Ebene der Spezifikation wird als ein rechteckiger Kasten symbolisiert, der den Namen der Komponente beinhaltet und den Stereotyp «`comp spec`». Der

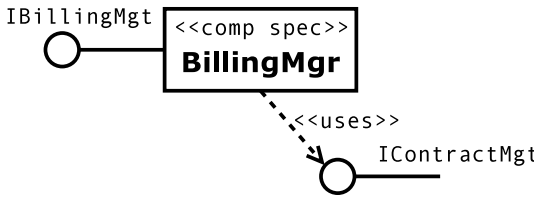
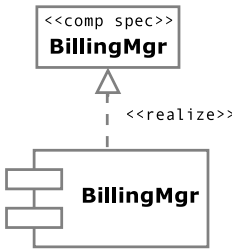


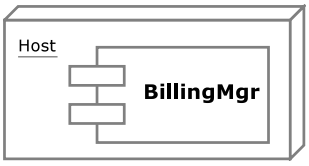
Abbildung 2.1: Komponentenspezifikation der Komponente `BillingMgr`.

Stereotyp ist erforderlich, um den "Kasten" als Komponente zu markieren (im Gegensatz zu einer Klasse mit ausgeblendeten Attribut– und Methodenfeldern). Die "Lollipops" symbolisieren die Schnittstellen. Die in der Abbildung gezeigte Komponente `BillingMgr` offeriert demnach eine Schnittstelle `IBillingMgt` und erwartet im Gegenzug die Schnittstelle `IContractMgt`[3].

Betrachtet man eine Komponentenspezifikation aus der Sicht des "Zusammensteckens", ist die Implementierung einer Komponente *nicht* entscheidend, denn eine Komponente innerhalb einer Konfiguration (*'assembly'*) sollte durch eine andere Komponente (mit gleicher Spezifikation) ersetzbar sein. Entscheidend ist also, welche Abhängigkeiten zwischen den einzelnen Teilen bestehen, und nicht, wie diese intern implementiert sind. Daher muß eine klare Trennung zwischen der Komponentenspezifikation und der *Komponentenimplementierung* gewahrt werden. Besteht auf der Ebene der Spezifikation keine Abhängigkeit mehr zwischen einer Spezifikation und deren Implementierung, kann die Implementierung später in einer beliebigen (Programmier–) Sprache erfolgen und jede Art von Datenspeicher verwenden.



Zur graphischen Repräsentation einer Komponentenimplementierung wird das in UML definierte Symbol einer Komponente verwendet. Die Idee dabei ist, daß das UML–Konzept der "Komponente" ohnehin mit Blick auf die Implementation und Auslieferung ('deployement') einer Komponente entworfen wurde. Die hier gezeigte Abbildung verdeutlicht den Zusammenhang zwischen einer Komponentenspezifikation und einer Komponentenimplementierung. Die Komponentenimplementierung `BillingMgr` erfüllt die Komponentenspezifikation `BillingMgr`.



Die Auslieferung ('deployement') einer Komponente bedeutet immer, eine Komponentenimplementierung auf einem Rechner zu installieren (und ggf. der Umgebung bekannt zu machen). Wurde eine Komponente ausgeliefert, spricht man von einer *installierten Komponente* . Die nebenstehende Abbildung zeigt die in dieser Arbeit verwendete Notation zur graphischen Darstellung einer installierten Komponente (der Kubus

[3]Die Erfüllung einer «`uses`» Beziehung wird in Kap. 2.3 gezeigt.

symbolisiert in UML einen Knoten / Rechner).

Bisher wurden Komponenten nur in ihren "statischen" Zuständen betrachtet; doch natürlich besitzen Komponenten zur Laufzeit (des Systems) auch einen Zustand. D.h., ebenso wie die von einer Komponente erbrachten Dienste wichtig sind, sind auch die Informationen, die durch eine Komponente verarbeitet werden, von entscheidender Bedeutung. Wird eine Komponente ausgetauscht, muß sichergestellt werden, das der letzte Zustand erhalten bleibt. Eine Einheit, aus Zustand und Funktionalität bestehend, ist die primäre Eigenschaft von Objekten. Daher wird eine Instanz einer Komponente als *Komponentenobjekt* bezeichnet. Die graphische Notation eines Komponentenobjektes folgt dem Symbol einer Klasseninstanz, d.h., es wird ein "Kasten" gezeichnet der den Namen der Komponente trägt. Der Name ist, wie bei Instanzen üblich, mit einem Unterstrich versehen.

2.3 Komponentenarchitekturen

Für die Entwicklung komplexer System sind häufig Regeln zum Entwurf und zur Implementierung der einzelnen Elemente erforderlich: Es bedarf einer (System–) Architektur.

In Bezug auf komponentenbasierte Systeme bedeutet dies, daß die Abhängigkeiten zwischen Komponenten und deren Zusammenspiel, erfaßt werden muß. Cheesman & Daniels definieren eine Komponentenarchitektur wie folgt:

> **Def.: Komponentenarchitektur** [8]
> Eine Komponentenarchitektur ist eine Menge von Komponenten mit der Granularität einer Anwendung, deren strukturalen Beziehungen sowie ihrer verhaltensbezogenen Abhängigkeiten.

Eine strukturale Beziehung umfaßt die Assoziations– und Vererbungsbeziehungen zwischen Komponentenspezifikationen und Komponentenschnittstellen sowie die Kompositionsbeziehungen zwischen Komponenten. Verhaltensbezogene Abhängigkeiten sind Abhängigkeitsbeziehungen zwischen Komponenten und anderen Komponenten, Abhängigkeitsbeziehungen zwischen Komponenten und Schnittstellen sowie zwischen Schnittstellen und Schnittstellen.

Die obige Definition einer Komponentenarchitektur ist eine "logische" Definition und damit unabhängig von der verwendeten Technologie! Ein solche Sicht auf ein System bietet v.a. zwei Vorteile: Der Grad der Kopplung (von schwach bis stark gekoppelt) zwischen den einzelnen Komponenten kann auf diese Weise präzise bestimmmt werden. Ebenso sind die Seiteneffekte, die bei einem Austausch einer Komponente bzw. einer Modifikation auftreten, besser voraussehbar.

Komponentenarchitekturen können in vielfältiger Form auftreten, sie sind dabei abhängig vom Kontext des darzustellenden Sachverhaltes:

- Für die Spezifikation: Komponentenspezifikations–Architektur
- Für die Realisierung: Komponentenimplementations–Architektur
- Zur Laufzeit: Komponentenobjekt–Architektur

Die verschiedenen Architekturen werden in den folgenden Abschnitten einzeln erläutert.

Komponentenspezifikations–Architektur

Eine Komponentenspezifikations–Architektur besteht aus Komponentenspezifikationen und Schnittstellen sowie deren Abhängigkeiten von einander. Abhängigkeiten bestehen dabei zwischen Schnittstellen bzw. Komponentenspezifikationen und einer Schnittstelle. Die Idee dabei ist, daß eine Abhängigkeit einen Vertrag mit der Implementierung einer Komponente beschreibt, d.h., jede Komponentenimplementierung muß diese Abhängigkeiten berücksichtigen. Cheesman und Daniels begründen die geforderte Genauigkeit in der Spezifikation damit, daß für große Systeme die Komponentenspezifikations–Architektur ein kritischer Punkt in einer Systemarchitektur darstellt, denn es verbietet einzelnen Entwicklungsteams die freie Auswahl der (wieder–) zu benutzenden Implementierungen.

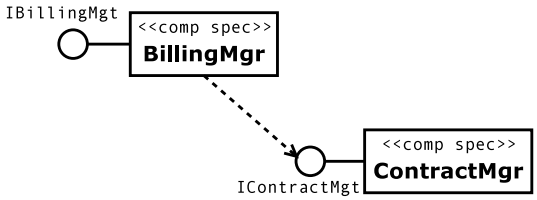


Abbildung 2.2: Komponentenspezifikations–Architektur–Diagramm.

Das Entscheidende bei dieser Form der Architektur ist also, daß eine erwartete Schnittstelle einer Komponente («`uses` `Beziehung`) an eine offerierte Schnittstelle einer anderen Komponenten gebunden wird («`offers`» Beziehung). Abbildung 2.2 zeigt eine Komponentenspezifikations–Architektur bestehend aus den Komponentenspezifikationen der Komponente `BillingMgr`, desssen erwartete Schnittstelle `IContractMgt` an die gleichnamige Schnittstelle der Komponentenspezifikation der Komponente `ContractMgr` gebunden wird.

Komponentenimplementations–Architektur

Abhängigkeiten zwischen einzelnen Komponentenimplementierung werden mit Hilfe der Komponentenimplementations–Architektur beschrieben. Diese umfaßt alle bereits in der Spezifikation festgelegten Abhängigkeiten sowie andere, während der Implementierung hinzugefügte, Abhängigkeiten. Das Beispiel in Abbildung 2.3 zeigt die Implementierung der Komponente `BillingMgr`, welche eine

Abhängigkeit zur Implementierung der Komponente `ContractMgr` besitzt (Abhängigkeit aus der Spezifikation übernommen), sowie eine SMTP–Bibliothek zur Erbringung ihrer Funktionalität benötigt[4].

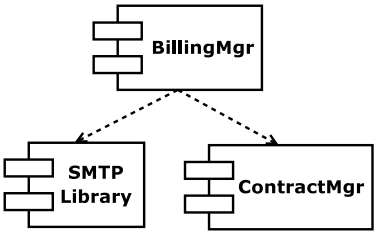


Abbildung 2.3: Komponentenimplementations–Architektur–Diagramm

Komponentenobjekt–Architektur

Zum Schluß soll nun die Komponentenobjekt–Architektur erläutert werden. Die Notwendigkeit dieser Architektur ist damit begründet, daß Komponenten zustandsbehaftete Entitäten sind. Dieser Umstand erfordert es, genau zu definieren, welche Komponetenobjekte benutzt werden. Eine Komponentenobjekt–Architektur spezifiziert exakt, auf welche "Instanzen" zugegriffen wird.

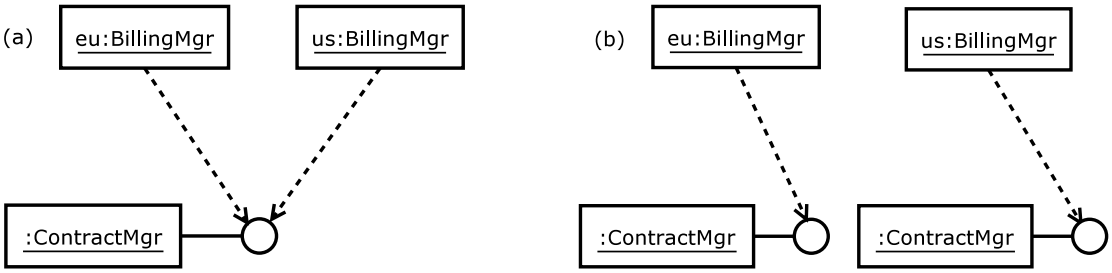


Abbildung 2.4: Alternative Komponentenobjekt–Architekturen

Abbildung 2.4 zeigt zwei verschiedene Objekt–Architekturen, denen beide die gleiche Komponentenspezifikations–Architektur zugrunde liegt. In Abbildung 2.4 (a) benutzen die Instanzen `eu` und `us` der Komponente `BillingMgr` dieselbe Instanz der Komponente `ContractMgr`, während in Abbildung 2.4 (b) die Instanzen `eu` und `us` unterschiedliche Instanzen der Komponente `ContractMgr` nutzen. Eine Komponente zur Verwaltung von Verträgen ist mit Sicherheit zustandbehaftet, so daß eine genaue Spezifikation der benutzten Komponentenobjekte erforderlich ist.

[4]SMTP = Simple Mail Transport Protocol

2.4 Design by Contract

Hinter *'design by contract'* verbirgt sich der Gedanke, die Beziehungen zwischen zwei (oder mehr) Entitäten durch einen Vertrag zu regeln. Ein Vertrag legt eindeutig fest, welche Leistungen von einer Entität erbracht werden müssen, wenn die leistung–fordernde Entität alle im Vertrag definierten Voraussetzungen erfüllt hat. Wie die geforderte Leistung erbracht wird, wird i.d.R. offen gelassen. Zusätzlich sollte in einem Vertrag festgeschrieben sein, welche Konsequenzen eine Vertragsverletzung hat.

Eine bekannte Verwendung von Verträgen im Bereich der Informatik ist die Belegung einer Operation mit Vor– und Nachbedingungen. Darin wird festgeschrieben, welche Bedingungen vor einem Aufruf der Operation erfüllt sein müssen; die Nachbedingung beschreibt, was nach der Rückkehr der Operation gewährleistet wird. Das objektorientierte Paradigma hat zusätzlich die *Klasseninvariante* eingeführt. Alle darin definierten Bedingungen müssen von der Klasse jederzeit erfüllt sein [5].

Im Bereich der Komponententechnologie eignen sich v.a. zwei Vertragstypen:

- Benutzung: Der Vertrag zwischen der Schnittstelle eines Komponentenobjektes und dessen Klienten
- Realisation: Der Vertrag zwischen einer Komponentenspezifikation und dessen Implementierung.

Diese Verträge hängen mit den Rollen des Lebenszyklußes einer Komponente zusammen: Ein Realisationsvertrag schreibt einem Komponentenentwickler vor, welche Leistungen die zu entwickelnde Komponente zu erbringen hat; ein Nutzungsvertrag definiert, wie diese Komponente von Klienten zu verwenden ist.

Nutzungsverträge

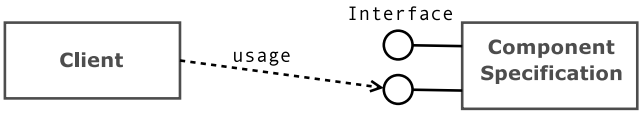


Abbildung 2.5: Vertrag der Benutzung

Ein Nutzungs–Vertrag regelt das Zusammenspiel zwischen einem Klienten und einer Schnittstelle eines Komponentenobjektes, wobei der "Name" des Vertrages i.d.R. dem Namen der Schnittstelle entspricht. Ein Benutzungsvertrag ist immer ein Laufzeitvertrag. Die Spezifikation des Vertrages erfolgt dabei über die Schnittstelle, wobei eine Schnittstelle aus einer Menge von Operationen besteht

[5] "Genauer: Nach jeder Objekterzeugung und nach jedem Aufruf einer Methode aus der Klasse."[10]

und ein (Schnittstellen–) Informationsmodell definiert. Eine Operation ist ein fein–granularer Vertrag an sich. Vor– und Nachbedingungen müssen von jeder Operation erfüllt sein. Eine formale Spezifikation eines Nutzungsvertrages kann z.B. mit Hilfe der 'Object Constraint Language' (OCL, [18]) vorgenommen werden. OCL ist ein Bestandteil der UML–Spezifikation (seit Vers. 2.0).

Ein *Schnittstelleninformationsmodell* ist eine abstrakte Definition aller Daten und Zustände, die durch die Operationen der Schnittstelle verändert werden können sowie eine Menge von Einschränkungen, die für diese Informationen definiert sind (vgl. Klasseninvariante). Ein Schnittstelleninformationsmodell wird dann besonders wichtig, wenn eine Komponente mehr als eine Schnittstelle offeriert. Die möglicherweise verschiedenen Informationsmodelle der Schnittstellen müssen, wenn eine Beziehung zwischen den Daten dieser besteht, sowohl syntaktisch als auch semantisch übereinstimmen.

Realisationsverträge

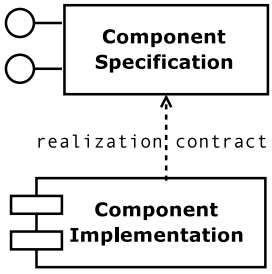
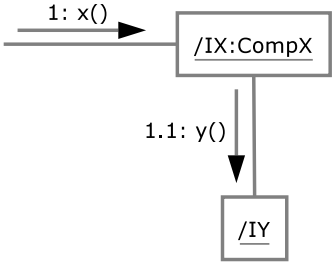


Abbildung 2.6: Realisierung einer Komponentenspezifikation

Im Gegensatz zum Nutzungsvertrag ist ein Realisationsvertrag ein 'designtime'–Vertrag. Darin wird festgeschrieben, welche Bedingungen für die Realisierung einer Komponentenspezifikation durch eine Komponentenimplementation gelten müssen. Eine Komponentenspezifikation definiert die Implementations– und Auslieferungsgrenze, sowie alle Interaktionen mit anderen Komponenten. Letztere sind Teil des Realisierungsvertrages einzelner Operationsimplementierungen. Diese können entweder prozedural (mit Hilfe eines Kollaborationsdiagrammes) oder deklarativ (Einschränkungen zwischen den Daten der offerierten und benutzten Informationsmodelle) festgehalten werden.

In dieser Arbeit werden zur Spezifikation eines Realisierungsvertrages meistens Kollaborationsdiagramme verwendet. Es gelten dabei die Namensregeln wie sie in UML verwendet werden, jedoch mit der Abweichung, daß für den Rollenbezeichner ein Schnittstellenbezeichner eingesetzt wird. Daraus ergibt sich folgende Namensregel: `object/interface:component`



Die Abbildung im Absatz zeigt ein Beispiel. Es wird darin vertraglich festgeschrieben, daß immer, wenn die Operation `x()` der Schnittstelle `IX` aufgerufen wird, die Komponentenimplementierung `CompX` die Operation `y` der Schnittstelle `IY` aufrufen muß. Welche Komponente die Schnittstelle `IY` realisiert, bleibt unspezifiziert.

2.5 Evolution

Eingangs wurde behauptet, daß sich die Anforderungen an Software–Systeme ändern, und gleichzeitig, daß ein komponentenbasiertes System die Menge der nötigen Veränderungen gering hält bzw. überschaubar macht. Jede Änderung einer System– oder Komponentenanforderung resultiert i.d.R. in einer (Weiter–) Entwicklung der betroffenen Elemente. Es werden daher die Begriffe *Evolution* oder '*continious engineering*' verwendet.

Die folgenden Abschnitte erläutern mögliche Formen der Evolution in komponentenbasierten Systemen.

Evolution in der Spezifikation

Evolution auf der Ebene der Spezifikation kann bedeuten: Änderung einer Komponentenspezifikation bzw. einer Komponentenspezifikations–Architektur.

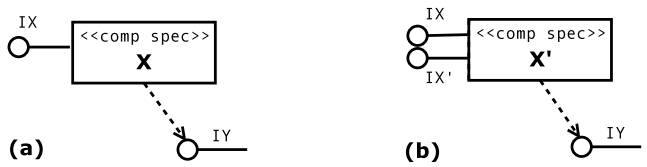


Abbildung 2.7: Evolution einer Komponentenspezifikation.

Abbildung 2.7 zeigt eine mögliche Evolution einer Komponentenspezifikation. Die Komponente `X` in Abb. (a) offeriert die Schnittstelle `IX`, die Komponente `X'` in Abb. (b) erweitert die Spezifikation der Komponente `X` um die neue Schnittstelle `IX'`. Der Nutzungsvertrag mit existierenden Klienten wird durch diese Form der Evolution nicht gebrochen, da die vertraglich festgelegte Schnittstelle (`IX`) weiterhin erfüllt wird. Der Realisierungsvertrag ändert sich sehr wohl! Eine Implementierung der Komponente `X'` muß die zusätzlich offerierte Schnittstelle `IX'` berücksichtigen.

Eine zweite Form der Evolution einer Komponentenspezifikation ist die Veränderung einer erwarteten Schnittstelle, bzw. eine Änderung der Menge der erwarteten Schnittstellen.

Eine Evolution in der Komponentenspezifikations–Architektur bezieht sich auf die Veränderung der verhaltensbezogenen Abhängigkeiten. Es findet eine Evolution

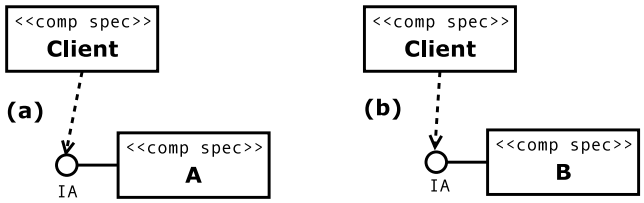


Abbildung 2.8: Evolution einer Komponentenspezifikations–Architektur.

der Beziehungen zwischen Komponentenspezifikationen statt (vgl. Abb. 2.8, (a) -> (b)).

Evolution der Implementation

Die in der Komponentenspezifikation beschriebenen Änderungen müssen, soll der Realisationsvertrag erfüllt bleiben, in der Komponentenimplementation nachvollzogen werden.

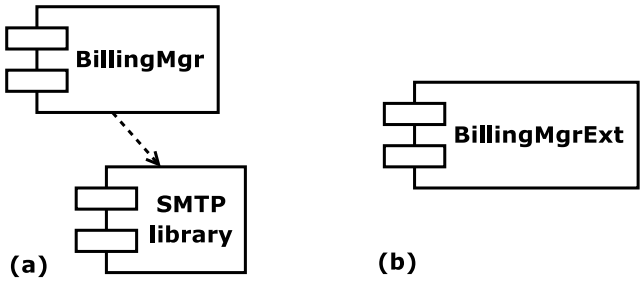


Abbildung 2.9: Evolution einer Komponentenimplementations–Architektur.

Es kann jedoch auch Änderungen geben, die aus einer Änderung der Spezifikation resultieren, z.B., kann es eine neue Version einer Komponentenimplementation geben, die effizienter (in Bezug auf Speicherverbrauch, Performanz, etc.) arbeitet. Auch ist die Evolution der Implementation in Bezug auf die Architektur möglich. So kann, z.B., eine Abhängigkeit zu einer Bibliothek entfallen (vgl. Abb. 2.9).

Konfiguration als Mittel zur Evolution

Konfiguration als Mittel zur Evolution bezieht sich auf die Auslieferung ('deployement') von Komponentenimplementierungen auf ein Zielsystem. Dies kann im einfachsten Fall bedeuten, daß eine gesamte Anwendung (neu) installiert wird, aber auch eine feiner–granulare Installation von einzelnen Komponenten. Während des Startens einer Komponentenumgebung bzw. Anwendung werden die vorhandenen Implementierungen eingelesen und entsprechend verknüpft ('*component wirering*') . Der Sachverhalt wird daher auch mit '*bootstrap configuration*' bezeichnet.

Die technische Realisierung erfolgt dabei meist mit Hilfe von *Komponentendescripto-*

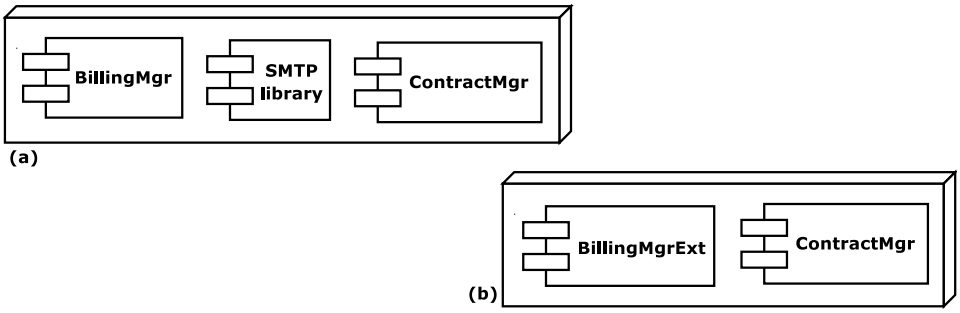


Abbildung 2.10: Evolution zum Zeitpunkt des 'deployement'

ren , welche alle offerierten und erwarteten Schnittstellen der Komponente enthalten, sowie eine Adresse, unter der die Implementierung der Komponente gefunden werden kann. Die Speicherung dieser Informationen erfolgt, z.B., in einer XML Datei (z.B. CCM [15]) oder innerhalb eines Java–JAR Manifestes (z.B. OSGi [20]). Auf der nächst höheren Ebene werden Architekturdescriptoren oder Anwendungsdescriptoren benutzt, um das Zusammenspiel der Komponentenobjekte festzulegen (z.B. CCM [15], TCM).

Evolution zur Laufzeit

Evolution zur Laufzeit eines Systemes wird immer dann benötigt, wenn eine extrem hohe Verfügbarkeit gefordert wird und daher ein "Neustart" nicht akzeptabel ist. Gerade im 'embedded' Bereich, wo die Gesamtlaufzeit von Systemen mehrere Jahre erreichen kann, ist es wünschenswert, einen Komponentenaustausch vorzunehmen, ohne dazu das gesamte System anhalten zu müssen. Aber auch für große Systeme, deren einzelnen Komponenten die Granularität einer Anwendung aufweisen, kann ein 'runtime update' durchaus notwendig sein. Für Webportale mit tausenden von gleichzeitig aktiven Nutzern ist ein Neustart womöglich ebenfalls inakzeptabel.

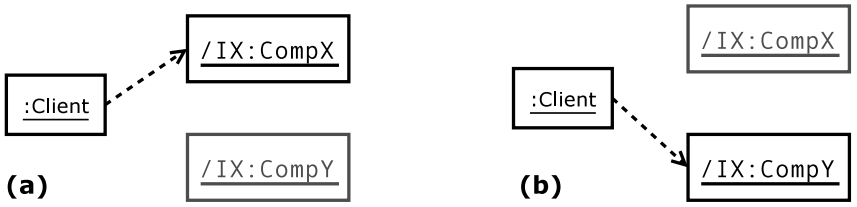


Abbildung 2.11: 'Runtime'–Evolution

Zur technischen Realisierung eines 'runtime update' gibt es eine Reihe von Lösungen (z.B. CORBA Online Upgrades [17], OSGi [20]). Für einen Laufzeitaustausch von Komponenten bedarf es i.d.R. sowohl eines Komponentenstandards als auch einer Unterstützung durch die Umgebung ('*Container*', '*component framework*').

Die gemeinsame Idee ist, den Austauschprozess in mehrere Abschnitte zu unterteilen, wobei jeder Abschnitt als ein Zustand innerhalb des Lebenszyklusses einer Komponente aufgefaßt wird (*'lifcycle management'*). Darin wird eine auszutauschende Komponente "angehalten" (und ggf. auch alle benutzenden Klienten), dann erfolgt der "Austausch" der Komponente, und schließlich wird die neue Komponente "Aktiviert" und die angehaltenen Klienten zur "Weiterarbeit" aufgefordert. "Austauschen" bedeutet dabei, die Abhängigkeitsbeziehungen der Klienten neu zu "verdrahten" ('component wirering', vgl. Abb. 2.11). Technisch gesehen ist ein Komponentenaustausch zur Laufzeit nicht weiter schwierig.

Die Probleme befinden sich eher auf der semantischen Ebene: Wie kann der Zustand einer Komponente übertragen werden, ohne die Anforderung der Maximalspeicherbelegung zu verletzten [3]? Auch wird diskutiert, wie sichergestellt werden kann, daß die "neue" Komponente das System korrekt nutzt und die spezifizierten Anforderungen einhält. Gefordert wird ein sogenannter "Sandkasten" zur Erprobung des Verhaltens(*'sandboxing'* [23]).

2.6 Motivation aspektorientierter Techniken für komponentenbasierte Systeme

Die bisher vorgestellten Konzepte, welche komponentenbasierten Systemen zu Grunde liegen, ermöglichen eine saubere Spezifikation von Abhängigkeiten und bieten einen hohen Grad an Flexibilität. Jedoch werden häufig zwei Punkte übersehen [9]:

- Entwurfsmethoden, welche eine freie Spezifikation von Schnittstellen erlauben
- Heutige, kommerzielle Systeme basieren auf einer aggresiven Nutzung von kommerziellen Komponenten (HTTP–Server, ORB, etc.)

Werden Anwendungen aus kommerziellen Komponenten zusammengebaut, so folgt daraus, daß Software–Entwickler weniger Kontrolle über die Komponentenarchitektur besitzen, und der (Komponenten–) Markt entscheidet, welche Komponenten verfügbar sind und wie diese miteinander interagieren. Eine weitere Problematik von kommerziellen Komponenten ist eine möglicherweise untrennbare Verbindung zwischen einem Komponentenhersteller und dessen Klienten ('*vendor lock*' [9]). Dabei sind die Kosten eines Herstellerwechsels exorbitant, so daß sich Softwareentwickler dem Willen der Hersteller unterordnen müssen.

Schwerer noch wiegt bei kommerziellen Komponenten, wenn die erworbene Software eine "Lücke" in Bezug auf die Anforderungen aufweist. Dabei können zwei Kategorien von "Lücken" definiert werden [9]:

Nichterfüllung der benötigten Funktionalität:
: Eine Komponente stellt nicht exakt die Funtionalität bereit, die von der Applikation gefordert wird ('*design mismatch*'). Die Ursache ist in der Vielzahl der Anforderungen zu suchen, der eine kommerzielle Komponente unterliegt. Daraus folgt für die Systementwicklung, daß die möglichen Anforderungen an Komponenten durch den Markt bestimmt werden.

Integrationsschwierigkeiten:
: Eine Komponente kann nicht oder nur schwer in ein bestehendes System integriert werden. Dies hat mehrere Ursachen: Unterschiedliche Hersteller von Komponenten machen falsche Annahmen darüber, wie ihre Komponenten intergriert werden sollen ('*architectural mismatch*'). Auch könnten Hersteller versucht sein, ihre Kunden durch eigene, nicht standardisierte Intergrationstechniken, an sich zu binden. Und letztendlich wird immer versucht, den Stand der Technik zu überbieten, so daß neu eingebrachte Funktionalitäten zu Integrationschwierigkeiten führen können.

Das bedauerliche Fazit daraus ist, daß die Entwicklung mit kommerziellen Komponenten eines Prozeßes bedarf, in dem alle nicht erfüllten Anforderungen erkannt,

extrahiert und ausgeglichen werden müssen.

Unter anderem daher ist es wünschenswert, Techniken zu verwenden, die eine nachträgliche Adaption des Verhaltens von Komponenten ermöglichen. Liegen die Quellen einer Komponentenimplementierung nicht vor oder können diese aus sonstigen Gründen nicht verändert werden, erweitert sich der Bedarf auf nicht–invasive Adaptionstechniken.

Das Paradigma der aspektorientierten Softwareentwicklung bietet Techniken zur Adaption von Softwaresystemen. Dabei wird ein spezifisches Verhalten einer Anwendung als ein *Aspekt* aufgefaßt, welcher u.U. separat entwickelt werden kann, um nachträglich in ein Softwaresystem integriert zu werden.

Die Möglichkeiten der Aspektorientierung können, bezogen auf Komponenten, präzise mit Hilfe der Nutzungs– und Realisationsverträge erfaßt werden.

Eine Adaption des Verhaltens sollte nicht auf der Ebene der Nutzungsverträge erfolgen, denn dies würde gravierende Folgen für Dienstanbieter und Benutzer nach sich ziehen. Das hinter einer Schnittstelle befindliche Schnittstelleninformationsmodell kann, ohne Vertragsbedingungen zu verletzen, verändert (erweitert) werden!

Richtig interessant wird es allerdings erst mit Blick auf einen Realisationsvertrag. Ohne in das Innenleben der Komponentenimplementierung einzugreifen, wird die Implementierungs– und 'deployement'–Grenze aufgehoben und die definierten Interaktionen einzelner Operationsimplementierungen können beliebig verändert werden.

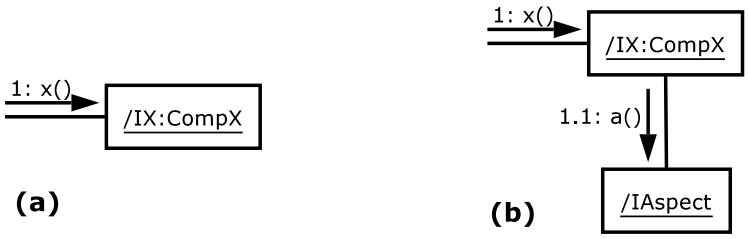


Abbildung 2.12: Aspektorientierte Erweiterung eines Realisationsvertrages.

Abbildung 2.12 zeigt eine Änderung des Realisationsvertrages der Komponente `CompX`. Teilabbildung (a) zeigt den ursprünglichen Vertrag der Komponente. Der Operationsaufruf `x` wird intern verarbeitet, es existieren keine Einschränkungen der Implementierung. Der Vertrag wird in Teilabbildung (b) derart erweitert, daß bei jedem Aufruf der Operation `x`, *danach* die Operation `a` an einer Schnittstelle `IAspect` aufgerufen werden soll. Eine solche "Vertragserweiterung" ist mit den bestehenden Komponententechnologien nur *invasiv* möglich. Das Bestreben dieser Arbeit ist es, zu beweisen, daß solche Erweiterungen *nicht–invasiv* erreicht werden können.

Eine weitere Anforderung an Komponentensysteme kommt neuerdings aus der Domäne der Luftsicherheit. Mit zunehmender Technologisierung der Cockpits von Flugzeugen steigt die von Piloten zu verarbeitende Informationsmenge rasant an. Für die Flugsicherheit entscheidende Daten gehen dadurch u. U. verloren. Es wird daher versucht, die Informationsmenge gering zu halten, in dem Informationen in Abhängigkeit zur jeweiligen Flugphase angezeigt werden. Die Menge der Informationen wird dabei bestimmt durch den Kontext, in dem sich ein Flugzeug befindet. (Z.B., kann die Wetterlage bei einem Flug von Berlin nach München extremen Schwankungen unterliegen, so daß der Pilot in der Schlechtwetterzone einen anderen Informationsbedarf hat als in einer Schönwetterlage.) Die Forschung geht daher in Richtung kontext–sensitiver Informationssysteme, welche die bestehenden Komponenten der sensorischen Erfassung nutzen[1].

Anhand dieses Beispiels möchte der Verfasser belegen, wie erforderlich kontextabhängige Informationssysteme sind. In dieser Arbeit wird deshalb eine innovative Technologie zur Realisierung kontextabhängiger Softwaresysteme dargestellt.

3 CORBA als Infrastruktur für Komponenten

Die *Common Object Request Broker Architecture* (CORBA) ist eine der weit verbreitetesten Plattformen zur Realisierung von verteilten Softwaresystemen. Die *Object Management Group*, ein Konsortium aus über 600 Firmen, veröffentlichte die erste CORBA–Spezifikation 1991. CORBA ist mittlerweile für nahezu alle Kombinationen aus Hardware und Betriebssystemen verfügbar und wird von einer Vielzahl von Programmiersprachen unterstützt. Alle großen Softwarehersteller bieten eigene Implementierungen der CORBA-Spezifikation an, die erfreulicherweise ein hohes Maß an Interoperabilität aufweisen. Die Vielfalt wurde in den letzten Jahren noch durch robuste *open source* Implementierungen vergrößert, wobei hier auf die *Java* Implementierung *OpenORB*[19] des Exolab hingewiesen werden soll. Diese wurde zusammen mit *MICO*[12] in dieser Arbeit verwandt.

Für den Einbau von aspektorientierten Techniken in verteilte Systeme wurde CORBA in der Version 2.4 [14] gewählt, da es eine Fülle von Erweiterungsmöglichkeiten bietet und eine hohe Industrierelevanz aufweist. Dies vor allem, wenn durch neue Techniken Möglichkeiten geschaffen werden, um bestehender CORBA-Anwendungen (Altsysteme) zu adaptieren und damit eine kosteneffiziente Integration dieser Systeme in neue Anwendungsfelder erreichbar wird.

Bestehende Implementierungen der Spezifikation sind, anders als im neueren *CORBA Component Model* (CCM), weitestgehend ausgereift und zum Teil kostenlos verfügbar.

3.1 Object Management Architecture

Die *Object Management Architecture* (OMA) der OMG definiert eine umfangreiche und flexible Architektur, welche für eine breite Menge von verteilten Systemen verwendbar ist. In zwei sich ergänzenden Modellen wird beschrieben, wie verteilte Objekte und deren Interaktionen plattformunabhängig spezifiziert werden können. Das *Object Model* definiert, wie Schnittstellen von verteilten Objekten in heterogenen Umgebungen beschrieben werden und das *Reference Model* charakterisiert die Interaktionen zwischen diesen Objekten.

Das Objektmodell definiert Objekte als eine gekapselte Einheit mit dauerhafter und eindeutiger Identität, dessen Dienste nur durch definierte Schnittstellen erreichbar sind [11]. Klienten nutzen diese Dienste, indem sie *requests* zu einem Objekt senden. Die Implementierungsdetails des Objektes und dessen Lokalität bleiben dem Klienten verborgen (*Location Transparency*).

Im Referenzmodell werden Schnittstellenkategorien definiert, welche ein allgemeines Gruppierungsschemata für Objektschnittstellen bilden. Es besteht aus den folgenden Entitäten (s. Abbildung 3.1):

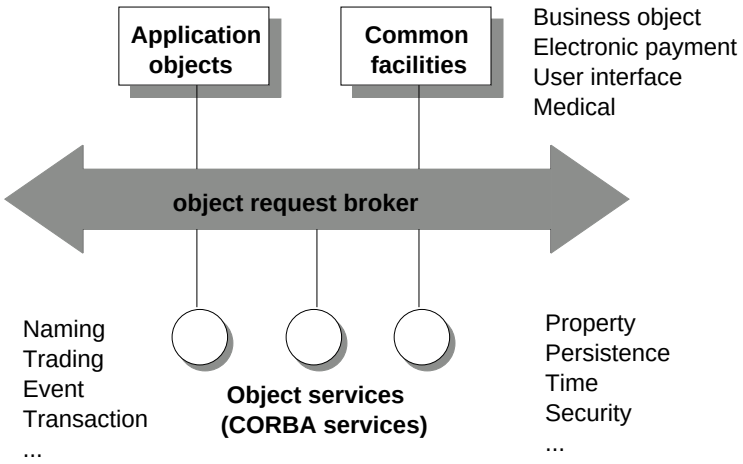


Abbildung 3.1: "Object Management Architecture" der OMG

Object Request Broker (ORB) ermöglichen den transparenten Zugriff auf entfernte Objekte und den Empfang von Rückgabewerten. Sie bilden die Grundlage zur Erstellung von Anwendungen, welche aus verteilten Objekten bestehen und ermöglichen Interoperabilität zwischen Applikationen unterschiedicher Programmiersprachen, Ausführungsumgebungen und Plattformen. Alle ORB's zusammen bilden eine Busstruktur, in die diese Objekte einfach "hineingehängt" werden können.

Object Services sind eine Sammlung von Diensten (Schnittstellen und Objekten) welche Basisfunktionalitäten für die Entwicklung und Nutzung von Objekten bereitstellen. So sind z.B. der *NameService* (white pages) oder der *TradingService* (yellow pages) Standarddienste, welche fast immer benötigt werden, um Objekte zu finden. Diese Dienste sind für verteilte Applikationen von großer Wichtigkeit und daher unabhängig von konkreten Anwendungsdomänen. Z.B. stellt der *Event Service* einen allgemeinen Mechanismus zum Propagieren von Nachrichten bereit, welcher von einer Anwendung mit domänenspezifischen Nachrichtentypen genutzt werden kann. Zuverlässigkeit und Sicherheit haben in verteilten Systemen eine besonders hohe Bedeutung. Applikationen können sich dabei durch die CORBA–Dienste *TransactionService* und *SecurityService* unterstützen lassen.

Alle Dienste zu erläutern würde den Rahmen dieses Dokumentes sprengen. Für eine exzellente Übersicht der CORBA–Dienste, deren Funktion und Anwendung, sei auf [21] verwiesen.

Common Facilities bezeichnen eine Sammlung von Spezifikationen, Standards und Diensten, um Interoperabilität verschiedener Anwendungen zu erreichen. Facilities sind keine elementaren Basisdienste sondern eher high–level Konstrukte mit hoher Relevanz für bestimmte Domänen. Unterschieden wird dabei in *horizontal facilities* und *vertical facilities*. Erstere bezeichnen domänenübergreifende Standards und Dienste wie z.B. *Internationalization and Time* oder *Workflow Management*. In der zweiten Kategorie finden sich Standards, wie sie in speziellen Domänen benötigt werden, z.B. "Air Traffic Control" (Domäne Transportwesen) oder "Biomolecular Sequence Analysis" (Domäne Humanmedizinforschung) [1]

Application Objects stellen die Implementierung einer (domänenspezifischen) Anwendung. Deren Schnittstellen werden nicht von der OMG standardisiert und obliegen den Softwareentwicklern bzw. Herstellern. Anwendungsobjekte befinden sich auf der höchsten Ebene des Referenzmodells.

3.2 Common Objects Request Broker Architecture

In der *Object Management Architecture* wird ein plattformunabhängiges Objektmodell für verteilte Anwendungen definiert, welches durch die CORBA-Spezifikation konkretisiert wird. Die folgende Abschnitte führen durch die CORBA Welt.

3.2.1 CORBA-Schnittstellenbeschreibung

Die ausschließliche Kommunikation über Schnittstellen ist ein fundamentales Konzept von Softwarekomponenten. Mit der *Interface Definition Language* (IDL) der OMG steht dafür ein mächtiges und vielseitiges Notationswerkzeug bereit. Softwarearchitekten können dadurch Schnittstellen von Komponenten (in OMG Terminologie: Objekte) elegant in einer C++ ähnlichen Notation beschreiben. Sie ist notwendig, um die für CORBA geforderte Programmiersprachen- und Plattformunabhängigkeit zu erlangen. Dies ermöglicht es, Anwendungen in unterschiedlichen Programmiersprachen zu realisieren, bei gleichzeitiger Wahrung von Interoperabilität zwischen den beteiligten Objekten. IDL kann somit als eine "sprachübergreifende" Sprache aufgefasst werden. Die Sprachunabhängigkeit der IDL ist von entscheidender Bedeutung für CORBA in heterogenen Systemen und zur Integration von getrennt voneinander entwickelten Anwendungen.

IDL weist keinerlei prozedurale Strukturen oder Variablen auf und kann somit nicht zum Entwickeln von lauffähigen Programmen verwendet werden. Sie dient ausschließlich zur Vereinbarung von Schnittstellen und Datentypen, welche für die Kommunikation mit CORBA-Objekten notwendig sind. Die IDL trennt die

[1] Eine vollständige Liste der CORBA–Facilities findet sich unter:
http://www.omg.org/technology/documents/domain_spec_catalog.htm

Schnittstelle eines Objektes von dessen Implementierung. Die Hauptelemente der CORBA-IDL sind:

Module definieren einen Namensraum zur Gruppierung von Schnittstellen. Es wird das Schlüsselwort `module` verwendet. Module haben einen eindeutigen Namen, der aus einem oder mehreren Bezeichnern besteht. In Zeichenkettenform werden die einzelnen Bezeichner durch `::` voneinander getrennt.

Schnittstellen definieren eine Menge von Methoden (in OMG-Terminologie: Operationen), welche von Klienten aufgerufen werden können. Innerhalb einer *Schnittstellendeklaration* können, wie innerhalb von Modulen auch, Ausnahmen, Datentypen und Attribute vereinbart werden. Attribute sind eine spezielle Form von Operationen, denn es wird im Übersetzungsvorgang automatisch eine Implementierung der `get-` und `set`-Funktionen erzeugt. Attribute können auch `readonly` deklariert werden, woraufhin nur für die `get`-Funktion eine Implementierung erzeugt wird. CORBA-IDL erlaubt *Mehrfachvererbung* von Schnittstellen, allerdings weder die Redefinition (*overriding*) noch das Überladen (*overloading*) von Operationen oder Attributen. Ein Operationsname ist dadurch eindeutig.

Operationen sind die Dienste, welche Klienten aufrufen können. IDL definiert den Namen der Operation, deren Parameter und Rückgabewert. Parameter bestehen aus einem Typ, einem Namen und zusätzlich einem Modus `in`, `out` und `inout`, welcher die Richtung des Datenflusses angibt. Da es in verteilten Systemen, abhängig von der zugrundeliegenden Netzwerkinfrastruktur, zu starken Laufzeitverzögerungen kommen kann, ist es möglich, für jede Operation einen Aufrufstil zu deklarieren. Operationen können synchron, d.h blockierend, aufgerufen werden (voreingestellt), `deferred` (nicht-blockierend, asynchron) mit explizitem Erfragen des Rückgabewertes (polling) oder `oneway`, wobei kein Rückgabewerte erwartet wird. Für jede Operation kann zusätzlich eine Menge von *Ausnahmen* festgelegt werden.

Datentypen werden in IDL in zwei Kategorien eingeteilt: einfache und konstruierte Typen. Zu den einfachen Typen gehören: `short, long, unsigned short, unsigned long, float, double, char, boolean, octet, string, enum, any`. Der Typ `any` nimmt dabei eine Sonderstellung ein. Er kann jeden IDL-Datentyp repräsentieren. Damit ist es möglich, zur Laufzeit beliebige Typen zu erzeugen und innerhalb eines `any` mit Werten zu belegen. Die konstruierten Typen umfassen: `struct, array, union, sequence`. Durch `typedef`-Ausdrücke können beliebig eigene Typen definiert werden. Jeder Datentyp wird mit geeigneten Sprachbindungen auf einen nativen Datentyp abgebildet.

Von der Schnittstellenspezifikation zur IDL-Beschreibung

Wird die zu beschreibende Schnittstelle als Schnittstelle einer Komponente aufgefasst und liegt hierfür ein *'Interface Information Model'* [8](IIM) vor, läßt sich eine IDL-Definition i.d.R. direkt daraus ableiten.

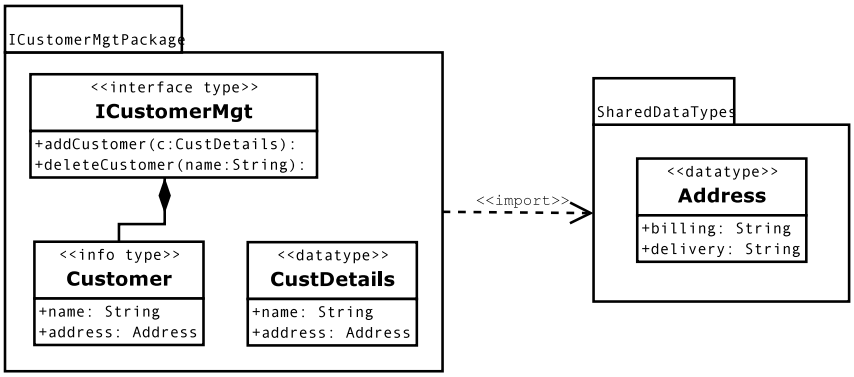


Abbildung 3.2:
'Interface Specification Package' einer einfachen Kundenverwaltung

Das 'Interface Information Model' definiert bereits alle für eine Schnittstelle notwendigen Informationen und Datentypen (vgl. 'Interface Specification Package' in Abbildung 3.2). Wichtig ist es, die Eigentumsbeziehungen der Schnittstelle zu den mit dieser Schnittstelle definierten Typen zu beachten. Diese werden nun in einzelne IDL-Ausdrücke umgesetzt. Das gruppierende Paket kann als Namensraum (`module`) definiert werden und die Schnittstellen des IIM werden zu IDL-Schnittstellen (`interface`). Definierte Datentypen werden zu IDL-Typen (einfach oder komplex) innerhalb des Namensraumes oder der Schnittstelle und importierte Pakete zu IDL-Ausdrücken der Form `#include "XXX.idl"`.

Das Beispiel in Listing 3.1 zeigt die Schnittstellendefinition einer sehr einfachen Kundenverwaltung.

Innerhalb des Namensraumes `ICustomerMgtPackage` (Z. 3) wird der konstruierter Datentyp `CustDetails` vereinbart (Z. 6–11), wobei auf vorgefertigte Typen (z.B. firmenweite Standards) zurückgegriffen wird (Z. 10) bzw. diese neu benannt werden (*declared subtyping*[23, 6.5], Z. 5). Für den Typ `CustDetails` wird zusätzlich ein Sequenztyp vereinbart (Z. 12).

Nachdem alle benötigten Typen importiert oder deklariert wurden, folgt die Definition der Schnittstelle `ICustomerMgt` (Z. 14). Die Operationen `addCustomer` (Z. 16) erlaubt es, Kunden anzulegen. Deren Daten werden aus dem hineingereichten Typen `CustDetails` übernommen. Eine durch die Implementierung der Schnittstelle erzeugte, eindeutige Kundennummer wird zurückgeliefert. Die Operation `deleteCustomer` (Z. 18) erwartet diese Kundennummer, um den damit referenzierten Kunden aus der Kundenverwaltung zu löschen. Kundeninformationen können mit Hilfe der Operation `findCustomer` (Z. 17) erfragt werden. Als Eingabe dient eine partiell gefüllte Struktur `CustDetails`, deren Werte mit denen der Kundenverwaltung verglichen werden.
Zurückgeliefert wird eine Liste von "Treffern."

```
1  #include <sharedTypes.idl>
2
3  module ICustomerMgtPackage
4  {
5      typedef sharedTypes::ID CustomerID;
6      struct CustDetails
7      {
8          CustomerID id;
9          string name;
10         sharedTypes::Address address;
11     };
12     typedef sequence<CustDetails> CustDetailsSeq;
13
14     interface ICustomerMgt
15     {
16         CustomerID addCustomer( in CustDetails c );
17         CustDetailsSeq findCustomer( in CustDetails c);
18         void deleteCustomer( in CustomerID id );
19     };
20 };
```

Listing 3.1: IDL-Schnittstellendefinition einer sehr einfachen Kundenverwaltung

In einer realistischen Schnittstellenbeschreibung würden z.B. mit Sicherheit weitere Operationen zum Lesen und Verändern von Kunden definiert sowie Ausnahmen hinzugefügt werden.

3.2.2 CORBA-Schnittstellenverzeichnis

Einer der Basisdienste der CORBA-Architektur ist das *Schnittstellenverzeichnis* ('*Interface Repository*', IR). Es ist eine Laufzeitdatenbank mit Schnittstellenspezifikationen aller CORBA-Objekte. Seit CORBA Version 2.0 ist eine Föderation von Schnittstellenverzeichnissen möglich. Dies ermöglicht lokalen Administratoren, die Autonomie über ihre Schnittstellen zu wahren (*Lokale Datenherren*), ohne Einschränkung der globalen Verfügbarkeit.

Das *Schnittstellenverzeichnis* hält Metadaten in Form von IDL über Objekte bereit, welche der Selbstbeschreibung von Objekten dienen (*introspection*). Für jede Art von dynamisch erzeugten Methodenaufrufen ist die Selbstbeschreibung unabdingbar. Auch werden dadurch zur Laufzeit Typüberprüfungen von Objekten, Operationen und deren Signatur möglich. Diese Techniken bilden eine ideale Voraussetzung für die Realisierung von aspekt-orientierten CORBA-Erweiterungen, denn sie erlauben z.B. typsicheres *Weben* (s.h. K. 20, S. 60) von Aspekt–Implementierungen zur Laufzeit des Systems.

Die CORBA-Spezifikation legt fest, wie die Informationen im Verzeichnis organisiert und gelesen werden. Es ist ein Satz von Klassen spezifiziert, dessen Instanzen die Informationen repräsentieren, die im Verzeichnis abgelegt sind. Veröffentlicht

werden IDL-Schnittstellen entweder durch einen IDL-Übersetzer oder durch die Schreiboperationen des Schnittstellenverzeichnisses. Jeder Schnittstelle sowie jedem definierten Datentypen wird dabei eine (global) eindeutige *RepositoryId* zugewiesen, welche es ermöglicht, diese eindeutig zu referenzieren. Die Informationen sind dann jederzeit abrufbar, aber auch zur Laufzeit modifizierbar. Eine Änderung von Schnittstellen ist allerdings nur in Spezialfällen sinnvoll, z.B., wenn Objekt–Implementierungen durch neue Versionen ausgetauscht werden, deren Schnittstellen modifiziert bzw. erweitert wurden.

Informationen aus dem Schnittstellenverzeichnises können auf unterschiedlichen Wegen erfragt werden:

1. Durch einen Aufruf der ORB-Methode `Object::get_interface_def`. Diese Operation kann auf jeder gültigen Objekt-Referenz angewandt werden und liefert das `InterfaceDef`-Objekt der am weitest spezialisierten Schnittstelle zurück. Die Schnittstelle ist damit vollständig beschrieben.
2. Ist die *RepositoryId* der Schnittstelle bekannt, so kann durch die Methode `Repository::lookup_id` das `InterfaceDef`-Objekt erreicht werden.
3. Mit Hilfe einer Folge von Namen kann durch den Namensraum (-baum) des Verzeichnisses navigiert werden. Hierzu steht die Operation `Repository::lookup` zur Verfügung.

3.2.3 Transparenzeigenschaften

Transparenzeigenschaften sind wesentlich für jede Middleware-Technologie. Aus der Sicht der Anwender, entscheiden Transparenzeigenschaften darüber, wie das System gesehen wird, bzw., um genauer zu werden, was sie nicht sehen. Transparenzen ermöglichen eine einheitliche Sicht auf ein System. Das *Reference Model Open Distributed Processing* (RM–ODP) [7] spielt dabei eine Vorreiterrolle. Darin werden viele, für verteilte Systeme relevanten, Transparenzeigenschaften aufgelistet und in ihrem Wesen definiert:

- access transparency
- failure transparency
- location transparency
- migration transparency
- persistence transparency
- relocation transparency
- replication transparency

- transaction transaction

Herrausgegriffen werden hier nur die nach [6] bezeichneten *primären Transparenzeigenschaften*, welche, wo möglich, die Verteiltheit des Systems verbergen sollen. Dazu zählen:

> **Def.: Ortstransparenz** [6]
> Der Zugriff auf Ressourcen ist ohne Kenntnis des Ortes, an dem sich die Ressourcen im System befinden (lokal/entfernt), durchführbar.

Diese beinhaltet die folgenden Eigenschaften:

> **Def.: Zugriffstransparenz** [6]
> Der Zugriff auf lokale und entfernte Ressourcen erfolgt mit der gleichen Operation.

> **Def.: Namenstransparenz** [6]
> Der Name einer Ressource ist auf allen Konten gleich.

Unter den sekundären Transparenzeigenschaften findet sich (nach [6]) die Migrationstransparenz:

> **Def.: Migrationstransparenz** [6]
> Prozeßverlagerung und Ortswechsel von Ressourcen bleiben dem Benutzer verborgen.

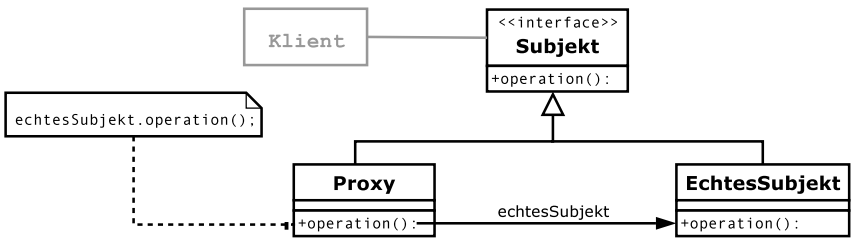


Abbildung 3.3: Stellvertreter-Muster (Proxy) [2]

Zugriffstransparenz und Migrationstransparenz werden in verteilten Systemen häufig durch *Stellvertreter*-Objekte erreicht. Das Stellvertretermuster (*Proxy*) [2, Proxy] besteht aus drei Entitäten (s. Abbildung 3.3):

Ein Stellvertreter (Proxy) verwaltet eine Referenz, welche den Zugriff auf das eigentliche Subjekt ermöglicht. Das `Subjekt` definiert die gemeinsame Schnittstelle,

sodass ein `Proxy` überall dort benutzt werden kann, wo ein `EchtesSubjekt` erwartet wird. `EchtesSubjekt` definiert das Objekt, das durch das `Proxy` repräsentiert wird. `Proxy` und `EchtesSubjekt` stehen in einer N:1 Beziehung, d.h. viele Proxies können auf ein `EchtesSubjekt` verweisen.

Das Proxy-Muster ermöglicht die Zugriffstransparenz durch ausschliessliche Verwendung von Schnittstellen (`Subjekt`) auf Klienten-Seite, dessen realisierende Klasse ein `Proxy` ist. Für lokale Zugriffe kann die Referenz `echtesSubjekt` ein einfacher Zeiger sein, für entfernte Zugriffe z.B. eine Netzwerkadresse mit Endpunktangabe. Der klientenseitige Zugriff auf Ressourcen erfolgt nun in jedem Fall über die gleiche Operation.

Zeigt ein Proxy auf ein entferntes Objekt, so bedarf es spezieller Mechanismen zur Übertragung der Argumente und des Rückgabewertes einer Operation. Den Vorgang des Ver– und Entpackens von zu versendenen bzw. empfangenen Daten wird mit *marshalling* bzw. *unmarshalling* bezeichnet. Proxies übernehmen häufig diese Funktionalität. Automatisch generierter Programmtext erspart dabei ein mühseliges "per Hand" schreiben.

Migrationstransparenz kann durch Neuzuweisung der Referenz `echtesSubjekt` erlangt werden. Prozeßverlagerung und Ortswechsel von Ressourcen bleiben dem Benutzer dadurch verborgen, dessen `Subjekt` ist weiterhin gültig.

3.2.4 Elemente der Architektur

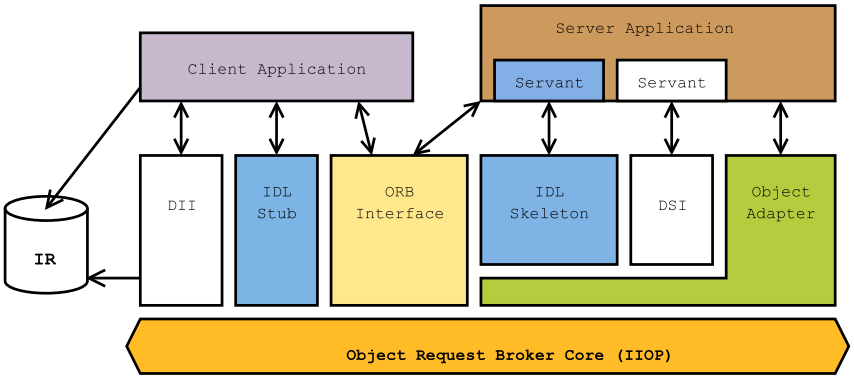


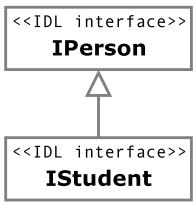
Abbildung 3.4: Struktur eines CORBA-2.0-ORB

Abbildung 3.4 zeigt den schematischen Aufbau der *Common Object Request Broker Architecture*. Die einzelnen Elemente der Architektur werden nun vorgestellt.

In CORBA findet sich das Proxy-Muster in Form von *Stubs* und *Skeletons* wieder. Der Stub (engl. Stümmel) übernimmt dabei die Rolle des `Proxy`, das Skeleton (engl. Gerippe) die Rolle des `EchtesSubjekt`. Eine IDL-Schnittstelle wird durch das `Subjekt` repräsensiert.

Stubs dienen Klienten als *Objektreferenzen* auf CORBA-Objekte und für den Aufruf von Operationen. Die Typisierung eines Stub ist statisch, d.h., sie liegt zum Zeitpunkt des Übersetzens fest und folgt der IDL-Schnittstellenbeschreibung.

Skeletons sind ein server-seitiges Implementierungsgerippe, dessen fehlende Domänenlogik durch *Vererbung* oder *Delegation* (*TIE-Approach*) hinzugefügt wird.



Das in Abbildung 3.5 gezeigte Klassendiagramm weist die Vererbungs– und Implementierungsbeziehugen der nebenstehenden IDL–Schnittstellen aus, wie sie für die Sprache Java definiert sind. Zugriffe eines Klienten auf ein CORBA–Objekt erfolgen immer über (programmiersprachenspezifische) Schnittstellen. Diese sind in der Abbildung 3.5 die Java–Interfaces `IPerson` und `IStudent`. Die Stub–Klassen `_IPersonStub` und `_IStudentStub` implementieren nun diese Interfaces und ermöglichen den Zugriff auf die entsprechenden Objekte.

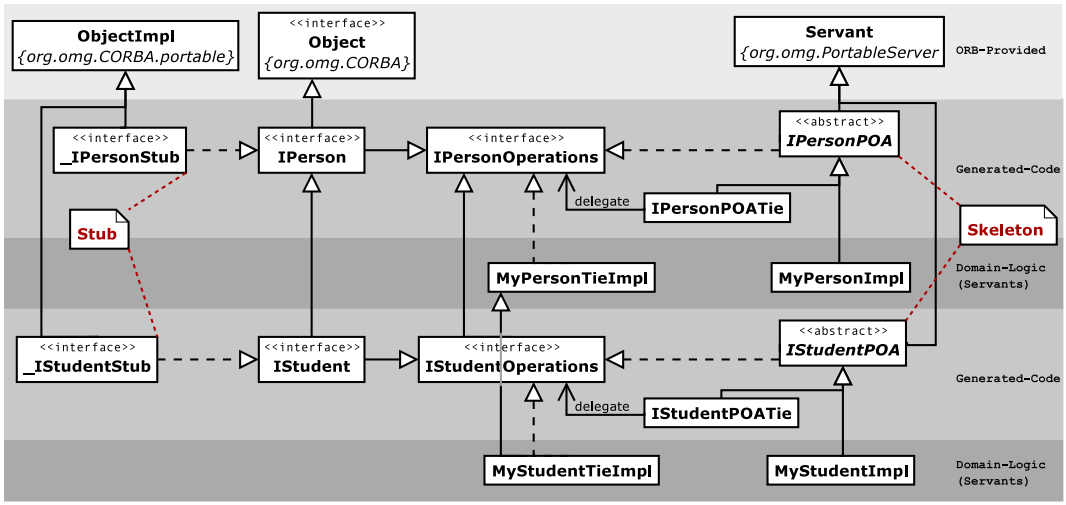


Abbildung 3.5: Beziehungen zwischen Stubs, Skeletons und Domänenlogik

Alle in der Skeleton–Klasse `IPersonPOA` als `abstract` deklarierten Methoden werden im Falle des Vererbungsansatzes mit konkreten Implementierungen überschrieben (`MyPersonImpl`). Die Klasse `IPersonPOATie` hingegen ist vollständig (jedoch ohne Domänenlogik) und kann instanziiert werden. Operationsaufrufe werden zur Erbringung der Funktionalität über die Referenz `delegate` an ein Objekt delegiert, welches die Schnittstelle `IPersonOperations` implementiert. Damit sichergestellt ist, das die Referenz `delegtate` gesetzt wird, erwartet jeder Konstruktor der Klasse `IPersonPOATie` eine solche.

Der Delegationsansatz ist dann interessant, wenn in Programmiersprachen mit nur einfacher Vererbung (*'single–inheritence'*) der Supertyp durch das Domänenmodell vorgegeben ist oder eine Implementierung weiter spezialisiert werden soll. Die

Klasse `MyPersonTieImpl` und deren Spezialiserung `MyStudentTieImpl` nutzen diesen Ansatz.

Die Implementierung der domänenspezifischen Funktionalität, unabhängig vom verwendeten Ansatz, wird mit '*servant*' bezeichnet. Die Programmierung von 'servants' erfolgt immer in der vom ORB unterstützten Zielsprache.

Die oben gezeigten Stub– und Skeleton–Klassen werden durch einen Übersetzungsvorgang aus einer IDL Schnittstellenbeschreibung generiert. Ein '*IDL-Compiler*' erzeugt dabei den für die Zielsprache spezifischen Quelltext. Eine von der OMG standardisierte Zuordnungen von Datentypen der IDL-Beschreibung zu denen der Zielprogrammiersprache, wird in Auszügen für Datentypen und Konstrukte und deren entsprechende Repräsentation in *Java* und *C++*, in Tabelle 3.1, gezeigt.

IDL	Java	C++
long	int	corba/Long
float	float	corba/Float
string	String	corba/String_var
any	CORBA.Any	corba/Any
struct	class	struct
enum	class	enum
sequence	[]	class
interface	interface	abstract class
include	import	include
module	package	namespace

Tabelle 3.1:
Übertragung von IDL-Typen und -Konstrukten zu Java bzw. C++ [16][13]

Zusätzlich zu den Stubs und Skeletons werden alle neu definierten Datentypen sowie *Helper*– und *Holder*–Klassen durch den IDL-Compiler generiert. Helper–Klassen ermöglichen das sichere Überführen von Objekttypen (*casting*). Dies wird durch die Operation `narrow` der entsprechenden Klasse bereitgestellt und beinhaltet i.d.R. eine (entfernte) Laufzeit-Typvalidierung. Das Ver- und Entpacken von Daten, welche druch ein Netzwerk übertragen werden (*marshalling*, *unmarshalling*), wird ebenfals von Helper-Klassen übernommen. Holder–Klassen dienen dazu, *out* und *inout*-Parameter einer Operation in Sprachen wie z.B. Java zu realisieren.

Die einzelnen Schritte, von der IDL-Beschreibung zur fertigen Client-Server-Anwendung, werden in Abbildung 3.6 noch einmal schematisch dargestellt.

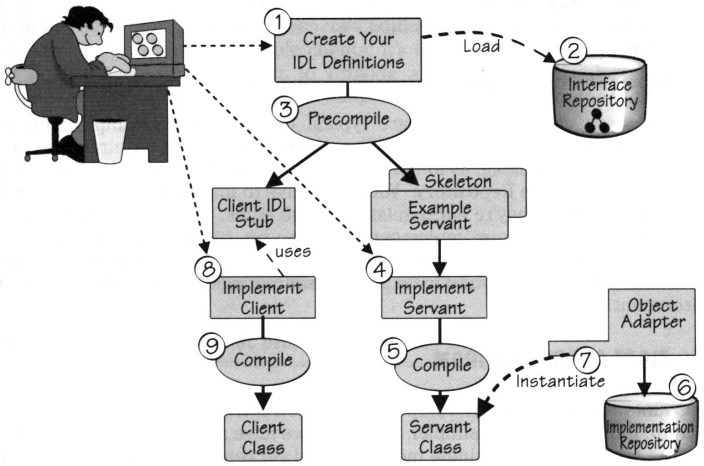


Abbildung 3.6: Enwicklungsprozesss einer CORBA-Applikation

Dynamische Operationsaufrufe und dynamische Servant–Implementierungen

Stubs und Skeletons ermöglichen eine statische Typprüfung des Programmtextes und bieten einen komfortablen Zugang zu entfernten Operationsaufrufen. In speziellen Anwendungen steht jedoch u.U. der Typ des aufzurufenden Objektes nicht zum Zeitpunkt des Übersetzens fest und muß zur Laufzeit bestimmt werden. Anders herum sollte es möglich sein, die Menge der Schnittstellen, die eine Servant–Implementierung bedient, zur Laufzeit zu konfigurieren. Eine bekanntes Beispiel, welches hochgradig dynamisch typisierte Implementierungen benötigt und beliebige Objekte aufrufen muß, ist eine Brücke zwischen zwei verschiedenen Middleware–Architekturen (z.B. CORBA und DCOM). CORBA bietet daher zwei Techniken an, um soch ein dynamisches Verhalten zu ermöglichen.

Auf Klienten–Seite steht das *Dynamic Invocation Interface* (DII) zur Verfügung. Das DII ermöglicht es, Operationen an einem Objekt aufzurufen, ohne daß dafür ein statischer Stub vorhanden sein muß. Für jedes CORBA–Objekt kann durch die Operation `Object.create_request(opName)` ein `ClientRequest`–Objekt erzeugt werden. Fordern die Operation Parameter, so müssen diese dem Aufrufobjekt in Form einer 'name–value–list' übergeben werden. Mit einem Aufruf '`invoke`' an dem `ClientRequest`–Objekt wird der Aufruf initiiert.

Ein Servant, der beliebige Operationsaufrufe entgegenehmen soll, kann mit Hilfe des *Dynamic Skeleton Interface* (DSI) realisiert werden. Ein solcher Servant ist eine Spezialisierung der Klasse *PortableServer.DynamicImplementation*. Darin ist

die abstrakte Methode `invoke` definiert, an die eingehende Aufrufe weitergeleitet werden; es erfolgt also ein invertierter Kontrollfluß ('Don't call us, we call you'). Der Name der aufgerufenen Operation und alle Parameter sind in einem `ServerRequest`–Objekt verpackt. Welche Schnittstellen ein solcher Servant bedienen kann, legt er durch die Implementierung der Methode `_all_interfaces` fest. Diese hat als Rückgabewert alle durch den Servant bereitgestellen Schnittstellen.

Beide Techniken (DII und DSI) zusammen ermöglichen es, vollständig zur Laufzeit typisierte Anwendungen zu entwickeln. Dabei kann auf das CORBA–Schnittstellenverzeichnis (s. Kap. 3.2.2, S. 24) zurückgegriffen werden, um Typinformationen von Objekten zu erlangen.

Objekt–Adapter und Objekt–Referenzen

Objekt-Adapter sind das Verbindungsglied zwischen einem 'ORB-Core' und dessen Servants. Sie folgen dem *Adapter*-Muster [2], um eingehende Aufrufe an Operationen eines '*servant*' weiterleiten zu können. Objektadapter mit sehr ähnlicher Funktionalität finden sich unter dem Begriff *Container* häufig in Komponentenumgebungen, wie, z.B., *Enterprise Java Beans* (EJB) [22], wieder. Die Hauptaufgaben eines Objektadapters sind:

- Objektreferenzen zu erzeugen, welche der Adressierung von Objekten dienen.
- Sicherzustellen, daß Servants der Ziel–Objekte leibhaftig sind ('incarnated').
- In den den ORB eingehende Aufrufe an einen Servant zu übergeben.

Seit der CORBA-Version 2.2 wird als Objektadapter ausschließlich der *Portable Object Adapter* (POA) verwendet. Dieser erlaubt es, zwischen einem statischen oder einem dynamischen Weiterleitungsmechanismus zu wählen. Im statischen Fall wird eine *object map* verwendet, welche die Zuordnung von *Zielobjekt* zu '*servant*' ermöglicht. Für den dynamischen Fall kann ein speziell entwickelter und im POA registrierter '*servant activator*' bzw. '*servant locator*' benutzt werden, welcher die passende Servant-Instanz ermittelt.

> **Def.: Policy** (Morris Sloman)
> "Policy is a rule that defines a choice in the behaviour of a system."

Das Verhalten des POA kann durch einen Satz von *Policies* beeinflußt werden. Je nach Bedarf können neue POA-Instanzen erzeugt werden und durch Policies, z.T. bekannt aus EJB [22], für Session-, Entity-, Zustandbehaftete- oder Zustandlose-Objekte konfiguriert werden. Das Verhalten für Nebenläufigkeit ist ebenso einstellbar. Auch kann durch Policies definiert werden, ob Objektreferenzen dauerhaft gül-

tig sind (*Persistent Lifespan Policy*). Das bedeutet, daß einmal veröffentlichte Referenzen weiter verwendet werden können, obwohl die Server–Anwendung, dessen Objektreferenzen betroffen sind, neu gestart wurde.

Objektreferenzen

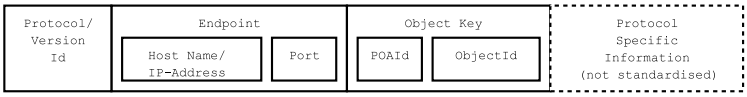
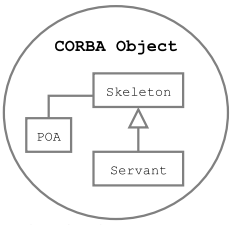


Abbildung 3.7: Aufbau einer CORBA-Objektreferenz

Anders als in statisch verbundenen Anwendungen reicht eine einfache (Haupt-) Speicheradresse, zur Referenzierung von Objekten in verteilten Systemen, nicht aus. Ein CORBA-ORB benötigt für den Aufbau einer Verbindung zu einem entfernten Objekt zusätzliche Informationen. Abbildung 3.7 zeigt die wichtigsten Elemente einer CORBA-Objektreferenz. Dies sind das zur Kommunikation verwendetet Netzwerkprotokol, ein Endpunkt, d.h. der Name des entfernten Rechners und eine Portnummer, an welcher der Server-ORB lauscht, sowie ein '*object key*', mit dessen Hilfe der Servant bestimmt wird. Optional können noch protokoll-spezifische Informationen hinzugefügt werden.

Die in Kapitel 3.2.3 (S. 26) geforderte primäre Transparenz des Ortes bedingt die Namenstransparenz, welche durch CORBA–Objektreferenzen erreicht wird. Diese sind für Klienten völlig transparent, können beliebig kopiert und verteilt werden. Zur einfacheren Handhabung wurde durch die OMG das *Interoperable Object Reference*-Format (IOR) spezifiziert. Oben genannte Informationen werden dabei in einer hexadezimalen Zeichenkette kodiert. Für die Umwandlung einer Objektreferenze in eine IOR und vice versa bietet der ORB zwei Operationen an: `object_to_string` und `string_to_object`. Diese Techniken stellen sicher, daß der Name einer Ressource auf allen Knoten gleich ist.

Was genau wird eigentlich durch den Ausdruck *CORBA-Object* bezeichnet ? Es gibt nicht *das* CORBA-Objekt. Es existiert eine Schnittstelle mit gleichem Namen. Aber wo ist das Objekt?



Mit *CORBA-Object* wird landläufig ein Konglomarat aus mehreren Entitäten bezeichnet. Die kleine Abbildung rechts soll dieses "virtuelle" Objekt verdeutlichen. Die Außensicht eines Klienten ist die der Schnittstelle `corba/Object`, bzw. eines Subtypen davon. Schaut man allerdings in den "Kreis" hinein, finden sich die drei bekannten Entitäten POA, Skeleton und Servant. Jeder von ihnen trägt seinen Teil zu einem CORBA-Objekt bei.

3.2.5 Zusammenspiel der Elemente

Um die Interoperabilität verschiedener ORB Implementierungen zu gewährleisten, müßen alle dieselbe "Sprache" sprechen. Dazu wurde von der OMG zuerst das abstrakte *General Inter ORB Protocol* (GIOP) spezifiziert. Abstrakt bedeutet dabei, daß es unabhängig von der darunterliegenden Transportschicht ist, solange diese einen verbindungsorientierten Charakter aufweist. Die OMG hat gleich noch eine Umsetzung des GIOP auf das meist verwendete verbindungs-orientierte Protokoll mitspezifiziert: Das *Internet Inter ORB Protocol* (IIOP) beschreibt, wie eine Implementierung des GIOP für TCP/IP basierte Netzwerke auszusehen hat. Jeder ORB, der von sich behauptet, CORBA 2.0 kompatibel zu sein, muß diese beiden Prokolle unterstützen.

Eine Aufrufsequenz von einem Klienten, durch den ORB hindurch, zu einem Servant gestaltet sich nun folgendermaßen:

1. Klienten haben grundsätzlich zwei Möglichkeiten, Operationen an einem Server-Objekt aufzurufen. Sie verwenden entweder einen statischen Stub, welcher in die Anwendung des Klienten hineingebunden ist oder das 'Dynamic Invocation Interface' (DII). In beiden Fällen wird der Aufruf an den, ebenfalls an die Anwendung gebundenen ORB, weiter gereicht.
2. Der klienten-seitige ORB überträgt den Aufruf an einen Server-ORB.
3. Der auf Seiten des Diensterbringers ('server') eingehende Aufruf wird durch den ORB analysiert und anhand des '*object key*' an den Objektadapter weitergeleitet, welcher das Zielobjekt erzeugt hat.
4. Mit Hilfe der '*Object-Id*' ermittelt der Objektadapter den Servant, welcher das Zielobjekt implementiert. Dieser kann entweder auf einem statischen Skeleton basieren oder mit Hilfe des 'Dynamic Skeleton Interface' (DSI) implementiert sein. Letzteres ermöglicht es einem 'servant', beliebige, vorher nicht in IDL definierte, Operationsaufrufe entgegen zu nehmen.
5. Nach Bearbeitung der Operation durch den 'servant' wird der Aufruf an den Klienten zurückgeleitet.

Request Interception

Der oben beschriebene Weg eines Aufrufes von einem Klienten zu einem Servant kann noch verfeinert werden. In der Version 2.4 wurde CORBA um die *Portable Interceptor*-Architektur erweitert. Dies ermöglicht es, *Request-Interceptor* in den ORB einzuhängen. Diese sind von fundamentaler Bedeutung für spezielle CORBA-Dienste. Der Transaktionsdienst z.B. muß sicherstellen, das mit jedem Aufruf eine Transaktions-Id propagiert wird. Daher benötigt er einen *hook* in den ORB, um sich an dem Aufrufprozess zu beteiligen.

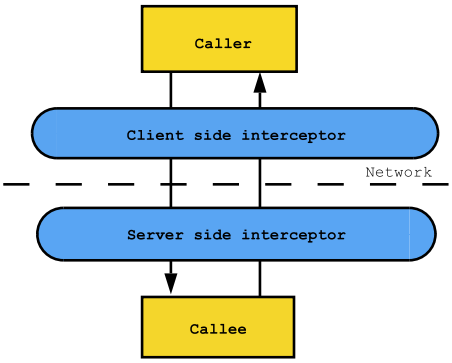
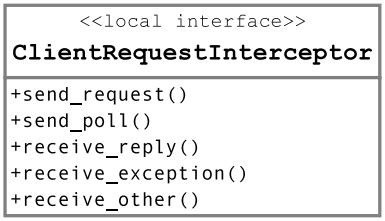


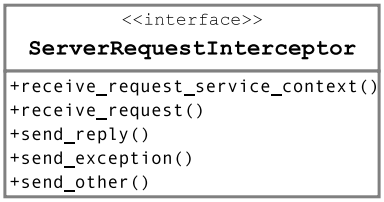
Abbildung 3.8:
Schematische Darstellung einer Aufruf- / Antwort- Sequenz durch zwei 'interceptor' hindurch.

Abbildung 3.8 zeigt die schematische Darstellung einer Aufruf / Antwort-Sequenz (request / reply sequence) durch zwei 'interceptor' hindurch. An den entscheidenen Punkten innerhalb dieser Sequenz befinden sich registrierte 'interceptor', welche die Aufruf / Antwort-Informationen einsehen und *Service-Kontexte* zwischen Klient und Server übermitteln können.

Auf Klienten–Seite dienen 'interceptor' dazu, ausgehende Operationsaufrufe abzufangen sowie eine Unterbrechung bei deren Rückkehr zu ermöglichen. Die CORBA 'Portable Interceptor'-Architektur sieht dafür fünf Operationen vor, die alle Arten von ausgehenden oder rückkehrenden Nachrichten empfangen und verarbeiten können.



Das Gegenstück bildet ein 'server request interceptor'. Sie werden an zwei Punkten bei eingehenden Nachrichten aufgerufen, was mit der zweischichtigen Architektur von ORB und Objektadapter zusammenhängt. Drei weitere Operationen sind vorgesehen, um auf Antwortnachrichten zu reagieren.



Zusätzlich kann in jeder Operation die Ausnahme 'ForwardRequest' geworfen werden, welche als Argument ein neues Ziel–Objekt erhält. Dadurch kann der Aufruf zu einem anderen Objekt (`effective_target`) umgeleitet werden. Vor allem für die Realisierung eines *Objekt-Adapter*[2, Adapter] (*Wrapper*) ist diese Technik sehr interessant. Es kann so sichergestellt werden, das alle Aufrufe den Adapter und nicht das zu adaptierende Objekt erreichen. Das Weben von Aspekten in ein System zur Laufzeit wird u.a. dadurch möglich (s. Kapitel 5).

Dienst-Kontexte sind kleine Informationspakete, die mit jeder Nachricht von ei-

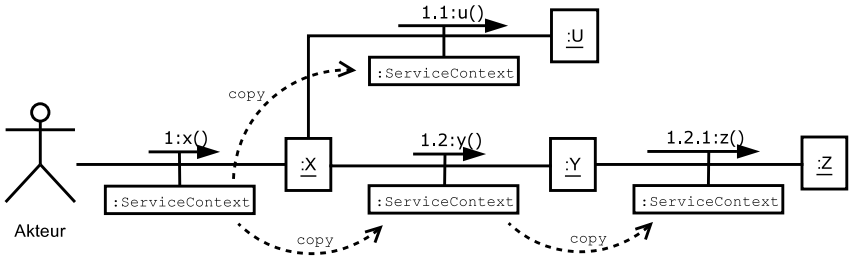


Abbildung 3.9:
Ein Service–Kontext propagiert an Aufrufe angehängt durch das System.

nem Klienten zu einem Server (CORBA–Objekt) mitreisen können (s. Abb. 3.9). Sie ermöglichen es, beliebige Daten aus dem Ausführungskontext des Klienten (client–thread) zu dem eines Diensterbringers (server–thread) zu übertragen. Client- und Server-Prozess stehen dabei sogenannte *Current-Objekte* zur Verfügung, um den einen Aufruf umgebenden Service-Kontext zu erreichen. Es liegt in der Verantwortung der Interceptor, diese Daten aus den Current-Objekten zu lesen und in Form eines Service-Kontext an den Aufruf anzuhängen, bzw. auf Seite des Diensterbringers, auszulesen und in ein Current-Objekt zu übertragen.

3.3 Zusammenfassung und Diskussion

Nachdem die Middleware CORBA in Aufbau und Funktionsweise erläutert wurde stellt sich die Frage, ob und in wie weit sie dazu geeignet ist, als Grundlage für Softwarekomponenten zu dienen.

Eine der wesentlichen Eigenschaft von Komponenten, Interaktionen und damit Abhängigkeiten auf Schnittstellen zu begrenzen, wird von CORBA-Objekten erfüllt. Dazu steht die OMG-IDL bereit, mit deren Hilfe Schnittstellen beschrieben werden, um daraus durch einen IDL-Compiler programmiersprachenabhängige Stubs und Skeletons zu generieren. Stubs dienen als Objektreferenz (typisiert auf eine Schnittstelle) und Skeletons als Gerüst zur Implentierung der Funktionalität. Durch die IDL werden alle Informationen erfaßt, welche zum Verpacken und Versenden von Daten bei der Kommunikation über Prozess-, CPU-, oder Maschinengrenzen hinaus notwendig sind. IIOP ermöglicht dabei den interoperablen Transport der Daten zwischen zwei ORBs (*physical pluggability*[23] und *syntactical pliggability*[9]). All dies ist notwendig, um die Komplexität von verteilten, interagierenden Systemen weitestgehend zu verbergen. Ausreichend ist es leider nicht, weil Softwareentwickler zwar dadurch Objekte bzw. Komponenten miteinander “verdrahten” (wiring) [23] können; darüber hinaus aber bietet ihnen diese Technik keine zusätzlichen Möglichkeiten.

Dies tritt insofern zu Tage, als daß CORBA “nur” ein Objektmodell definiert. Ein

explizietes Komponentenmodell ist in der CORBA Version 2.4 nicht eingebaut. Software-Entwickler müssen v.a. an zwei Stellen eigene, nicht standardisierte Wege gehen.

CORBA-Objekte besitzen nur eine Schnittstelle. Diese kann durch (multiple) Vererbung aus mehreren Schnittstellen zusammengesetzt sein, sodaß eine Schnittstelle auch ein Subtyp der erwarteten sein kann (Polymorphie). Bezeichner innerhalb einer Schnittstelle müssen einmalig sein, overloading oder overriding von Methoden ist ausgeschlossen. Komponentenerweiterungen durch Hinzufügen einer neuen oder erweiterten Schnittstelle bringen dadurch Namens-, Versions- und Implementierungsprobleme mit sich.

Schwerer wiegt, durch die Abwesenheit eines Komponentenmodells, der Umstand, daß CORBA-Objekte nur Schnittstellen *anbieten* können (*'offered interfaces'*) . Eine Ausdrucksmöglichkeit zur Definition von benötigten Schnittstellen ist nicht vorgesehen (*'required interfaces'*). Dies führt dazu, daß es keinen Standard gibt, wie Referenzen zu anderen Komponenten, gelangen. Jede Implementierung geht dabei ihren eigenen Weg. Umso wichtiger sind deshalb die CORBA Dienste. Sie ermöglichen das Entdecken von Objekten in allen erdenklichen Formen. Der *NameService* (white pages) und der *TradingService* (yellow pages) sind wohl die am meisten verwendeteten Dienste.

IDL Schnittstellenbeschreibungen sind leider auch ausschließlich auf die Definition von Datentypen, Operationen und deren Signaturen beschränkt. Ausdrucksmöglichkeiten zur formalen Spezifikation von Verträgen zwischen Klient und Schnittstelle, Schnittstelle und Implementierung, bzw., Schnittstellen und Schnittstellen, sind nicht vorhanden (Nutzungsverträge bzw. Realisationsverträge). Diese werden weitestgehend, formell oder informell, menschlichen Softwarearchitekten überlassen. Dabei bedarf es gerade einer explizieten Verbindung zwischen einer Schnittstelle und ihrer vertraglichen Spezifikation.

In COM / DCOM von Microsoft sind Schnittstellen nach deren Veröffentlichung deshalb unveränderlich (*'immutable interfaces'*). Die Idee ist, daß Schnittstellen immer einer Spezifikation zugrunde liegen; mit der zuvor notwendigen Voraussetzung einer unveränderlichen Spezifikation. D.h. die notwendige Vorraussetzung ist, das diese Spezifikation unveränderlich ist. COM's *'unique interface id'* kann als eine Art Verdingung zwischen Spezifikation und Schnittstelle angesehen werden.

CORBA hat mit der Version 2.0 eine *RepositoryId* eingeführt, welche zur eindeutigen Referenzierung von Schnittstellen benutzt werden kann. Diese wird i.d.R. automatisch durch einen IDL-Compiler aus dem vollqualifizierten Namen der Schnittstelle generiert. Änderungen innerhalb der Schnittstelle werden dadurch nicht berücksichtigt. Die Möglichkeit, durch *pragma*-Ausdrücke für jede Schnittstelle eine explizite 'RepositoryId' zu definieren (z.B. in Form einer GUID [2]), ist weitestgehend unbekannt und wird dadurch selten benutzt.

[2] Global Unique ID

In Bezug auf Evolution ist eine Versionierung der Schnittstellen und deren Implementierungen von entscheidener Bedeutung. Verwenden Klienten und Dienstanbieter verschiedene Versionen einer Schnittstelle, so müssen Lösungen zur Wahrung von Kompatibilität gefunden werden. Die OMG adressiert dieses Problem durch die Erarbeitung eines *Change Mangement Service*. Unterstützt werden soll *Version Tracking* sowie das Sicherstellen von Kompabilität in sich verändernden Systemen. Allerdings sind dazu noch keine nennenswerten Veröffentlichungen erschienen.

Offen bleiben in CORBA auch Fragen nach *Deployement* und *Assembly*. Die Spezifikation sieht keinen Standardbehälter für Komponenten vor. Auch Komponenten- bzw. Komponentenarchitekturdeskriptoren [8][15] wurden nicht definiert. Das Einbinden von CORBA-Komponenten unterschiedlicher Hersteller erfolgt immer nach proprietären Verfahren.

Mit der Version 3.0 wurden in CORBA einige der oben beschriebenen Probleme gelöst. Das *CORBA Component Model* (CCM) erweitert CORBA - Objekte durch *Ports*, welche in Form von *Facets* (angebotene Schnittstellen) oder *Receptacles* (benutzte Schnittstellen) vorkommen. Diese ermöglichen es, CORBA - Komponenten sehr einfach um Schnittstellen und Funktionalität zu erweitern, bzw. explizit das Erwarten einer Schnittstelle auszudrücken. Auch werden Behälter und Deskriptoren standardisiert. Damit wird CORBA's Attraktivität , wenn der Markt es schafft, vollständige Implementierungen zu liefern, noch deutlich steigen.

Damit ist das Problem der nicht–invasiven Komponentenadaption aber noch nicht gelöst. Bisher stellt sich CORBA als ein bedingt brauchbares Instrument dar, weist aber für den Bereich der Komponententechnologie Schwächen auf.

4 Object Teams für verteilte Systeme

In diesem Kaptitel soll die aspektorientierte Technik *Distributed Object Teams* (DOT) vorgestellt werden. Es ist eine Variante des Programmiersprachenmodells *Object Teams* [5] für veteilte Systeme auf Grundlage der CORBA-Technologie.

Das Konzept von 'Object Teams' wurde für verteilte Systeme mit Hilfe der CORBA–Technologie umgesetzt. CORBA wurde gewählt, da diese Technik der defacto standard in verteilten Industrieanwendungen ist und damit einen möglicherweise großen Interessentenkreis, z.B., im Bereich der Adaption von Altsystemen erreicht.

Außerdem hält diese Technik eine Vielzahl von Ansatzpunkten bereit, um in das Laufzeitverhalten des ORB eingreifen zu können. Letzteres ist durch die seit der Version 2.4 vorhandene *Portable Interceptor Architecture* (s. Kap. 3.2.5, S. 33) möglich. Hinzu kommt die hervorragende Introspektionsfähigkeit von CORBA-Objekten mit Hilfe des Schnittstellenverzeichnisses. Typinformationen von Objekten und die Möglichkeit, in den Nachrichtenfluss zwischen Objekten einzugreifen, sind damit zur Laufzeit verfügbar. Dies erschienen die idealen Voraussetzungen für eine Realisierung von Object Teams in einem verteilten CORBA–System zu sein.

Distributed Object Teams wird als Variante des Object Teams – Modell bezeichnet, da eine eins-zu-eins Umsetzung aufgrund der Verteiltheit des Systems und den speziellen Eigenschaften der CORBA nicht sinnvoll erscheint bzw. möglich ist. Das CORBA Objektmodell gibt dabei den Rahmen vor.

Mit DOT sollte ein Maximum an Kompabilität zur CORBA Version 2.4 gewahrt werden. Die DOT–Technologie ist dadurch unproblematisch in bestehende Systeme (*legacy systems*) integrierbar. Dies ist besonders für bestehende Systeme interessant, da gerade hier nicht-invasive Adaptionstechniken einen großen Gewinn bei der Erweiterung um neue Funktionalität bedeuten können, denn bestehende Systeme können in binärer Form vorliegen und mit einem Minimum an Wissen über die Struktur (genauer der Schnittstellen und Schnittstelleninformationspakete) adaptiert werden. Für DOT wurde daher versucht, eine Technik zu entwickeln, die eine späte oder sogar nachträgliche Integration in verteilte CORBA–Systeme ermöglicht.

Der zwingend erforderliche Entwicklungsprozess aus Schnittstellenbeschreibung in IDL und dessen Übersetzungsvorgang durch einen IDL–Compiler bietet weitere Ansatzpunkte. Es drängte sich zur Realisierung von Object Teams die Überlegung auf, die IDL um neue Schlüsselwörter zu erweitern und diese dann durch einen eigenen Compiler zu interpretieren, um daraus spezielle *Stubs* und *Skeletons*

zu generieren. Dadurch wären allerdings nur Operationsaufrufe adaptierbar, denen statische Stubs bzw. Skeletons zugrunde liegen. Auch sind solche generierten Stubs und Skeletons nicht mehr CORBA–kompatibel, und daher (z.B. für bestehende Systeme) inakzeptabel. Es sollte ein anderer Ansatz gefunden werden.

Für eine prototypische Realisierung der DOT–Konzepte wurde die Progammiersprache Java gewählt und als zugrunde liegender CORBA–ORB die Implementierung *OpenORB*[19] des Exolab.org . Das entstande DOT–Framework wird daher mit *DOT/J* bezeichnet.

In den folgenden Unterkapiteln werden am Beispiel eines Bonussytems die Konzepte von DOT und deren Modellierung vorgestellt, sowie die Umsetzung der Konzepte auf CORBA–IDL und, wo nötig, die sich daraus ergebenen Bedingungen zur Implementierung von Java–Klassen.

4.1 Aspekte

4.1.1 Von Objekten zu Komponenten

Der folgende Absatz motiviert die Idee von '*Object Teams*'. Eine ausführliche Beschreibung des Ansatzes findet sich in [5] bzw. [25](in deutscher Sprache). Die Übertragung der Konzepte von 'Object Teams' für verteilte Systeme (DOT) erfolgt ausführlich ab Kapitel 4.3 (S. 47).

Das objetorientierte Paradigma benutzt die "Klasse" als statischen Strukturierungsmechanismus zur Modellierung einer Domänenentität. Die von einer Anwendung zu erbringende Funktionalität (Systemfunktionalität) verteilt sich allerdings häufig auf verschiedene Klassen (*code scattering*). Andersherum müssen in OO-Implementierungen verschiedene Funktionalitäten oder Zuständigkeiten in ein und derselben Methode bearbeitet werden. In diesem Fall spricht man von '*tangling*'. Passen zwei oder mehr Zuständigkeiten nicht in die zu Grunde liegende Struktur (Klassen) der ersten Zuständigkeit, sondern schneiden statt dessen mehrere, werden diese als '*crosscutting concern*' [5] bezeichnet.

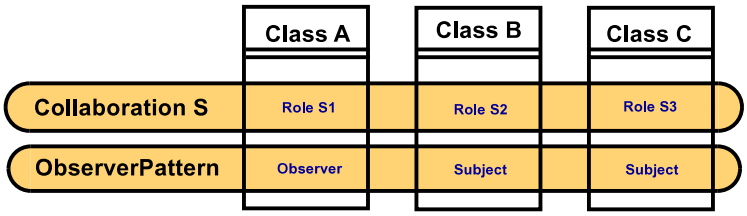


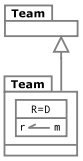
Abbildung 4.1: Orthogonalität von Klassen und Kollaborationen

Dieser Streuung von Zuständigkeiten versucht das aspektorientierte Paradigma entgegen zu treten. Ein Aspekt kapselt eine Zuständigkeit, die losgelöst ist von dem zugrundeliegenden Domänenmodell. Die Vermischung verschiedener Zuständigkeiten in ein und der selben Methode ('tangling code') kann so vermieden werden. Wird eine Funktionalität, verteilt über mehrern Klassen, erbracht, muß jede dieser Teilfunktionalitäten in einen einzelnen Aspekt verpackt werden. Daraus ergeben sich zusammenhängende, mit–einander interagierende Aspekte, so genannte Kollaborationen, die es erlauben, 'scattered code' zu kapseln. Abbildung 4.1 zeigt den Sachverhalt am Beispiel des *Observer Pattern* [2, ObserverPattern]. Mehrere Klassen (`A, B, C`), sind an mehreren Kollaborationen beteiligt (`S, ObserverPattern`), während eine Kollaboration (`ObserverPattern`) mehrere Klassen umspannt.

'Object Teams' führt als Strukturierungsmechanismus für kollaborierende Aspekte ein neues Modul, das '*Team*', ein. Ein Team kapselt eine Menge von Aspekten. Da jeder der beteiligten Aspekte eine spezielle Zuständigkeit innerhalb des Kontextes der Kollaboration übernimmt (z.B. `Observer`), wird der jeweilige Askpekte in 'Object Teams' Terminologie als *Rolle* bezeichnet. Das in

einer Rolle gekapselte Verhalten macht außerhalb des Teams wenig Sinn. Daher ist eine Rolle an das umschließende Team gebunden. Die Definition einer Rolle wird, dem OO–Paradigma folgend, mit einer "Klasse" beschrieben.

Um eine Wiederverwendung des in einem Team durch Rollen implementierten Verhaltens zu erlangen, können diese unvollständig (abstrakt) definiert werden. Für die Methode `update` eines `Observer`, z.B., ist erst zum Zeitpunkt des Bindens an eine Domänenentität definiert, welches Verhalten durch Aufruf der Methode ausgelöst werden soll. Ist eine Rolle abstrakt definiert, so ist auch ihr zugehöriges Team abstrakt. Eine solche Rolle kann zu einem späteren Zeitpunkt durch Vererbung oder durch Delegation an eine Domänenklasse konkretisiert werden.



Bindungen zwischen Kollaborationen und Klassen des Domänenmodells werden in Object Teams unter zu Hilfenahme eines speziellen Teams – eines Konnektors – ausgedrückt. Ein Konnektor ist eine Spezialisierung eines (Basis–) Teams. Alle im Basisteam definierten Rollen werden durch '*implicit inheritence*' in den Konnektor eingefügt und können Klassen der Domäne zugeordnet werden. Bindungen erfolgen auf der Ebene von Methoden, wobei zwei Aufrufstile in Object Teams definiert sind: *callin* bzw. *callout*. Die Definition erfolgt immer aus Sicht der Rolle. Für eine `callin`–Bindung wird eine Rollenoperation vor (`before`), nach (`after`) oder anstatt (`replace`) einer Operation des Domänenobjektes ausgeführt. Ein Aufruf erfolgt als in–die–Rolle–hinein. `Callout`–Bindungen hingegen erfolgen aus der Rolle hinaus zu einer Operation des Domänenobjektes. Dabei werden in der Rolle als `abstract` definierte Methoden (z.B., die Methode `update` eines `Observer`) an eine Implementierung der Domänenklasse gebunden. Alternativ dazu können abstrakte Rollenmethoden auch durch eine *in–place* Implementierung im Konnektor realisiert werden. Die so definierten Adaptionen können an der Konnektor–Instanz zur Laufzeit an– und ausgeschaltet werden.

Da Teams und deren Rollen i.d.R. eine vom Kontext des Domänenmodell unabhängige Funktionalität erbringen, liegt sicherlich keine Übereinstimmung der Methodennamen und deren Signatur vor. Daher kann in 'Object Teams' zu jeder Bindung definiert werden, wie die Signatur zweier Methode angepasst werden soll (z.B. durch Weglassen oder einfache Umrechnungen von Parametern).

Ein Team kann in einem weiteren Schritt wiederum verfeinert werden. Es ist dadurch möglich, einmal definierte Adaptionen zu einem späteren Zeitpunkt neuen Bedingungen anzupassen oder das System um weitere Aspekte zu bereichern.

4.1.2 Verhaltensadaption von Komponenten

Die Konzepte des aspektorientierten Paradigmas können nicht eins–zu–eins vom OO-Modell (Object Teams) auf komponentenbasierte Systeme übertragen werden.

Folgt man dem Ansatz, daß eine Komponente aus mehreren, miteinander inter-

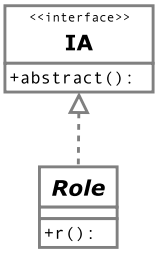
agierenden Klassen besteht und die äußere Grenze einer Komponente durch den Namensraum (oder in UML: Paket) beschrieben ist, so ist eine komponenteninterne Verhaltensadaption durch Rollen eines Object–Team sicherlich eingänglich. Dabei muß eine solche Komponente allerdings eine sogenannte *white box* sein, bei der die Implementierung der Komponente vollständig vorliegt und studiert werden kann.

Werden Komponenten jedoch als *black box* betrachtet, wobei im Idealfall nur die Schnittstellen und deren Spezifikationen einem Klienten bekannt sind [23], können Klassen nicht mehr als Ausgangsbasis zur Adaption dienen. D.h., die interne Implementierung einer Komponente ist nicht bekannt (und liegt im ungünstigsten Fall in Binärform vor). Ein Eingreifen in das komponenteninterne Verhalten ist also schwer möglich.

Jedoch sollten alle mit einer Komponente ausführbaren Interaktionen durch deren Schnittstellen (Nutzungvertrag) beschrieben sein. Die funktionale "Lücke" des Realisationsvertrages kann durch Adaptions gelöst werden; es bedarf dazu allerdings eines 'trigger' innerhalb des Nutzungsvertrages der Komponente. Eine Adaption von 'black box'–Komponenten muß daher daher auf der Ebene der Schnittstellen einer Komponente erfolgen.

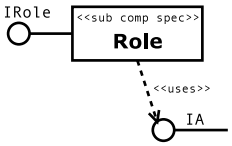
Daraus folgt, für den Schritt aus der objektorientierten Welt, hin zu komponentenbasierten Systemen und dessen Adaption durch aspektorientierte Techniken:

- Klassen werden zu Schnittstellen
- Kollaborationen manifestieren sich in eigenständigen (Team–) Komponenten
- Rollen werden zu Subkomponentenschnittstellen
- Alle abstrakten Methoden einer Rolle definieren eine eigenständige Schnittstelle



Der letzte Punkt adressiert ein Problem, wonach Schnittstellen keine abstrakten Operationen beinhalten können [1]. In der OOP werden abstrakte Methoden dazu verwandt, eine zu erbringende Funktionalität in einer Klasse zu definieren (z.B. `update`), um diese durch Vererbung mit einer Implementierung zu konkretisieren. Eine abstrakte Klasse erwartet also die Erfüllung / Implementierung einer bestimmten Schnittstelle durch einen Subtypen ('subtyping contract' [23]).

[1] Schnittstellen definieren ja gerade das Verhalten, welches von einer Implementierung erwartet werden kann, die diese Schnittstelle realisiert.



In der Welt der Komponenten ist die Benutzung einer Schnittstelle, ohne Kenntnis der konkreten Implementierung, ein fundamentales Konzept ('expected interface', «uses»). Daher soll dieses Konzept auf abstrakte Rollen übertragen werden. Alle "abstrakten" Methoden einer Rolle werden in einer eigenständigen Schnittstelle definiert. Die Funktionalität dieser erwarteten Schnittstelle kann durch 'forwarding' von einer (anderen) Komponente erbracht werden.

Die in Abbildung 4.1 (S. 41) gezeigte Orthogonalität von Klassen und Kollaborationen wird, in abgeänderter Form, auch für Komponentensysteme darstellbar. Abbildung 4.2 zeigt ein Beispiel mit drei Schnittstellen: `IA, IB, IC`. Diese sind mit mehreren Kollaborationen durchzogen (`Colaboration C`, `Bonus Collection`), wobei jede Schnittstelle (z.B. `IA`) eine bestimmte Rolle (z.B. `ICollector`) innerhalb des Kontextes einer Kollaboration (`Bonus Collection`) spielt. Dabei wird das Rollenverhalten durch eine Schnittstelle definiert.

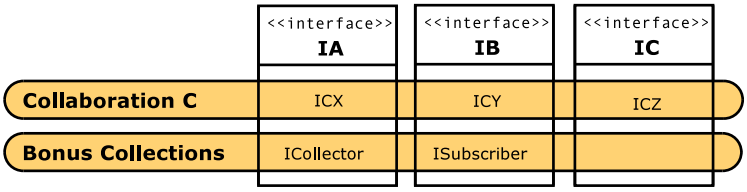
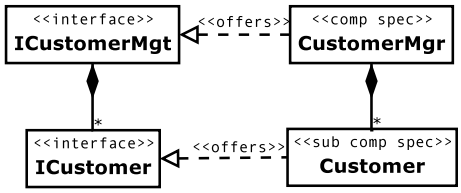


Abbildung 4.2: Kollaborierende Aspekte an Komponentenschnittstellen

Granularität möglicher Adaptionen

Eine Adaption auf der Ebene der Komponentenschnittstellen bietet vielleicht nicht die feine Granularität an Adaptionsmöglichkeiten, der es häufig bedarf. U.U. ist es zur Definition eines Aspektes notwendig, daß ein Teil des "Innenlebens" einer Komponente nach außen hin sichtbar ist. Wird ein Teil einer Komponentenimplementierung enthüllt, spricht man von einer *gray box*–Komponente [23]. Dabei kann die partielle Enthüllung der internen Funktionalität als Bestandteil der Komponentenspezifikation angesehen werden.

Ein bekanntes Beispiel ist die Belegung einer Operationsimplementierung mit *Constraints* [8]. Dabei wird, z.B., mit Hilfe eines Kollaborationsdiagrammes festgelegt, daß eine Implementierung der Operation `IX.x()` einer Komponenteninstanz des Types `X`, was immer sie auch tut, auf jeden Fall die Operation `IY.y()` einer (anderen) Komponente aufrufen muß.



Eine weitere Möglichkeit, den internen Aufbau einer Komponente zu beschreiben, bietet sich unter Verwendung der *Subkomponenten*–Modellierung . Cheeseman und Daniels folgen dabei in 'UML Components'[8] der Idee, wonach es u.U. angebracht sein kann, die hinter einer Schnittstelle stehenden *Informationstypen* nicht über primitive Datentypen als Eingangs– und Ausgangsparameter einer Komponentenschnittstelle zu bearbeiten. Statt dessen werden die Informationstypen als Subkomponenten mit eigenen Schnittstellen modelliert (s. nebenstehende Abb.), deren Referenzen die umgebene Komponente verlassen können [2]. Dieser Ansatz ist u.U. aus anderen Komponentenmodellen, wie EJB [22] oder CCM [15] bereits bekannt. In der EJB–Modellierung wird allerdings eine solche Komponente nicht expliziet durch den Stereotyp «`sub comp spec`» markiert. Der Umgang mit derart modellierten Komponenten ist natürlicher, ähnelt dem objektorientierten Modell und bietet v.a. mehr Ansatzpunkte zur Definition von Aspekten. Ein Teil der inneren Struktur einer Komponente ist dadurch nach außen sichtbar – und damit adaptierbar [3].

4.2 Domänenbeispiel

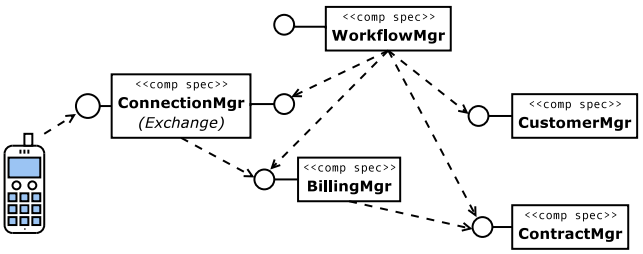


Abbildung 4.3: Komponentenarchitekturspezifikation

An einem Beispiel läßt sich vieles leichter erklären. Dieses Kapitel verwendet daher ein einfaches Beispiel, um die DOT–Technologie zu verdeutlichen. Die in Abbildung 4.3 dargestellte *Komponentenspezifikationsarchitektur* [8, Component Specification Architecture] zeigt eine Menge von Komponenten und deren Schnittstellen, die zusammen ein einfaches EDV System zur Verwaltung von Kunden, Telekommunikationverträgen und zur Abrechnung von Telefonkosten bilden.

Die Schnittstellenspezifikationspakete der Komponenten `CustomerMgr` und `BillingMgr` befinden sich in einem übergeordneten Paket `Telecom` (s. Abb. 4.4).

Über die Schnittstelle `ICustomerMgt` können Kunden hinzugefügt, gelöscht und

[2] Für eine ausführliche Diskusion der beiden Ansätze siehe [8].
[3] Zu Voraussetzungen zur Adaption von CORBA–Komponenten s. 4.5 (S. 72)

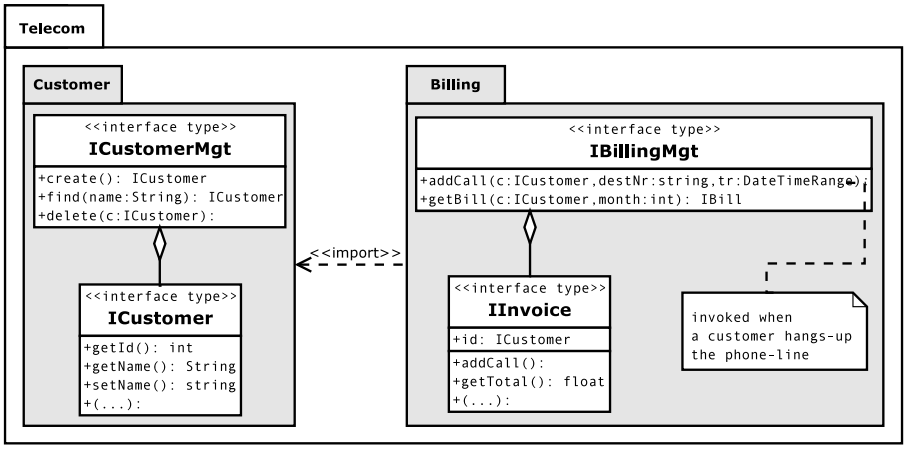


Abbildung 4.4: Schnittstellenspezifikationspakete `Customer` und `Biling`

anhand des Namens eines Kunden gefunden werden. Die eigentlichen Daten des Kundens werden über die Schnittstelle `ICustomer` bearbeitet.

Die Schnittstelle `IBillingMgt` der Abrechnungskomponente ist sehr einfach gehalten. Hat ein Kunde einen Anruf getätigt, so wird anschließend (durch die Vermittlungsstelle) die Operation `addCall` aufgerufen. Als Parameter wird der Kunde, die gewählte Zielrufnummer sowie der Begin- und Endzeitpunkt des Gespräches übergeben. Es wird die Gesprächsgebühr errechnet und anschließend als neue Position der Rechnung hinzugefügt, welche über die Schnittstelle `IInvoice` erreichbar ist.

4.3 Teams

Ein *Team* kapselt eine Menge von partiellen, miteinander interagierenden Aspekten, wobei jeder Aspekt als Rolle einer Domänenentität aufgefaßt wird. Teams und dessen Rollen befinden sich in einer sehr speziellen bzw. ausschnitthaften Sicht auf die (außenliegende) Welt und zwar sowohl in der Typebene als auch in der Ebene der Instanzen.

Die Definition einer Verhaltensadaption durch Aspekte (Rollen) erfolgt auf Typebene. Daher sind in DOT zwei Basisschnittstellen vorgegeben (`ITeam` und `IRole`), welche durch Vererbung den speziellen Eigenschaften der zu realiserenden Aspekte angepasst werden müssen.

Team– und Rollenschnittstellen werden in DOT in Form von Komponenten realisiert. Aufgrund der Kontextabhängigkeit einer Rolle darf diese nicht ihr (umschließendes) Team verlassen. D.h., es muß eine strikte Trennung der beiden Welten, Team und Domäne, gewahrt bleiben. Rollen sind an ihr Team gebunden, wohingegen *Basisobjekte* (Domänenkomponenten) kontextübergreifende Entitäten sind. In DOT werden deßhalb Rollen als Subkomponenten einer Team–Komponente modelliert, da sie dadurch fest an ihre Teamkomponente gebunden sind.

Für die Sprache Java manifestiert sich eine Team–Komponent und dessen Subkomponenten (Rollen) in zwei Klassen: `TeamImpl` bzw. `RoleImpl`. Diese müssen, parallel zu ihren Schnittstellen, durch Vererbung spezialisiert werden.

Am Beispiel eines Bonus–Teams, mit dessen Hilfe ein Telefonkunde beim Tätigen eines Anrufes “Bonuspunkte” sammeln kann (Payback, HappyDigits), werden die einzelnen Schritte zur Definition eines Teams gezeigt.

4.3.1 Modellierung

UML For Aspects (UFA) [4] ist eine auf UML [18] basierende, graphische Notationstechnik zur Modellierung von Askpekten unter Verwendung des ’Object Team’ Modells. Ein Team wird darin als UML–Paket dargestellt. Es umschließt die mit dem Team definierten Rollen, wobei eine Rolle als Klasse modelliert wird. Zusätzlich werden zwei neue Felder (’List Compartments’[18]) eingeführt. Team–Paketsymbole lassen sich durch ein Feld erweitern um Methoden auf Ebene des Teams zu definieren (*team–level Methods*). Das andere Feld ergänzt das Symbol der (Rollen–) Klasse, d.h., zu den Feldern für Attribute und Methoden kommt ein drittes Feld, welches sich unterhalb des Methodenfeldes befindet. Darin werden Callin– und Callout–Bindungen zwischen Methoden einer Rolle und eines Domänenobjektes definiert.

Die Notationstechnik UFA wurde für ’Object Teams’ entwickelt, d.h., sie folgt dem OO-Modell. Sie bedient sich daher der Klasse als Konstrukt zur Modellierung von Rollen. Anders als in ’Object Teams’ werden Rollen nicht als Klassen sondern als

Schnittstellen einer Subkomponente definiert. Die für 'Object Teams' verwendete Notationstechnik (UFA) ist daher nicht idealerweise geeignet, DOT–Teams zu modellieren. Zudem sollte eine graphische Repräsentation eines DOT–(CORBA) Teams möglichst 1–zu–1 in die IDL umgesetzt werden können. Daher liegt es nahe, die für Komponenten ohnehin schon verwendeten Schnittstellenspezifikationspakete als Grundlage zur Definition eines Teams zu verwenden.

Die in DOT verwendete Darstellungstechnik nutzt nicht das UML–Symbol "Paket" zur Darstellung der Kompositionsbeziehung zwischen einem Team und dessen Rollen sondern weist die Kompositionsbeziehungen explizit aus. Dadurch vereinfacht sich die Transformation der sepzifizierten Schnittstellen in die Sprache "IDL" deutlich. Es müssen deßhalb keine weiteren Vereinbarungen getroffen werden, wie die Überführung der graphischen Repräsentation eines Teams in IDL zu erfolgen hat.

4.3.2 Schnittstellenspezifikation

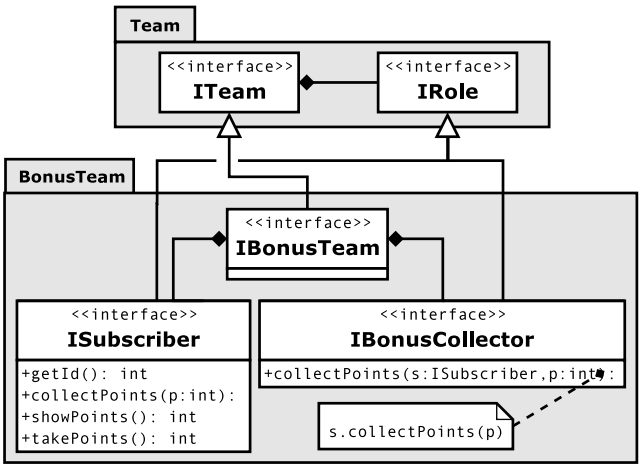


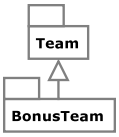
Abbildung 4.5: Schnittstellenspezifikationspaket eines einfachen Bonus–Team

Abbildung 4.5 zeigt die Definition des `BonusTeam`. Das gruppierende Paket (IDL–Module) trägt den Namen des Teams. Die Teamschnittstelle `IBonusTeam` selbst enthält keine Operationen ('team–level methods') [4].

Die Schnittstelle `ISubsriber` definiert Operationen, um einer an dem Bonussystem teilnehmenden Entität Punkte gut zu schreiben (`collectPoints`), den aktuellen Punktestand abzufragen (`showPoints`) sowie die gesammelten Punkte zu benutzen (`takePoints`) und dadurch den Punkteakkumulator zu leeren. Mit Hilfe der Schnittstelle `IBonusCollector` können ausschließlich Punkte gesammelt

[4] Fungiert dieses Team nach eine Spezialisierung als Konnektor (s. Kap. 4.4, S. 56), kann es durch Operationen angereichert werden

werden. Dabei werden als Parameter ein Teilnehmer sowie ein Punktewert übergeben.



In der Abbildung 4.5 sind die Vererbungsrelationen zwischen den einzelnen Schnittstellen explizit eingezeichnet. Diese können, für eine einfachere Darstellung, wie nebenstehend zu sehen, durch eine Vererbungsrelation zwischen den Paketen ersetzt werden. Es soll jedoch verdeutlicht werden, das eine Teamschnittstelle und dessen Rollenschnittstellen immer die Abbildung 4.5 eingezeichneten Vererbungsrelationen in IDL aufweisen müssen.

4.3.3 Teamspezifikationspaket

Die in der Schnittstellenspezifikation definierten Schnittstellen werden in der *Komponentenspezifikation* einzelnen Komponenten zugeordnet. Zudem werden alle Abhängigkeiten einer Komponente zu anderen Schnittstellen festgehalten. Für DOT bedeutet dies, welche Teamkomponente welche Schnittstelle implementiert und welche andere(n) Schnittstelle(n) das Team benötigt.

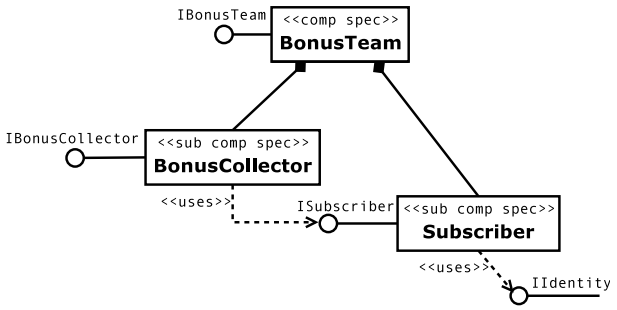
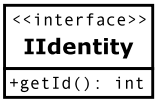


Abbildung 4.6: Komponentenspezifikation der Komponente `BonusTeam`

Der Sachverhalt wird mit Hilfe eines Komponentenspezifikationsdiagrammes [8] dargestellt (s. Abb. 4.6). Darin ist die Komponente `BonusTeam` mit deren Subkomponenten eingezeichnet. Die Schnittstelle `IBonusTeam` wird durch die Komponente `BonusTeam` realisiert. Dessen Subkomponenten `Subscriber` und `BonusCollector` realisieren die Schnittstellen `ISubsriber` und `IBonusCollector`.

Das Konzept von 'Object Teams' ermöglicht es, Methoden einer Rolle abstrakt zu definieren. Diese können durch einen Subtyp der Rolle entweder implementiert, oder, durch einen Konnektor (s. Kap. 4.4, S. 56), an Methoden einer Domänenentität gebunden werden (*'callout'*).



Um eine späte Bindung "abstrakter" Operation einer Rolle durch einen Konnektor zu ermöglichen, müssen in DOT diese Operationen in eine seperate Schnittstelle ausgelagert werden, welche durch die Rolle benutzt wird (s. Kap. 4.1.2, S. 42). Die in der Rollenschnittstelle `ISubscriber` definierte Operation `getId()` soll als abstrakte Operation behandelt werden. Sie wird dazu innerhalb der neuen Schnittstelle `IIdentity` definiert.

Mit Hilfe der Komponentenspezifikation der Subkomponente `Subscriber` wird zwar dargestellt, daß eine Abhängigkeitsbeziehung («`uses`») zur Schnittstelle `IIdentity` besteht (s. Abb. 4.6), es bleibt jedoch offen, wann die Operation `IIdentity.getId()` aufgerufen wird. Es muß explizit festgelegt werden, daß die Operation `IIdentity.getId()` immer dann aufgerufen wird, wenn ein Aufruf der Operation `ISubsriber.getId()` erfolgt. Dieser Realisationsvertrag wird mit Hilfe eines seperaten Komponenteninteraktionsdiagrammes [5] dargestellt (s. Abb. 4.7).

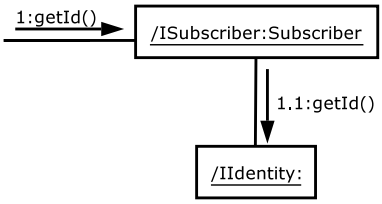


Abbildung 4.7: Komponenteninteraktionsdiagramm der Komponente Subscriber

Offerierte und benötigte Schnittstellen einer Komponente sowie Informationen über die Implementierung einer Komponente werden häufig in einem *Komponentendeskriptor* zusammen gefaßt. Eine durch Menschen lesbare Form eines Deskriptors ist, z.B., ein PDF–Dokument, in dem die Spezifikation der Komponente festgehalten ist. Die technische Realisierung eines Komponentendeskriptors kann, z.B., mit XML-Dateien realisiert werden [15] oder durch Hinzufügen von neuen Schlüsselwörtern in das, in einer Java–Jar–Datei eingebetteten, Manifest[20].

Für DOT sollen Komponentendeskriptoren zusätzlich Informationen über "abstrakte" Operationen einer Rolle beinhalten. D.h., die in der Komponentenspezifikation definierten «`uses`» Beziehungen und deren Verträge zur Implementierung der einzelnen Operationen werden in den Komponentendeskriptor aufgenommen. Dieses Konstrukt wird mit *Teamspezifikation* bezeichnet. Es umfaßt also die von einer Teamkomponente bereitgestellten Schnittstellen und definiert die erwartete Schnittstelle jeder Rolle inklusive des Vertrages zur Implementierung der darin enthaltenen Operationen.

Die graphische Darstellung erfolgt mit Hilfe eines erweiterten Schnittstelleninformationspaketes. Dieses wird mit *Teamspezifikationspaket* bezeichnet.

[5] Für ein Komponenteninteraktionsdiagramm gilt die Namensregel [8]:
`Instanzname/Schnittstelle:Komponente`

Für das `BonusTeam` ergibt sich das in Abbildung 4.8 gezeigte Teamspezifikationspaket. Darin enthalten sind alle schon aus Abildung 4.5 (S. 48) bekannten Schnittstellen (die Operationen wurden zur kürzeren Darstellung ausgeblendet). Hinzugekommen ist die Schnittstelle `IIdentity`, welche von einer Implementierung der Schnittstelle `ISubscriber` erwartet wird. Dieser "Vertrag" wird durch die eingezeichnete Abhängigkeit und den Stereotyp «`expected`» geschlossen.

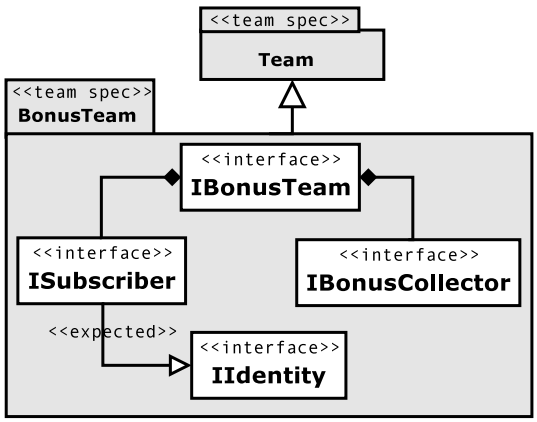


Abbildung 4.8: Teamspezifikationspaket `BonusTeam`

4.3.4 IDL–Definition

Ein Teamspezifikationspaket wird vollständig in CORBA–IDL umgesetzt. Es sind also keine weiteren Hilfsmittel (z.B. XML-Dateien) notwendig! Listing 4.1 zeigt die IDL–Beschreibung für das `BonusTeam`.

Ein Team wird immer innerhalb eines eigenen Namensraumes definiert (Z. 2). Die Schnittstelle der Team–Komponente muß eine Spezialisierung der Schnittstelle `DOT::ITeam` sein (Z. 3), alle Rollenschnittstellen erben von der Schnittstelle `DOT::IRole` (Z. 8, 13). Erwartete Schnittstellen werden ebenfalls innerhalb des Namenraumes des Teams definiert (Z. 5).

Um die Zuordnung einer Rolle zu einer erwarteten Schnittstelle auszudrücken, wird sich eines besonderen Tricks bedient. Die IDL erlaubt die Deklaration von konstanten Datentypen (IDL: `const`). Dies wird genutzt, um eine konstante Zeichenkette, innerhalb des Namensraumes des Teams, zu deklarieren. Sie enthält alle Metainformationen eines Teams.

Leider kann eine Zeichenkette belieben Text beinhalten, sodaß dem Entwickler verdeutlicht werden muß, das es sich nicht um einen einfachen *'string'* handelt, sondern um einen Typen, für den bestimmte Regeln gelten. IDL bietet die Möglicheit neue Typen durch ein `typedef` zu erzeugen. Dies wird in DOT genutzt, um den Typen `string` mit `DOT::TeamInfo` zu benenen. `DOT::TeamInfo` ist also weiter-

```
1 #include <DOT.idl>
2 module BonusTeam {
3   interface IBonusTeam : DOT::ITeam {
4   };
5   interface IIdentity {
6     long getId();
7   };
8   interface ISubscriber : DOT::IRole, IIdentity {
9     void collectPoints(in long p);
10    long showPoints();
11    long takePoints();
12  };
13  interface IBonusCollector : DOT::IRole {
14    void collectPoints(in ISubscriber s, in long p);
15  };
16  const DOT::TeamInfo TEAMINFO = ""
17     "dependency{"
18         "roleIf{ISubscriber}"
19         "expectedIf{IIdentity}"
20     "}";
21 };
```

Listing 4.1: Umsetzung des `BonusTeam` in IDL

hin eine Zeichenkette, besitzt jetzt jedoch semantische Eigenschaften.

Alle Informationen, die statisch verfügbar sein müssen und nicht direkt in IDL auzudrücken sind, werden in eine Instanz namens `TEAMINFO` des Types `DOT::TeamInfo` deklariert. Diese befindet sich genau *ein* Mal innerhalb des Namensraumes des Teams. Tabelle 4.1 gibt eine Übersicht über die für ein Team [6] verwendeten Schlüsselwörter innerhalb der Zeichenkette `TEAMINFO`

Innerhalb einer `TEAMINFO` muß der Name einer Schnittstelle vollqualifiziert sein, wenn er sich nicht auf den lokalen Namensraum bezieht. Schnittstellen innerhalb der Vererbungshirarchie des Teams können jedoch mit ihrem einfachen Namen angesprochen werden.

Schlüsselwort	Bedeutung
`dependency{<roleIf{}><expectedIf{}>}`	Abhängigkeit
`roleIf{<Interface Name>}`	Rollenschnittstelle
`expectedIf{<Interface Name>}`	Erwartete Schnittstelle

Tabelle 4.1: Schlüsselwörter innerhalb einer `TEAMINFO` eines Teams.

Die konstante Zeichenkette `TEAMINFO` in Listing 4.1 (Z. 16) definiert eine Abhängigkeit (`dependency{}` , Z. 17) zwischen einer Rollenschnittstelle (`roleIf{}`, Z. 18) und einer erwarteten Schnittstelle (`expectedIf{}`, Z. 19).

[6] Ein Konnektor (s. Kap. 4.4, S. 56) erweitert die Menge der in einer `TEAMINFO` möglichen Schlüsselwörter.

Eine Überprüfung der Korrektheit der Ausdrücke innerhalb der `TEAMINFO`–Zeichenkette kann nicht durch den IDL–Compiler übernommen werden (es erfolgt nur eine Validierung der IDL–Syntax). Es ist ein seperates Werkzeug notwendig [7].

4.3.5 Implementierung für DOT/J

Nachfolgend sollen kurze Hinweise gegeben werden, was zur Implementierung eines Teams unter Verwendung des DOT Java Frameworks beachten werden muß.

Die Schnittstellen eines Teams müssen von den Basisschnittstellen `DOT::ITeam` bzw. `DOT::IRole` erben (s. oben). Parallel dazu, muß die Implementierung eines Teams, daß in den Klassen `DOT::TeamImpl` bzw. `DOT::RoleImpl` vordefinierte Verhalten durch Vererbung nutzen. Es ergibt sich daher das in Abbildung 4.9 gezeigte Klassendiagramm (für Rollen gilt ein entsprechendes Klassendiagramm mit anderen Bezeichnern).

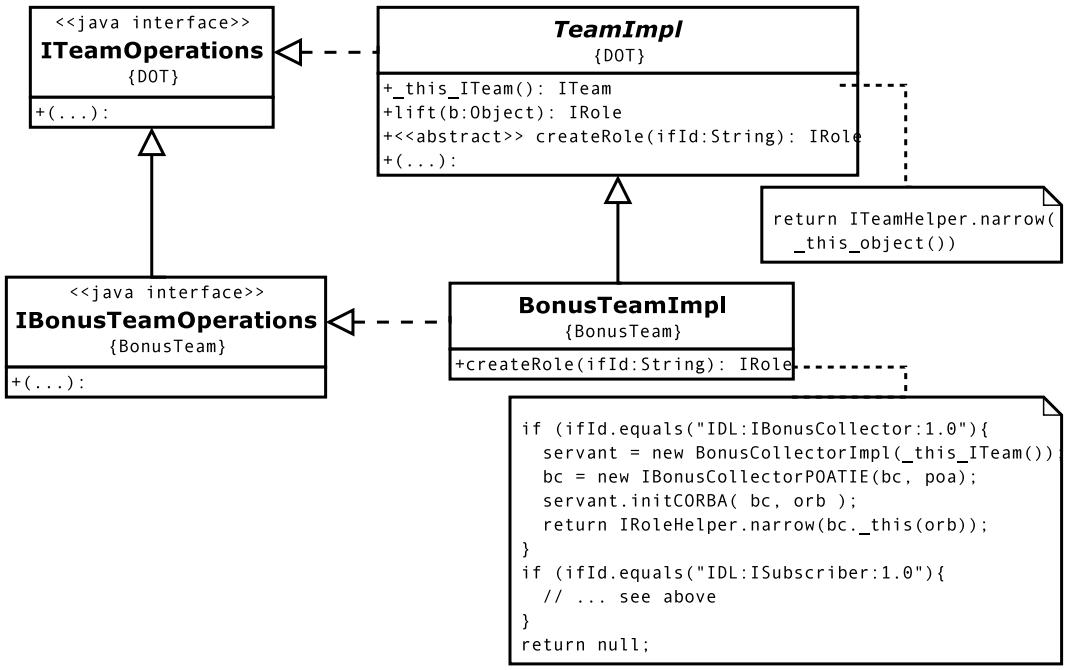


Abbildung 4.9:
Klassendiagramm mit Abhängigkeiten zwischen den Namensräumen `DOT` und `BonusTeam`. Die Schnittstellen werden durch einen IDL–Compiler generiert.

Immer wenn ein Basisobjekt (Objekt der Domäne) den Kontext eines Teams betritt, wird dieses zu der dazu gehörigen Rolle geliftet (Translationspolymorphie, s. Kap. 4.4.1, S. 61). Ist keine Rolle mit diesem Basisobjekt assoziiert, muß ein neues

[7] Siehe '`GlobalTeamMgr`', Kap. 5.1, S. 76

Rollenobjekt erzeugt werden. Da in DOT Rollen und Basisobjekte auf der Ebene der Schnittstellen typisiert sind, kann die Instanziierung einer Rolle durch ein generisches Team nicht erfolgen. Dem generischen Team ist nicht bekannt, welche Klasse welche (Rollen–) Schnittstelle implementiert. Daher ist in der Teamimplementierung `DOT::TeamImpl` ein *hook* zur Erzeugung von Rollenobjekten notwendig. Das Lifting wird von der *Schablonenmethode* `lift` übernommen, welche die *Einschubmethode* `createRole` zur Erzeugung einer Rolle verwendet. Die Methode `createRole` stellt eine Fabrik–Methode [2, Fabrik] für Rollenobjekte dar. Der Einfachheit halber gibt es pro Team nur eine Rollenfabrik. Damit alle Rollen eines Teams erzeugt werden können, erhält die Methode `createRole(...)` als Argument die *RepositoryId*[8]der Schnittstelle, deren Implementierung erzeugt werden soll (s. Notiz in Abb. 4.9).

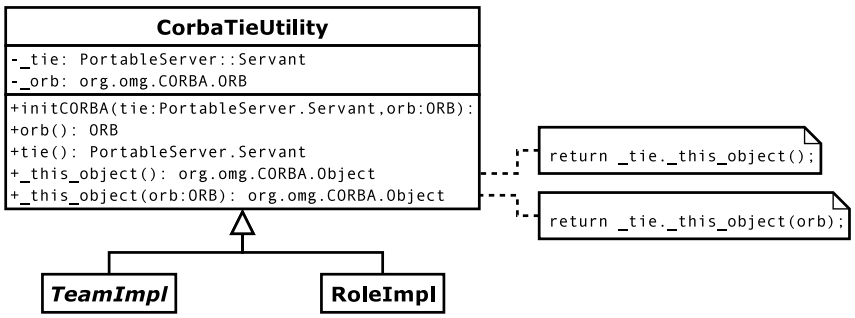


Abbildung 4.10:
`CorbaTieUtility` ist die Basis für alle Team– und Rollenklassen

Teams und Rollen sind CORBA–Objekte und müssen daher einen `Servant` bereitstellen. Dies erfolgt normalerweise durch Vererbung eines `Skeleton`. Java bietet jedoch nur 'single inheritence', sodaß eine Vererbung zwischen den Klassen `DOT::TeamImpl` und `BonusTeam::TeamImpl` eine weitere Vererbung zur Klasse IBonusTeamPOA ausschließt [9]. Daher muß für DOT/J der *TIE* Ansatz gewählt werden. DOT bedient sich der Klasse `CorbaTieUtility`, welche die Verwendung von TIE–Referenzen vereinfacht (s. Abb. 4.10). Es wird v.a. die Methode `_this_object()` bereitgestellt, mit deren Hilfe eine *Stub*–Referenz auf das CORBA–Objekt einer Servant–TIE–Implementierung erlangt werden kann.

Mit Hilfe der (Super–) Klasse `CorbaTieUtility` verläuft die Erzeugung einer Team– oder Rollenimplementierung mit assoziiertem PortableServer wie in der Notiz in Abbildung 4.9 gezeigt: (1) Es wird eine die Schnittstelle `XXX` implementierende Klasse instanziiert (`XXXImpl`). (2) Danach wird eine Servantinstanz (`XXXPOATIE`) erzeugt, deren Konstruktor die voher erschaffene Implementierung (`delegate`) übergeben bekommt. (3) An der Implementierung (`XXXImpl`) wird

[8]Eine `RepositoryId` ist eine eindeutige Zeichenkette zur Identifikation einer Schnittstelle innerhalb des CORBA–Schnittstellenverzeichnisses.
[9]Details zur Schnittstellen– und Skeleton– Vererbung in CORBA s. bitte Kap. 3.2.4, S. 27

`initCORBA` aufgerufen um damit den assoziierten Servant und den ORB bekannt zu machen. (4) Der Servant wird z.B. mit `_this(orb)` aktiviert.

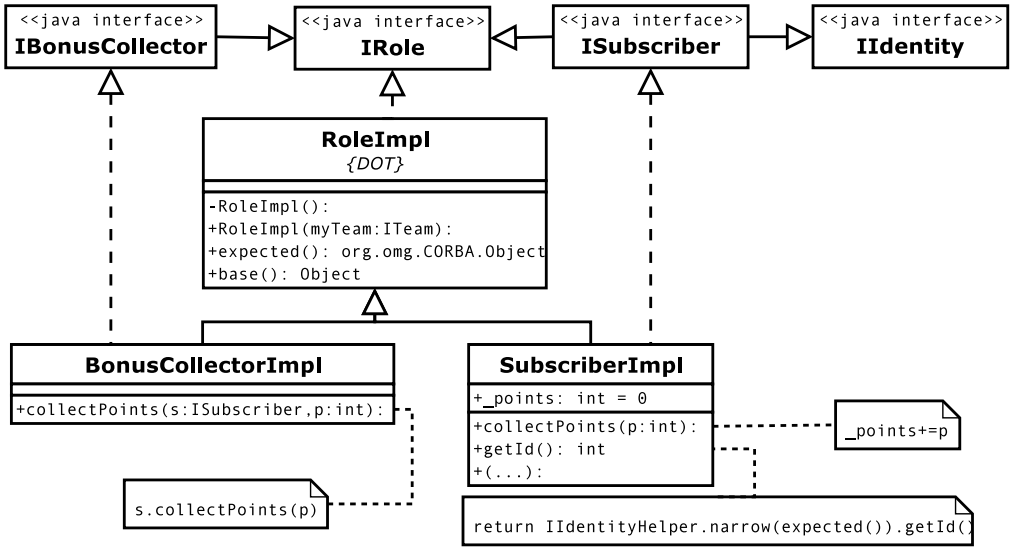


Abbildung 4.11:
Vererbungs– und Realisierungsbeziehungen der Rollen des `BonusTeam`.

Genauso wie Teamimplementierungen eine Vererbungsbeziehung zur Klasse `DOT.TeamImpl` haben müssen, wird von Rollenimplementierungen gefordert, daß sie eine Spezialisierung der Klasse `DOT.RoleImpl` sind. Alle erwarteten Operationen einer Rollenschnittstelle müssen in der realisierenden Klasse implementiert werden [10], denn es bedarf *einer* Zeile Quelltext, um eine späte Bindung dieser Operation mit Hilfe eines Konnektors [11] zu ermöglichen; die Implementierung einer solchen, erwarteten ("abstrakten") Rollenoperation folgt immer demselben Muster:

```
IXXXHelper.narrow(expected()).xxx()
```

In obiger Zeile muß der Ausdruck "IXXX" durch den Namen der erwarteten Operation ersetzt werden, der Ausdruck "xxx" durch den Namen der darauf aufzurufenden Operation.

Die Methoden `expected()`[12] liefert eine Referenz auf die Implementierung der erwarteten Schnittstelle. Diese muß in den Typen der erwarteten Schnittstelle gewandelt werden (`narrow`), um anschließend darauf die entsprechende Operation aufzurufen.

[10] In OT/Java werden diese Methoden als «`abstract`» deklariert und durch eine verfeinerte Klasse implementiert bzw. gebunden.
[11] s. K. 4.4, S. 56
[12] s. Kap. 5.2.2, S. 93

4.4 Der Konnektor

Nachdem kollaborierende Aspekte in einem Team zusammengefaßt wurden, bleibt die Frage, wie diese in ein Domänenmodell gewebt, d.h. eingefügt werden können. Dazu definiert 'Object Teams' ein Konstrukt namens *Konnektor*. Ein Konnektor ist eine spezielle Form eines Team, welcher die Schnittstelle `DOT::ITeam` und dessen Implementierung `DOT::TeamImpl` verfeinert. Er bindet Rollenschnittstellen an Schnittstellen der Domäne (Basisschnittstelle) – jede Komponente der Domäne *spielt* also eine bestimmte Rolle eines Teams.

Ein Beispiel: Es soll ein Konnektor `TelecomBonusTeam` definiert werden, der mit Hilfe einer Komponente `BonusTeam` das in Abbildung 4.3 (S. 45) gezeigte System (Paket: `Telecom`, Abb. 4.4, S. 46) erweitert. Daraus können sich folgende *Spieler*–Beziehungen ergeben: Alle Subkomponenten mit der Schnittstelle `ICustomer` spielen die Rolle der Schnittstelle `ISubscriber`, die Komponente `BillingMgr` mit der Schnittstelle `IBillingMgt` spielt die Rolle, welche die Schnittstelle `IBonusCollector` implementiert.

Eine Rolle kapselt eine Teilfunktionalität einer Domänenkomponente, d.h. eine Bindung dieser Teilfunktionalität muß an der Stelle erfolgen, wo sie benötigt wird. Daher erfolgen Bindungen auf der Ebene der Operationen von Schnittstellen. Ein Operationsaufruf an einer gebundenen Schnittstelle einer Domänenkomponente wird zu einer Rollenoperation delegiert. Das Konzept 'Object Teams' definiert zwei Arten von Delegation (aus der Sicht der Rolle): *Callin* und *Callout*. Deren Übertragung in DOT wird nachfolgend erläutert.

Callin

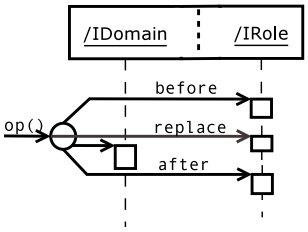


Abbildung 4.12:
'Callin': Delegation von Operationsaufrufen einer Schnittstelle einer Domänenkomponente zu Operationen einer Rollenschnittstelle.

Eine 'callin'–Bindung definiert, daß jeder Aufruf einer Operation an einer Schnittstelle einer Domänenkomponente zu einer definierten Operation der gespielten Rollenschnittstelle delegiert wird.

Abbildung 4.12 soll den Vorgang der Delegation anhand eines Sequenzdiagram-

mes erläutern. Der gemeinsame Kasten um die Komponenteninstanzen herum (sie implementieren die Schnittstellen `IDomain` bzw. `IRole`), und die gestrichelte Linie sollen verdeutlichen, daß es sich zwar um zwei seperate Instanzen handelt, diese aber unwiderbringlich durch einen Konnektor "zusammengeschweißt" sind.

Eine 'Callin'–Delegation kann auf drei Arten erfolgen:

- *Before*: Die Operation der Rollenschnittstelle wird *vor* der Operation der Basisschnittstelle ausgeführt
- *Replace*: Die Operation der Rollenschnittstelle wird *anstatt* der Operation der Basisschnittstelle ausgeführt (Die Implementierung der Rollenoperation kann jedoch einen *basecall* absetzen, der einen Aufruf der ursprünglichen Funktionalität zur Folge hat.)
- *After*: Es wird zuerst die Operation der Basisschnittstelle, *danach* die Operation der Rollenschnittstelle ausgeführt.

Eine Delegation von einer Domänenschnittstelle zu einer Rollenschnittstelle hin, ist in keinem Domänenkonstrukt definiert! Der Konnektor ist verantwortlich dafür, das Domänenschnittstellen, und damit Domänenkomponenten, durch die Definition von Delegationen adaptiert werden. Eine Namensgleichheit zwischen Schnittstellen oder Operationen der Domäne und der Rollen ist nicht notwendig. Auch können Differenzen in der Signatur einzelener Operationen angepasst werden (s. Kap. 4.4, S. 59.

Im Konnektor `TelecomBonusTeam` wird z.B. eine 'Callin'–Delegation zwischen den Operationen `IBillingMgt::addCall(...)` und (der neuen Operation) `IBonusCollector::collectCallPoints(...)` definiert.

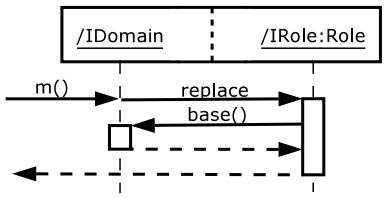


Abbildung 4.13:
Eine Rollenoperation spezialisiert eine Operation einer Domänenschnittstelle.

Ersetzt eine Rollenoperation eine Operation einer Domänenschnittstelle (*replace*), kann die Implementierung der Rollenoperation die ursprüngliche Operation mit Hilfe des Schlüsselwortes *base* weiterbenutzen (s. Abbildung 4.13). Ein solcher Aufruf wird mit *'basecall'* bezeichnet. Jede Operation einer Rollenschnittstelle kann eine Operation einer Domänenschnittstelle ersetzen; es ist keine gesonderte Deklaration in der Rollenschnittstelle notwendig. Benutzt die Implementierung einer Rollenoperation jedoch das Schlüsselwort *base*, muß diese Operation als *replace* deklariert

werden. Nur so ist sichergestellt, daß eine entsprechende Domänenoperation (welche durch *base* aufgerufen wird) gebunden wurde.

Callout

Das Gegenstück zur 'callin'–Delegation wird mit '*callout*' bezeichnet, es erfolgt eine Delegation von einer Rolle zu ihrer gebundenen Domänenkomponente. Eine Teamspezifikation (s. Kap. 12, S. 50) enthält neben der Teamschnittstelle und deren Rollenschnittstellen auch alle von deren Rollen erwarteten Schnittstellen («`excepted`»). Diese können in einem Konnektor auf zwei Arten realisiert werden.

Die fehlende Funktionalität kann in einer abgeleiteten Rollenimplementierung erbracht werden, d.h., die Operationen der erwarteten Schnittstelle werden überschrieben. Die spezialisierte Rolle muß dazu einer eigenen Schnittstelle folgen, welche durch eine «`expected`» Beziehung auf eine Schnittstelle der noch übriggebliebenen erwarteten Operationen zeigt. Damit die spezialisierte Rolle auch benutzt wird, muß zusätzlich in der Teamkomponente die Methode `createRole(...)` überschrieben werden, sodaß auf Anforderung hin, eine Instanz der spezialisierten Rollenimplementierung erzeugt wird.

Abbildung 4.14: Callout–Bindung

Alternativ zur Implementierung von Operationen erwarteter Schnittstellen, können Rollenoperationen an Operationen einer Domänenschnittstelle gebunden werden (s. Abbildung 4.14). Damit werden Aufrufe von Operationen erwarteter Schnittstellen an definierte Operationen von Domänenschnittstellen delegiert. Eine Domänenkomponente implementiert also die erwartete Funktionalität einer Rolle.

Für das Beispiel des Konnektors `TelecomBonusTeam` erfolgt eine 'callout'–Bindung in der Spielerbeziehung zwischen `ISubscriber` und `ICustomer`. Darin wird die Operation `getId()` der Rollenschnittstelle an die Operatione `getId()` der Domänenschnittstelle gebunden.

Eine Teamspezifikation (Konnektorspezifikation) wird als *vollständig* bezeichnet, wenn keine «`expected`» Beziehung mehr besteht oder alle Operationen der erwarteten Schnittstellen an Operationen von Domänenschnittstellen gebunden sind. Ein solches Team kann instanziiert werden.

Signaturanpassung

Bindungen zwischen Rollen– und Domänenschnittstellen erfolgen immer auf der Ebene der Operationen. Da ein Team unabhängig von einer Domäne entwickelt werden kann, ist eine Übereinstimmung der Signaturen von gebundenen Operationen nicht immer gegeben. Daher kann für jede Art von Bindung zweier Operationen definiert werden, wie die Signaturen anzupassen sind. Dabei können Parameter vertauscht, weggelassen oder transformiert werden. Auch der Rückgabewert, er gehört schließlich auch zur Signatur, kann verändert oder als Argument einer anderen Operation übergeben werden. Für Operationen ohne Parameter werden alle Argumente automatisch versteckt.

Das Laufzeitsystem von DOT paßt, vor einer Delegation zu einer Rolle oder Domänenkomponente, alle Parameter entsprechend der Definition an und transformiert anschließend den Rückgabewert.

Aktivierung der definierten Adaptionen

Die in einem Konnektor definierten Bindungen treten erst nach einer expliziten Aktivierung einer Konnektorinstanz in Kraft. Dazu muß an einer Konnektorinstanz die Operation `activate` aufgerufen werden [13].

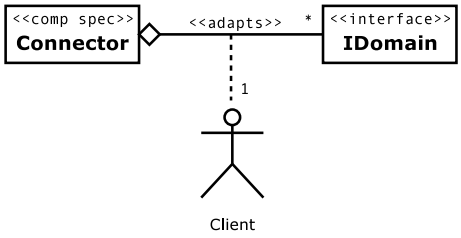
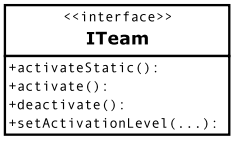


Abbildung 4.15: Per–Klienten Adaption von Domänenkomponenten

Die Aktivierung eines Konnektors erfolgt in DOT auf einer Per–Klienten–Basis (s. Abb. 4.15). Es wird so eine klientenspezifische Sicht auf das System ermöglicht [24]. D.h., aktiviert ein Klient einen Konnektor, sind alle Bindungen des Konnektors innerhalb des Kontextes des Klienten aktiv.Der Kontext des Klienten wird mit allen Aufrufen propagiert, so daß auch für entfernte Komponenten die definierten Adaptionen aktiv sind. Deaktiviert ein Klient einen zuvor aktivierten Konnektor, werden alle seine Bindungen für den Kontext des Klienten inaktiv. Die Funktionalität zur Aktivierung und Deaktivierung ist in der Basisschnittstelle `DOT::ITeam` definiert, und damit in jdem Team vorhanden.

[13] Mit Hilfe der Operation `setActivationLevel` können zusätzlich fein–granulare Aktivierungsmodi eingestellt werden: INACTIVE, NOCALLIN, FROZEN, ACTIVE



Wird eine Konnektorinstanz aktiviert, so ändert sich das Verhalten der Domänenkomponenten entsprechend der definierten Bindungen. Wird die Instanz deaktiviert, sind alle definierten Bindungen inaktiv und die Domänenkomponenten nehmen ihr ursprüngliches Verhalten wieder an. Die Funktionalität der Domänenkomponenten ist also abhängig vom Zustand einer Konnektorinstanz. Diese kann daher als Kontext aufgefasst werden (im folgenden mit *Konnektorkontext* bezeichnet). Eine Domänenkomponente nimmt also innerhalb des Kontextes eines Konnektors ein rollentypisches Verhalten an, und zwar abhängig vom Vorhandensein eines klienten–spezifischen Kontextes. Dadurch ist es möglich, kontextsensitive Informationssysteme zu modellieren.

Um das Beispiel des Informationsbedarfes eines Piloten während eines Fluges durch verschieden Wetterzonen wieder aufzugreifen, ist eine klienten–spezifische Aktivierung der definierten Adaptionen, und damit, z.B. der Menge der darzustellenden Informationen, erforderlich.

Anders als die eben beschriebene dynamische Aktivierung können Konnektoren auch statisch aktiviert werden. Ein Konnektor, der über die Operation `ITeam::activateStatic` statisch aktiviert wurde, bindet alle seine Rollen statisch an Domänenkomponenten. Diese Bindungen sind für alle Klienten der Domänenkomponenten aktiv.

Anders als in 'Object Teams' kann *jeder* Konnektor statisch aktiviert werden (natürlich muß er vollständig sein). Es wird also nicht voraussgesetzt, das ein statischer Konnektor nur statische Methoden besitzt. Es gelten allerdings folgende Bedingungen:

- Es darf pro Konnektor nur eine statisch aktivierte Instanz geben.
- Der Aufruf `activateStatic` erfolgt genau ein Mal pro Konnektor und zwar vor Aufrufen der Operation `activate` eines dynamischen Teams.
- Ein (klientenseitiger) Aufruf von `activate` oder `deactivate` an einem zuvor statisch aktiviertem Konnektor ist nicht erlaubt.

Ein aktiver Konnektor verändert das Verhalten einer Domänenkomponente, indem deren Funktionalität zur Laufzeit durch Delegation um das in Rollen definierte Verhalten erweitert wird. Dieser Vorgang wird mit *dynamischem Weben* bezeichnet. Da weder die Schnittstellen der Domänenkomponenten noch deren Implementierungen verändert werden, erfolgt eine nicht–invasive Adaption von Komponenten: Der Quelltext der Komponentenimplementierung hat sich nicht geändert, wohl aber der Realisationsvertrag der Domänenkomponente!

4.4.1 Translationspolymorphie

Domänenkomponenten und Teamkomponenten werden unabhängig voneinander entwickelt und sollten dadurch kein wechselseitiges Wissen haben. Sprich: Rollen kennen nur ihr Team und die darin befindlichen Rollen, Domänenkomponenten wissen nichts von Rollen.

Erst ein Konnektor stellt durch eine Spieltbeziehung eine Verbindung zwischen einer Rollenschnittstelle und einer Domänenschnittstelle her. Der Konnektor setzt also Schnittstellen in Beziehung zueinander. Eine Instanz, die einer Domänenschnittstelle folgt, genügt auch einer Rollenschnittstelle und vice versa. Eine Beziehung zweier Schnittstellen gilt allerdings nur innerhalb eines aktiven Konnektorkontextes, da für eine Domänenschnittstelle mehrere Spielt–Beziehungen definiert sein können. Eine Domänenschnittstelle genügt also genauso vielen Rollenschnittstellen wie Spielt–Beziehungen definiert sind. Allgemein wird dieser Sachverhalt mit *Polymorphie* bezeichnet.

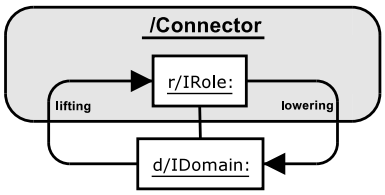
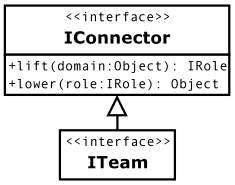


Abbildung 4.16:
Eine Typumwandlung findet nur innerhalb eines Konnektorkontextes statt.

Die Umwandlung eines Typen in einen anderen ('casting') wird in DOT durch zwei Operationen realisiert; dabei ändert sich nicht nur der Typ (wie z.B. in Java), sondern, da Rollen eigenständige Subkomponenten sind, auch die Referenz [14]. Eine Rolle kann durch einen Aufruf der Operation `ITeam::lift()`, mit einer Referenz einer Domänenschnittstelle als Argument, erhalten werden. Für den umgekehrten Fall, also die Wandlung einer Rolle in eine Domänenkomponente, steht die Operation `ITeam::lower()` zur Verfügung. Dadurch, daß zwei Schnittstellenreferenzen unterschiedlichen Typs die gleiche Komponente repräsentieren, spricht man von *Translationspolymorphie*. Nur im Kontext des Konnektors ist definiert, welche Domänenschnittstelle zu welcher Rollenschnittstelle gehoben werden soll. Deshalb findet der Prozeß des 'lifting' und des 'lowering' immer innerhalb eines Konnektors statt. Existiert zu einer Referenz einer Domänenschnittstelle bereits eine Referenz auf eine Rollenschnittstelle (durch eine vorangegangenes lifting), wird diese wieder benutzt. Zu jeder Instanz der Domäne existiert innerhalb eines Konnektors eine Rolleninstanz und zu jeder Rolle gehört genau eine Domäneninstanz.

[14] Der Begriff *Identität* wurde bewußt vermieden. Wird die Identität als logisches Konstrukt begriffen, so ändert sie sich nicht; Domänenkompnenten und Rollen bilden eine Einheit. Aus der Sicht der dinglichen Welt ändert sich die Identität sehr wohl. Die Instanz der Rolle und die der Domänenkomponente unterscheiden sich; es liegen also auch zwei verschiedenen Referenzen vor.



Die Typumwandlung durch 'lifting' und 'lowering' ist völlig automatisiert. [15] Erfolgt innerhalb eines Konnektorkontextes ein Aufruf einer Domänenoperation, für welche eine Spieltbeziehung existiert, so wird die Domänenreferenz und alle Parameter der Operation (soweit für sie eine Spieltbeziehung innerhalb des Konnektors vorliegt) zu entsprechenden Rolleninstanzen geliftet. Bei einem 'Callout' oder 'Basecall', also einem Aufruf von einer Rolle zu einer Domänenkomponente hin, wird die Rolleninstanz (und alle Parameter plus Rückgabewert) in die entsprechende Instanz der Domänenkomponente umgewandelt. Dadurch gelangt kein Komponentenobjekt der Domäne, was eine Rolle innerhalb eines Teams spielt in dieses hinein, und keine Rolleninstanz aus dem Team heraus. Die Menge aller (Sub–) Komponenten der Domäne und Rollen eines Teams bilden zwei disjunkte Mengen.

4.4.2 Modellierung

Ein Konnektor ist ein spezialisiertes Team mit besonderen Eigenschaften. Daher wird ebenso wie für Teams, das Paket–Symbol zur Darstellung von Konnektoren verwendet, welches durch den Stereotyp («`connector spec`») als Konnektorspezifikationspaket definiert wird.

Die durch den Konnektor definierten Adaptionen der Domäne werden mit dem Symbol einer Aggregationsbeziehung dargestellt, welches den Stereotyp «`adapts`» trägt. Die Aggregationsbeziehung geht dabei immer von dem Team zu einem Schnittstellenpaket der Domäne. Rollenschnittstellen können nur an Schnittstellen der Domäne gebunden werden, zu denen es eine «`adapts`» Beziehung gibt.

Zur Bindung einer Rollenschnittstelle an eine Domänenschnittstelle wird eine spezielle Form des Klassensymbols verwendet. Das Namensfeld ('*name compartment*') beinhaltet einen Eintrag in der Form | Rollenschnittstelle = Domänenschnittstelle |.

Zu den bekannten Feldern eines Klassensymbols (Name, Attribute, Methoden) kommt für Bindungen auf der Ebene von Operationen ein zusätzliches Feld ('*list compartment*') unterhalb des Methodenfeldes hinzu. Einträge zur Bindung zweier Operationen werden darin in der Form
| Rollenoperation Bindungstypsymbol Domänenoperation | dargestellt. Für die Bindungstypen werden die in Tabelle 4.2 gezeigten Symbole verwendet.

[15] An wenigen Stellen ist es in DOT notwendig, eine Typumwandlung "per hand" vorzunehemen. Siehe bitte K. 4.4.4, S. 68.

Bindungstyp	Symbol
callout	$\rightarrow$
callin: davor	$\leftarrow before$
callin: danach	$\leftarrow after$
callin: ersetzend	$\leftarrow$

Tabelle 4.2: Symbole zur Darstellung der Bindungstypen.

Beispiel

Die graphische Modellierung eines Konnektors soll anhand des `TelecomBonusTeam` verdeutlicht werden. Abbildung 4.17 zeigt das Teamspezifikationspaket `BonusTeam` und dessen spezialisiertes Konnektorspezifikationspaket `TelecomBonusTeam`, welches die Pakete `Telecom::Billing` und `Telecom::Customer` adaptiert.

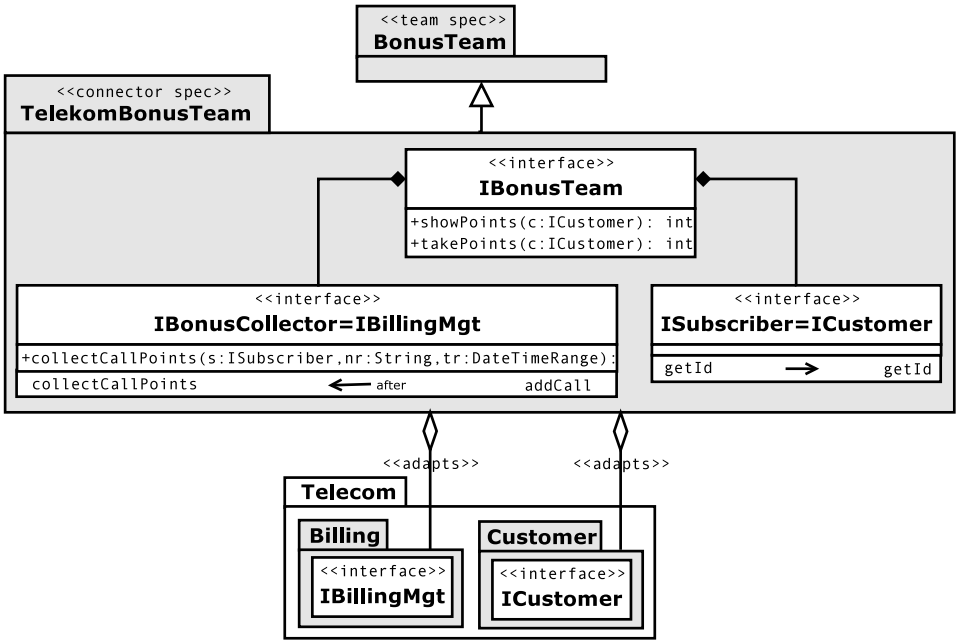


Abbildung 4.17: Konnektormodellierung am Beispiel des `TelecomBonusTeam`.

Der Konnektor definiert zwei Spielt–Beziehungen. Es wird eine Spielt–Beziehung zwischen den Schnittstellen `ISubscriber` und `ICustomer` definiert, wobei die Operation `getId` der erwarteten Schnittstelle `IIdentity` durch ein 'callout' an die Operation `getId` der Schnittstelle `ICustomer` gebunden wird. Die erwartete Schnittstelle `IIdentity` befindet sich im Paket `BonusTeam` und muß im Konnektor nicht erneut eingezeichnet werden.

Die zweite Spielt–Beziehung bindet die Schnittstellen `IBonusCollector` und `IBillingMgt`. Da das Klassensymbol dieser Spielt–Beziehung eine neue Operati-

on definiert, wird damit automatisch auch eine neue Schnittstelle definiert. Diese ist von `BonusTeam::IBonusCollector` abgeleitet und enthält die neue Operation `collectCallPoints`. Im Feld für Operationsbindungen wird eine 'Callin'–Bindung definiert: Die Operation `collectCallPoints` soll nach der Operation `addCall` aufgerufen werden.

Die Schnittstelle `IBonusTeam` wird durch zwei *team level* Operationen angereichert. Mit Hilfe der Operation `showPoints(...)` kann der aktuelle Punktestand einer übergebenen `ICustomer`–Referenz erfragt werden. Die Operation `takePoints(...)` liefert ebenfalls den Punktestannd eines Kunden, zusätzlich wird jedoch der Punktestand des Kunden auf "0" gesetzt.

4.4.3 IDL–Definition

Jedes Team wird, auch wenn es eine Spezialisierung eines bereits definierten Teams ist, in einem eigenen Namensraum (IDL: `module`) definiert. Dies gilt auch für Konnektoren, da diese immer eine Spezialisierung eines Teams sind.

Innerhalb dieses Namensraumes *muß* eine Schnittstelle definiert sein, die eine direkte oder indirekte Vererbungsbeziehung zur Schnittstelle `DOT::ITeam` besitzt. Dadurch kann von DOT erkannt werden, welche Teams (und damit Namensräume) in einer Subtyp–Beziehung stehen. D.h. ein Konnektor hat immer eine eigene Schnittstelle, auch wenn darin keine neuen Operationen definiert sind.

Werden in einer Spielt–Beziehung zweier Schnittstellen neue Operationen definiert, müssen diese in einer neuen Rollenschnittstelle definiert werden. Diese Schnittstelle ist eine Spezialisierung der im 'Super'–Team definierten Rollenschnittstelle und trägt deren Namen.

Andersherum gilt: Eine Rollenschnittstelle, die ohne Veränderung der Signatur aus einem Super–Team in einer Spielt–Beziehung verwendet wird, muß nicht noch einmal im Namensraum des Teams definiert werden. In diesem Fall erfolgt eine *impliziete Vererbung* der Rollenschnittstelle. Natürlich dürfen dort, wo es Sinn macht, in einem Konnektor neue Rollenschnittstellen definiert werden.

In einer Rollenschnittstelle darf als Attribut oder Argument einer Operation keine Schnittstelle der Domäne vorkommen, wenn für diese eine Spielt–Beziehung in der Vererbungshirarchie des Konnektors definiert ist.

Die eigentliche Definition der Spielt–Beziehungen erfolgt über die bereits in Kapitel 4.3.4 (S. 51) beschriebene Zeichenkette `TEAMINFO`. Diese hat aus der Sicht des IDL–Compiler keinerlei Beziehung zu der `TEAMINFO` des Super–Teams, da sie sich in zwei verschiedenen Paketen befinden. Aus der Sicht von DOT jedoch werden zwei Zeichenketten namens `TEAMINFO` als eine betrachtet, wenn es in den Namensräumen der beiden Zeichenketten jeweils eine Schnittstelle gibt, die dem Typ `DOT::ITeam` folgt und diese Schnittstellen in einer direkten oder indirekten Vererbungsbeziehung zu einander stehen. Ein Konnektor erweitert die Menge der in

einer `TEAMINFO` gültigen Schlüsselwörter (s. Tabelle 4.3, S. 66).

Eine Spielt–Beziehungen wird mit dem Schlüsselwort `playedBy` eingeleitet, welches als Argument zwei Schnittstellen übergeben bekommt sowie optional eine Menge von Operationsbindungen. Die übergebenen Schnittstellen werden dabei durch die Schlüsselwörter `roleIf` u. `baseIf` als Rollen – bzw. Domänenschnittstelle ausgewiesen.

Für die vier möglichen Arten einer Bindung auf der Ebene der Operationen sind folgende Schlüsselwörter definiert: `callout`, `before`, `after` und `replace`. Jedem dieser Schlüsselwörter werden zwei Operationsnamen übergeben, welche mit `roleOp` als Operation der Rollenschnittstelle und mit `baseOp` als Operation der Domänenschnittstelle ausgewiesen werden.

Stimmen die Signaturen der beiden Operationen nicht überein oder erwartet eine Operation der Rollenschnittstelle weniger Argumente als die Operationen der Domänenschnittstelle, muß eine Signaturanpassung vorgenommen werden. Dies ist nicht notwendig, wenn die Signatur einer Operationen der Rollenschnittstelle *keine* Parameter aufweist.

Signaturanpassung

Verschiedene Signaturen zweier Operationen einer Spielt–Beziehung können mit Hilfe einer Menge von einzelnen Parameteranpassungen angeglichen werden. Eine Parameteranpassung hat immer die Form: `Rollenargument = Basisargument` bzw. `Basisargument = Rollenargument`. Argumente sind alle Parameter einer Operation plus deren Rückgabewert. Auf den Rückgabewert wird innerhalb einer Parameteranpassung mit "`RETVAL`" zugegriffen. Argumente einer Rollenoperation werden über den Ausdruck `roleArg` eingeleitet, es wird zuerst das anzupassende (Ziel–) Argument angegeben, gefolgt von einem "=" und einer Menge von Argumenten der Domänenoperation, welche durch einfache mathematische Ausdrücke miteinander verknüpft werden können. Für Argumente einer Domänenoperation verhält es sich genau umgekehrt. Eine Anpassung wird mit `baseArg` definiert, gefolgt von dem Namen des Argumentes welches angepasst werden soll, dann ein "=" und schließlich eine Menge von Argumentnamen der Rollenoperation. Auch diese können durch einfache, mathematische Ausdrücke miteinander verknüpft werden.

Die Deklaration einer CORBA–Operationssignatur umfaßt neben dem Typ und dem Namen auch die Datenflussrichtung eines Parameters. In IDL markiert das Schlüsselwort `in`, daß der Parameter in die Operation hinein geht, `out` daß der Parameter als "Rückgabewert" (aus der Operation hinaus) aufgefaßt wird und schließlich `inout` daß der Parameter sowohl Daten in die Operation hinein als auch heraus transportiert. Der Rückgabewert einer Operation wird immer als "`out`" aufgefaßt. Erfolgt in einer Operationsbindung eine Signaturanpassung, müssen die Datenflussrichtungen der einzelnen Parameter beachtet werden.

Spielt–Beziehung	Beschreibung
`playedBy{ <roleIf{}> <baseIf{}> [Operationsbindung][...]}`	Def. einer Spieltbeziehung zweier Schnittstellen mit mind. einer Operationsbindung
`roleIf{<InterfaceName>}`	Name der Rollenschnittstelle
`baseIf{<InterfaceName>}`	Name der Domänenschnittstelle
Operationsbindung	
`callout{ <roleOp{}> <baseOp{}> [Parameteranpassung][...]}`	Def. einer 'callout'–Bindung, optionale Signaturanpassung
`replace{ <roleOp{}> <baseOp{}> [Parameteranpassung][...]}`	Def. einer 'callin'–Bindung (ersetzend), optionale Signaturanpassung
`before{ <roleOp{}> <baseOp{}> [Parameteranpassung][...]}`	Def. einer 'callin'–Bindung (davor), optionale Signaturanpassung
`after{ <roleOp{}> <baseOp{}> [Parameteranpassung][...]}`	Def. einer 'callin'–Bindung (danach), optionale Signaturanpassung
`roleOp{<OperationName>}`	Name einer Operation der Rollenschnittstelle
`baseOp{<OperationName>}`	Name einer Operation eine Domänenschnittstelle
Parameteranpassung	
`roleArg{<ArgNameOfRoleOp>=<ArgNameOfBaseOp [Exp]>[...]}`	Anpassen eines Argumentes einer Rollenoperation
`baseArg{<ArgNameOfBaseOp>=<ArgNameOfRoleOp [Exp]>[...]}`	Anpassen eines Argumentes einer Domänenoperation
Exp	
`< + || - || * || / > [const]`	Mathematischer Ausdruck, opt. Konstante(n)

Tabelle 4.3:
Schlüsselwörter innerhalb einer `TEAMINFO` eines Konnektors.Bedeutung der Zeichen: <X>: Term X *muß* vorhanden sein. [X]: Term X *kann* vorhanden sein. [...]: Der letzte Term *kann wiederholt* angegeben werden.

Tabelle 4.4 zeigt die möglichen Kombinationen von Parameterzuweisungen in Abhängigkeit ihrer Datenflussrichtung. Die möglichen Kombinationen sind für alle Bindungstypen aufgelistet.

Bindungstyp	$dfr(roleArg) \leftarrow dfr(baseArg)$	$dfr(roleArg) \rightarrow dfr(baseArg)$
before	IN ← IN	OUT → IN
after	IN ← {IN,OUT}	OUT → OUT
replace	IN ← IN	OUT → OUT
callout	OUT ← OUT	IN → IN

Tabelle 4.4:
Mögliche Zuweisungen von Parametern einer CORBA–Operation in Abhängigkeit der Datenflußrichtungen. IDL–Schlüsselwörter: `in`, `inout`, `out`, Def. IN: {`in`,`inout`}, Def. OUT: {`inout`, `out`, `RETVAL`}, Def. $dfr(arg)$: "Datenflußrichtung eines Argumentes"

Da in der verteilten Komponentenwelt häufig mit zusammengesetzten Datentypen (`struct`) gearbeit wird (Steigerung der Netzwerkeffizienz), kann, auf der rechten Seite einer Argumentanpassung, auf die Elemente des zusammengesetzten Datentypes mit Hilfe des Punktes (".") zugegriffen werden.

IDL–Beispiel

Das Konzept der Konnektordefinition in IDL soll am Beispiel des Konnektor `TelecomBonusTeam` verdeutlicht werden. Listing 4.2 zeigt den IDL–Quelltext.

Um alle Typen des Super–Teams `BonusTeam` und der Domäne `Telecom` bekannt zu machen, werden die entsprechenden IDL–Dateien inkludiert (Z. 1, 2). Die Definition des Konnektors erfolgt innerhalb des Namensraumes `TelecomBonusTeam` (Z. 4). Ob eine Vererbungsbeziehung zwischen zwei Teams vorliegt, wird von DOT anhand der Vererbungshirarchie der Schnittstelle `DOT::ITeam` erkannt. Daher ist es notwendig die Schnittstelle `IBonusTeam` als Subtyp der Schnittstelle `BonusTeam::IBonusTeam` zu definieren (Z. 6). Darin werden zwei Operationen deklariert (`showPoints` u. `takePoints`, s.o.) welche als Argument eine Kundenreferenz erhalten (Z. 7, 8).

Der Konnektor verfeinert die Rollenschnittstelle `BonusTeam::IBonusCollector`, indem eine davon erbende Schnittstelle `IBonusCollector` definiert wird (Z. 10). Darin wird die Operationen `collectCallPoints` deklariert.

Die eigentliche Aufgabe eines Konnektors, die Definition von Spielt–Beziehungen, erfolgt mit Hilfe der Zeichenkette `TEAMINFO` (Z. 15). Darin werden zwei Spielt–Beziehungen festgelegt. Die Rollenschnittstelle `IBonusCollector` wird der Domänenschnittstelle `Telecom::Billing::IBillingMgt` zugewiesen (Z. 16). Es wird eine 'callin'–Bindung derart definiert, daß die Operation `collectCallPoints` *nach* der Operationen `addCall` augerufen wird (Z. 19). Die

```
1 #include "BonusTeam.idl" // inheritence
2 #include "Telecom.idl" // «adapts»
3
4 module TelecomBonusTeam // seperate module !!!
5 {
6   interface IBonusTeam : BonusTeam::IBonusTeam {
7     long showPoints(in Telecom::Customer::ICustomer c);
8     long takePoints(in Telecom::Customer::ICustomer c);
9   };
10  interface IBonusCollector : BonusTeam::IBonusCollector{
11    void collectCallPoints(in BonusTeam::ISubscriber s,
12       in string destNr,
13       in Telecom::Billing::DateTimeRange range);
14  };
15  const DOT::TeamInfo TEAMINFO = ""
16    "playedBy{"
17      "roleIf{IBonusCollector}"
18      "baseIf{Telecom::Billing::IBillingMgt}"
19      "after{ roleOp{collectCallPoints} baseOp{addCall} }"
20    "}"
21    "playedBy{"
22      "roleIf{ISubscriber}"
23      "baseIf{Telecom::Customer::ICustomer}"
24      "callout{ roleOp{getId} baseOp{getId} }"
25    "}";
26 };
```

Listing 4.2: Umsetzung des `TelecomBonusTeam` in IDL.

zweite Spielt–Beziehungen bindet die Rollenschnittstelle `ISubscriber` an die Domänenschnittstelle `Telecom::Customer::ICustomer` (Z. 21) [16]. Die erwartete Operation `getid` der Schnittstelle `ISubscriber` (bzw. `IIdentity`) wird durch eine 'callout'–Bindung an die Operation `getId` der Schnittstelle `IBillingMgt` gebunden (Z. 24).

Alle erwarteten Operationen des Teams `BonusTeam` sind durch den Konnektor gebunden – der Konnektor ist vollständig.

4.4.4 Implementierung für DOT/J

Jedes Team (und damit auch ein Konnektor) befindet sich in einem eigenen Namensraum (IDL: module / Java: package) und definiert eine neue Schnittstelle. Für diese Schnittstelle muß eine Implementierung bereitgestellt werden, welche eine Vererbungsbeziehung zur Implementierung des Super–Teams aufweist.

[16] Die Angabe der Schnittstelle `ISubscriber` erfolgt nicht voll–qualifiziert, obwohl sie sich nicht in dem die Zeichenkette `TEAMINFO` umgebenen Namensraum befindet. Durch die Definition der Schnittstelle `IBonusTeam` wurde eine "Vererbungsbeziehung" zwischen den Namensräumen geschaffen, sodaß DOT die Schnittstelle `ISubscriber` finden kann.

Das Argumentkonstrukt innerhalb einer Konnektormethode 'IDomain as IRole' wird nicht von DOT unterstützt, da es nicht in IDL spezifiziert werden kann. In solchem Fall muß eine Operation definiert werden, welche den Typ der Domänenschnittstelle als Parameter erhält. An den Stellen der Implementierung, in denen der Typ der Rolle benötigt wird, muß ein explizites '*lifting*' erfolgen.

Wird in einem Team (oder Konnektor) eine neue Rollenschnittstelle definiert oder eine Rollenschnittstelle des Super–Teams durch neue Operationen erweitert, muß in der Implementierung der Teamkomponente die Fabrik–Methode `createRole` überschrieben werden. Die Implementierung der Methode `createRole` muß Instanzen der neuen Rollen erzeugen können. Die Implementierung folgt dabei immer demselben Schema: Die übergebene `RepositoryId` wird auf die in dem Team neu definierten Rollenschnittstellen verglichen und es wird ggf. eine Instanz der erzeugt, welche diese Schnittstelle implementiert (s. Kap. 4.3.5, S. 54). Wurde eine `RepositoryId` übergeben, welche nicht in diesen Namensraum gehört, muß natürlich trotzdem eine Rolle zurückgeliefert werden. Daher erfolgt standardmäßig der Aufruf `super.createRole(ifid)`, wenn vorher keine der Vergleichsbedinungen erfüllt werden konnte.

Für alle in einem Konnektor definierten Rollenschnittstellen, d.h., für alle Rollenschnittstellen, die sich im Namensraum des Konnektors befinden, muß eine Implementierung bereitgestellt werden. Jede Rollenschnittstelle wird durch *mindestens eine* Klasse realisiert, die von der entsprechend allgemeineren (Rollen–) Klasse des Super–Teams abgeleitet ist.

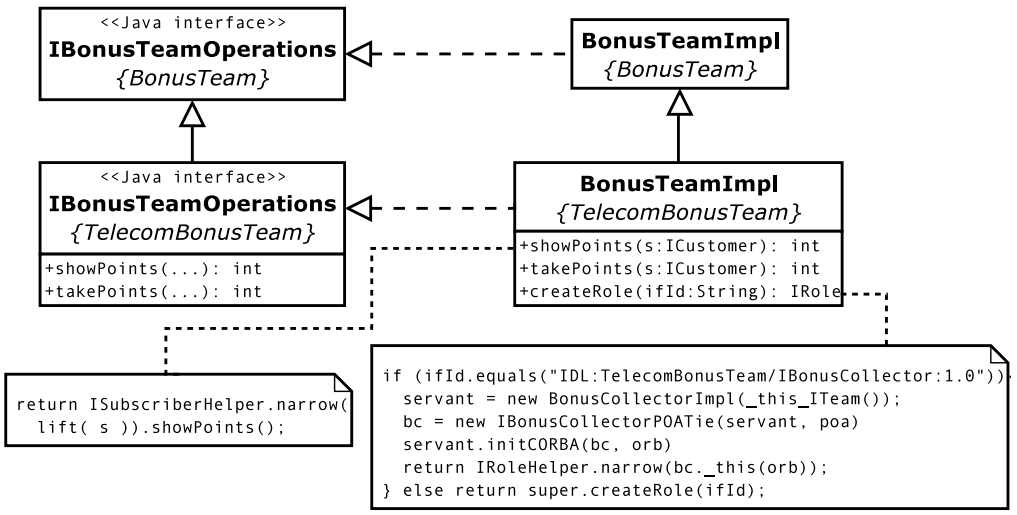


Abbildung 4.18:
Implementierung der Klasse `TelecomBonusTeam.BonusTeamImpl`.

Die Implementierung eines Konnektors soll nun wieder an dem Beispiel des `TelecomBonusTeam` verdeutlicht werden. Abbildung 4.18 zeigt die Beziehun-

gen, in die die Konnektorimplementierung `BonusTeamImpl` eingebettet ist. Die Klasse `TelecomBonusTeam.BonusTeamImpl` ist eine Spezialisierung der Klasse `BonusTeam.BonusTeamImpl` und realisiert die generierte Schnittstelle `TelecomBonusTeam.IBonusTeamOperations`.

Es werden die Methoden `showPoints` und `takePoints` implementiert, welche als Argument eine Referenz des Typs `ICustomer` erhalten. Aus den oben genannten Gründen muß daher ein explizites *lifting* der `ICustomer`–Referenz zu einer `ISubscriber`–Referenz erfolgen. Nach einem *narrowing*, der zurückerhaltenen Referenz, auf den Types `ISubscriber`, kann darauf die Operation `showPoints` bzw. `takePoints` aufgerufen werden.

Die Fabrik zur Erzeugung von Rollen (`createRole`) wird überschrieben. Die Implementierung der Methode wird erweitert, sodaß auf Anfrage hin eine Rolleninstanz erzeugt wird, die die Schnittstelle `TelecomBonusTeam::IBonusCollector` erfüllt.

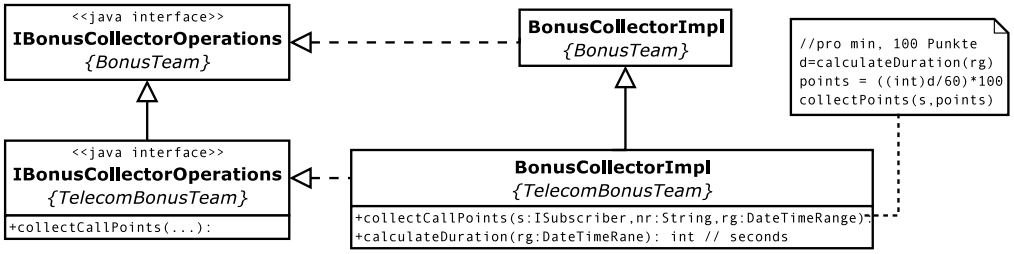


Abbildung 4.19:
Implementierung der Klasse `TelecomBonusTeam.BonusCollectorImpl`.

Die Rollenschnittstelle `TelecomBonusTeam::IBonusCollector` wird durch den IDL–Compiler in das Java–Interface `IBonusCollectorOperations` übersetzt und durch die Klasse `BonusCollectorImpl` realisiert. Diese Klasse ist eine Spezialisierung der Klasse `BonusTeam.BonusCollectorImpl`. Sie implementiert die Methode `collectCallPoints`, welche mit Hife der Methode `calculateDuration` die Dauer eines Telefongespräches ermittelt und die Zahl der Bonuspunkte als "pro Minute, 100 Punkte" errechnet. Die errechnete Punktezahl wird dem Teilnehmer durch den Aufruf der Operation `collectPoints` gutgeschrieben.

Beispielquelltext

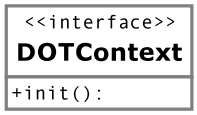
Zum Verständnis soll jetzt ein Minimalbeispiel gezeigt werden, welches das oben definierte und implementierte `BonusTeam` instanziiert und aktiviert.

Für dynamisch aktivierbare Teams muß das `DOTContext`–Objekt innerhalb jedes *thread* ein Mal initialisiert werden, es empfiehlt sich im allgemeinen, bereits am Anfang das `DOTContext`–Objekt zu initialisieren.

```
1  // initialize orb, poa, ...
2  // initialize poamanager
3  // set FORCE_MARSHAL_POLICY
4
5  // initialize DOTContext
6  DOTContext context = DOTContextHelper.narrow(
7    orb.resolve_initial_references("DOTCurrent"));
8  context.init();
9
10 // create Teamimpl & POATie & activate object
11 servant = new TelecomBonusTeam.BonusTeamImpl();
12 tie = new TelecomBonusTeam.IBonusTeamPOATie( servant );
13 servant.initCORBA(tie, orb);
14
15 // activate Team
16 iBonusTeam = tie._this(orb);
17 iBonusTeam.activate(); // customer collect for this thread
18
19 // invoke Domaincomponent
20 billingMgr = // ...
21 cutstomer = // ...
22 dateTimeRange = // ...
23 billingMgr.addCall(customer, "+49303140", dateTimeRange);
24
25 // show Points
26 System.out.println("Points:␣" + iBonusTeam.showPoints(customer));
```

Listing 4.3: Beispielquelltext einer `main`–Methode.

Listing 4.3 zeigt den Quelltext. Die Initialisierung des ORB und das Setzen der `FORCE_MARSHAL_POLICY` (s. K. 4.5, S. 72) sind im Listing nicht enthalten.



Das `DOTContext`–Objekt wird zur Startzeit des ORB geladen und kann über die Operation `resolve_initial_references` vom ORB erfragt werden (Z. 6,7). Die Initialisierung erfolgt in Zeile 8. Das `BonusTeam` wird in Zeile 11 instanziiert, der entsprechende POA dazu in Zeile 12. Danach erfolgt die Initialisierung des `BonusTeam`–Servant. Der Stub zur Schnittstelle des `BonusTeam` wird in Zeile 16 durch den Aufruf `_this` erlangt. Ab dem Aufruf `activate` (Z. 17) erfolgt das Punktesammeln für alle Kunden, die von diesem 'thread' aus einen "Anruf" tätigen (Z. 23) Der erreichte Punktestand wird in Zeile 26 vom `BonusTeam` erfragt und auf die Konsole ausgegeben [17].

[17] Beim Start des Programms muß dem ORB der `dot.ORBInitializer` übergeben werden. Initialisierungsklassen werden Java–ORBs mit Hilfe einer Kommandozeilenoption mitgeteilt:
`-Dorg.omg.PortableInterceptor.ORBInitializerClass.dot.ORBInitialzer`

```
1  // create a FORCE_MARSHAL_POLICY
2  org.omg.CORBA.Policy forcePolicy =
3  orb.create_policy(
4    org.openorb.policy.FORCE_MARSHAL_POLICY_ID.value,
5    orb.create_any());
6  org.omg.CORBA.Policy policies[] =
7    new org.omg.CORBA.Policy[]{ forcePolicy };
8  // set the policy in the policy manager
9  org.omg.CORBA.PolicyManager pm =
10   ( org.omg.CORBA.PolicyManager )
11   orb.resolve_initial_references( "ORBPolicyManager" );
12 pm.set_policy_overrides( policies,
13   org.omg.CORBA.SetOverrideType.ADD_OVERRIDE);
```

Listing 4.4:
Erzwingen des (Un-) Marshaling zwischen allen Objekten des selben ORB (Beispiel für OpenORB)

4.5 Voraussetzungen zur Adaption von CORBA–Komponenten

Distributed Object Teams ermöglicht eine Adaption von beliebigen CORBA–Objekten, wenn (1) deren Schnittstellen in IDL spezifiziert wurden und (2) diese im *'interface repository' (IR) abgelegt sind. Auch muß eine verteilte Anwendung auf der CORBA–Version 2.4 basieren (3), damit die Portable Interceptor–Architektur durch den ORB unterstützt wird. In jedem ORB des Systems muss, unter zu Hilfenahme des (4)* `dot.ORBInitializer`, ein *Client Interceptor* sowie ein *'server interceptor'* registriert werden. Dies wird in den meisten Fällen einen Neustart des Systems bedingen.

Damit alle Operationsaufrufe durch Interceptor abgefangen werden können, muß die Anwendung gewährleisten, daß Zugriffe auf Objekte ausschließlich über CORBA–Objektreferenzen erfolgen.Dies ist insbesondere für Aufrufe von Komponenten zu dessen Subkomponenten zu beachten, wenn auch diese Aufrufe abgefangen werden sollen, um damit eine Adaption zu ermöglichen. Jedes CORBA–Objekt stellt zur Referenzierung dafür die Operation `_this()` bereit. Diese muß anstelle programmiersprachenspezifischer Zeiger (häufig `this`) in der Implementierung verwendet werden. Oftmals muß der ORB durch properitäre Techniken konfiguriert werden, sodaß Aufrufe zwischen Objekten innerhalb des selben ORB, nicht aus Gründen der Performanz den Prozess des (Un-) Marhalling umgangen werden können (s. Listing 4.4).

Achtung! Bei häufigen Aufrufen von Komponentenfunktionalitäten über CORBA–Referenzen (z.B. Suchen eines Kunden anhand von Attributen), kann dies zu erheblichen Leistungseinbußen führen, da jeder Aufruf und dessen Parameter einem Marshalling– und Unmashallingprozess durch den ORB unterliegt. Daher sollte in solchem Fall (weiterhin) eine einfache, programmiersprachenspezifischer Zeiger zur Referenzierung eingesetzt werden.

4.6 Zusammenfassung

Ein Team kapselt eine Menge von kollaborierenden Aspekten, welche als einzelne Rollen modelliert werden. Die Definition eines Teams und dessen Rollen erfolgt auf der Ebene der Schnittstellen und wird mit *Teamspezifikation* bezeichnet. Darin enthalten sind ausserdem die von einer Rolle erwarteten Schnittstellen («`expected`»). Zur graphischen Modellierung eines Teams wurde das *Teamspezifikationspaket* eingeführt, dessen Umsetzung in IDL u.a. mit Hilfe der konstanten Zeichenkette `TEAMINFO` erfolgt.

Die in einem Team definierten Adaptionen werden mit Hilfe eines Konnektors auf der Ebene der Schnittstellen an Komponenten der Domäne gebunden. Diese Verknüpfung erfolgt, indem eine Rollenschnittstelle einer Schnittstellen der Domäne zugeordnet wird ("Spielt–Beziehung"). Eine Adaption des Verhalten erfolgt dort, wo sie benötigt wird: auf der Ebene der Operationen. Eine Bindung zweier Operationen kann die Form 'callout' (von der Rolle zur Basis) oder 'callin' (von der Basis zur Rolle) haben. Dabei ist eine Namensgleichheit der Operationen nicht notwendig. Auch können ggf. verschiedene Signaturen durch Weglassen, Umsortieren oder Transformieren von Parametern ineinander überführt werden. Alle definierten Adaptionen werden erst nach Aktivierung einer Konnektorinstanz in das Domänenmodell gewoben, Das *runtime weaving* erfolgt dabei auf einer Per–Klienten–Basis, d.h., jeder Klient hat eine eigene Sicht auf das System. Zur Laufzeit können Referenzen auf Schnittstellen der Domäne in Referenzen auf Rollenschnittstellen überführt werden. Dies geschieht mit Hilfe der Operationen `lift` und `lower`. Dadurch, daß zwei Schnittstellenreferenzen unterschiedlichen Typs die gleiche Komponente repräsentieren, spricht man von Translationspolymorphie.

Die Modellierung eines Konnektors erfolgt, wie Teams auch, mit einem Paketsymbol. Der Stereotyp «`adapts`» wurde eingeführt, um eine Adaptierungsbeziehung zwischen einem Konnektorspezifikationspakete und einem Domänenschnittstellenpaket herzustellen. Die Zuordnung einer Rollenschnittstelle zu einer Schnittstelle der Domäne erfolgt mit einer speziellen Form eines Klassensymbols. Im Namensfeld steht dazu ein Ausdruck in der Form "Rollenschnittstelle = Domänenschnittstelle". Um eine Bindung zweier Operationen zu modellieren, wurde ein neues Listenfeld eingeführt, welches sich unterhalb des Operationsfeldes befindet. Eine Bindung erfolgt dann mit Hilfe der Zeichen "<–" ('Callin') bzw. "–>"('callout').

Zur Realisierung eins Konnektors wurde speziell auf die in der Zeichenkette `TEAMINFO` erlaubten Schlüsselwörter eingegangen. Eine Spielt–Beziehung wird mit `playedBy` definiert; für die verschiedenen Arten von Operationsbindungen stehen die Schlüsselwörter `callout`, `before`, `after` und `replace` zur Verfügung. Für jede Bindung kann eine Menge von Parameteranpassungen definiert werden, wobei die Datenflussrichtung jedes Parameters zu beachten ist.

Fazit: Konnektoren ermöglichen die Adaption von (entfernten) Domänenkomponenten, deren IDL–Schnittstellen bekannt sind. Ein Eingreifen in den Quelltext der Komponentenimplementierungen ist dazu nicht erforderlich.

5 DOT/J Framework: Interne Struktur & Laufzeitverhalten

Die im vorherigen Kapitel geschilderten Konzepte und Schnittstellen beschreiben die Idee von "Object Teams" und die Umsetzung für verteilte (CORBA–) Systeme. Die genannten Techniken können in jeder Progammiersprache realisiert werden, sofern ein entsprechender ORB dafür existiert. Das DOT/J Framework ist eine konkrete Realisierung der DOT–Konzepte in der Sprache Java. Als zugrunde liegender ORB wurde der OpenORB [19] des *Exolab.org* verwandt.

Es soll zuerst ein kurzer Überblick der Komponentenspezifikations–Architektur des DOT/J Framework gegeben werden. In den nächsten Unterkapiteln erfolgt dann eine genaue Darstellung der zur Realisierung verwendeten Konzepte von DOT.

Abbildung 5.1 zeigt alle Komponenten des DOT/J Framework sowie deren offerierten und erwarteten Schnisttstellen.

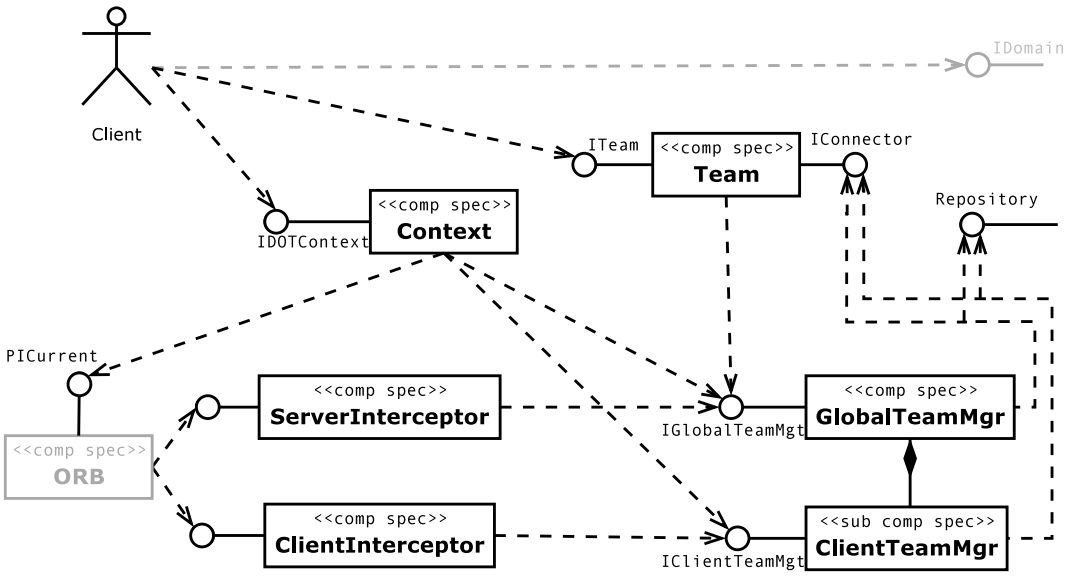


Abbildung 5.1: Komponentenspezifikations–Architektur des DOT/J Framework

Klienten nutzen die bekannten Schnittstellen `IDOTContext` zur Initialisierung eines klientenspezifischen Kontextes und `ITeam` zur (De–) Aktivierung von Teams (s. Kap. 4.4.4, S. 71). Die 'Interceptor'–Komponenten werden durch den ORB aufgerufen und leiten 'Callin'–Aufrufe zu dem `GlobalTeamMgr` bzw. einem `ClientTeamMgr` um (s. Kap. 5.4, S. 95). Alle statisch aktivierten Teams werden

durch die Komponente `GlobalTeamMgr` verwaltet, alle dynamisch aktivierten Teams durch dessen Subkomponente `ClientTeamMgr` (s. Kap. 5.1, S. 76). Beide benötigen zur Erbringung ihrer Funktionalität eine Referenz auf das CORBA–Schnittstellenverzeichnis (Schnittstelle: `Repository`) (s. Kap. 3.2.2, S. 24) und arbeiten auf einer Menge von `IConnector` Schnittstellen (s. Kap. 5.2.1, S. 91).

5.1 GlobalTeamManager

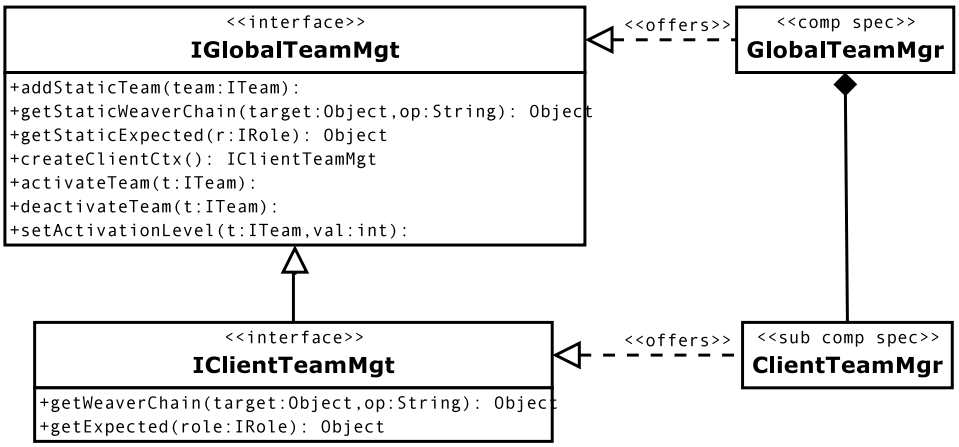


Abbildung 5.2: Schnittstelle des `GlobalTeamMgr`.

Die Hauptaufgabe des `GlobalTeamMgr` ist die Verwaltung von statisch aktivierten Teams. Um die DOT–Technologie verwenden zu können, muß sich in jedem "System" *eine* Instanz des `GlobalTeamMgr` befinden. Alle DOT/J 'server interceptor' benötigen eine Referenz auf die 'singleton' [2, Singleton] Instanz des `GlobalTeamMgr` [1].

Zur Realisierung der in DOT verwendeten "per–Klienten–Aktivierung" von dynamischen Teams besitzt der `GlobalTeamMgr` eine Menge von Subkomponenten — die `ClientTeamMgr`. Die Idee dabei ist folgende: Möchte ein Klient ('–thread') die Möglichkeit der dynamischen Teamaktivierung nutzen, so wird von ihm gefordert, die Operation `init` des `DOTContext`–Objektes aufzurufen. Das `DOTContext`–Objekt fordert daraufhin den konfigurierten [2] `GlobalTeamMgr` auf, eine neue Instanz des `ClientTeamMgr` zu erzeugen und dessen Referenz zurückzuliefern. Diese Referenz wird im `DOTContext`–Objekt vermerkt und fortan mit allen Aufrufen des Klienten ('–thread') propagiert.

[1] Zur einfachen Referenzverteilung registriert sich der `GlobalTeamMgr` während der Initialisierungsphase in dem voreingestellten *NameService*.

[2] Das `DOTContext`–Objekt des Klienten kann eine solche Referenz mit Hilfe des 'NameService' erhalten oder alternativ durch eine Kommandozeilenoption.

Die folgenden Abschnitte beschreiben im Detail die Konzepte, die zur Umsetzung des `GlobalTeamMgr` und dessen `ClientTeamMgr` verwendet werden.

5.1.1 Repräsentation von Bindungen zur Laufzeit

DOT bedient sich einer konstanten Zeichenkette `TEAMINFO`, um die erwartete Schnittstelle einer Rolle sowie die Bindungen eines Konnektors zu definieren.

Wenn eine Konnektorinstanz aktiviert wird, erfragt der `GlobalTeamMgr` dessen Schnittstelle aus dem CORBA–Schnittstellenverzeichnis. War der Typ der Schnittstelle bis dato unbekannt, läuft ein Prozess ab, in dem die `TEAMINFO`–Zeichenkette des Namensraumes, in dem sich die Schnittstelle des Konnektors befindet, mit allen `TEAMINFO`'s der Namensräume der geerbten Schnittstellen vereinigt wird. Die resultierende Zeichenkette muß zur Laufzeit des Systems ausgewertet und validiert werden. Die gewonnen Informationen werden in einer objektorientierten Datenstruktur gespeichert. Abbildung 5.3 zeigt das Klassendiagramm, aus dessen Klassen eine solche Datenstruktur erbaut ist.

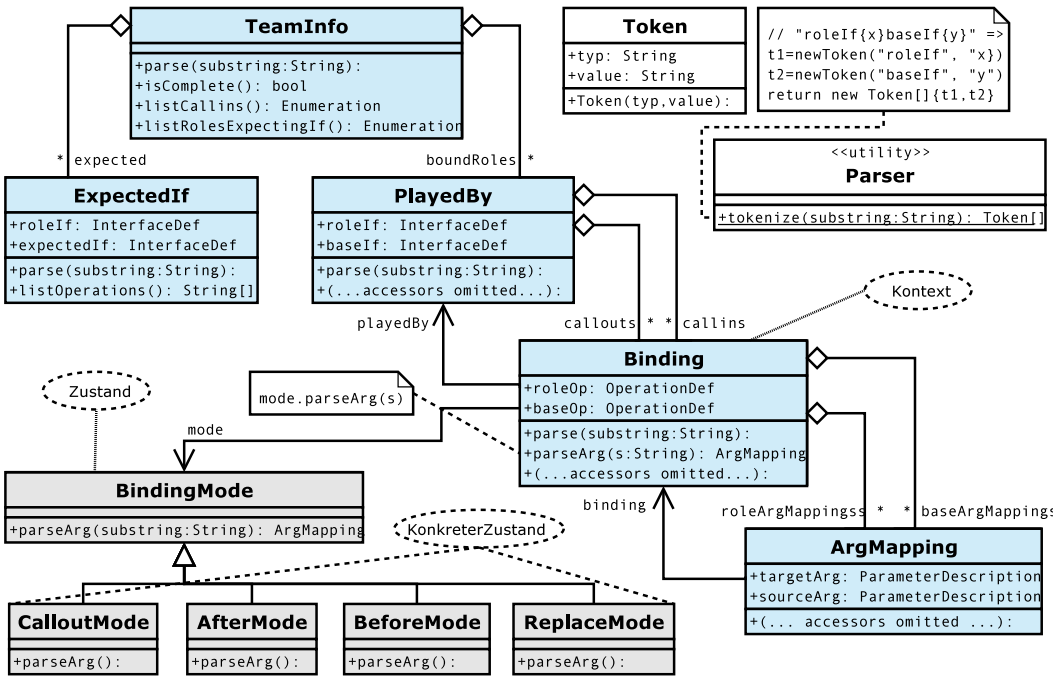


Abbildung 5.3:
Datenstruktur zur Aufnahme der in einer `TEAMINFO` deklarierten Entitäten.

Die Idee dabei ist, daß das Objekt die Auswertung der `TEAMINFO` übernimmt, welches die Informationen später repräsentiert. Dazu bietet jede Klasse die Methode `parse` an, welche die Wurzelelemente der übergebenen Zeichenkette analysiert und entsprechende Kindobjekte erzeugt. Das Kindobjekt ist dafür verantwortlich,

die restlichen Elemente der Zeichenkette auszuwerten. Alle Objekte können auf die Hilfsmethode `Parser.tokenize(...)` zur Zerlegung der Zeichenkette in `Token` zurückgreifen.

Bindungen eines Konnektors werden innerhalb der Datenstruktur durch die Klasse `Binding` repräsentiert, das Attribut `mode` definiert dabei den Bindungsmodus. Jeder Bindungsmodus ist durch eine eigenständige Klasse definiert (`CalloutMode, BeforeMode, AfterMode, ReplaceMode`). Diese Struktur wurde gewählt, da sich eine interessante Vereinfachung bei der Validierung von Argumentanpassungen ergibt. Das *Zustandsmuster* [2, Zustand] diente als Vorlage. Die Validität einer Argumentanpassung setzt nicht nur voraus, daß die Argumente in den Operationen vorhanden und richtig typisiert sind, sondern auch, daß die Datenflussrichtungen der Argumente zueinander passen. Dieses ist wiederum abhängig von der Art der Bindung. Daher erfolgt die Auswertung der Argumentanpassungen einer `TEAMINFO` innerhalb der entsprechenden Unterklasse von `BindingMode`.

Die oben beschriebene Datenstruktur wird für jeden Konnektortyp (=Namensraum) aufgebaut, wenn ein Konnektor zum ersten Mal aktiviert wird (statisch oder dynamisch).

5.1.2 Realisierung von Callin–Bindungen

Bevor auf die konkreten Techniken zur Realisierung von 'Callin'–Bindungen in DOT/J eingegangen wird, werden ein paar allgemeine Überlegungen zum Laufzeitverhalten eines mit 'Callin'–Bindungen adaptierten Systems präsentiert.

In DOT werden 'Callin'–Bindungen auf der Ebene der Operationen definiert, d.h. zu einer Operation einer Domänenschnittstelle können eine Menge von Bindungen zu Operationen einer Rollenschnittstelle definiert werden. Für 'Callin'–Bindungen wird zwischen drei Modi unterschieden ('before', 'replace' und 'after'), wobei jeder Modus eine Reihenfolge definiert, in der die Operationen der Rollen– bzw. Domänenschnittstelle aufgerufen werden. Die folgende Aufzählung zeigt die Aufrufreihenfolgen der drei Modi:

'Before'–Bindung

- Rollenoperation
- Domänenoperation

'Replace'–Bindung

- Rollenoperation (Rolle kann wiederum Domänenoperation aufrufen)

'After'–Bindung

- Domänenoperation
- Rollenoperation

Konnektoren und deren Bindungen werden statisch in IDL deklariert, d.h., es ist für jede Bindung bekannt, welches Verhalten sie adaptiert, es ist jedoch nicht bekannt, in welcher Reihenfolge die einzelnen Bindungen bearbeitet werden sollen. Dies hängt von der Aktivierungsreihenfolge der einzelnen Konnektoren ab.

`TA.IRA.b()`	$\leftarrow before$	`IDomain.m()`
`TB.IRB.r()`	$\leftarrow$	`IDomain.m()`
`TC.IRC.a()`	$\leftarrow after$	`IDomain.m()`
`TD.IRD.b()`	$\leftarrow before$	`IDomain.m()`

Tabelle 5.1: Beispiel: Vier Teams definieren 'Callin'–Bindungen auf dieselbe Operation.

Beispiel: Angenommen, es werden vier Konnektoren `TA`, `TB`, `TC` und `TD` definiert, welche die Operation `IDomain.m()` mit den in Tabelle 5.1 gezeigten 'Callin'–Bindungen belegen. Werden die Konnektoren in der Reihenfolge 1: `TA`, 2: `TB`, 3: `TC` und 4: `TD` aktiviert, ergibt sich eine Aufrufsequenz, wie sie in Abbildung 5.4 dargestellt ist. Der Aufruf der Operation `m` an der Domänenkomponente `d` wird abgefangen (s. Kap. 5.4, S. 95) und zuerst an die Rolle `rd` delegiert. Von da aus geht

der Aufruf zur Rolle `rb`. Würde die Rolle `rb` keinen *basecall* absetzen, würde der Aufruf nicht an die Rolle `ra` gelangen sondern direkt in der Rolle `rc`. Die Rolle `rb` setzt einen 'basecall' ab, sodaß der Aufruf weiter zur Rolle `ra` (letzte 'before'–Bindung) geht, von dort zur Domänenkomponente `d` und schließlich zur Rolle `rc` ('after'–Bindung).

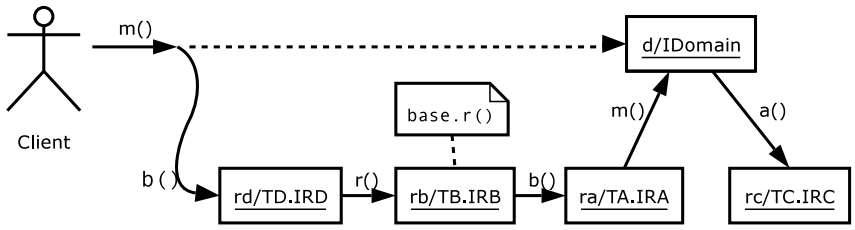


Abbildung 5.4:
Aufrufreihenfolge der Rollen und der Domänenkomponente bei einer 'Callin'–Verarbeitung der Operation `IDomain.m()`. Die Teams aus Tabelle 5.1 wurden in der Reihenfolge TA, TB, TC, TD aktiviert.

Das DOT Laufzeitsystem muß sicherstellen, daß die einzelnen Rollen in Abhängigkeit der Konnektoraktivierungsreihenfolge aufgerufen werden. Wie kann dies erreicht werden?

Konnektoraktivierungen erfolgen immer sequenziell, d.h., mit jedem neu aktivierten Konnektor wird das Verhalten der Domänenoperation wieder verändert, oder, anders gesagt, das Verhalten der Domänenoperation wird immer spezieller (s. Abb. 5.5).

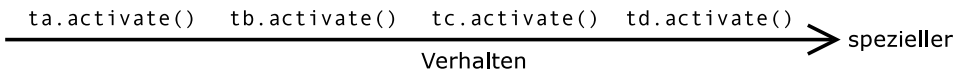


Abbildung 5.5: Verhaltensänderung durch Konnektoraktivierung

Jeder der aktivierten Konnektoren trägt seinen Teil zum spezifischeren Verhalten einer Domänenoperation bei. *Ein* Konnektor ist also zuständig für *eine* bestimmte Verhaltensänderung pro Bindung. Es bietet sich daher an, im Laufzeitsystem von DOT eine verkettete Liste von Objekten zu benutzen, in der jedes Objekt eine Zuständigkeit repräsentiert. Das Objekt, dessen Zuständigkeit die spezifischste Verhaltensänderung auslöst, steht dabei ganz vorne in der Liste.

Das Rad muß hier nicht neu erfunden werden, es gibt das objektbasierte Verhaltensmuster *Zuständigkeitskette* [2, Zuständigkeitskette], mit dessen Hilfe das gewünschte Verhalten erlangt werden kann. In einer Zuständigkeitskette wird das vorderste Objekt der Liste aufgefordert, seine Zuständigkeit zu erledigen. Dieses fordert dann (zu einem beliebigen Zeitpunkt) seinen Nachfolger auf, auch seine

Zuständigkeit abzuarbeiten. Der Prozess läuft so weiter, bis entweder ein Objekt keinen Nachfolger mehr hat (das Ende der Kette ist erreicht), oder der Nachfolger aus bestimmten Gründen nicht aufgerufen werden soll. Abbildung 5.6 zeigt die Struktur der Klassen, aus der eine Aufrufkette für 'Callin'–Bindungen etabliert wurde[3].

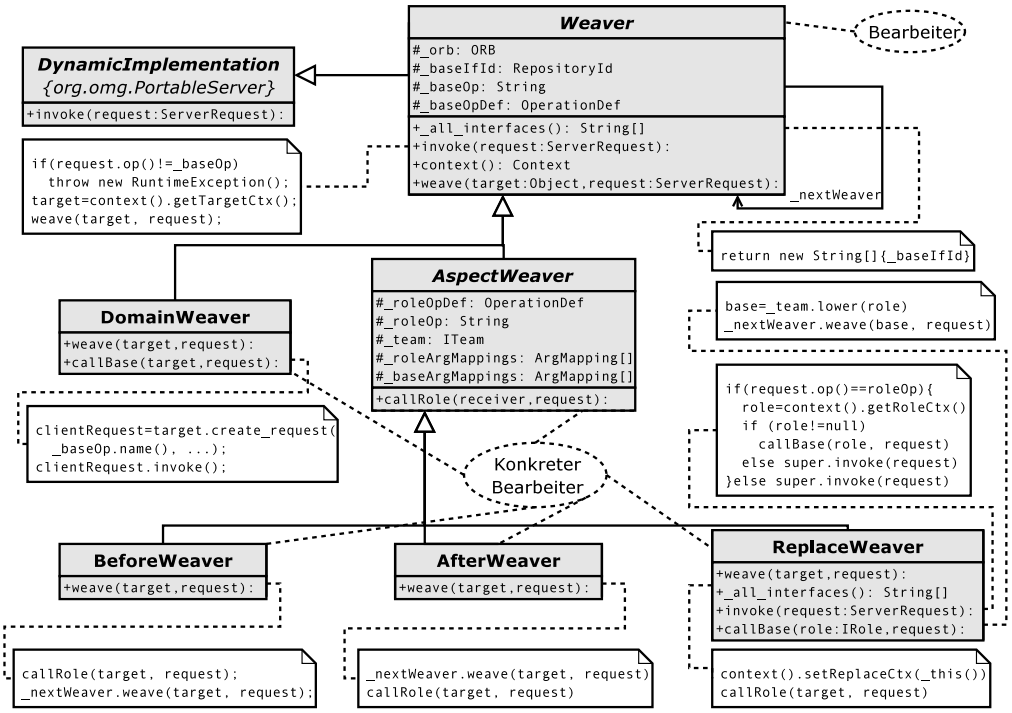


Abbildung 5.6:
Struktur der Zuständigkeitskette zur Bearbeitung von 'callin'–Aufrufen.

Die Glieder der Kette werden durch die Klasse `Weaver` (engl: Weber) repräsentiert. Der Grundgedanke dabei ist, daß jedes Glied seinen Teil der Verhaltsänderung in die Kette hineinwebt (abstrakte Methode `weave`) und anschließend seinen Nachfolger (`_nextWeaver`) aufruft. Bei einer Konnektoraktivierung werden entsprechend der im Konnektor definierten Bindungen `Weaver` erzeugt und an den *Anfang* der zur Bindung gehörenden Kette angehangen. Die Kettenglieder des zuletzt aktivierten Konnektors stehen also immer ganz vorne in den Ketten. Konnektoren können in belieber Reihenfolge deaktiviert werden. Dabei werden alle zu einem Konnektor gehörenden `Weaver` aus den Ketten entfernt.

Der erste `Weaver` in der Kette wird durch den 'Callin'–Interzeptionsmechanismus (s. Kap. 5.4, S. 95) von DOT aufgerufen. Um weitere Indirektionen zu vermeiden, ist jeder `Weaver` in der Lage, CORBA–Aufrufe direkt vom ORB entgegen zu nehmen. Die Klasse `Weaver` nutzt dazu das *Dynamic Sekeleton Interface* (DSI) (s. Kap. 3.2.4, S.

[3]Die Klassenstruktur wurde aus [25] übernommen.

30), um zur Laufzeit festzulegen, welche Schnittstellen bedient werden. Die Menge der Schnittstellen ist dabei fest auf die zu adaptierenden Domänenschnittstelle konfiguriert (Attr. `_baseIfId`). Eingehende Aufrufe werden durch die Elternklasse an die Methode `invoke` delegiert. Diese überprüft zuerst, ob der Name des Aufrufes mit dem konfigurierten Namen der Domänenoperation übereinstimmt. Anschließend wird das eigentliche Ziel des Aufrufes (vor der Interzeption) aus dem Kontext–Objekt gelesen und zusammen mit den Aufrufinformationen an die abstrakte Methode `weave` übergeben. Unterklassen implementieren in der Methode `weave` das spezifische Verhalten (s. Abb. 5.6, Notiz oben–links).

Der `DomainWeaver` webt (Methode: `weave`) durch einen Aufruf der Domänenoperation die eigentliche Domänenfunktionalität in die Gesamtfunktionalität ein. Er nutzt dazu daß durch die Methode `weave` übergebene `target`, welches eine Referenz auf die Schnittstelle der Domänenkomponente ist und mit Hilfe des *Dynamic Invocation Interface* (DII) (s. Kap. 3.2.4, S. 30) aufgerufen werden kann (s. Abb. 5.6, Notiz mitte–links). In jeder Kette befindet sich genau *ein* `DomainWeaver`. Er ist *immer* das letzte Glied in der Kette.

Ein `AspectWeaver` webt durch einen Auruf einer Rollenoperation einen "Aspekt" in die Gesamtfunktionalität ein. Wie genau dies geschieht, bleibt jedoch offen, die Methode `weave` ist nicht implementiert, die Klasse bleibt abstrakt. Die Methode `callRole` kann von einer Subklasse benutzt werden, um die in `_roleOp` definierte Operation der Rolle aufzurufen. Die Methode `callRole` erledigt die notwendigen Signaturanpassungen und ruft mit Hilfe des DII das Rollenobjekt auf.

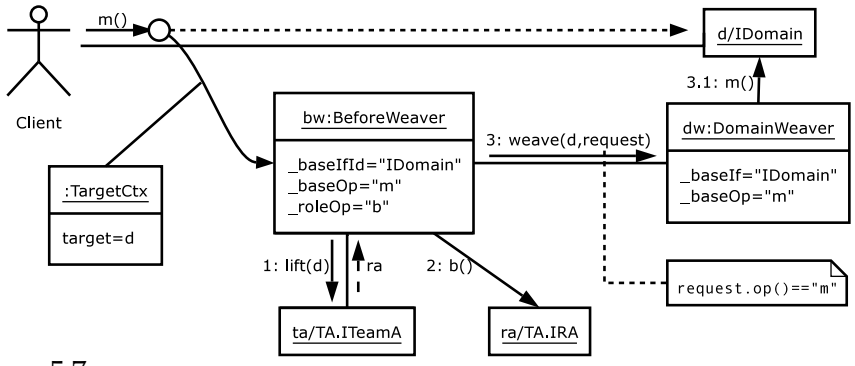


Abbildung 5.7:
Kollaborationsdiagramm zur Erläuterung der Funktionalität eines `BeforeWeaver`. Die in Tabelle 5.1 (S. 79) gezeigten Bindungen dienen als Grundlage; der Konnektor `TA` wurde aktiviert.

Die Klassen `BeforeWeaver`, `AfterWeaver` und `ReplaceWeaver` legen fest, in welcher Reihenfolge ein Aspekt dazu gewoben wird. Der `BeforeWeaver` ruft die Methode `callRole` auf, bevor er seinem Nachfolger das "Weben" überläßt (s. Abb. 5.6, Notiz unten–links). Ein `AfterWeaver` arbeitet in genau der umgekehrten Reihenfolge: Er ruft zuerst an dem nächste Glied in der Kette `weave` auf und

anschließend seine Rolle mit `callRole` (s. Abb. 5.6, Notiz unten–mitte). Abbildung 5.7 erläutert das Verhalten eines `BeforeWeaver` noch einmal am Beispiel eines Aufrufes `m` an der Domänenschnittstelle `IDomain` für den einzig aktivierten Konnektor `TA` aus Tabelle 5.1 (S. 79). Der `BeforeWeaver bw` ist auf die Schnittstelle "IDomain" konfiguriert (Attr. `_baseIf`). Eingehende Aufrufe der Operation "m" werden zuerst an die Rollenoperation "b" weitergeleitet. Anschließend wird `weave` des nachfolgenden `DomainWeaver` aufgerufen.

Der `ReplaceWeaver` ruft nur die Methode `callRole` auf (s. Abb. 5.6, Notiz unten–rechts). Allerdings kann eine Rolle einen *basecall* absetzen, d.h., die Rolle wünscht, daß die ursprüngliche Domänenoperation aufgerufen wird. Daher speichert der `ReplaceWeaver` vor dem Aufruf der Rollenoperation seine CORBA–Referenz mit Hilfe der Operation `setReplaceCtx` innerhalb des `Context`–Objektes. Initiiert die Rolle einen 'basecall', so landet der Aufruf bei diesen `ReplaceWeaver` (Rückruf). Ein `ReplaceWeaver` folgt also zwei Schnittstellen: Der der Domänenkomponente und der der Rolle. Damit der Aufruf nicht durch die Elternklasse `Weaver` an die Methode `weave` delegiert wird, ist die Methode `invoke` überschrieben. Die Implementierung der Methode `invoke` prüft zuerst, ob der Aufruf den Namen der Rollenoperation trägt. Stimmen die Namen überein, wird die Rolle aus dem Context–Objekt gelesen. Konnte eine Rolle gelesen werden, liegt ein 'basecall' vor und die Rolle wird zusammen mit dem Request–Objekt an die Methode `callBase` weiter gereicht. Diese senkt die Rolle zu ihrer Basis und ruft schließlich die Methode `weave` des Nachfolgegliedes der Kette auf. Liegt kein 'basecall' vor, wird die ursprüngliche Methode `invoke` des DSI aufgerufen.

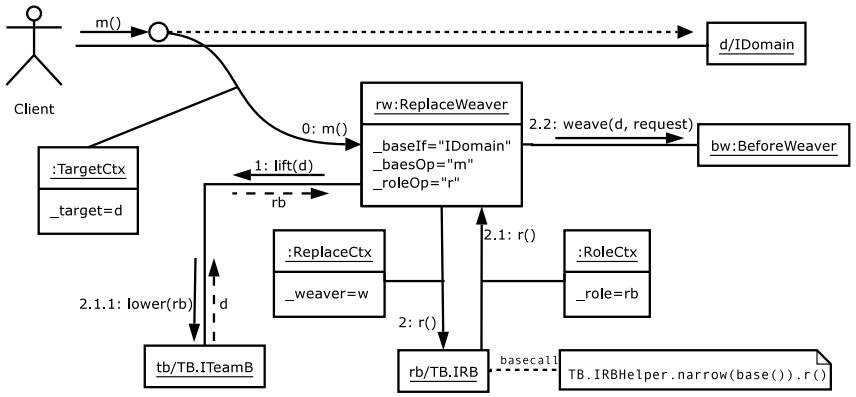


Abbildung 5.8:
Kollaborationsdiagramm zur Erläuterung der Funktionalität eines `ReplaceWeaver`. Die in Tabelle 5.1 (S. 79) gezeigten Bindungen dienen als Grundlage, der Konnektor `TB` wurde zusätzlich zum Konnektor `TA` aktiviert, das Diagramm setzt die Kette aus Abb. 5.7 fort.

Abbildung 5.8 verdeutlicht den Prozeß beispielhaft. Die darin gezeigte Kette be-

steht aus drei Elementen, deren ersten beiden im Diagramm eingezeichnet sind. Die Kette ist eine Fortsetzung der Kette aus Abbildung 5.7. Der umgeleitete Aufruf der Operation `m` gelangt zu dem `ReplaceWeaver rw`. Dieser nutzt sein Team zum Heben der (Domänen–) Referenz `d` und ruft auf der erhaltenen (Rollen–) Referenz `rb` die Operation `r` auf. Mit diesem Aufruf wird ein `ReplaceCtx` transportiert, der die Quelle des Aufrufes enthält. Nachdem der Aufruf an der Rolle angekommen ist, setzt diese einen 'basecall' ab, sodaß der `ReplaceWeaver` "zurückgerufen" wird. Die Rollenreferenz aus dem stillschweigend mitübertragenen `RoleCtx` wird durch den `ReplaceWeaver rw` genutzt, um die Rolle zu ihrer Basis zu senken. Mit der vom Team zurückerhaltenen Domänenreferenz wird schließlich an dem Nachfolger die Methode `weave` aufgerufen.

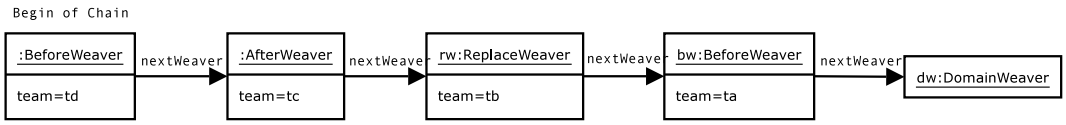


Abbildung 5.9: Für das Beispiel resultierender `WeaverChain`.

Die sich ergebene Gesamtkette für das Beispiel aus Tabelle 5.1 (S. 79) wird in Abbildung 5.9 gezeigt. Die Kette ist von rechts (Ende der Kette) nach links (Anfang der Kette) aufgebaut. Das Kettenglied des zuletzt aktivierten Teams (`td`) befindet sich daher ganz links. Die geforderte Aufrufsequenz aus Abbildung 5.4 (S. 80) wird von dieser Zuständigkeitskette erfüllt.

Aktivierung statischer Teams in Bezug auf 'Callin'–Bindungen

Der `GlobalTeamMgr` verwaltet alle Ketten, die notwendig sind, um die 'Callin'–Bindungen statisch aktivierter Teams abzubilden. Die Spitze einer jeden Kette ist dabei über die `RepositoryId` der gebundenen Domänenoperation assoziiert (vgl. Abb. 5.10).

Die Aktivierung eines Teams als "statisch" resultiert in einem Aufruf der Methode `addStaticTeam` des `GlobalTeamMgr`, welche für alle 'Callin'–Binungen des Teams Weaver erzeugt und diese an die Spitze der jeweiligen Kette setzt (s. obere Notiz in Abb. 5.10).

'Callin'–Bindungen statisch aktivierter Teams werden durch 'server interceptor' abgefangen (s. Kap. 5.4.2). Diese fragen den `GlobalTeamMgr` nach einem 'forward object' mit Hilfe der Operation `getStaticWeaverChain`. Der `GlobalTeamMgr` bestimmt aus der Schnittstelle der übergebenen Domänenreferenz und dem Namen der daran aufgerufenen Operation die `RepositoryId` der Operation und erhält dadurch den ersten Weaver der entsprechenden Kette (s. erster Teil der unteren Notiz in Abb. 5.10).

Ale problematisch erweisen sich die definierten Anforderungen von DOT: Bindun-

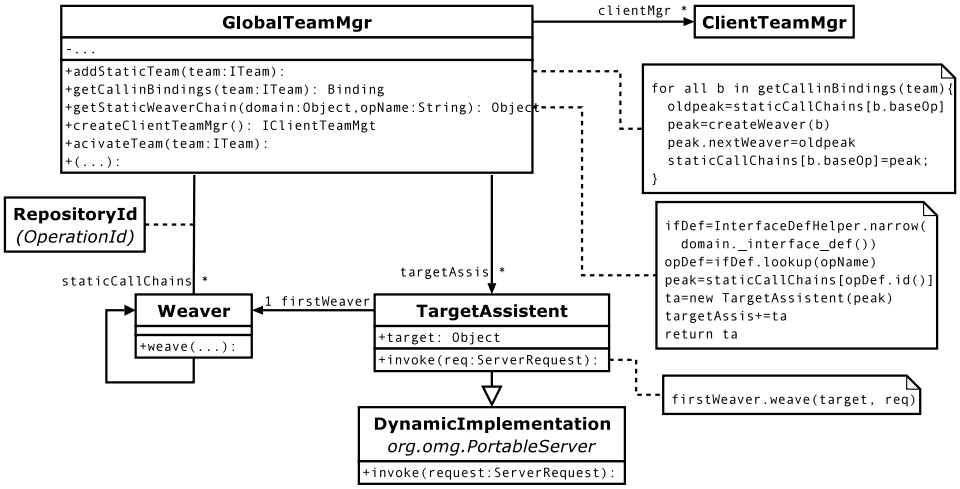


Abbildung 5.10:
Der `GlobalTeamMgr` assoziiert `Weaver` über die `RepositoryId` einer gebundenen Domänenoperation.

gen statischer Teams sollen allein durch die Installation eines 'server interceptor' auf Seiten der zu adaptierenden Domänenkomponente realisiert werden können. Es soll nicht notwendig sein, einen 'client interceptor' auf Seiten des Aufrufers zu registrieren[4]. Die Schwierigkeit mit dieser Anforderung ist, daß in diesem Fall der 'server interceptor' zwar einen Aufruf umleiten kann (`ForwardRequest`), jedoch nicht in der Lage ist, einen `TargetCtx` mit dem ursprünglichen Ziel des Aufrufes anzuhängen [5]. Ein `WeaverChain` basiert aber auf der Annahme, daß das Zielobjekt in die Methode `weave` hineingereicht wird.

Um das Problem zu lösen, wurde eine Klasse `TargetAssistent` eingeführt (s. Abb. 5.10). Fragt ein 'server interceptor' den `GlobalTeamMgr` nach einem entsprechenden `WeaverChain`, so wird nicht direkt das erste Glied der Kette zurückgegeben, sondern ein auf das ursprüngliche Ziel konfigurierter `TargetAssistent`. Der 'server interceptor' leitet den Aufruf an den Zielassistenten weiter. Dort angekommen, ruft dieser auf seinem `firstWeaver` die Methode `weave` mit dem konfigurierten `target` auf. Jetzt wird die Kette normal abgearbeitet. Es ergibt sich daher eine Struktur, wie sie Abbildung 5.11 zeigt.

[4]Für frei zugängliche Dienste mit anonymen Klienten kann u.U. nicht sichergestellt werden, daß alle Klienten den notwendigen 'client interceptor' in ihrem ORB registriert haben.

[5]Ein `ForwardRequest` fordert lediglich den ORB des Klienten auf, seinen Aufruf zu einem neuen Ziel umzuleiten. Da der klientenseitige ORB jedoch keinen DOT–'client interceptor' registriert hat, kann der notwendige `TargetCtx` nicht an die Zuständigkeitskette gelangen.

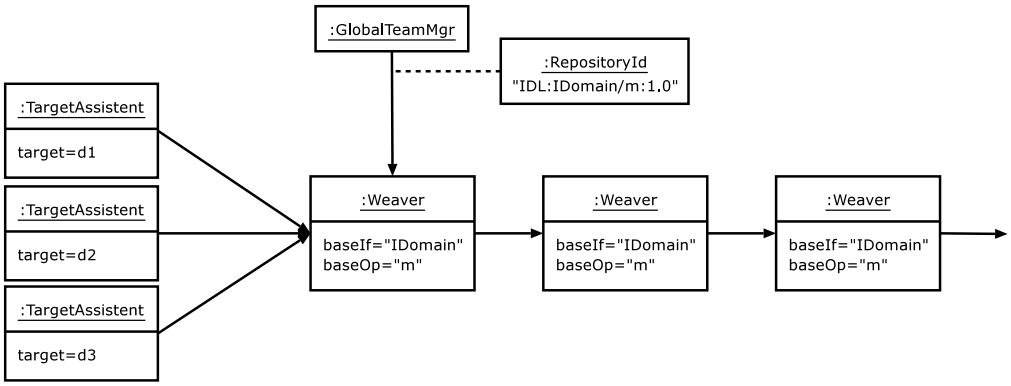


Abbildung 5.11:
Für jeden Aufruf einer Domänenoperation wird ein `TargetAssistent` erzeugt und mit dem ersten Glied einer Kette konfiguriert.

Aktivierung dynamischer Teams

Bei Aktivierung eines dynamischen Teams wird die Operation `activateTeam` des `GlobalTeamMgr` aufgerufen. Diese liest aus dem `ClientCtx` des Aufrufes den `ClientTeamMgr` aus und übergibt das Team an die Methode `activateTeam` dieses `ClientTeamMgr`.

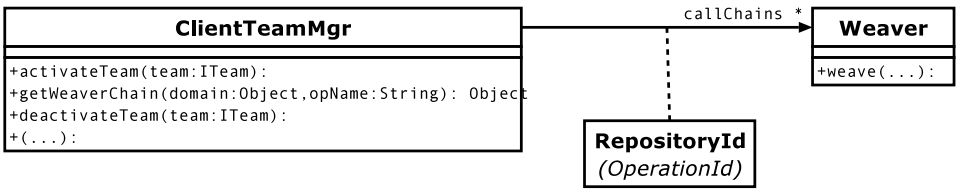


Abbildung 5.12:
Ein `ClientTeamMgr` verwaltet `WeaverChains` über die `RepositoryId`'s gebundener Domänenoperation.

Ein `ClientTeamMgr` verwaltet alle Ketten die notwendig sind, um die 'Callin'–Bindungen dynamisch aktivierter Teams für den Kontext des Klienten abzubilden. Dazu wird ein assoziativer Speicher verwendet, in dem unter der `RepositoryId` der Domänenoperation ein `WeaverChain` gespeichert ist (vgl. Abb. 5.12). Für jede aktive Bindung ist eine `WeaverChain` registriert.

Ein `ClientTeamMgr` verwaltet die Ketten, die sich aus allen dynamisch aktivierten Teams ergeben. Für eine Operation können allerdings sowohl Bindungen von statisch aktivierten Teams als auch von dynamisch aktivierten Teams vorliegen. D.h., die von einem `ClientTeamMgr` zu erbringende Funktionalität erstreckt sich über dessen eigene Ketten und die vorher (statisch) aufgebauten Ketten des `GlobalTeamMgr`. Wenn im `GlobalTeamMgr` "statische" Ketten zu einer bestimm-

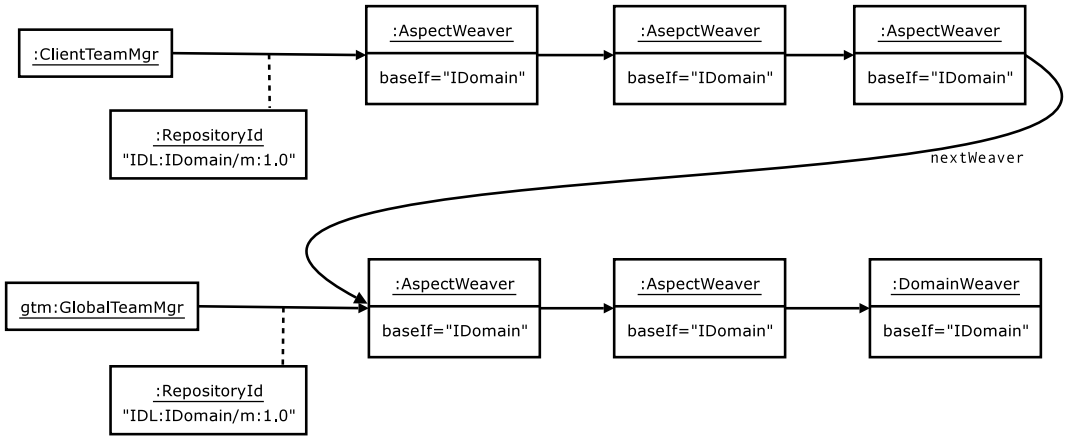


Abbildung 5.13:
Die Ketten des `ClientTeamMgr` sind mit den Ketten des `GlobalTeamMgr` verknüpft.

ten Bindung vorliegen, wird das erste Element dieser Kette an die entsprechende Kette des `ClientTeamMgr` angehangen, und zwar *anstatt* des `DomainWeaver` (letztes Element in einer Kette) (vgl. Abb. 5.13).

Jeder 'client interceptor' (s. Kap. 5.4.1) versucht zuerst, einen `ClientCtx` zu lesen, die darin enthaltene Referenz zeigt auf einen `ClientTeamMgr`. Der 'client interceptor' ruft an diesem die Operation `getWeaverChain` auf, und überprüft damit, ob für den abgefangenen Operationsaufruf eine 'Callin'–Bindung aktiv ist. Ist diese Bindung aktiv, liefert der `ClientTeamMgr` eine Referenz auf das erste Glied in der Kette und der 'client interceptor' führt einen `RequestForward` aus. Der Aufruf gelangt in die Kette, das gewünschte Ergebnis wird erbracht.

Deaktivierung dynamischer Teams

Teams können jederzeit deaktiviert werden. Die Deaktivierung eines Teams bedeutet, das alle `Weaver`, die mit dem Team assoziiert sind, aus den Ketten entfernt werden. Dabei wird der Vorgänger des entfernten Gliedes mit dem Nachfolger des entfernten Gliedes verknüpft. Ist in einer Kette nur noch ein `DomainWeaver` enthalten, so wird auch dieser entfernt und die Kette komplet aus der Liste `callChains` gelöscht.

Eine Anfrage `getWeaverChain` durch einen 'client interceptor' liefert dadurch "null" zurück, der 'interceptor' führt keinen `ForwardRequest` aus. Das ursprüngliche Verhalten der Domänenkomponente wird ausgeführt.

5.1.3 Callout–Verarbeitung

Das Gegenstück zu 'Callin'–Aufrufen sind 'Callout'–Aufrufe, der Nachrichtenfluß geht dabei von einer Rolle zu einer Domänenkomponente.

'Callout'–Aufrufe sind in DOT nur für Operationen möglich, die in einer seperaten Schnittstelle definiert wurden und diese durch eine «`expected`» Beziehung mit der Schnittstelle der Rolle verbunden ist. Ein Konnektor definiert eine `playedBy` Relation zwischen einer Rollenschnittstelle und einer Domänenschnittstelle sowie 'Callout'–Bindungen zwischen Operationen dieser Schnittstellen. Implementierungen einer erwarteten Rollenoperation müssen, um später durch einen Konnektor gebunden werden zu können, den Aufruf weiterleiten. Sie tun dies, indem sie mit Hilfe der Operation `expected` ihrer Elternklasse (`RoleImpl`) eine Referenz auf die erwarteten Schnittstellen besorgen und darauf die 'Callout'–Operation ausführen.

Eine erwartete Schnittstellen wird in DOT durch einen `InterfaceAdapter` erfüllt. Die Klasse implementiert das *Adapter*–Muster [2, Adapter] in einer leicht abgewandelten Form des *Objektadapters* (s. Abb. 5.14). Die Realisierungsbeziehung

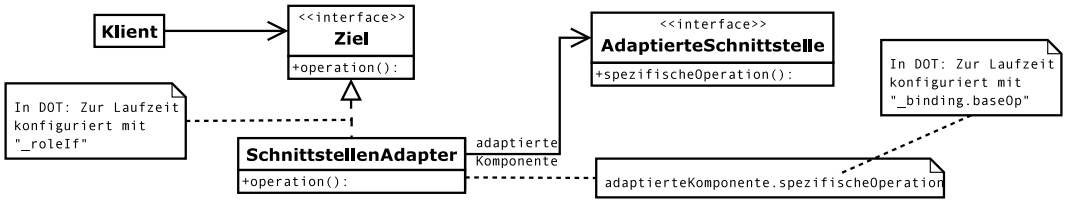


Abbildung 5.14: *Objektadapter für Schnittstellen*

zwischen der Schnittstelle `Ziel` und dem `SchnittstellenAdapter` kann zur Laufzeit konfiguriert werden. Welche Operation an `AdaptierteSchnittstelle` aufgerufen wird, kann ebenfalls zur Laufzeit bestimmt werden. Der DOT `InterfaceAdapter` ist damit ein generischer Schnittstellenadapter.

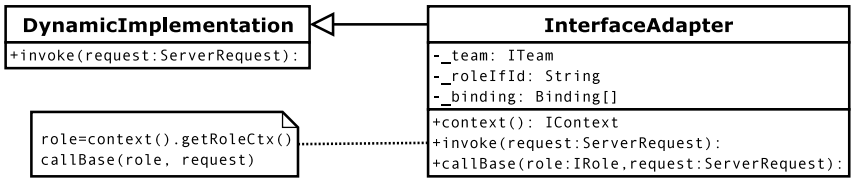


Abbildung 5.15: Die Klasse `InterfaceAdapter` ist ein *Objektadapter*.

Abbildung 5.15 zeigt die Attribute und Methoden des `InterfaceAdapter`. Über das Attribut `_roleIfId` wird die angebotene Schnittstelle konfiguriert. In `_binding` (s. Kap. 5.1.1, S. 77) sind alle Operationsbindungen und deren Vorschriften zur Signaturanpassung enthalten. Eingehende Aufrufe werden mit Hilfe des *Dynamic Skeleton Interface* (s. Kap. 3.2.4, S. 30) entgegengenommen (`invoke`) und

an die Methode `callBase` weitergereicht. Diese bittet das ebenfalls konfigurierte Team, die übergebene Rolle zu senken (`lower`) und ruft auf der zurückerhaltenen Referenz (Domänenkomponente) schließlich die in `_binding` angegebene Domänenoperation mit Hilfe des '*Dynamic Invocation Interface*' (DII) auf.

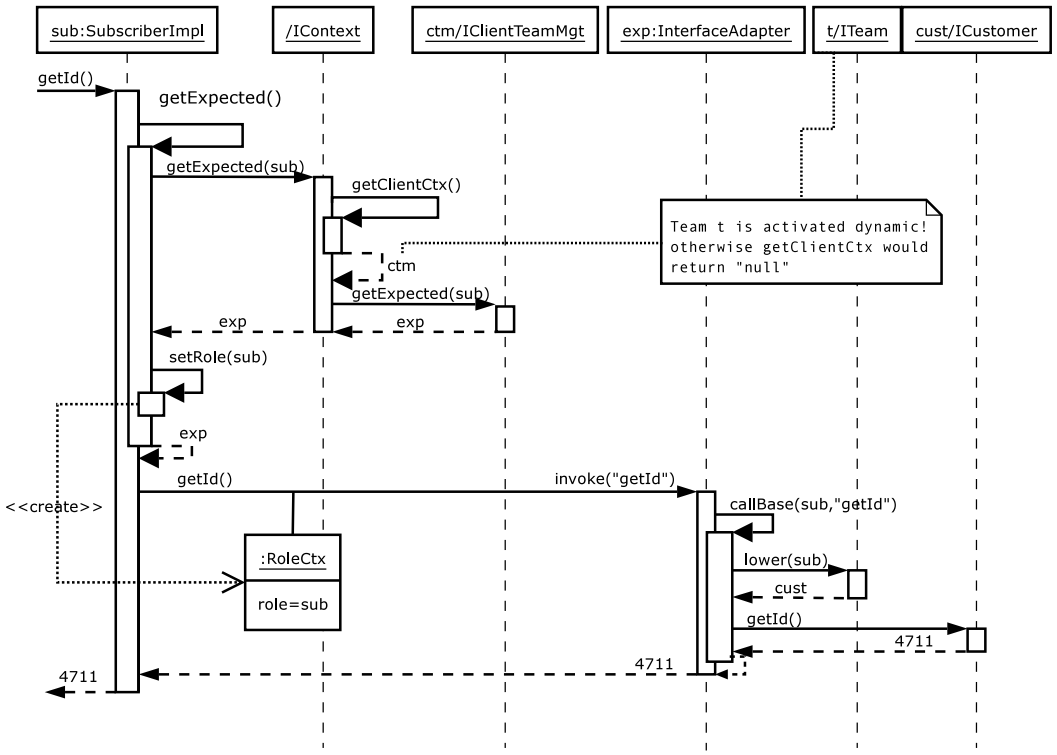


Abbildung 5.16:
Sequenz eines 'Callout'–Aufrufes am Beispiel der Operation `ISubscriber.getId()`.

Abbildung 5.16 zeigt die einzelnen Schritte eines 'Callout'–Aufrufes im Detail, am Beispiel der Rolle `ISubscriber` des `TelecomBonusTeam`. Die Sequenz beginnt mit einem Aufruf der Operation `getId`. Die Implementierung der Operation folgt dem Standardschema für 'Callout'–Implementierungen (s. Kap. 18, S. 55). Es wird zuerst die erwartete Schnittstelle mit Hilfe der Operation `expected` von der Elternklasse (`RoleImpl`) erfragt und anschließend darauf die Operation `getId` aufgerufen. An den Aufruf wird stillschweigen ein `RoleCtx` angehangen, die darin enthaltene Rolle nutzt der `InterfaceAdapter`, um über das Team eine Referenz auf die zur Rolle gehörende Domänenkomponente zu erhalten. Damit kann die Operation `getId` des Kunden aufgerufen werden.

Aktivierung statischer Teams in Bezug auf 'Callout'

Der `GlobalTeamMgr` verwaltet alle `InterfaceAdapter` von statisch aktivierten Teams. Jeder `InterfaceAdapter` ist dazu eindeutig über die *RepositoryId* der Rollenschnittstelle referenziert (vgl. Abb. 5.17). Bei einer statischen Teamak-

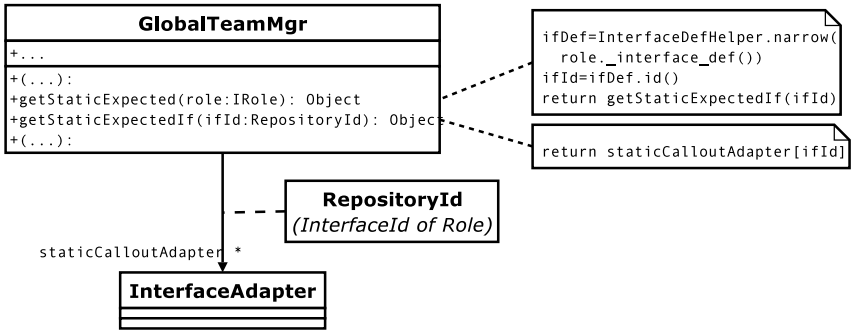


Abbildung 5.17:
Der `GlobalTeamMgr` verwaltet die `InterfaceAdapter` aller statisch aktivierten Teams.

tivierung erzeugt der `GlobalTeamMgr`, entsprechend der Bindungen des Teams, `InterfaceAdapter` für alle 'Callout'–Bindungen, und fügt sie in den assoziativen Speicher `staticCalloutAdapter` ein. Als Schlüssel dient die 'RepositoryId' der Rollenschnittstelle.

Kardinalität der `staticCalloutAdapter`: Es existiert für jede Rollenschnittstelle, für die mindestens eine 'Callout'–Bindung definiert ist, *ein* `InterfaceAdapter` innerhalb des `GlobalTeamMgr`, wenn das Team, welches die obigen Bindungen besitzt, statisch aktiv ist.

Wird von einer Rolle eine Referenz auf eine erwartete Schnittstelle angefordert, findet der `GlobalTeamMgr` den entsprechenden `InterfaceAdapter` anhand der 'RepositoryId' der Rolle und liefert den Fund zurück.

Aktivierung dynamischer Teams in Bezug auf 'Callout'

Der `ClientTeamMgr` verwaltet alle `InterfaceAdapter` eines 'client thread'. Diese sind ebenfalls über einen assoziativen Speicher mit einer *RepositoryId* verknüpft (vgl. Abb. 5.18).

Wird innerhalb eines 'client thread' ein Team aktiviert, wird der Aufruf von dem zu aktivierenden Team über den `GlobalTeamMgr` zu dem `ClientTeamMgr` weitergeleitet, welcher durch den `ClientCtx` mit dem Klienten assoziiert ist. Darin werden, entsprechend der 'Callout'–Bindungen des Teams, `InterfaceAdapter`

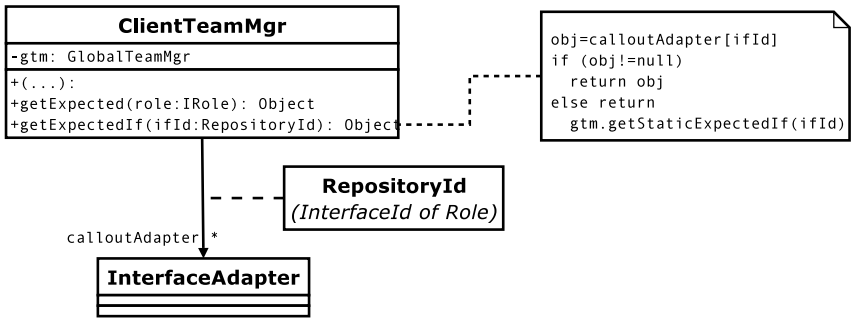


Abbildung 5.18:
Der `ClientTeamMgr` verwaltet alle `InterfaceAdapter` eines 'client thread'.

erzeugt [6] und in den assoziativen Speicher eingefügt. Als Schlüssel dient dazu die 'RepositoryId' der Rollenschnittstellen.

Wird für eine Rolle eine erwartete Schnittstelle angefordert, wird als erstes versucht, einen passenden `InterfaceAdapter` aus der Liste `calloutAdapter` zu lesen. Wurde kein passender Eintrag gefunden, wird der `GlobalTeamMgr` nach einer erwarteten Schnittstelle gefragt.

Deaktivierung dynamischer Teams in Bezug auf 'Callout'

Ein `InterfaceAdapter` wird entfernt, wenn alle dynamisch aktivierten Teams, deren Rollen aufgrund einer 'Callout'–Bindung einen `InterfaceAdapter` benötigen, deaktiviert wurden (Reduzierung des Speicherbedarfs).

5.2 Implementierung der Team–Basisklassen

Der `GlobalTeamMgr` übernimmt, wie oben gesehen, die eigentliche Funktionalität des Webens (für 'Callin') und der Schnittstellenadaption (für 'Callout'). Die Implementierungen der Klassen `TeamImpl` bzw. `RoleImpl` sind dadurch denkbar einfach. Die nächsten beiden Abschnitte zeigen die Details.

5.2.1 Klasse `TeamImpl`

Die Elternklasse aller Team–Implementierungen ist die Klasse `TeamImpl` (s. Abb. 5.19). Sie stellt Methoden zur statischen und dynamischen Aktivierung zur Verfügung und übernimmt das 'lifting' und 'lowering'.

[6] Eine Erzeugung erfolgt nur, wenn noch kein auf diese Bindung konfigurierter `InterfaceAdapter` vorliegt.

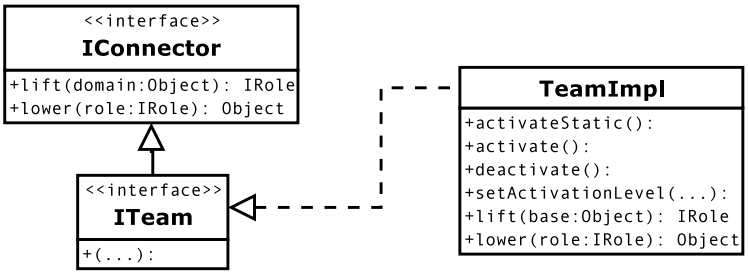


Abbildung 5.19: Die Klasse `TeamImpl` realisiert die Schnittstelle `ITeam`.

Eine Aktivierung eines Teams (statisch und dynamisch) führt innerhalb des Teams nur dazu, eine Zustandsvariable auf den neuen Zustand (aktiviert) zu setzen.

Ein Aufruf der Operation `activateStatic` wird an die Operation `activateStatic` des `GlobalTeamMgr` weitergeleitet. Als Argument erhält der Aufruf das statisch aktivierte Team. Innerhalb des `GlobalTeamMgr` werden daraufhin die entsprechenden Kettenstrukturen für 'Callin'–Bindungen um Kettenglieder, entsprechend der Bindungen des Teams, erweitert. Außerdem werden die notwendigen `InterfaceAdapter` für 'Callout'–Bindungen registriert.

Bei einer dynamischen Aktivierung wird der Aufruf ebenfalls an den `GlobalTeamMgr` weitergeleitet. Dieser ergänzt jetzt allerdings die Kettenstrukturen innerhalb des `ClientTeamMgr`, welcher dem aktivierenden Klienten zugeordnet ist und registiert darin die entsprechenden `InterfaceAdapter`.

Die Operation `lift` stellt die Funktionalität zum Heben einer Referenz auf eine Domänenkomponente bereit. Dazu wird in einem assoziativen Speicher nachgesehen, ob eine zugehörige Rolle vorhanden ist. Ist diese vorhanden, terminiert die Operation `lift` mit der Rolle als Rückgabewert. Ansonsten wird aus dem CORBA–Schnittstellenverzeichnis die IDL–Schnittstellenbeschreibung der Domänenreferenz geholt und damit die Methode `createRole` aufgerufen. Spezialisierte Teamimplementierungen müssen die Methode `createRole` überschreiben, um die nötige Funktionalität zur Erzeugung eine Rolle bereit zu stellen. Das neu erzeugte Rollenobjekt wird dem assoziativen Speicher, zusammen mit der Referenz der Domänenkomponente, hinzugefügt. [7]

Um zu einer Rolle die zugehörige Domänenkomponente zu erlangen, bietet die Klasse `TeamImpl` die Operation `lower` an. Aus einem zweiten assoziativen Speicher wird versucht, eine Referenz mit der zur Rolle assoziierten Domänenkomponente zu lesen. War die Suche erfolgreich, wird die Referenz zurückgegeben. Konnte keine Referenz gelesen werden, so ist die Rolle nicht mit dieser Team–Instanz

[7] Der Prozess ist in Wirklichkeit noch etwas komplizierter: Es kann leider nicht auf den Java–Objektreferenzen gesucht werden, da diese nur auf einen mit dem CORBA–Objekt verbundenen Stub zeigen. Gesucht werden muß aber über die Menge der CORBA–Objekte. Daher wird die in der Schnittstelle `org.omg.CORBA.Object` definierte Operation `_hash(int max)` genutzt, um ein CORBA–Objekt, unabhängig vom benutzten Stub, eindeutig zu referenzieren.

assoziiert.

5.2.2 RoleImpl

Die Implementierung der Klasse `RoleImpl` muß v.a. zwei Aufgaben erfüllen (s. auch Abb. 5.20):

Unterklassen müssen mit einer Referenz ihrer erwarteten Schnittstelle versorgt werden. Unterklassen nutzen dazu die Methoden `expected`, deren Implementierung in der mittleren Notiz in Abbildung 5.20 gezeigt ist. Die erwartete Referenz wird aus dem `IContext`–Objekt erfragt. Anschließend schreibt sich die Rolle selbt in den `RoleCtx` und gibt die erwartete Schnittstellen–Referenz zurück. Die zurückgegebene Referenz zeigt auf eine Implementierung, die der erwarteten Schnittstelle folgt (`InterfaceAdapter`, s. Kap. 5.1.3, S. 88).

Spezialisierte Klassen der `RoleImpl` können einen 'basecall' absetzen. Die Klasse `RoleImpl` bietet die Methode `base` an, um eine Referenz auf ein Objekt zu erhalten, auf dem der 'basecall' ausgeführt werden kann (`ReplaceWeaver`, s. Kap. 5.1.2, S, 80). Das Verfahren entspricht dem der Methode `expected` (s. Abb. 5.20).

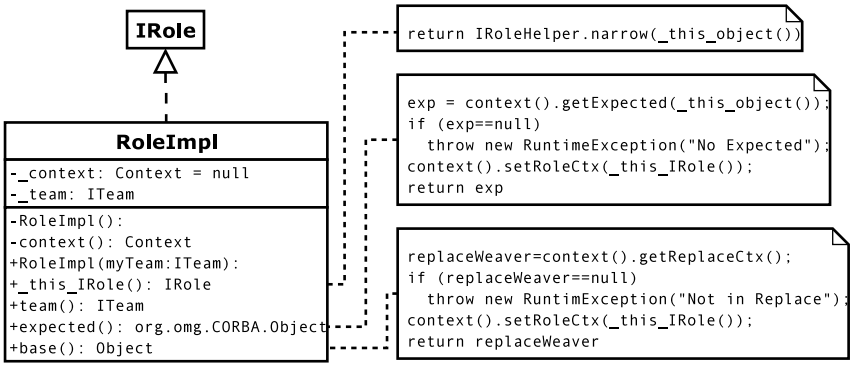


Abbildung 5.20: Implementierung der Klasse `RoleImpl`

5.3 Kontextinformationen

DOT benötigt zur Erbringung der "magischen" Funktionalität an manchen Stellen zusätzlich Informationen, zu den bereits vorhandenen, die in CORBA schon vorgesehen sind (z.B. muß eine Rollenoperation feststellen können, ob sie sich in einem 'basecall' befindet). Aufrufabhängige Informationen (*Kontexinformationen*), können in CORBA mit Hilfe eines *Current*–Objektes in den aktuellen Ausführungkontext ('thread') übertragen werden. 'Request interceptor' lesen die Informationen aus

den 'Current'–Objekten aus und hängen sie mit Hilfe eines *service context* an Operationsaufrufe an (s. auch Kap. 3.2.5, S. 34). DOT verwendet vier Dienstkontexte:

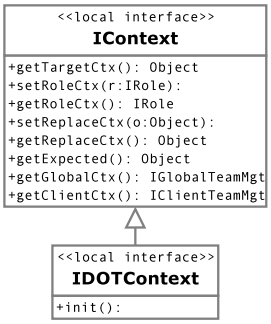
- Ein `ClientCtx` enthält eine eindeutige Identifikation des Klienten in Form einer Referenz auf einen `ClientTeamMgr`.
- Erfolgt eine Aufrufumleitung aufgrund einer aktiven 'callin'–Bindung, so wird das ursprüngliche Zielobjekt des Aufrufes in einen `TargetCtx` geschrieben.
- Wenn eine Rolle einen 'basecall' absetzt oder an der Rolle eine Operation einer erwarteten Schnittstelle aufgerufen wird, überträgt ein `RoleCtx` eine Referenz auf dieser Rolle.
- Wird eine Rollenoperation aufgerufen, für die eine 'replace'–Bindung definiert ist, so wird zur Rolle hin ein `ReplaceCtx` transportiert. Dieser enthält eine Referenz, auf der die Rolle einen 'basecall' absetzen kann.

Tabelle 5.2 faßt die Transportwege der einzelnen Dienstkontexte noch einmal zusammen.

Dienstkontext	Quelle	Ziel
`ClientCtx`	`:Object`	`:Object`
`TargetCtx`	`:ClientInterceptor`	`:Weaver`
`RoleCtx`	`:RoleImpl`	`:InterfaceAdapter`
`ReplaceCtx`	`:ReplaceWeaver`	`:RoleImpl`

Tabelle 5.2: Transportwege von DOT–Dienstkontexten.

Klienten, die die Möglichkeit der dynamischen Teamaktivierung nutzten, müssen vor der ersten Aktivierung eines Teams die Operation `init` des `DOTContext`–Objektes aufrufen. Ein solcher Aufruf ordnet einem Klienten eindeutig einen `ClientTeamMgr` zu. Eine Referenz auf diesen wird fortan mit *jedem* Aufruf propagiert.



DOT spezifische Kontextinformationen werden mit Hilfe der Schnittstelle *IContext* des *Current*–Objektes "DOTContext" gesetzt und gelesen. Das 'Current'–Objekt wird zum Zeitpunkt der ORB–Initialisierung geladen und ist danach durch die ORB–Operation `resolve_initial_references` über den Namen "DOTContext" erreichbar. Die zurückgelieferte Schnittstelle hat den Typ `IDOTContext`. Nur die darin enthaltene Operation `init` ist für Anwender der DOT–Technologie zu benutzen! Alle geerbten Operationen werden DOT–intern verwendet.

5.4 Interzeption von 'Callin'–Aufrufen

Das DOT/J Framework muß sicherstellen, daß alle definierten 'Callin'–Bindungen zur Laufzeit umgesetzt werden. Es müssen sowohl Metainformationen in Form von 'service contexts' übertragen werden (s. Kap. 5.3, S. 93), als auch alle an Domänenkomponenten gerichtete Aufrufe, für die eine Bindung definiert ist, an den entsprechenden `WeaverChain` umgeleitet werden.

Eine Möglichkeit der Umleitung von Aufrufen zu `WeaverChain` war, die Referenzen, die Klienten nutzen, um Aufrufe abzusetzen, gleich auf den entsprechenden `WeaverChain` zeigen zu lassen. Da es in CORBA ein gängiger Weg ist, Referenzen durch den *NameService* oder *TradingService* zu erlangen, müßten die dort registrierten Einträge nur durch Referenzen auf den entsprechenden `WeaverChain` ersetzt werden. Die Referenzen, die sich Klienten von diesen Diensten besorgen, würden dann auf die "richtige" Adresse zeigen. Leider hat diese Technik einige Schwachstellen: (1) Es gibt in CORBA nicht *den* Weg, Referenzen zu erlangen. D.h., es ist implementierungsabhängig, wie entfernte Objekte entdeckt werden und welche Dienste dabei zum Einsatz kommen. (2) Auch ist unklar, zu welchem Zeitpunkt eine solche Registrierung erfolgt. Eine nachträgliche Änderung einmal registrierter Referenzen ist sowohl technisch schwierig als auch semantisch: Wie wird die Gültigkeit der Referenzen überprüft, und wann wird eine Überprüfung vorgenommen? (3) Vor allem bleibt die Frage, ob wirklich alle Objekte in einem Dienst registriert sind. Kurzlebige, d.h., temporär erzeugte Objekte aber auch persistente Domänenentitäten mit Millionen von Instanzen werden sicherlich nicht in einem Standard–Dienst veröffentlicht. Schlußfolgerung: Diese Technik kann nicht sicherstellen, daß für alle Objekte eines Systems die definierten Adaptionen umgesetzt werden.

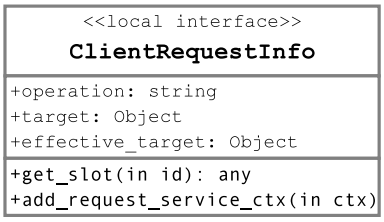
Es muß also eine Möglichkeit bestehen, alle von einem Klienten ausgehenden Aufrufe abzufangen, zu analysieren und ggf. umzuleiten. Der Vorgang des Abfangens und Umleitens von Aufrufen bedarf dabei einer Unterstützung durch den verwendeten ORB. Diese besteht seit der CORBA Version 2.4 in Form der *Portable Interceptor*–Architektur (s. auch Kap. 3.2.5, S. 33). DOT verwendet '*client interceptor*' für dynamische aktivierte Teams (s. Kap. 5.4.1) und '*server interceptor*' für statische aktivierte Teams (s. Kap. 5.4.2, S. 99).

Tabelle 5.3 zeigt die im ORB zu registrierenden 'interceptor' in Abhängigkeit der vom ORB verwalteten Komponenten.

Component instances	`ClientInterceptor`	`ServerInterceptor`
Client only	(dynamic activation)	
Server only (Domain)		(static activation)
Team only	needed	needed
`GlobalTeamMgr` only	needed	needed

Tabelle 5.3:
Notwendige 'interceptor' in Abhängigkeit von Komponenteninstanzen.

5.4.1 Client Interceptor



Ein 'client interceptor' wird bei allen ausgehenden Aufrufen eines ORB's involviert. Dabei ruft der ORB die Methode `send_request` des ClientInterceptor auf und übergibt ein `ClientRequestInfo`–Objekt. Die Schnittstelle dieses Objektes ist in verkürzter Form nebenstehend zu sehen. Das Attribute `operation` beinhaltet den Namen der aufgerufenen Operation, `target` eine Referenz auf das Zielobjekt und `effective_target` das eigentliche Ziel des Aufrufes (s. unten). Mit `get_slot` können Informationen gelesen werden, die ein Klient in sein 'thread' abhängiges Kontextobjekt geschrieben hat. Dienstkontexte ('service context') können mit Hilfe der Operation `add_request_service_ctx` an den ausgehenden Aufruf angehangen werden.

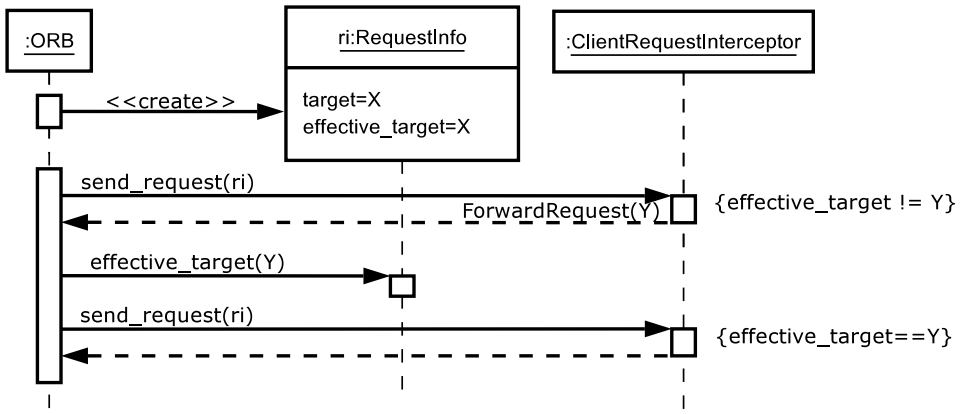


Abbildung 5.21:
Interaktionen zwischen `ORB` und `ClientRequestInterceptor`

Der Zusammenhang zwischen den Attributen `target` und `effective_target` bedarf einer besonderen Erläuterung (s. Abb. 5.21). Initial zeigen `target` und `effective_target` auf das selbe Objekt (`X`). Wirft ein 'client interceptor' die Ausnahme `ForwardRequest`, in der die "neue" Zielreferenz (`Y`) enthalten ist,

wird das Attribut `effective_target` durch den ORB auf die neue Zielreferenz gesetzt. Der 'client interceptor' wird anschließend vom ORB erneut aufgerufen. Ein Aufruf der Vergleichsoperation `is_equivalent` zwischen `effective_target` und `Y` liefert ein wahres Ergebnis

Die Referenzen innerhalb eines `ClientRequestInfo`–Objektes, `target` und `effective_target`, resultieren aus der von CORBA bereit gestellten Migrationstranzparenz (s. Abb. 3.2.3, S. 26). Die "Orginal"–Referenz muß aufbewahrt werden, da sich darin auch der orginale *object key* (s. Abb. 3.2.4, S. 32) befindet. Dieser kann, z.B., eine Kundennummer enthalten und darf somit nicht geändert werden. [8].

Aufrufinterzeption in DOT

Ein Ausschnitt des Algorhythmus zur Umleitung von Aufrufen für dynamisch aktivierte Teams wird unter Verwendung der Sprache Java in Listing 5.1 gezeigt.

(0) Zuerst wird anhand des Operationsnamens überprüft, ob der Interceptor aufgrund einer administrativen Operation aufgerufen wurde. Administrative Operationen sind z.B. `_is_equivalent` (Gleichheitstest) oder `_is_a` (Typprüfung). Solche Operationen werden nicht durch Interceptor verarbeitet.

(1) Anschließend wird der mit dem Aufruf assoziierte `ClientCtx` ausgelesen. Die Methode `getClientCtx()` greift dabei auf `ri.get_slot()` zurück, denn nur über das Request–Objekt können aufrufabhängige Informationen erlangt werden. Konnte kein `TeamContext` gelesen werden, kann auch kein Team dynamisch aktiviert worden sein und der 'interceptor' kehrt zurück.

(2) Auf der im `ClientCtx` enthaltenen Referenz des `ClientTeamMgr` wird die Operation `getWeaverChain` ausgeführt. Ist im `ClientTeamMgr` eine Bindung für die Schnittstelle des Zielobjektes und die gerade aufgerufene Operation vorhanden, liefert dieser die Spitze der entsprechenden `WeaverChain` zurück. Wurde ein solches Objekt nicht geliefert, kann der 'interceptor' zurückgehen. Vorher muß allerdings noch der *ClientCtx* eingepackt und an den Request angehängt werden.

[8] Der präsentierte Sachverhalt ist das Ergebnis einer langwierigen Versuchsreihe zur Verhaltensanalyse von CORBA–ORB's im Zusammenspiel mit Aufrufinterzeptoren. Die CORBA–Spezifikation [14] enthält leider dazu keine brauchbaren Hinweise; die Fehlermeldungen der getesteten ORB's (OpenORB, JacORB, ORBacus) sind ebenfalls nicht aussagekräftig. Die in der CORBA–Spezifikation angegebenen Wege zur Vermeidung von Rekursion sind nicht machbar. Darin wird empfohlen, eine Rekursionsmarke in einen 'slot' zu schreiben, und diese in dem nächsten Aufruf des 'interceptor' auszuwerten. Die Spezifikation definiert die 'slots' allerdings als 'readonly' und ein ORB–Referenz zum direkten Setzen des 'slot' in das `PICurrent`–Objekt ist innerhalb eines 'interceptor' nicht erreichbar!

```
1  public void send_request(ClientRequestInfo ri) throws ForwardRequest
2  {
3 (0)
4    // immediatly return on administrative operations (not shown)
5 (1)
6    // get the ClientCtx from DOTContext
7    ClientCtx ctx = getClientCtx( ri );
8    if (ctx == null) return; // no clientCtx — do nothing
9 (2)
10   // get weaver chain from ClientTeamMgr
11   org.omg.CORBA.Object forwardObject =
12     ctx.clientTeamMgr.getWeaverChain(ri.target(), ri.operation());
13   if (forwardObject == null)
14   {
15     // nothing to do, ensure only Context propagation
16     if (ctx != null) appendClientCtxToRequest( ctx, ri );
17     return;
18   }
19 (3)
20   // check, if request is already delivered to forwardObject
21   if (ri.effective_target().is_equivalent( forwardObject ))
22   {
23     // correct target set, only ensure
24     // propagation of Client and Target Ctx
25     appendClientCtxToRequest( ctx, ri );
26     appendTargetCtxToRequest( ri.target(), ri );
27     return;
28   }
29 (4)
30   // forward request to Team
31   throw new ForwardRequest( forwardObject );
32 }
```

Listing 5.1: Algorhythmus zur klientenseitigen Aufrufinterzeption

Dadurch wird sichergestellt, daß der Kontext weiterhin propagiert und damit die darin enthaltene Referenz weiteren verfügbar bleiben.

(3) Im nächsten Schritt wird überprüft, ob der Aufruf bereits das `forwardObject` als Ziel hat. Der Vergleich erfolgt über das Attribute `effective_target`. Ist der Aufruf bereits an das Team gerichtet, so wird wieder der `ClientCtx` an den Aufruf angehangen und der Interceptor kehrt zurück.

(4) Hat der Aufruf ein anderes Ziel als das `forwardObject`, wird eine `ForwardRequest`–Ausnahme ausgelöst, deren Argument das `forwardObject` ist. Die Dienstkontexte werden im zweiten Aufruf des 'interceptor' angehangen (vgl. Abb. 5.21).

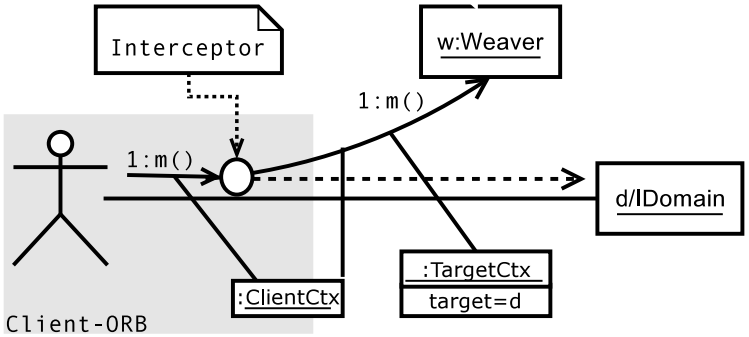


Abbildung 5.22:
Ein DOT `ClientInterceptor` leitet 'Callin'–Aufrufe zu einem `WeaverChain` um und erledigt die Propagation von Dienstkontextobjekten.

5.4.2 Server Interceptor

Aufrufe, für die eine 'Callin'–Bindung innerhalb statisch aktivierter Teams definiert sind, werden von '*server interceptor*' abgefangen und verarbeitet. Dazu wird innerhalb jedes ORB's, der eine zu adaptierende Domänenkomponente beherbergt, mit Hife des `DOT.ORBInitializer` ein `ServerInterceptor` registriert.

Der ORB ruft bei eingehenden Aufrufen die Operation `receive_request` des `ServerInterceptor` auf und übergibt dabei ein `ServerRequestInfo`–Objekt, in welchem die Zielobjektreferenz und der Name der aufzurufenden Operation enthalten ist. Der `ServerInterceptor` überprüft zuerst, ob der Aufruf nicht bereits durch einen `ClientInterceptor` an die "richtige" Adresse umgeleitet wurde[9]. Anschließend wird mit Hilfe der Operation `getStaticWeaverChain` der im `ServerInterceptor` konfigurierte `GlobalTeamMgr` nach einem `WeaverChain` für die aufgerufene Operation und für das Zielobjekt gefragt. Wurde ein 'chain' geliefert, wird als nächstes überprüft, ob der Aufruf bereits den 'chain' zum Ziel hat. Wenn nicht, wird der Aufruf durch den Wurf einer `ForwardRequest`–Ausnahme, mit dem 'chain' als Argument, umgeleitet. Ansonsten läßt der 'interceptor' den Aufruf passieren, überträgt allerdings vorher noch alle mit dem Aufruf mitpropagierten 'service contexts' in das `Context`–Objekt des 'server thread'.

Ein weitere Aufgabe des `ServerInterceptor` ist es, bei einem Aufruf von *team level* Operationen das Ziel–Team vorher zu aktivieren, so es nicht bereits statisch aktiv ist. Kehrt ein solcher Operationsaufruf zurück, deaktiviert der `ServerInterceptor` das Team. Der Zusammenhang zwischen einem Aufruf und dessen Rückkehr kann aus Sicht des `ServerInterceptor` an der vom ORB erzeugten *request–id* erkannt werden.

[9] Ist ein `TargetCtx` vorhanden, wurde der Aufruf bereits durch einen `ClientInterceptor` verarbeitet.

5.5 Verteilungskonzepte

Abbildung 5.23 zeigt die maximal mögliche Verteiltheit der einzelnen Komponenten des DOT/J Framework und der Domäne, wobei die Rechnergrenzen mit einem Kubus symbolisiert sind[10]. Die Verbindungslinien geben die Richtungen der Datenflüße zwischen den Komponenten an; die Stärke symbolisiert die Häufigkeit von Nachrichten.

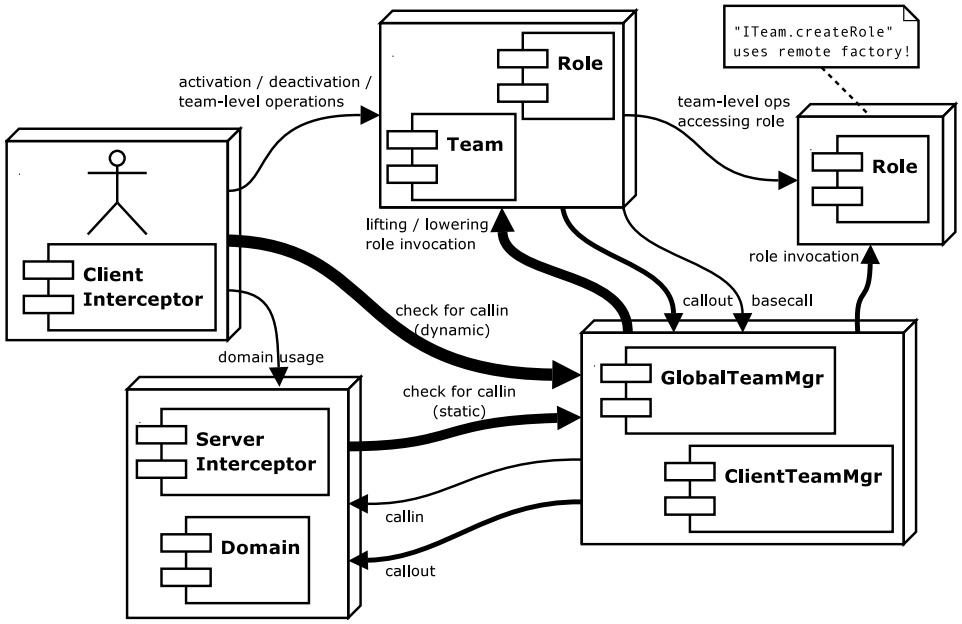


Abbildung 5.23:
Maximale Verteiltheit der Komponenten und deren Interaktionen.

Verteiltheit Klienten und deren benutzte Domänenkomponenten können sich auf unterschiedlichen Rechnern befinden. Befinden sich Klient und Domänenkomponente jedoch innerhalb des selben ORB, gelten die in Kap. 4.5 (S. 72) aufgeführten Bedingungen.

Der `GlobalTeamMgr` ist auf einen einzelnen Rechner auslagerbar. Allerdings befinden sich alle von diesem erzeugten `ClientTeamMgr` auf dem selben Rechner.

Teams und dessen Rollen bilden eine Einheit. Solche Einheiten können sich auf verschiedenen Rechnern befinden. Der Grad der Verteiltheit kann noch erhöht werden: Dazu wird die Rollenfabrik (s. Kap. 4.3.5, S. 54) derart implementiert, das die Rollenerzeugung an eine entfernte Rollenfabrik weitergeleitet wird. Welche entfernte

[10] Rechnergrenzen als Grundlage der Verteilung sind für DOT nicht entscheidend. Die eigentliche Grenze ist durch den ORB definiert, der die einzelnen Komponenten verwaltet.

Rollenfabrik dabei zum Einsatz kommt, kann natürlich zur Laufzeit entschieden werden. Es ist also möglich, Rollen, unabhängig von ihren Teams, über ein System zu verteilen.

Interaktionen Wie in der Abbildung eingezeichnet, ist die Menge der Interaktionen zwischen dem `ClientInterceptor` des Klienten bzw. der Domänenkomponente und der Instanz der `GlobalTeamMgr`–Komponente am größten. Der Sachverhalt resultiert aus der Notwendigkeit, das für jeden Aufruf überprüft werden muß, ob eine 'Callin'–Bindung vorliegt. Gerade diese Interaktionen könnten jedoch unter Verwendung einer 'cache'–Strategie deutlich reduziert werden. Da ein entfernter Aufruf ca. 1000 mal langsamer ist als ein programminterner Aufruf, kann selbst ein langsamer Such–Algorhythmus zur Auffindung der 'cache'–Informationen noch ein Gewinn sein.

Eine ebenfalls große Menge an Interaktionen findet zwischen der Komponenteninstanz des `GlobalTeamMgr` bzw. dessen `ClientTeamMgr` und den Team–Komponenteninstanzen statt. Jeder `Weaver` muß eingehende Referenzen zu der entsprechenden Rollenreferenz 'liften' und ggf. auch alle Argumente während der Signaturanpassung. Eine Optimierung mit Hilfe eines 'caches' ist auch hier möglich. Ein 'lifting' oder 'lowering' durch ein entferntes Team müßte dann nur noch für Referenzen erfolgen, die sich bis dato nicht im 'cache' befinden. Zur Steigerung der Leistungsfähigkeit ist es sicherlich angebracht, den `GlobalTeamMgr` und dessen im ORB registrierten 'interceptor' in "schnellen" Sprachen, wie z.B. C++, zu implementieren, da von dieser Komponente u.U. alle Aufrufe von Domänenkomponenten, bearbeitet werden müssen. Zu überlegen wäre auch, die `ClientInterceptor` ebenfalls über entfernte Fabriken zu erzeugen. Dadurch könnt, zumindest für alle dynamisch aktivierten Teams eines Klienten, der "zurückgelegte Weg" der Nachrichten zwischen dem 'client interceptor' und dem `ClientTeamMgr` deutlich reduziert werden. Alternativ dazu könnte das System auch so konfiguriert sein, daß die Abarbeitung der 'Callin'–Ketten eines Klienten immer auf dem am wenigsten ausgelasteten Rechner erfolgt.

6 Fazit

In dieser Arbeit wurden die Konzepte aus den drei Bereichen

- Komponententechnologie
- Verteilte Systeme
- Aspektorientierter Programmierung

vorgestellt und miteinander verknüpft.

Die Komponententechnologie ist in vielen Bereichen sinnvoll einsetzbar, denn sie hilft, Systeme in ihrer Ganzheit zu betrachten und modular aufzubauen, so daß Änderungen zu einem noch nicht absehbaren Zeitpunkt mit abschätzbaren Aufwand integrierbar sind.
Im Gegensatz zur Theorie zeigt sich in der Praxis, daß die klar spezifizierten Abhängigkeiten bisher durch wirtschaftliche Interessen sowie den Faktor "Mensch" weniger nachvollziehbar geworden sind und sein werden und dennoch einer Lösung zugeführt werden können.

Da kommerzielle Anbieter überwiegend ihre favorisierten Technologien aufgrund ihrer marktbeherschenden Stellung verbreiten, verzögert dies den generellen Prozeß des kooperativen Miteinanders und der daraus resultierenden "offenen Standards" auf hohem technologischen Niveau.

Zur Realisierung von verteilten Systemen wurde die CORBA–Technologie vorgestellt; deren Bewertung ergibt, daß CORBA als Infrastruktur für Komponenten gut geeignet ist, wenn die Spezifikation um eigene Komponentenstandards erweitert wird. Erstaunlich ist und bleibt die Tatsache, daß die bereits seit zehn Jahren existierende, komplexe CORBA–Spezifikation genügend Freiraum läßt und zusätzlich Techniken bereitstellt, die zur Integration der für aspektorientierte Programmierung notwendigen Laufzeitkonstrukte erforderlich sind.

Die aspektorientierte Programmierung wurde in der konkreten Ausprägung des 'Object Teams'–Modells eingeführt und auf verteilte, komponentenbasierte Systeme übertragen.

'Distributed Object Teams' bietet die Möglichkeit der Adaption von Komponenten, die u.a. verwendet werden kann, um:

- Anforderungsänderungen in Altsystemen elegant zu realisieren
- Funktionale Lücken von kommerziellen Komponenten zu schließen

Diese Lücken können durch 'Distributed Object Teams' nicht–invasiv geschlossen werden, anstatt wie bisher nur durch invasive (=aufwendige) Techniken. Die technische Realisierung erfolgt dabei über einen "Konnektor", der bereits verwendete "glue" Techniken um semantische Eigenschaften erweitert.

'Distributed Object Teams' bietet zusätzlich die Möglicheit, daß die Aktivierung und Deaktivierung von Aspekten zur Laufzeit erfolgen kann und damit ein Kontext geformt wird, der zur Modellierung von kontextsensitiven Informationssystemen erforderlich ist.

Zukunftsträchtig ist (eine der interessantesten Fragen), wie sich die Komponententechnologie generell entwickeln wird und ob darüber hinaus die wissenschaftlich / theoretischen Erkenntnisse der aspektorientierten Programmierung in die industrielle Anwendung eingehen und ob diese mit der Komponententechnologie verknüpft werden wird.

Danksagung

Meinem Betreuer, Herrn Dr. Stephan Herrman, möchte ich für seine konstruktiven fachlichen Ratschläge und sein stets offenes Ohr danken. Weiterhin danke ich Matthias Veit für die freundschaftliche Unterstützung bei der Verwirklichung dieser Arbeit, meiner Mutter, Reinhild Hoffer, für die grammatikalischen Korrekturen, und Joseph Bauer für den unbelasteten Blick auf die Arbeit und die sich daraus ergebenden Anregungen.

Für alles andere möchte ich Juliane Freudl ganz herzlich danken.

Literaturverzeichnis

[1] BAUER, JOSEPH: *Identification and Modeling of Contexts for Different Information Scenarios in Air Traffic*. , Technische Universität Berlin, März 2003.

[2] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON JOHN VLISSIEDES: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1995. Deutsche Übersetzung von Dirk Riehle.

[3] GORINSEK, J., S. VAN BAELEN, Y. BERBERS K. DE VLAMINCK: *EMPRESS: Component Based Evolution for Embedded Systems*. in ECOOP 2002 Workshop on Unanticipated Software Evolution (USE2002) G. Kniesel, P. Costanza and M. Dimitriev (eds.), http://joint.org/use2002/, June 2002.

[4] HERRMAN, STEPHAN: *Composable designs with UFA*. , Workshop on Aspect-Oriented Modeling with UML – (1st International Conference on Aspect-Oriented Software Development) (icse), 2002. Reviewed position paper.

[5] HERRMAN, STEPHAN: *Object Teams: Improving Modularity for Crosscutting Collaborations*. , Proc. of Net.ObjectDays, Erfurt, 2002. Forschungsbericht.

[6] HORST LANGENDÖRFER, BETTINA SCHNOR: *Verteilte Systeme*. Carl Hanser Verlag München, 1994. ISBN: 3-446-17468-0.

[7] ISO: *Reference Model Open Distributed Processing*. , International Organization for Standardization, ftp://ftp.dstc.edu.au/pub/arch/RM-ODP/.

[8] JOHN CHEESMAN, JOHN DANIELS: *UML Components*. Addison-Wesley, 2001. ISBN: 0-201-70851-5.

[9] KURT C. WALLNAU, SCOTT A. HISSAM, ROBERT C. SEACORD: *Building Systems from Commercial Components*. Addison-Wesley, 2001. ISBN: 9-780201-700640.

[10] MEYER, BERTRAND: *Objektorientierte Softwareentwicklung*. Hanser-Verlag, 1988. ISBN: 3-446-15773-5.

[11] MICHI HENNING, STEVE VINOSKI: *Advanced CORBA Programming with C++*. Addison Wesley, 1999. ISBN: 0-201-37927-9.

[12] MICO.ORG, www.mico.org: *mico is CORBA*, 2001. Mico ist ein freier C++ ORB (Unix und Windows, Aktuelle Version 2.3.8).

[13] OMG: *C++ Language Mapping Specification*. , Object Management Group, www.omg.org, June 1999. Dokument: formal/99-07-42.

[14] OMG: *The Common Object Request Broker: Architecture and Specification*. , Object Management Group, www.omg.org, May 2002. Version 2.6.

[15] OMG: *Corba Components*. , Object Management Group, www.omg.org, June 2002. Version 3.0.

[16] OMG: *IDL to Java Language Mapping Specification*. , Object Management Group, www.omg.org, August 2002. Version: 1.2, Dokument: formal/02-08-05.

[17] OMG: *Online Upgrades*. , Object Management Group, www.omg.org, July 2002. Draft Adopted Specification.

[18] OMG: *Unified Modeling Language Specification*. , Object Management Group, http://www.omg.org/cgi-bin/doc?formal/03-03-01 (letzter Zugriff: April 2003), 2003. Version 1.5.

[19] OPENORB, COMMUNITY: *OpenORB*. openorb.sourceforge.net, 2002. OpenORB ist ein freier Java ORB (aktuelle Version 1.3.0). Ursprünglich entwickelt von exolab.org.

[20] OSGI: *OSGi Service Platform*. , Open Services Gateway Initiative, www.osgi.org, Oct 2001. Release 2.

[21] ROBERT ORFALI, DAN HARKEY, JERI EDWARDS: *Instant CORBA*. Addison-Wesley, 1999. ISBN: 3-8273-1325-2.

[22] SUN: *Enterprise JavaBeans Documentation*. , Sun Microsystems, 2002. WEB: http://java.sun.com/products/ejb/docs.html (letzter Zugriff: März 2003).

[23] SYZPERSKI, CLEMENS: *Component Software*. Addison–Wesley, 1999. ISBN: 0-201-17888-5.

[24] U.A., EDDY TRUYEN: *Dynamic and Selective Combination of Extensions in Component–Based Applications*. , DistriNet, Dept. of Computer Science, K.U. Leuven, http://www.cs.kuleuven.ac.be/ eddy/lasagne.html.

[25] VEIT, MATTHIAS: *Evaluierung Modularer Softwareentwicklung mit "Object Teams" am Beispiel eines Projektmanagementsystems*. , Technische Universität Berlin, Dezember Dez. 2002.

Abbildungsverzeichnis

Listings

Index