Das Rollenkonzept der Objektorientierten Programmierung als ein Service für CORBA

Christian Niederhuber
Franz Noll

Inhaltsverzeichnis

1 Motivation

1.1 Programmiersprachenentwicklung im Wandel der Zeit

Nach der Software-Krise der 60er-Jahre, als es aufgrund der fehlenden Abstimmung innerhalb der Software-Entwicklungs-Projekte und der unzureichenden Dokumentation Probleme mit der Aufwandsschätzung und Wartung von Software gab, suchte man nach Methoden und Vorgehensweisen der „Strukturierten Programmierung“. Man betrachtete Software-Entwicklung als Produktionsprozeß, in dem Module nach genau festgelegten Spezifikationen isoliert vom konkreten Einsatzbereich hergestellt werden sollten.

Im Laufe der Zeit wurden die Grenzen dieser Strukturierung deutlich:

- nicht alle Anforderungen können in zufriedenstellendem Maße formalisiert werden
- lange Kommunikationswege zwischen Programmierer und User
- mangelnde Flexibilität

Deshalb wurde vermehrt versucht, den User durch Prototyping-Strategien, etc. in den Entwicklungsprozeß einzubeziehen und Software-Entwicklung als Designprozeß zu betrachten, in dem das Umfeld der abzubildenden Abläufe reorganisiert wird. [Info&Ges95]

1.2 Objektorientiertes Paradigma

Die Erkenntnis, daß nicht alle Details in streng strukturierten Formen darstellbar sind, führte zum Objektorientierten Paradigma, in dem ein Modell der realen Objekte anstelle von starren Abläufen in den Vordergrund tritt: Die Trennung von Daten und Verhalten wird zugunsten einer Kapselung in agierende Objekte aufgegeben; jeder Zugriff auf die Daten eines Objektes erfolgt nur von diesem selbst aus mit Hilfe von definierten Zugriffsmethoden. [Booch94], [Shlaer88], [Goldberg89], [Wirfs-Brock90]

Es ist zu beobachten, daß sich dieser Paradigmenwechsel parallel zu wesentlichen gesellschaftlichen Veränderungen vollzog: Herrschte zur Zeit der „Strukturierten Programmierung“ die Vorstellung vor, die Welt sei durch genaue Analyse ihrer Einzelheiten beherrschbar, so entstand das Objektorientierte Paradigma in einem Klima der gesamtheitlichen Sichtweise von komplexen Systemen.

Der Versuch dieses Abbild so realitätsnah wie möglich zu machen, führte unter anderem zu dem Bedürfnis, Objekte in unterschiedlichen Rollen darzustellen: Diese

Rollen sollten jeweils spezifische Eigenschaften der Objekte (Daten und Verhalten im Sinne des Objektorientierten Paradigmas) kapseln.

Das Ziel dieser Kapselungen ist es, wiederverwendbare Software-Bausteine (Software-Reuse) zu schaffen, die ohne großen Adaptierungsaufwand in neuen Projekten wiederkehrende Probleme lösen können. Durch Vereinbarung über Schnittstellen und Funktionalitäten wird ein hoher Grad an Wiederverwendbarkeit von abstrakten Software-Bausteinen erreicht. Mit Hilfe der Grundkonzepte Klassifizierung, Kapselung und Vererbung wird die gesamte Programmlogik in „lebenden" Objekten abgebildet.

1.3 Einsatz von verteilten Objekten

Daneben entwickelte sich durch den Fortschritt in der Netzwerk-Technologie die Möglichkeit, viele verbundene Einzelsysteme sinnvoll als ein Gesamtsystem zu betrachten und Applikationen dadurch mehrere unterschiedliche Objekt-Räume zugänglich zu machen. Da davon ausgegangen werden kann, daß in einem solchen Gesamtsystem die Einzelsysteme oft auf verschiedenen Plattformen liegen, entsteht der Bedarf, heterogene Objekt-Räume in einen gemeinsamen virtuellen Objekt-Raum zu integrieren, in dem über eine einheitliche Schnittstelle kommuniziert werden kann.

Das Ziel dieser Integrationen ist es, daß sich mehrere Applikationen ein konkretes Objekt teilen – somit werden Objekt-Instanzen wiederverwendet.

1.4 Einführung des Begriffs „Subjekt"

Mehrere Applikationen, die sich ein konkretes Objekt teilen, betrachten dieses Objekt im allgemeinen in unterschiedlichen Kontexten – das Objekt wird also wiederverwendet und in verschiedenen Rollen angesprochen.

Ein Objekt mit der Gesamtheit seiner Rollen kann als Subjekt bezeichnet werden und vereint so die Ideen des Software-Reuse mit den Ideen der verteilten Objekte.

In der vorliegenden Arbeit soll eine Entwicklungsumgebung geschaffen werden, in der verteilte Objekte kontextübergreifend wiederverwendet und dadurch Subjekte verwaltet werden können.

2 Rollenkonzept der Objekt-orientierten Programmierung

2.1 Motivation

Objekte, die im klassischen Objektorientierten Paradigma modelliert werden, weisen einige wesentliche Defizite auf, die die Grenzen des Objektorientierten Paradigmas aufzeigen. An diesen Grenzen will das Rollenkonzept einige entscheidende Erweiterungen anbringen.

Während des Abstraktionsprozesses können für jede Klasse von Objekten unterschiedliche Kriterien gefunden werden, an Hand derer ihre Instanzen unterschiedlich klassifiziert werden können [Booch94].

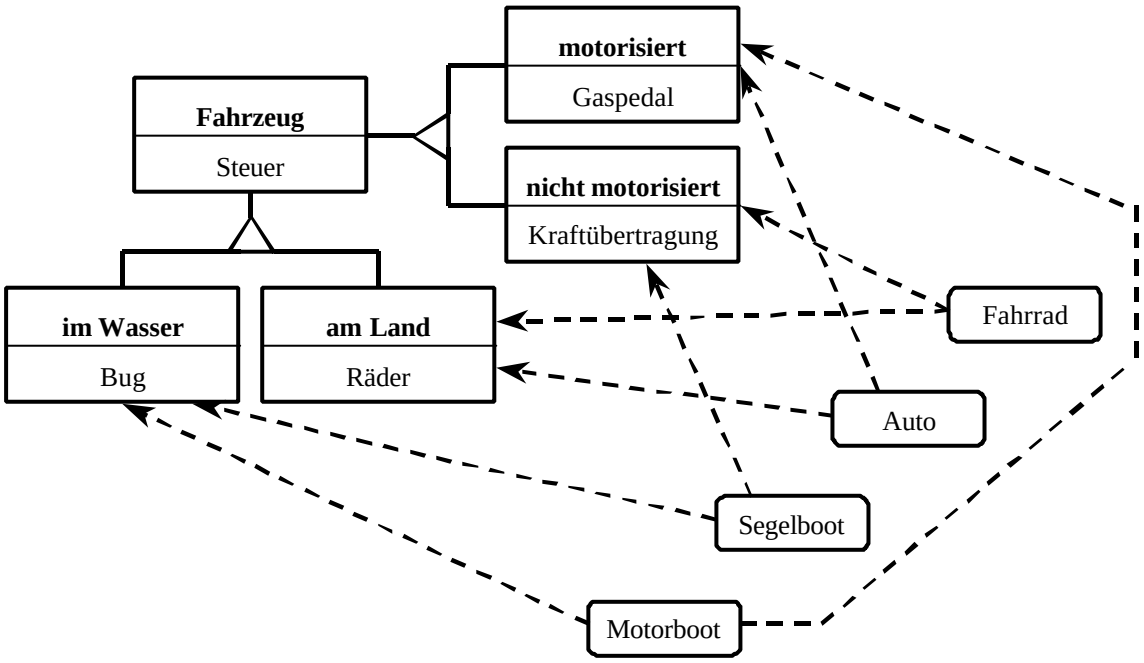

Abbildung 1: Mehrdimensionale Klassifizierung von Objekten

Die in Abbildung 1 dargestellten Objekt-Instanzen gehören alle der Klasse Fahrzeug an. Für jedes Objekt kann angegeben werden, ob es einen Motor besitzt oder nicht und ob es sich an Land oder im Wasser fortbewegt. Beide Klassifizierungen sind zulässig und können in unterschiedlichen Kontexten sinnvoll sein. Soll ein Objektmodell beide Klassifizierungen abbilden können, so kann dies auf verschiedene Arten erfolgen.

2.1.1 Mehrere Spezialisierungs-Kriterien

2.1.1.1 Problem

Das klassische Objektorientierte Paradigma enthält die Möglichkeit, Klassen durch Subklassenbildung zu spezialisieren und die neu entstehenden Subklassen über einen Vererbungs-Mechanismus mit der Funktionalität der Superklasse auszustatten.

Die gesamte Funktionalität eines Objektes kann über mehrere Schnittstellen angesprochen werden – jede Schnittstelle begründet einen Typ, dem ein Objekt angehört. Typen können ebenfalls in eine Hierarchie eingeordnet werden.

Da jeder Typ einer Ausprägung in genau einem Spezialisierungs-Kriterium entspricht, ist ein Objekt, das nach mehreren Kriterien spezialisiert werden soll, auch mehreren Typen zuzuordnen. Aufgrund der Forderung, daß jedes Objekt Instanz genau einer Klasse sein muß, ist es nur mit Einschränkungen möglich, die Typenhierarchie in der Klassenhierarchie abzubilden.

Ohne Rollenkonzept wäre für Überschneidungen in der Klassenhierarchie jeweils eine eigene Subklassen-Konstruktion mit Mehrfachvererbung notwendig, um die Extensionen der Subklassen disjunkt zu machen. Das führt sehr rasch zu kombinatorischen Explosionen in der Anzahl der notwendigen Subklassen.

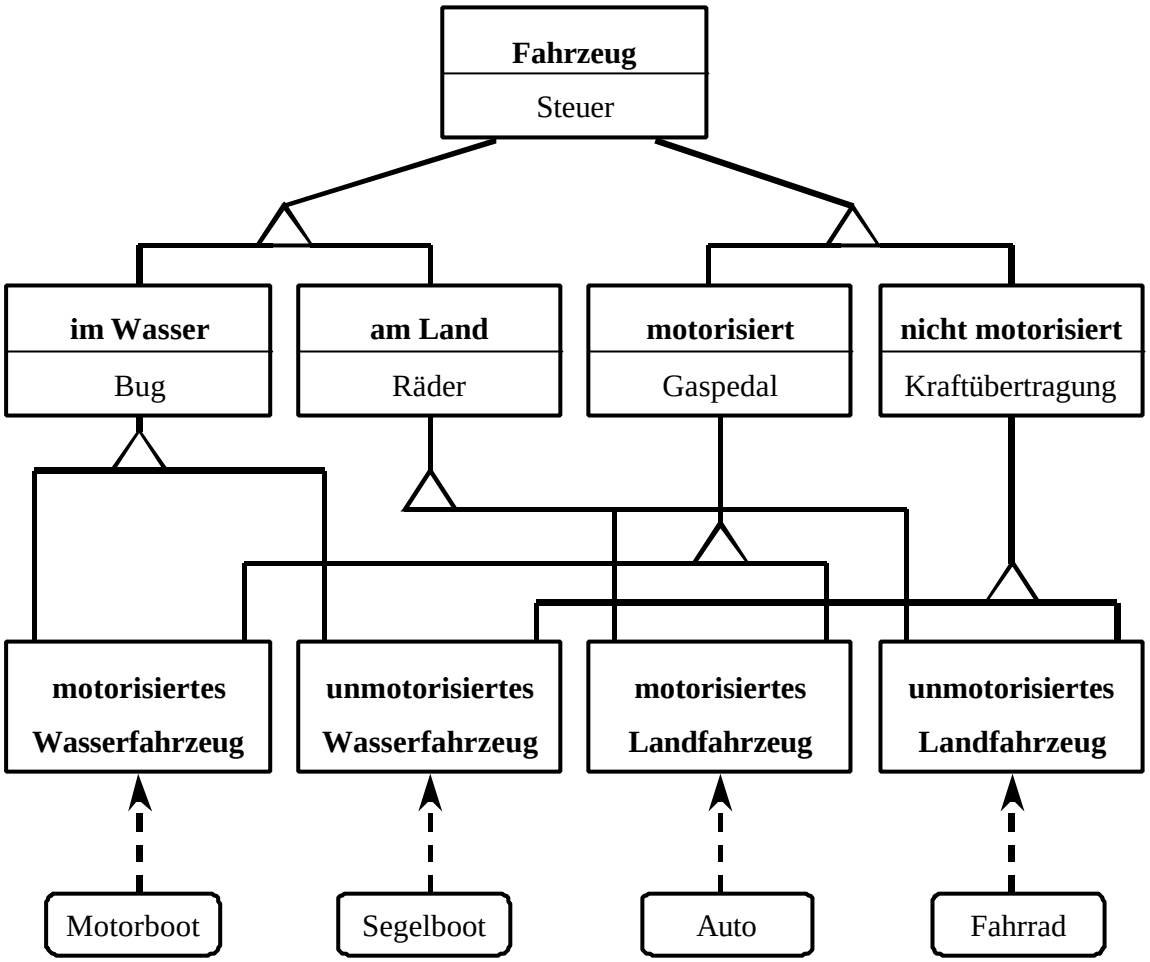

Abbildung 2: Mehrdimensionale Klassifizierung mit Mehrfachvererbung

Für jede Spezialisierungsdimension (Medium in dem sich das Fahrzeug fortbewegt und Antriebsart) muß es abstrakte Klassen für alle Möglichkeiten des jeweiligen Spezialisierungskriteriums geben. Diese Ebene stellt die Schnittstellen der einzelnen Typen zur Verfügung – kein Objekt kann Instanz einer dieser „Typen-Klassen" sein.

Zusätzlich muß es für jede Permutation dieser Möglichkeiten eigene Klassen (Intersection Classes) geben, die nur der Mehrfachvererbung dienen. Auf dieser Ebene werden die einzelnen Typen zu instanziierbaren Klassen zusammengeführt.

Die gesamte spezielle Funktionalität (Eigenschaften und Methoden) ist also in den abstrakten Klassen angesiedelt, wohingegen die Intersection Classes nur der Instanziierung dienen und keine zusätzliche Funktionalität hinzufügen. Jedes Objekt ist damit Instanz genau einer dieser Intersection Classes.

Mehrfachvererbung bringt einige Probleme mit sich (z.B. „Diamant"-Problem[1]) und ist deshalb in einigen Programmiersprachen (z.B. Smalltalk) nicht vorgesehen.

[1] Wird die gleiche Funktionalität über zwei unterschiedliche Wege geerbt, so muß entschieden werden, welche der beiden Versionen zur Anwendung kommt. In der konkreten Situation kann dieser Konflikt

Soll eine Klasse mehrmals nach unterschiedlichen Kriterien spezialisiert werden, ohne Mehrfachvererbung zu verwenden, so müssen diese Subklassenbildungen sequentiell erfolgen. Das zweite Kriterium spezialisiert also bereits spezialisierte Klassen noch einmal.

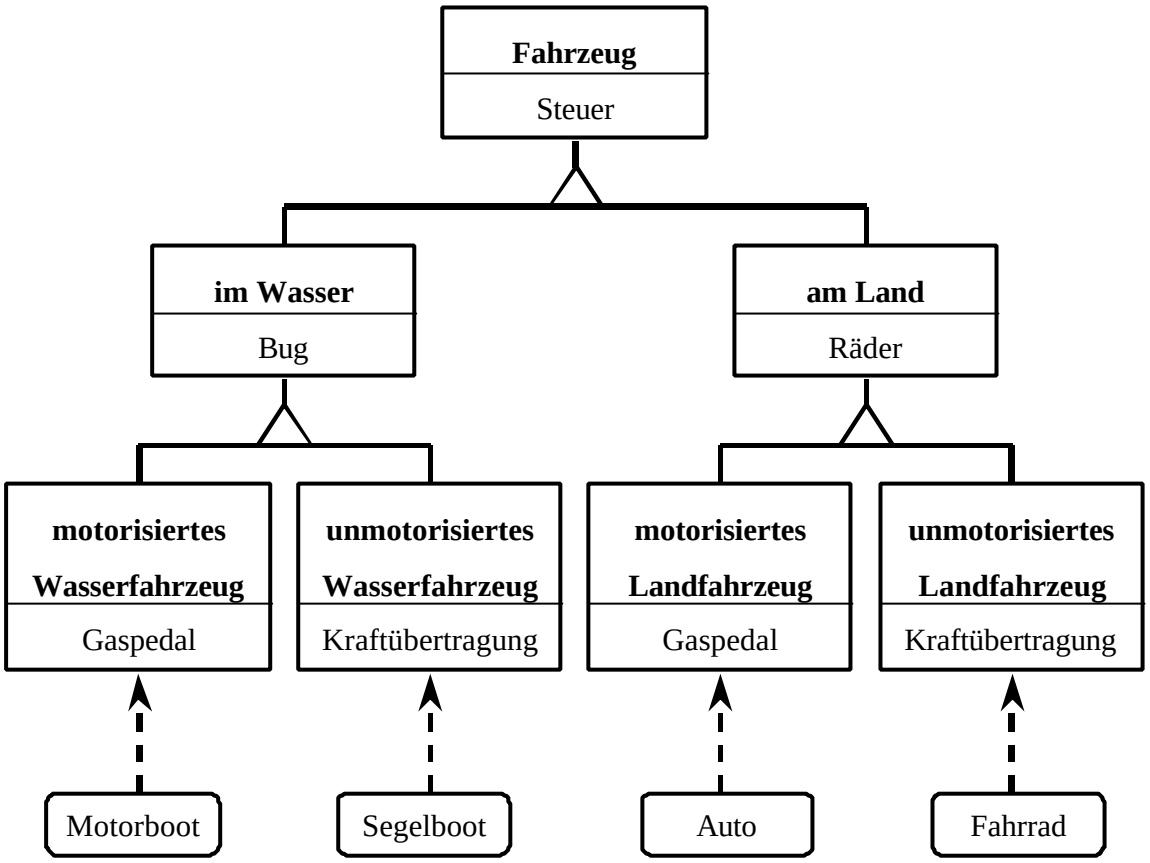

Abbildung 3: Mehrdimensionale Klassifizierung ohne Mehrfachvererbung

Die Vermeidung der oben erwähnten Intersection Classes kann nur durch den Preis redundant definierter Funktionalität erkauft werden: Die spezielle Funktionalität von motorisierten bzw. nicht motorisierten Fahrzeugen (Gaspedal/Kraftübertragung) muß an zwei unterschiedlichen Stellen im Klassenbaum doppelt implementiert werden.

Mehrmaliges Vorkommen in einer Rolle würde den Aufwand der Subklassen-Definition zusätzlich vergrößern.

2.1.1.2 Lösungs-Ansatz

Neben dem klassischen Vererbungs-Mechanismus bei der Subklassenbildung, der der Wiederverwendung der Klassen als abstrakte Software-Bausteine dient, wird ein weiterer Vererbungs-Mechanismus eingeführt, der über die Zuordnung eines Rollen-

aufgrund der mehrfachen Spezialisierung einer Klasse leicht eintreten, wenn diese mehrfachen Spezialisierungen wieder zusammengeführt werden.

Objektes zu seinem Kern-Objekt definiert wird; diese Form der Vererbung erfolgt somit eher auf der Instanzen-Ebene, als auf der Klassen-Ebene, und dient somit der Wiederverwendung der Instanzen.

Die Zuweisung von Rollen ist unabhängig von anderen Rollen, die ein Objekt bereits übernommen hat. So können Mehrfachvererbung sowie eigene Durchschnitts-Klassen vermieden werden, und auch das mehrmalige Zuweisen einer Rolle wird möglich.

Zur Veranschaulichung dieses neuen Beziehungstyps im Objekt-Modell wird die folgende Notation verwendet: Jede Rolle, die einem Objekt zugeordnet ist, wird durch eine halbrunde Ausbuchtung am Objekt dargestellt. An jede dieser Ausbuchtungen schließt sich die Beziehung zu jenem Objekt an, das diese Rolle verkörpert.

Diese Notationsform wurde in Analogie zur Darstellung von Aggregations-Beziehungen in OMT gewählt.

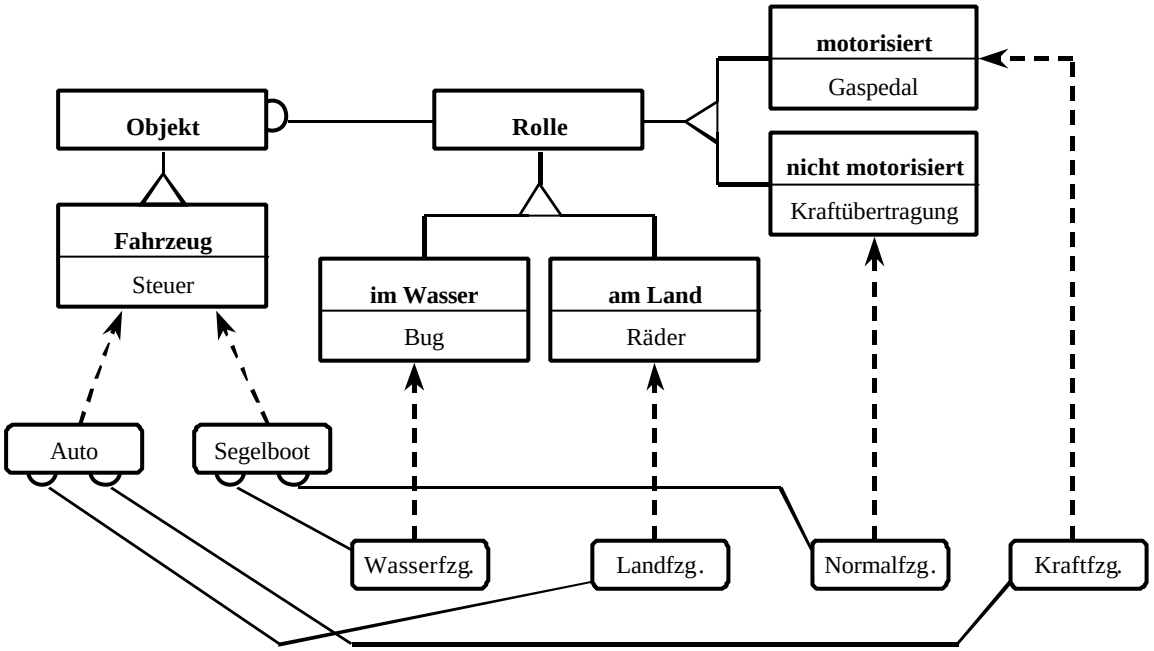

Abbildung 4: Mehrdimensionale Klassifizierung mit Rollen

2.1.2 Kontextabhängige Modellierung

2.1.2.1 Problem

Das klassische Objektorientierte Paradigma verlangt, daß alle Komponenten eines Objektes innerhalb der Objekt-Kapsel eingeschlossen sind; dazu ist es notwendig, von im jeweiligen Kontext irrelevanten Aspekten zu abstrahieren, alle notwendigen Funktionalitäten aber zum Zeitpunkt der Modellierung zu antizipieren. Da keine darüberliegenden Konstrukte - wie ein Hauptprogramm - vorgesehen sind, müssen

neben den grundlegenden (intrinsic) auch die vom Kontext abhängigen (extrinsic) Eigenschaften und Verhaltensweisen im Objekt enthalten sein.

Wird ein Objekt in unterschiedlichen Kontexten verwendet, müssen alle berücksichtigt werden. Soll ein Objekt dann in einem bestimmten Kontext verwendet werden, so ist es notwendig, in jedem Fall alle extrinsic Komponenten zur Verfügung zu stellen. Je mehr Anwendungen für ein Objekt bestehen, desto umfangreicher wird diese Aufblähung. Kommt eine Anwendung nachträglich hinzu, müssen bestehende Objekte erweitert werden.

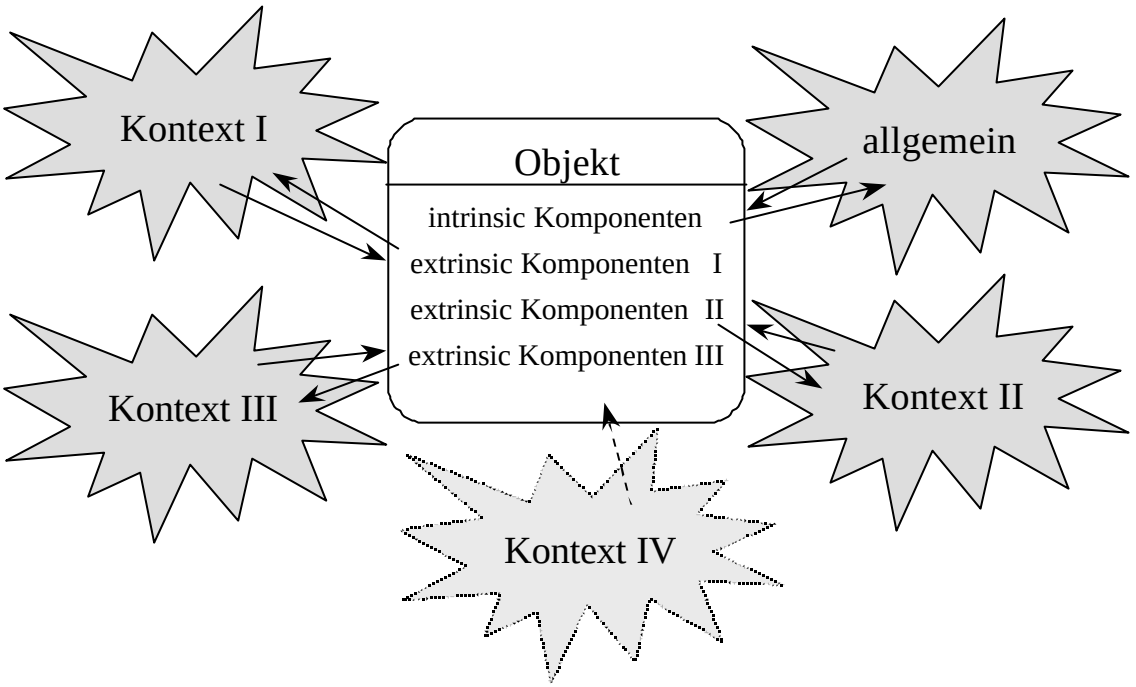

Abbildung 5 - Das klassische Objektorientierte Modell

2.1.2.2 Lösungs-Ansatz

Ein in die Klassen-Hierarchie eingeordnetes Objekt bekommt zusätzlich zu seinen (intrinsic) Komponenten weiteres (extrinsic) Verhalten aufgrund von zugeordneten Rollen-Objekten.

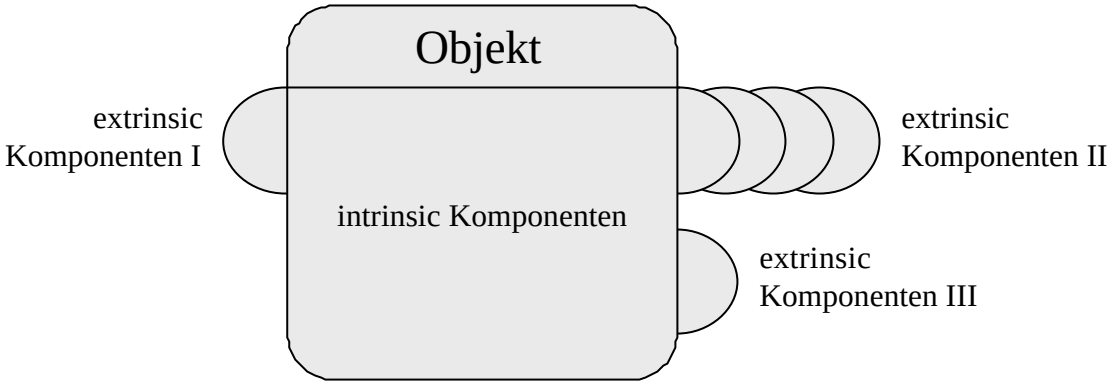

Abbildung 6 - Darstellung eines Objektes mit zugeordneten Rollen

2.1.3 Klassen-Migration bestehender Objekte

2.1.3.1 Problem

Jedes Objekt ist im klassischen Objektorientierten Paradigma Instanz genau einer – der am stärksten spezialisierten – Klasse.

Jede Veränderung in der Zuordnung zu einer Klasse würde die Geburt eines neuen Objektes der neuen Klasse, das Kopieren der entsprechenden Information in dieses neue Objekt und schließlich den Tod des alten Objektes notwendig machen. Damit würde jedoch eine der essentiellen Bedingungen für die Identität persistenter Objekte nach [Wieringa91] verletzt: Auch wenn der Inhalt des neuen Objektes mit dem Inhalt des alten Objektes in den relevanten Teilen übereinstimmt, hat das neue Objekt einen anderen Object-Identifier und ist somit ein anderes Objekt!

2.1.3.2 Lösungs-Ansatz

Durch die Zuordnung von Rollen zu einem Objekt erfolgt eine gewisse Form der dynamischen Klassifizierung; Veränderungen in der Rollen-Zuordnung entsprechen Migrations-Prozessen in dieser dynamischen Klassifizierung. Zuordnungen von Rollen lassen aber das Objekt unberührt und betreffen nur die Existenz der jeweiligen Rollen.

2.2 Begriffsbestimmung

2.2.1 Mögliche Definitionen des Begriffes Rolle

- Rolle: Zentrale Kategorie der Soziologie, die die Summe der gesellschaftlichen Erwartungen an das Verhalten eines Inhabers einer sozialen Position (Stellung im Gesellschaftsgefüge) bezeichnet. Diese Verhaltenserwartungen stellen sich als ein Bündel von Verhaltensnormen dar, deren Verbindlichkeit unterschiedlich streng ist. Der Widerspruch verschiedener Rollenerwartungen heißt Rollenkonflikt. [Brockhaus]
- A Role of an Object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects. [Kristensen95]
- Role: The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association role) or dynamic (e.g., a collaboration role). [UML]

Die traditionelle Definition des Begriffes Rolle, die [Brockhaus] bietet, stammt aus der Soziologie und legt ein Schwergewicht auf die Erwartungen, die von außen an eine Person herangetragen werden: Die Person richtet ihr Verhalten nach diesen Erwartungen, die abhängig von der gerade eingenommenen Position in der Gesellschaft sind. So kann also auch von der selben Person in unterschiedlichen Situationen unterschiedliches Verhalten erwartet werden.

Die anderen beiden Definitionen entstammen einschlägiger Literatur aus dem Bereich Objektorientierte Software-Entwicklung und machen in ähnlicher Art und Weise mit unterschiedlicher Terminologie deutlich, daß sich die Sichtweise der Objektorientierten Software-Entwicklung sehr stark an der Realität orientiert.

[Kristensen95] übersetzt die soziologische Definition auf konzeptioneller Ebene in die Begriffswelt der Objektorientierten Software-Entwicklung; [UML] interpretiert die gleiche Definition auf einer eher technischen Ebene.

Die Bedeutung des Begriffes ist somit nicht allzu weit von der einer Rolle eines Schauspielers in einem Theaterstück entfernt, wie auch in [Bachman77] gezeigt wird[2]. Auch ein Schauspieler würde auf die Frage nach seinem Namen im Rahmen des Stückes anders reagieren als abseits der Bühne, und auch das Wissen, das er in dieser Rolle hat, ist ein anderes als sein persönliches.

[2] Definition der Rolle in [Bachman77]: Eine Rolle wird von einem Akteur gespielt und spiegelt ein definiertes Verhaltensmuster wieder.

2.2.2 Definition der Begriffe intrinsic / extrinsic

Es gibt Komponenten (Eigenschaften und Verhaltensweisen), die ein Objekt bereits bei der Entstehung besitzt und die während der gesamten Lebenszeit ihre Struktur nicht verändern. Diese Eigenschaften nennt man intrinsic (grundlegend) – sie sind den Objekten statisch zugeordnet.

Andererseits gibt es Komponenten, die von Objekten nur zeitweise angenommen werden bzw. die nur in bestimmten Kontexten verwendet werden. Diese Eigenschaften nennt man extrinsic (kontextabhängig) – sie sind den Objekten dynamisch zugeordnet.

Diese Abgrenzung stellt eine wesentliche – und in der Praxis oft nicht einfach zu treffende – Modellierungsentscheidung dar, wie in Kapitel 2.3.2 näher beleuchtet wird.

2.2.3 Mögliche Definitionen des Begriffes Subjekt

Als Subjekt wird in der Literatur zumeist ein Objekt in der Summe aller implementierten Kontexte definiert. Werden also kontextabhängige Komponenten in Rollen-Objekten abgelegt und diese einem Objekt zugeordnet, so stellt sich das zugehörige Subjekt als die Gesamtheit aus Objekt und allen zugeordneten Rollen-Objekten dar:

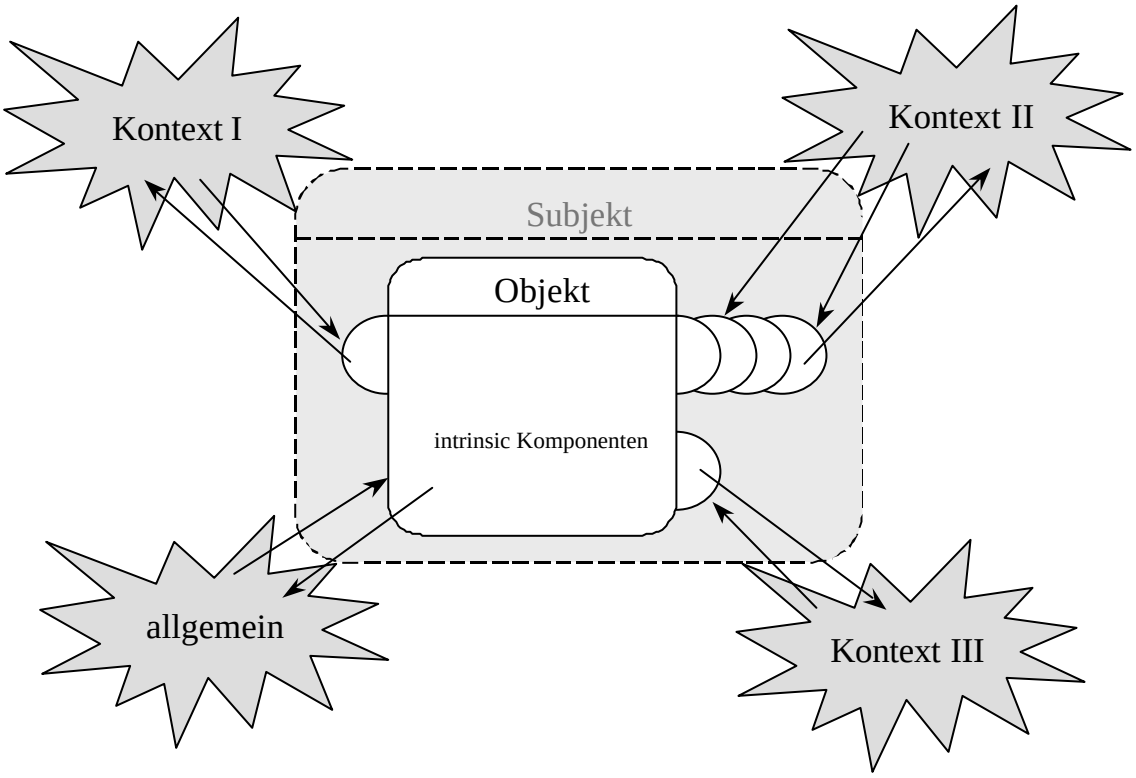

Abbildung 7 - Darstellung eines Subjektes im Objektorientierten Paradigma

Die zugeordneten Rollen sind „Teile“ des Subjektes; durch die Zuordnung der Rollen zum Objekt entsteht auch ein „spezialisiertes“ Objekt. Die Zuordnung (und damit auch die Spezialisierung und Vererbung) passiert aber auf Instanzenebene; so könnten sogar neue Klassifizierungen gemäß der zugeordneten Rollen eingeführt werden.

Darüberhinausgehend stellt [Harrison93] ein Konzept vor, das das klassische Objektorientierte Paradigma um eine Schicht erweitert, in der Subjekte angesiedelt sind, über die auf die Objekte zugegriffen wird; diese Subjekte entsprechen jeweils einem Kontext, enthalten die jeweiligen extrinsic Komponenten und verwenden diese während des „Agierens“ mit den Objekten.

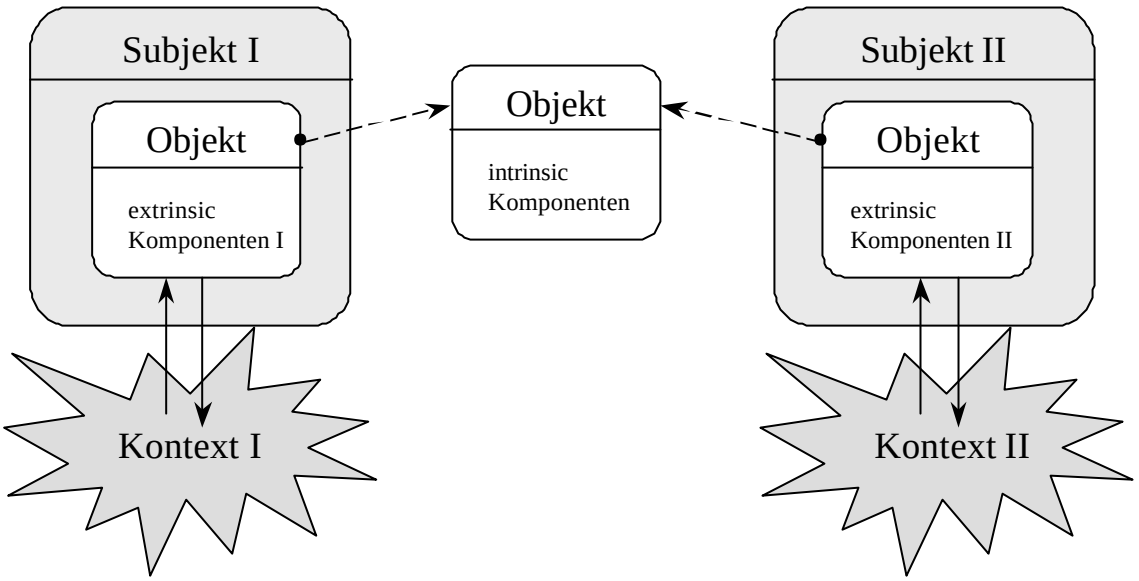

Abbildung 8 - Das Subjektorientierte Modell nach [Harrison93]

Da jedoch diese Definition weit über das klassische Objektorientierte Paradigma hinausgeht, wird im folgenden der Begriff Subjekt im Sinne der ersten angeführten Definition verwendet, da diese noch im Einklang mit dem Objektorientierten Paradigma erscheint. Das Modell nach [Harrison93] wird als Alternativvorschlag im Kapitel 2.4 kurz zusammengefaßt.

2.2.4 Definitionen verwandter in der Literatur verwendeter Begriffe

In der Literatur wird eine Reihe von Begriffen diskutiert, die im Zusammenhang mit Rollen stehen und ähnliche, aber zum Teil leicht unterschiedliche Bedeutungen haben:

- **Aspekt**
  [Richardson91] beschreibt ein Konzept, das OO-Datenbank-Objekten die Möglichkeit geben soll, ihren Typ zu einem speziellen Subtyp zu erweitern.
- **Clover**
  [Stein90] beschreibt ein Konzept, durch das Objekte in entsprechenden „Ausbuchtungen“ (Kleeblatt) extrinsic Komponenten ablegen können.
- **Perspektiven**
  [Sciore89] beschreibt ein Konzept, das die Definition von Template-Hierarchien erlaubt, über die Objekte kontextabhängiges Verhalten delegieren können.
- **View**
  [Shilling89] beschreibt ein Konzept, in dem für ein Objekt mehrere unabhängige Schnittstellen (interface) definiert werden können, die jeweils kontextabhängige Komponenten beinhalten können.
  [Kristensen95] jedoch kritisiert, daß der Begriff View suggeriert, es handle sich nur um einen Filter.
- **Filter**
  Durch einen Filter wird ein Interface definiert, das nur zu bestimmten Komponenten des Objektes Zugang ermöglicht; alle anderen Komponenten werden „ausgefiltert“. In einem Filter kann jedoch keine Erweiterung des Objektes um kontextabhängige Komponenten stattfinden.
- **Maske**
  In Weiterführung der Kritik von [Kristensen95] kann der in der Literatur noch nicht gebrauchten Begriff der Maske verwendet werden: Eine Maske kann bestimmte Komponenten filtern, andere Komponenten verändern und zusätzliche Komponenten hinzufügen.

Die Konzepte, denen diese Begriffe entstammen, unterscheiden sich größtenteils nicht in ihrem Ziel, sondern eher in Entstehungsgeschichte, Umfeld, Motivation und in technischen Details.

Der Begriff Rolle wurde in der Informatik zuerst in der Datenmodellierung ([Bachman77], [Richardson91], [Su91], [Bergamini93]) geprägt. Bei der Modellierung von Entitäten war sehr früh erkannt worden, daß eine Entität bei der Beziehung mit anderen Entitäten unterschiedliche Funktionen einnehmen kann. Diese Funktionen wurden auch als Rollen [Chen76] bezeichnet, so daß an den Enden einer Beziehung bestimmte Rollen von Entitäten zu finden sind.

Die Idee der Rolle wurde von der Objektorientierten Software-Entwicklung ([Sciore89], [Stein90], [Kristensen95], [Schrefl96]) aufgegriffen und weiterentwickelt, wobei von ähnlichen Überlegungen ausgegangen wurde. Auch die Objektorientierte Software-Entwicklung versucht die Objekte der realen Welt abzubilden und stößt dabei auf das

Problem, daß Objekte abhängig von ihrer Umgebung – ihrem Kontext – agieren, also kontextabhängiges Verhalten zeigen bzw. Eigenschaften besitzen.

[Bergamini93] versucht, seine Ideen in einem völlig neuen, extra dafür konzipierten System (Fibonacci) optimal umzusetzen – wohingegen [Schrefl96] versucht, ein bestehendes System (Smalltalk) mit möglichst wenig Aufwand so zu erweitern, daß seine Ideen darin abgebildet werden können.

Im folgenden werden einige der in dieser Literatur kontroversiell diskutierten Punkte aufgegriffen und besprochen.

2.3 Modellierungsaspekte

Wenn man die Abläufe des täglichen Lebens untersucht, findet man eine Vielzahl von Situationen, in denen Menschen in Rollen agieren oder Objekte des täglichen Lebens in bestimmten Rollen gesehen werden:

- Objekt Person in den Rollen: Mutter/Student/Angestellter
- Objekt Brett in den Rollen: Tisch/Sessel/Dach

Bei näherer Betrachtung erkennt man, daß es scheinbar unterschiedliche Kriterien gibt, nach denen man Eigenschaften entweder in Objekten selbst oder deren Rollen ansiedelt.

Es scheint Dinge zu geben, die es erlauben, ein Objekt als Person oder Brett zu identifizieren. Diese Eigenschaften gehen auch nicht verloren, wenn die Objekte eine bestimmte Rolle annehmen; man kann auch die Person oder das Brett wahrnehmen, wenn man hinter die Rolle blickt. Diese Eigenschaften - sie werden als „intrinsic“ bezeichnet - scheinen also charakteristisch zu sein und sich über die Lebensdauer hinweg in ihrer Struktur kaum zu verändern.

Eine Rolle jedoch existiert meist nur zeitlich begrenzt bzw. wird sie häufig erst im Laufe des Lebenszyklus und nicht bei der Entstehung eines Objektes angenommen. Die Eigenschaften dieser Rolle – sie werden als „extrinsic“ bezeichnet – erweitern die Objekte dynamisch. Das bedeutet aber auch, daß das eigentliche Objekt durch die Annahme einer Rolle nicht nur gleich bleibt, sondern daß es das selbe bleibt. Seine Identität bleibt unverändert.

Eine weitere interessante Frage stellt sich, wenn man einige unterschiedliche Beispiele betrachtet. Nimmt ein Objekt von sich aus eine Rolle an oder wird ihm eine solche von „außen“ gegeben, wird das Objekt also in einer bestimmten Weise betrachtet. Zum Annehmen einer Rolle gehört ein gewisses Bewußtsein dieser Rolle. Der Angestellte weiß, in welcher Rolle er agiert und wie er sich deshalb verhalten muß, während das Brett in seinem Verhalten keinen Unterschied zeigt, ob es als Tisch oder als Sessel eingesetzt wird.

Wie aber sind all diese Überlegungen mit dem Objektorientierten Paradigma zu vereinbaren? Welche Mechanismen des Objektorientierten Paradigmas kann man verwenden, um Rollen zu implementieren?

2.3.1 Probleme beim Objekt-Design

2.3.1.1 Konzept

Ausgangsbasis für jeden Modellierungsprozeß ist ein Abstraktionsprozeß. Diesem Abstraktionsprozeß liegt ein bestimmtes Modellierungsziel zugrunde. Details, die zur Erreichung dieses Ziels keinen Beitrag leisten, werden dabei vernachlässigt, um ein vereinfachtes Modell der Realität zu erhalten. In diesem Modell soll alles zur Erreichung des jeweiligen Modellierungsziels Notwendige enthalten sein.

Entscheidend bei der Modellierung der Objekte ist zum Beispiel die Unterscheidung zwischen den intrinsic und den jeweiligen extrinsic Komponenten. Ist diese Unterscheidung bei konkreten Objekten der realen Welt noch relativ einfach zu treffen, da man sich auf Beobachtbares und Meßbares beziehen kann, so sind diese Umstände nicht mehr anwendbar, wenn es sich um abstrakte Objekte handelt.

Für ein anderes Modellierungsziel kann diese Unterscheidung also anders ausfallen: Bestimmte Komponenten, die im Bezug auf ein Modellierungsziel als intrinsic qualifiziert werden, werden hinsichtlich eines anderen Modellierungszieles als extrinsic qualifiziert.

Gerade bei Objekten, die in Informationssystemen Verwendung finden, ist oft ein so hoher Grad an Abstraktheit festzustellen, daß es schwer fällt zu entscheiden, welche Komponenten als intrinsic betrachtet werden sollten.

2.3.1.2 Probleme

Dabei läßt sich das Problem beobachten, daß Spezialisierungen unterschiedlichster Natur in ein und dieselbe Klassenhierarchie gepackt werden müßten:

- Spezialisierungen hinsichtlich unterschiedlicher Dimensionen
- Rollenbildung, die eigentlich eigenständig identifizierbare Objekte erzeugt

In Zusammenhang damit steht die Frage, ob der Abstraktionsprozeß Rollenbildung (roleification) eine Form der Spezialisierung ist: Die Rolle Angestellter kann wohl nur einem Objekt der Klasse Person zugewiesen werden. Ist ein Angestellter daher eine spezialisierte Person? Die Antwort darauf ist wohl Ja, was das Naheverhältnis zwischen Rollenbildung zu Generalisierung/Spezialisierung ausdrückt.

Aber dieses „spezialisierte“ Verhalten kann nie das intrinsic Verhalten des Objektes redefinieren, sondern nur erweitern: Eine Telefonnummer, die aus der Rolle Angestellter stammt, ersetzt nicht die private Telefonnummer der Person, sondern wird dieser hinzugefügt; sowohl die Person, als auch der Angestellte können eigenständig identifiziert (Sozialversicherungsnummer/Personalnummer) und nach der jeweiligen Telefonnummer gefragt werden. Es ist jedoch auch möglich, von beiden auch die jeweils andere Nummer oder auch beide Nummern zu erfahren – dieses Verhalten ist aus der Aggregation von Teilen zu einem Ganzen bekannt (z.B: Preis des Ganzen ist die Summe der Preise seiner Teile).

Daher und aus später ausgeführten Gründen besteht also ein ebenso großes Nahverhältnis zu Aggregation/Dekomposition, wie auch zu Generalisierung/Spezialisierung.

2.3.1.3 Beispiele

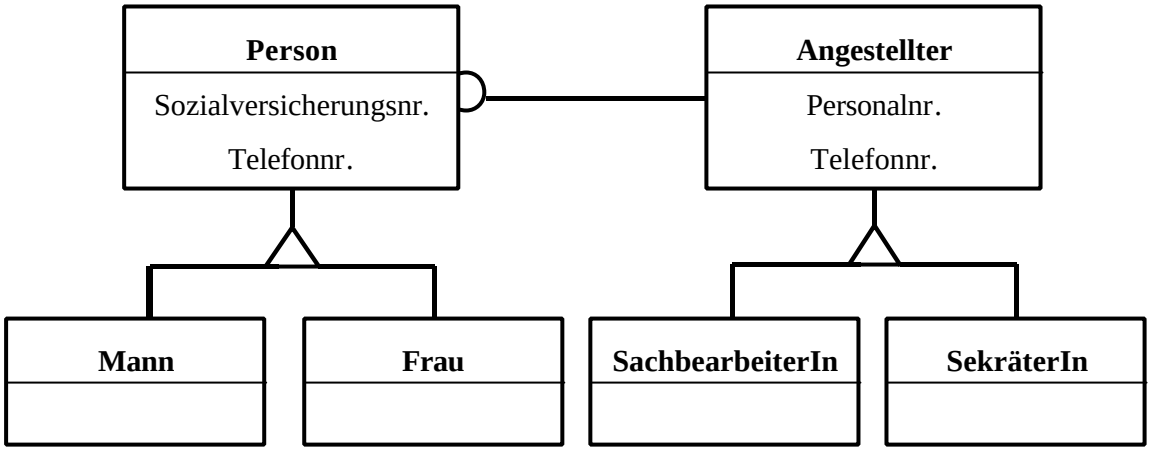

Abbildung 9: Das Zusammenspiel zwischen Spezialisierung und Rollenbildung in zwei getrennten Klassenhierarchien.

Sowohl die Klasse „Person“, als auch die Klasse „Angestellter“ sind abstrakte Klassen und in jeweils eigenen Klassenhierarchien weiter spezialisiert; die Rollenbeziehung zwischen einer Person und seiner Rolle als Angestellter wird jedoch auf der Ebene der abstrakten Klassen definiert. Dies hat zur Folge, daß jede Instanz der Subklassen der Klasse „Person“ eine Rolle annehmen kann, die Instanz einer Subklasse der Klasse „Angestellter“ ist.

Sowohl in der Klasse „Person“ als auch in der Klasse „Angestellter“ ist eine eigene Instanzvariable „Telefonnr.“ definiert.

2.3.1.4 Realisierungshinweise

[Wieringa91] grenzt die folgenden Begriffe ab:

- **Object Identifier (OID)**
  Ein in jedem möglichen Zustand der Welt global eindeutiger, unveränderlicher Wert, der keine sonstige Information ausdrückt.
- **Key**
  Ein in einem tatsächlichen Zustand der Welt global eindeutiger Wert, der auch Information tragen kann und daher veränderbar sein muß.
- **Surrogate**
  Ein Schlüssel, der nur in Bezug auf eine lokal definierte Umgebung (Datenbank) eindeutig ist.

Die Forderung, daß ein OID in jedem möglichen Zustand der Welt eindeutig sein muß, erfordert, daß jeder Wert von dem Zeitpunkt an, zu dem er irgendwo auf der Welt einem Objekt zugewiesen wurde, nirgends mehr einem anderen Objekt zugewiesen werden kann.

In der CORBA-Spezifikation 2.0 [CORBA] wird ein Verfahren vorgeschlagen, um dieser Forderung zu genügen, in das Parameter wie Zeit und Netzwerkadresse einfließen.

In diesem Zusammenhang stellt sich nun die Frage, wann eine Spezialisierung in einer eigenen Klassenhierarchie abgebildet wird und diese mittels Rollenbildung mit der ursprünglichen Klassenhierarchie verbunden wird?

[Wieringa91] schlägt dafür vor, immer dann eine eigene Klassenhierarchie einzuführen, wenn eine Klasse über ein eigenes Identifikationsmerkmal verfügt: Jede Instanz der Klasse „Mann“ verfügt zur eindeutigen Identifikation nur über die Sozialversicherungsnummer, die sie von der Klasse „Person“ geerbt hat; jede Instanz der Klasse „Angestellter“ ist aber über die Personalnummer eindeutig als eigenständiges – wenn auch abhängiges – Objekt identifizierbar.

Innerhalb der Klassenhierarchie, die von der Klasse „Angestellter“ abgeleitet wird, gilt wiederum, daß alle Instanzen der Subklassen nur über die geerbte Personalnummer identifiziert werden können. Außerdem kann neben der Anzahl aller vergebenen Sozialversicherungsnummern, die mit einer Rolle „Angestellter“ verbunden sind, auch die Anzahl aller vergebenen Personalnummern ermittelt und die beiden Ergebnisse sinnvoll unterschiedlich interpretiert werden[3]:

- Wieviele verschiedene Personen stehen in Anstellungsverhältnissen?
- Wieviele Anstellungsverhältnisse existieren?

2.3.2 Intrinsic Komponenten versus Extrinsic Komponenten

2.3.2.1 Konzept

Es gibt Eigenschaften und Verhaltensweisen, die ein Objekt bei der Entstehung besitzt und die während der gesamten Lebenszeit ihre Struktur kaum verändern. Diese Eigenschaften nennt man intrinsic.

Andererseits gibt es Eigenschaften und Verhaltensweisen, die von Objekten nur zeitweise angenommen werden bzw. die nur in bestimmten Kontexten verwendet werden. Diese Eigenschaften nennt man extrinsic.

2.3.2.2 Probleme

Es ist bei der Modellierung sehr häufig nicht möglich festzulegen, ob Eigenschaften intrinsic oder extrinsic sind. Diese Entscheidung kann von Modell zu Modell unterschiedlich sein. Ein Grund dafür ist, daß jedes Modell eine Vereinfachung der komplexen realen Situation darstellt. Welche Dinge vereinfacht werden beeinflußt auch eine Entscheidung zwischen intrinsic und extrinsic.

Gerade bei sehr abstrakten Objekten, die keine physische Manifestation besitzen sind kaum Grenzen zwischen intrinsic und extrinsic zu ziehen. Hier sind unter Umständen auch Objekte vorstellbar, die keine intrinsic Komponenten, sondern nur extrinsic Komponenten besitzen.

[3] [Wieringa91] erläutert in diesem Zusammenhang das folgende Beispiel: Soll die Auslastung einer Buslinie ermittelt werden, so ist es nicht sinnvoll, die Personen über ihre Sozialversicherungsnummern zu zählen, da viele der Personen im Beobachtungszeitraum öfter mit der Buslinie gefahren sein könnten; vielmehr müssen die verkauften Fahrkarten gezählt werden.

2.3.2.3 Beispiele

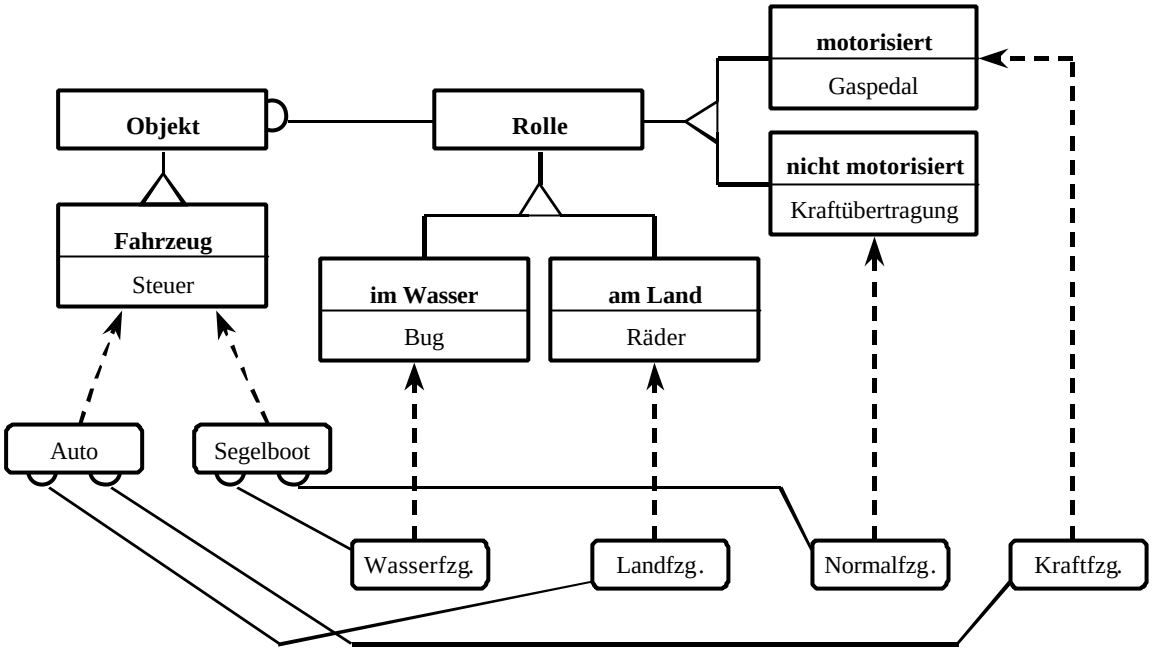

Abbildung 10: Trennung der Komponenten in intrinsic und extrinsic

Abbildung 10 zeigt ein mögliches Modell der Klasse Fahrzeug. Alle Instanzen der Klasse Fahrzeug müssen in diesem Modell auf irgend eine Art und Weise gesteuert werden – diese Eigenschaft wird also als intrinsic qualifiziert und direkt in der Klasse Fahrzeug angesiedelt.

Nicht alle Instanzen der Klasse Fahrzeug müssen in diesem Modell Räder aufweisen, sondern nur jene, die sich auf Land fortbewegen – diese Eigenschaft wird also als extrinsic qualifiziert und in einer der Rollenklassen angesiedelt.

Die konkrete Instanz der Klasse Fahrzeug „Auto“ enthält direkt ein Steuer und bekommt zusätzlich Räder in seiner Rolle als Landfahrzeug zugeordnet.

In einem anderen Modell kann davon abstrahiert werden, daß sich Fahrzeuge auch noch in einem anderen Medium fortbewegen können, sodaß die Räder in diesem Modell als intrinsic zu qualifizieren wären.

In den bisherigen Beispielen waren die einzelnen Eigenschaften immer gegenständlich erfaßbar. Bei Objekten wie: Compiler, Editor, Rechner, Sortierer, etc. ist dies nicht mehr möglich, daher fällt auch die Trennung in intrinsic und extrinsic Komponenten schwerer.

2.3.2.4 Realisierungshinweise

Eine Trennung von intrinsic und extrinsic-Eigenschaften sollte in einem Rollenkonzept realisiert werden, wobei intrinsic-Eigenschaften einem Objekt zugeordnet werden, während extrinsic Eigenschaften Teil einer Rolle sind.

2.3.3 Annehmen von Rollen versus Zuweisen von Rollen

2.3.3.1 Konzept

Es gibt sowohl Modelle, in denen Rollen von Objekten bewußt angenommen werden, als auch Modelle, in denen Rollen nur von anderen als solche gesehen werden, ohne daß das Objekt sich dessen bewußt ist und sein Verhalten selbst steuern kann. Will man diese Unterscheidung modellieren, könnte man dies über die Bindungsstärke bzw. Position der Rolle tun.

Während eine angenommene Rolle semantisch näher bei jenem Objekt liegt, das diese Rolle sowohl angenommen hat, als auch diese Rolle ausfüllt, müßte eine zugewiesene Rolle näher bei dem Objekt liegen, das die Rolle zugewiesen hat, das die Rolle letztendlich jedoch nicht spielt.

Diese Unterschiede in der semantischen Positionierung der Rolle haben auch Auswirkungen auf die Art der Kommunikation und die Frage, wie Objekte auf Veränderungen ihrer intrinsic Eigenschaften reagieren müssen. Auch die Kapselung ist eine andere und die Kontextinformation, die durch die Rolle repräsentiert wird, ist unterschiedlich positioniert.

In der realen Welt können beide Situationen parallel auftreten, was in einer Entwicklungsumgebung berücksichtigt werden müßte.

2.3.3.2 Probleme

Es ist auch bei diesem Konzept schwer zu sagen, wie abstrakte Objekte eingeordnet werden sollen. Ob diese Objekte bewußt Rollen annehmen oder nur in solchen gesehen werden, ist in vielen Fällen schwer zu entscheiden und daher schwer zu modellieren.

Während der Aufwand für die Umsetzung eines festgelegten Modells noch relativ einfach sein könnte, steigt der Aufwand und die Komplexität für die notwendigen Kommunikations- und Verwaltungsmöglichkeiten bei einer Implementation beider Varianten sehr rasch.

2.3.3.3 Beispiele

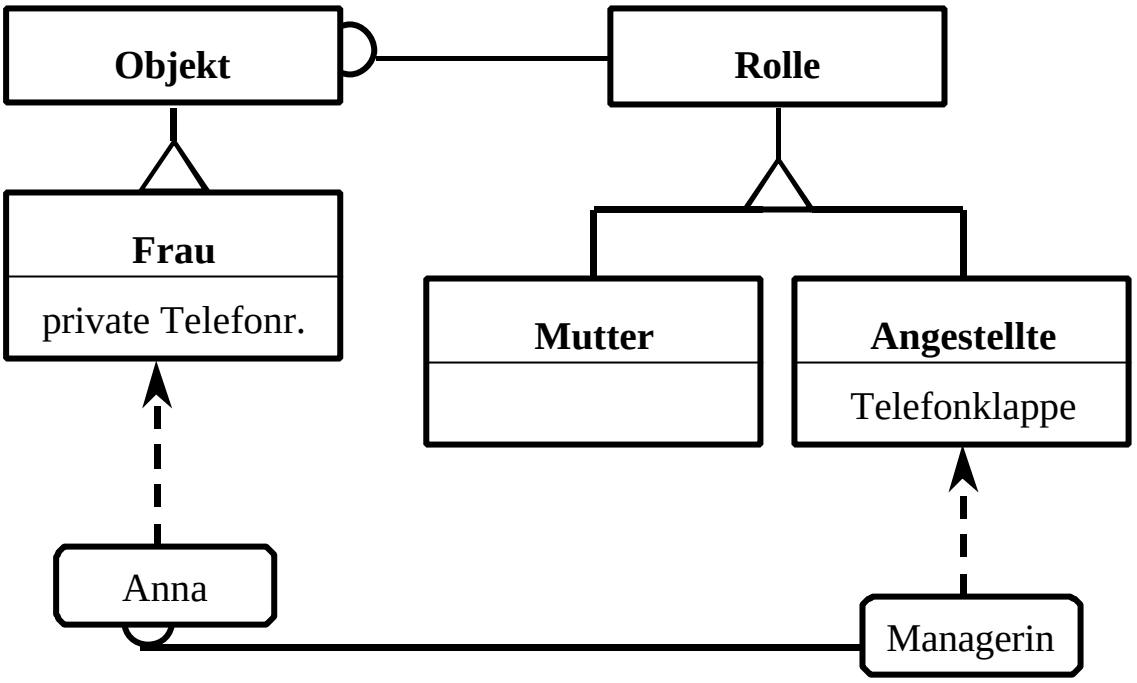

Abbildung 11: Das Annehmen von Rollen

In Abbildung 11 ist der Instanz „Anna“ der Klasse „Frau“die Rolle „Managerin“ zugeordnet. Da die Rolle „Managerin“ Ausprägung der Rollenklasse „Angestellte“ ist, verfügt sie in dieser Rolle über eine „Telefonklappe“. Sie selbst entscheidet abhängig vom Kontext, in dem sie angesprochen wird, welche Telefonnummer sie angibt: Sie nimmt also ihre Rolle selbständig an, wenn es der Kontext erfordert.

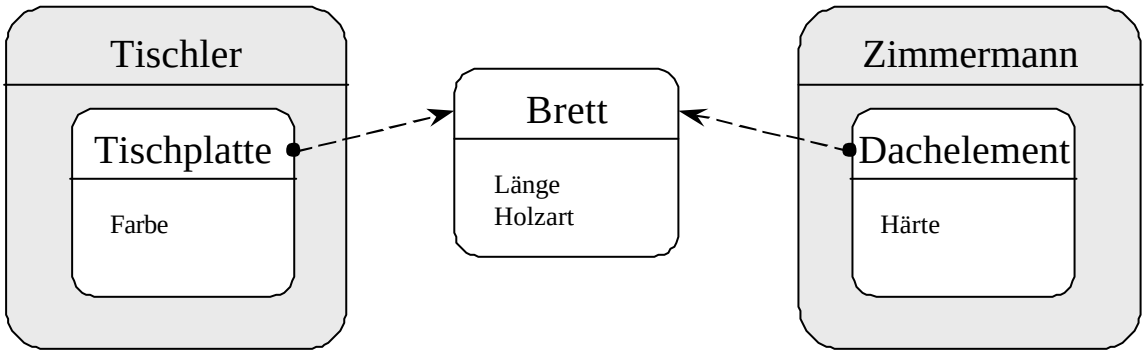

Abbildung 12: Das Zuweisen von Rollen nach [Harrison93]

In Abbildung 12 wird ein Brett in zwei verschiedenen Kontexten betrachtet. Im Kontext des Subjektes Tischler wird aus der intrinsic Eigenschaft „Holzart“ die extrinsic Eigenschaft „Farbe“ abgeleitet, die nur für den Tischler von Interesse ist. Im Kontext des Subjektes Zimmermann wird aus der intrinsic Eigenschaft „Holzart“ die extrinsic

Eigenschaft „Härte“ abgeleitet, die nur für den Zimmermann von Interesse ist. Das Brett hat kein Wissen über diese Interpretation seiner intrinsic Eigenschaft.

2.3.3.4 Realisierungshinweise

Eine Realisierungsmöglichkeit der unterschiedlichen Position bzw. Bindungsstärke wäre die Aggregation in die jeweiligen Partner. Dabei müßte eine angenommene Rolle beim annehmenden Objekt aggregiert werden, eine zugewiesene Rolle beim zuweisenden Objekt. Die Verwaltung müßte beides ermöglichen.

Das Rollenkonzept des Objektorientierten Paradigmas erlaubt die Abbildung des Annehmens einer Rolle, da dies auch eher der Vorstellung des selbständigen Agierens der Objekte entspricht. Demgegenüber steht der Ansatz des subjektorientierten Paradigmas nach [Harrison93], bei dem nur das Zuweisen von Rollen zu den Objekten durch agierende Subjekte im Vordergrund steht.

Eine saubere Modellierung sollte wahrscheinlich beide Varianten ermöglichen und auch durch Position und Bindungsstärke verdeutlichen. Da der Aufwand jedoch sehr hoch wird, könnte eine Lösung, die zwar beide Mechanismen (zum Annehmen wie zum Zuweisen) bietet, jedoch nur eine Position der Rolle erlaubt, die Komplexität entsprechend vermindern. Bei einer solchen Lösung müßte nur eine Kommunikations- und Verwaltungsstruktur vorgesehen werden.

2.3.4 Vererbung versus Delegation

2.3.4.1 Konzept

Vererbung auf Klassenebene ist eines der wesentlichsten Prinzipien des Objektorientierten Paradigmas. Klassen beschreiben dabei die Eigenschaften und das Verhalten von Objekten und können in Hierarchien derart geordnet werden, daß eine „is_a“ Beziehung entsteht, was einer Generalisierungs-/Spezialisierungsbeziehung entspricht. Die Extension der Subklasse ist dabei eine Teilmenge der Extension der Superklasse, wobei die zu erfüllenden Bedingungen restriktiver sind.

Im Objektraum existieren Objekte, die durch „Instanziierung“ einer Klasse entstehen. Die Klasse stellt so etwas wie einen Bauplan der Instanz dar. Der Instanziierungsprozeß einer Klasse entspricht also immer dem Kopieren des Bauplans in ein neues Objekt. Wird dieser Bauplan verändert, so wirkt sich diese Veränderung nur auf die danach entstandenen Objekte aus. Objekte, die schon vor der Veränderung vorhanden waren, bleiben in ihren Eigenschaften und ihrem Verhalten unangetastet.

Der Vererbung auf Klassenebene steht die Delegation gegenüber. Mit ihr sind vor allem Begriffe wie Prototyping [Lieberman86] und Klonung verbunden: Objekte werden dabei direkt im Objektraum abgeleitet; es gibt keinen klaren Bauplan, sondern es wird direkt ein schon vorhandenes Objekt referenziert und nur die Unterschiede zu diesem

neu beschrieben. Nicht jede Eigenschaft hat also auch einen eigenen Speicherplatz im Objekt, eine Kapselung existiert nur begrenzt.

Auch hier findet eine Wiederverwendung statt, nur wird nicht der Bauplan wiederverwendet, sondern das Objekt mit all seinen Werten, mit denen die Eigenschaften belegt sind. In einer solchen Umgebung ist es jedoch problemlos möglich die Referenz zu ändern und somit auch die wiederverwendeten Eigenschaften und das Verhalten. Ein solcher „Klassenwechsel“ (eine Änderung der Referenz entspricht im Klassenkonzept in etwa einer Klasse) ist also dynamisch und bei gleichbleibender Objektidentität durchführbar.

2.3.4.2 Probleme

Die Unterschiede in den Konzepten führen in jeweils reiner Implementation zu unterschiedlichen Systemen. Während in einem System mit Delegation und Prototyping der Überblick leicht verloren geht (es existiert keine Metaebene, auf der Objekte abstrakt beschrieben und zu Mengen zusammengefaßt werden könnten), und rasche unsaubere Implementationen besser unterstützt werden, fehlen bei einem starren Klassenkonzept mit Vererbung die dynamischen, flexiblen Möglichkeiten.

2.3.4.3 Beispiele

Der konzeptionelle Unterschied dieser beiden Konzepte läßt sich anschaulich in die reale Welt übertragen. Wie in [Lieberman86] beschrieben, ähnelt die Vererbung auf Klassenebene mehr der Wissensweitergabe in Buchform. Egal über wie viele Generationen aufbewahrt, ist der Inhalt immer der selbe; einmal festgeschrieben, bleibt er unverändert. Ein reines Delegationskonzept entspricht der Wissensweitergabe von Mund zu Mund, die über die Generationen hinweg Änderungen unterworfen ist und bei der die Art der Vermittlung stark abhängig von den beteiligten Person bleibt.

Wird an eine Person die Aufgabe gestellt, eine bestimmte Funktion zu integrieren, so kann diese Person diese Aufgabe erfüllen, indem sie selbst gelerntes Verhalten anwendet: Diese Person hat die Fähigkeit in sich aufgenommen. Eine andere Alternative besteht darin, die Aufgabe an eine andere Person zu delegieren, die diese Fähigkeit besitzt.

Der Vorteil der zweiten Alternative besteht darin, daß mehrere Personen ihre Aufgaben jeweils an diesen Fachmann delegieren können. Ändert sich nun die Verfahrensanweisung des Integrierens, so muß sich nur eine Person dieses neue Verfahren aneignen, damit alle Personen ihre Aufgaben mit Hilfe des neuen Verfahrens lösen.

Der Nachteil liegt darin, daß alle auch von der Verfügbarkeit dieser einen Person abhängig sind und eventuelle Fehler, die an diesem einen Punkt gemacht werden, sich entsprechend vervielfältigen.

2.3.4.4 Realisierunghinweise

[Stein87] zeigt, daß keines der beiden Konzepte für sich betrachtet stärker ist, sondern daß beide in der Lage sind, einander gegenseitig abzubilden. Dennoch wird klar, daß es je nach Problemstellung mit den verschiedenen Konzepten unterschiedlich einfach ist, das Problem zu lösen, bzw. die Lösungen unterschiedlich performant sind. [Stein87] plädiert daher dafür, hybride Konzepte zu verwenden, in denen möglichst viele Vorteile von Delegation und Vererbung vorhanden sind.

Das Kernobjekt kann in einer Klassenhierarchie stehen und somit die intrinsic-Teile im Bauplan der Klasse enthalten. Ebenso kann die Rolle selbst die Ausprägung einer Klasse in einer anderen Hierarchie sein. Für beide sind die dort gebotenen Mechanismen problemlos verwendbar. Baupläne können so wiederverwendet, Unterschiede in Klassen festgeschrieben oder in unterschiedlichen Hierarchiestufen unterschiedlich detailliert werden.

Die Verbindung dazwischen jedoch sollte unbedingt mittels Delegation erfolgen, um sowohl die Identitätsprobleme bei Klassenwechsel zu vermeiden als auch die Dynamik zu erhalten. Eine Implementation, die diese Überlegungen berücksichtigt, liefert [Schrefl96].

2.3.5 Wiederverwendung der Instanzen versus Wiederverwendung der Konzepte

2.3.5.1 Konzept

Die Intention des Rollenkonzeptes ist die Wiederverwendung von konkreten Objekt-Identitäten: Das selbe Objekt soll für unterschiedliche Kontexte modelliert werden.

Eine der wesentlichen Intentionen des klassischen Objektorientierten Konzepts ist aber die Wiederverwendung abstrakter Einheiten als Software-Bausteine zur Lösung von wiederkehrenden Problemstellungen.

2.3.5.2 Probleme

Wird nur der Intention des klassischen Objektorientierten Paradigmas gefolgt, so ist es möglich, in verschiedenen Anwendungen Objekte nach dem gleichen Bauplan zu erzeugen, ohne diese jeweils neu implementieren zu müssen.

Jede Wiederverwendung einer konkreten Instanz würde die Geburt eines neuen Objektes der wiederverwendeten Klasse und das Kopieren der entsprechenden Information in dieses neue Objekt notwendig machen. Damit würde jedoch eine der essentiellen Bedingungen für die Identität persistenter Objekte nach [Wieringa91] verletzt: Auch wenn der Inhalt des neuen Objektes mit dem Inhalt des alten Objektes in den relevanten Teilen übereinstimmt, hat das neue Objekt einen anderen Object-Identifier, und ist somit ein anderes Objekt!

2.3.5.3 Beispiele

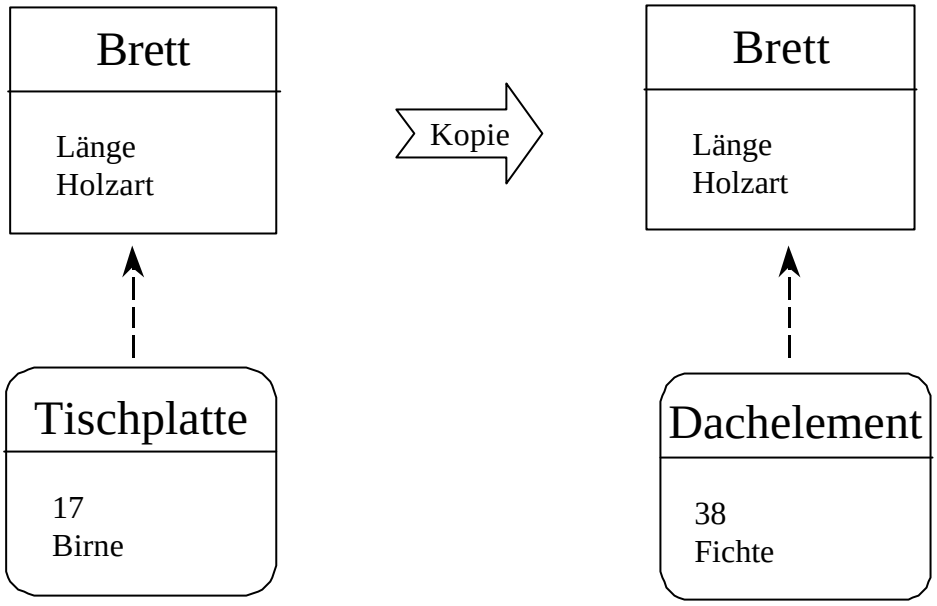

Abbildung 13: Wiederverwendung eines Konzeptes

In der klassischen Objektorientierten Intention ist das Ziel nur, eine Klasse Brett, die einmal für eine Applikation „Tischler“ modelliert und implementiert wurde, in einer anderen Applikation als abstrakten Softwarebaustein wiederverwenden zu können.

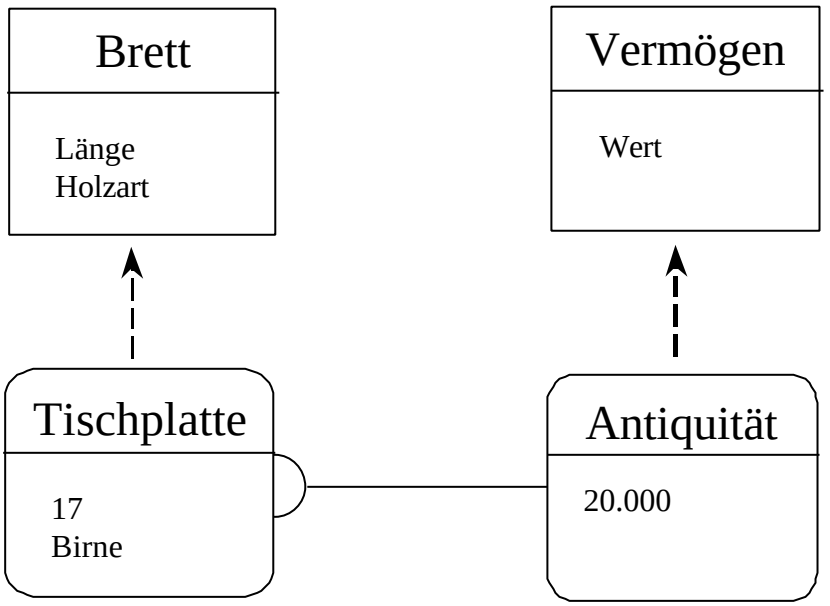

Abbildung 14: Wiederverwendung einer Instanz

In Abbildung 14 wird die konkrete Instanz der Klasse Brett „Tischplatte“ in einem anderen Kontext als „Antiquität“ wiederverwendet.

2.3.5.4 Realisierungshinweise

Das Rollenkonzept ermöglicht durch die unabhängige Zuordnung von Rollenobjekten zu einem Objekt die Wiederverwendung dieses Objektes in neuem Kontext.

Daneben ist es natürlich weiterhin möglich, auch die abstrakten Software-Bausteine zur Lösung von wiederkehrenden Problemstellungen wiederzuverwenden: Sowohl das Objekt selbst, als auch die Rollenobjekte sind als eigenständige Software-Bausteine implementiert und können so wiederverwendet werden.

Durch die Koppelung mehrerer Objekträume durch Vernetzung, wird es möglich, konkrete Instanzen, die in anderen Objekträumen beheimatet sind, in den eigenen Objektraum einzubinden. [CORBA]

In der Regel wird mit jedem einzelnen dieser Objekträume ein unterschiedlicher Kontext verbunden sein – es ist daher von besonderem Interesse, diese beiden Ansätze zu verbinden und das Rollenkonzept mit verteilten Objekten einzusetzen.

2.3.6 Lebenszyklus von Objekten versus Lebenzyklus von Rollen

2.3.6.1 Konzept

Wesentlich für die Umsetzung des Rollenkonzeptes ist die Dynamik in der Rollenzuordnung [Richardson91],[Pernici90]. Ein Objekt wechselt seine Rollen über die Zeit! Rollen sind nicht von der Geburt eines Objektes bis zu seinem Tod hin gleich – vielmehr kann über die Zuordnung von Rollen ein bestimmter Lebenszyklus eines Objektes abgebildet werden.

Damit in Zusammenhang steht die Frage, welche Voraussetzungen für die Zuordnung einer Rolle bestehen können: Bestimmte Rollen können nur in Verbindung mit anderen Rollen zugeordnet werden; die Zuordnung anderer Rollen hat bestimmte Folgen für das Objekt. Außerdem sollen gewisse Rollen nur bestimmten Objekten zugeordnet werden können.

Auch Rollen sind einem Lebenszyklus unterworfen, der eigentlich unabhängig von der Zuordnung zu Objekten sein soll.

2.3.6.2 Probleme

Laut [Kristensen95] wird durch Abgeben einer Rolle nicht unmittelbar deren Tod im Objektraum eingeleitet, sondern nur die letzte Phase in deren Lebenszyklus: Alle Objekte, die diese Rolle noch referenzieren, müssen bei Bedarf davon informiert werden, daß das Objekt diese Rolle nicht mehr inne hat, da das Rollenobjekt erst wirklich stirbt, wenn keine Referenzen mehr darauf existieren – bis zu diesem Zeitpunkt ist es aber dennoch bereits ungültig.

Außerdem muß zu diesem Zeitpunkt entschieden werden, welche Teile der Rolle in das Objekt übernommen werden und damit aus extrinsic zu intrinsic Eigenschaften mutieren.

2.3.6.3 Beispiele

Am Beginn des Lebenszyklus einer Person weist diese viele der Eigenschaften, die später als intrinsic betrachten werden, noch nicht auf. Diese werden erst im Laufe des Objekt-Lebenszyklus erlernt: In der Rolle Schüler werden gewisse Fähigkeiten aufgenommen, von denen einige nach dem Ablegen dieser Rolle in die Person selbst übernommen werden. Genau dies ist die Voraussetzung, daß die Person danach die Rolle Student annehmen kann.

Die Rolle Vater kann nur von Instanzen der Klasse Mann angenommen werden.

2.3.6.4 Realisierungshinweise

In einer Entwicklungsumgebung muß es möglich sein, in einer Rolle Abhängigkeiten und Lebenszyklen zu definieren.

[Pernici90] geht ausführlich auf die Bedeutung der Rollenzuordnungen für den Lebenszyklus von Objekten ein und stellt in diesem Zusammenhang die Notwendigkeit einer Basisrolle fest, die zum Zeitpunkt der Instanziierung eines Objektes aktiv ist. In der Interpretation dieser Arbeit stellt das Kernobjekt selbst diese Basisrolle bereits dar.

2.4 Subjektorientierte Konzept

[Harrison93] stellt ein Konzept vor, das das klassische Objektorientierte Konzept um eine neue Schicht erweitert, in der selbständig agierende Subjekte reduzierte Objekte interpretieren. Es handelt sich dabei um einen Ansatz, der ähnliche Ziele wie das Rollenkonzept des Objektorientierten Paradigmas verfolgt: Jedes Subjekt stellt einen eigenen Kontext dar und wird in diesem Sinne wie eine eigenständige Applikation behandelt.

2.4.1 Ziele

Die folgenden Punkte sind bei einer Entwicklungs-Umgebung, die dem Subjektorientierte Paradigma genügen soll, zu beachten:

- Applikationen, die mit den selben Objekten arbeiten, sollen unabhängig von einander entwickelt werden können.
- Applikationen, die mit den selben Objekten arbeiten, sollen nicht von einander abhängig sein müssen.
- Eine neue Applikation soll in einen Verbund eingefügt werden können, ohne die bestehenden Applikationen und die existierenden Objekte verändern zu müssen.
- Innerhalb einer Applikation sollen die Konzepte des Objektorientierten Paradigmas beibehalten werden.

2.4.2 Subjekt-Modell

Das Subjektorientierte Modell wird also als eine Hülle um einen Objektorientierten Kern eingeführt; in dieser Schicht soll es möglich sein, daß mehrere unabhängige Applikationen mit den selben Objekten arbeiten. Die Subjekte arbeiten also in einem gemeinsamen Objektraum, der durch ein gemeinsames Verzeichnis von Objekt-Identifiers repräsentiert wird.

Das Subjekt selbst hat keinen eigenen Zustand, sondern repräsentiert eine Auswahl der Eigenschaften der Objekte und fügt durch Interpretationen im jeweiligen Kontext weitere Zustandsinformationen hinzu. Das Subjekt ist also keine Sicht auf das Objekt im Sinne eines reinen Filters[4], sondern vielmehr eine Anreicherung durch eine subjektive Sichtweise.

Der Zustand des Kernobjektes ist nicht direkt (objektiv) ansprechbar, sondern nur in einer der subjektiven Sichtweisen. Ein Zustand des Objektes im Sinne des Objektorientierten Paradigmas existiert durch die Aufspaltung auf mehrere Subjekte nicht mehr.

Das klassische Objektorientierte Modell entspricht also der Sichtweise von einem Subjekt, die nun um andere Sichtweisen erweitert werden soll.

Die Ausprägung eines Subjektes wird auch als „Aktivierung“ bezeichnet.

[4] Das Konzept der „Sicht“ (View) als reiner Filter wird zum Beispiel in der MVC-Kopplung von Smalltalk eingesetzt. Das hier verwendete Konzept der Anreicherung mit subjektiven Informationen weicht von dieser traditionellen Verwendung des Begriffes „Sicht“ deutlich ab.

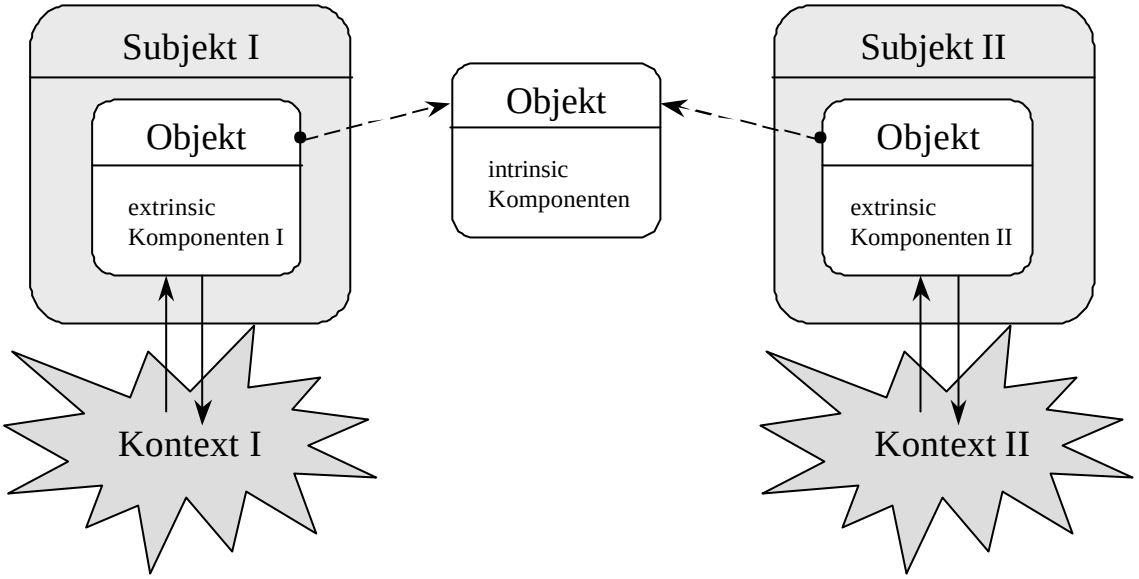

Abbildung 15 - Das Subjektorientierte Modell nach [Harrison93]

Jedes Subjekt teilt alle ihm bekannten Objekte in eine subjektive Klassenhierarchie ein, die dem jeweiligen Kontext genügt, und kann diese auch selbständig erweitern. Diese Erweiterung ist den anderen Subjekten nicht explizit bekannt; es muß also entsprechende Verfahren und Übereinkünfte geben, die eine jeweilige Einordnung und Verwendung erlauben.

Zur Einordnung unbekannter Objekte in die eigene subjektive Klassenhierarchie sind verschiedene Strategien denkbar:

- Durch Abstraktion wird eine Superklasse in der Hierarchie des erzeugenden Subjektes gesucht, die dem anfragenden Subjekt in seiner Hierarchie bereits bekannt ist; aufgrund fehlender genauerer Information wird das neue Objekt dort zugeordnet (explizit).
- Statt der Superklassen werden Geschwisterklassen für den Analogieschluß herangezogen; die Einordnung erfolgt wieder bei der gemeinsamen Superklasse (inferred).
- Gesucht wird eine Klasse, deren Protokoll dem des neuen Objektes ähnlich erscheint (interface based).

Wahrscheinlich wird in der Praxis eine mehrstufige Strategie, die alle Ansätze verfolgt und die jeweiligen Ergebnisse kombiniert, die besten Resultate erzielen. Kritisch erscheint jeweils die Abbruchsentscheidung, da nicht genau definierbar ist, welche Analogien noch zielführend sind, und wann das Objekt als „unbekannt“ zu klassifizieren ist, und welche Folgen das hat.

Analogieschlüsse sind generell nur möglich, wenn eine Klassenhierarchie bereits bis zu einer gewissen Tiefe vorhanden ist – ohne irgendein Wissen über die Umwelt sind auch keine Analogien erkennbar. Es stellt sich also das Problem der Ersteinführung neuer Subjekte in bestehende Universen.

2.4.3 Vergleich mit dem Rollenkonzept

2.4.3.1 Parallelen

Die Problemstellungen, die die beiden Konzepte motivieren, sind sehr ähnlich:

- konzeptionelle Trennung der intrinsic und der extrinsic Komponenten
- stärkere Betonung der Individualität von konkreten Identitäten
- Vermeidung von Mehrfachvererbung wegen Klassifizierungen nach verschiedenen Kriterien
- Umgang mit unterschiedlicher Behandlung von Objekten (zeitlich bzw. im Kontext)

2.4.3.2 Unterschiede

Die Lösungsansätze der beiden Konzepte gehen jedoch in grundsätzlich verschiedene Richtungen:

Das Rollenkonzept versucht das klassische Objektorientierte Modell evolutionär zu erweitern, indem die vorhanden Möglichkeiten zur Implementierung des Konzeptes verwendet werden; das Subjektorientierte Konzept führt jedoch – quasi revolutionär – eine neue Schicht ein, in der gänzlich neue Konzepte Anwendung finden.

Im Rollenkonzept bleibt das (erweiterte) Objekt weiterhin das einzige agierende Konstrukt, wohingegen im Subjektorientierten Konzept nur noch die Subjekte als Aktor fungieren.

Beispielsweise bedeutet unterschiedlicher Umgang mit Objekt-Identitäten im Rollenkonzept immer Veränderungen im Objekt und seinen Rollen; das Objekt handelt abhängig von seinen Rollen unterschiedlich. Im Subjektorientierten Konzept bleibt das Kernobjekt immer gleich – die Subjekte arbeiten in unterschiedlichem Kontext aber unterschiedlich damit.

2.5 Weiterführende Aspekte

Rollen, deren Zuordnung zu einem Objekt nicht mehr aktuell ist, können in bestimmten Fällen tatsächlich als nicht mehr existent betrachtet werden; im Allgemeinen wird man jedoch - unter Berücksichtigung des zeitlichen Aspektes im „Leben“ eines Objektes - annehmen können, daß das Verhalten einer einmal zugewiesenen Rolle das Objekt auch nach dem „Tod“ der Rolle weiter beeinflussen kann.

Von menschlicher Intelligenz verlangt man die Fähigkeit zu lernen. Warum soll dieser Aspekt in der Objektorientierten Programmierung gänzlich fehl am Platz sein? Es ist Verhalten vorstellbar, das aus Relevanz- oder Sicherheitsgründen auf eine bestimmte Rolle eingegrenzt werden kann und muß, aber es ist auch eine Art von Verhalten

vorstellbar, das gelernt wird und dem Objekt so über die Grenzen einer Rolle zur Verfügung steht. Gerade in der Kombination und Weiterentwicklung von erlerntem Verhalten liegt doch eine der Stärken menschlicher Intelligenz - warum sollten Objekte, die doch Dingen der realen Welt entsprechen sollen, das nicht dürfen? Allgemein sind Personen der realen Welt von außen durch die Maske einer bestimmten Rolle erreichbar - wie sie allerdings reagieren, entspricht bestimmten Verfügbarkeitsregeln innerhalb der Person.

Verhalten und Informationen, die sich eine Person aufgrund einer Tätigkeit in einer Firma angeeignet hat, können nach dem Ausscheiden aus dieser Firma aus Geheimhaltungsgründen explizit nicht mehr zur Verfügung stehen - auf der anderen Seite kann man aber von Personen, die einmal eine bestimmte Rolle inne gehabt haben, unter anderen Begleitumständen erwarten, daß diese Rolle sie als Person verändert hat und so auch im inaktiven Zustand weiter beeinflussend wirkt.

2.5.1 Lerntheorie

In[Milling95] wird Lernen als ein „Zielsuchender, zu einer Verhaltensänderung führender Rückkopplungsprozeß“ definiert.

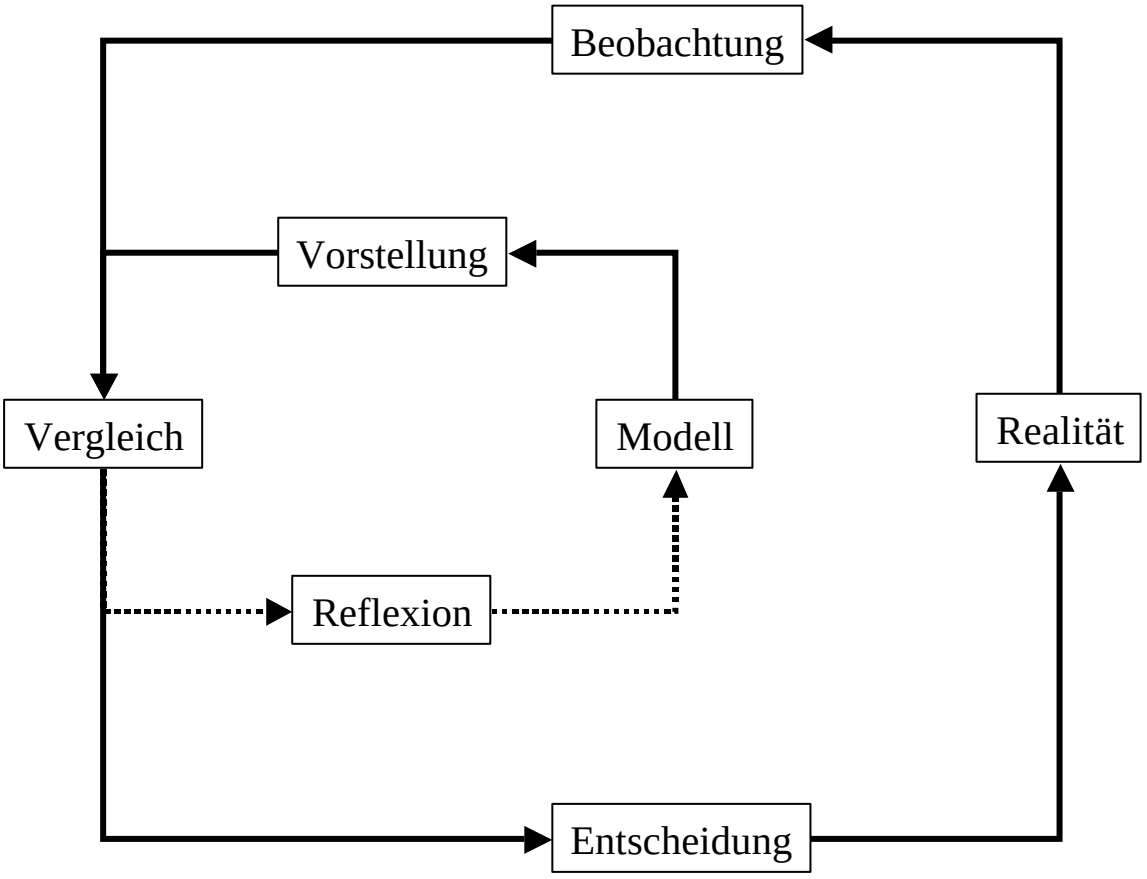

Abbildung 16: Lernprozesse nach[Milling95]

In Abbildung 16 werden Lernprozesse auf zwei Ebenen vorgestellt:

- In der ersten Stufe erfolgt ein bloßer Vergleich zwischen den Vorstellungen, die sich aus einem Modell der Realität ergeben, und der Beobachtung der Realität. Aus diesem Vergleich resultiert eine Entscheidung, die wiederum Auswirkungen auf die Realität hat (Lernen erster Ordnung).
- In der zweiten Stufe erfolgt zusätzlich nach dem Vergleich eine Reflexion der Vorstellungen, die das Modell gemäß der Abweichung von der Beobachtung weiterentwickelt (Lernen zweiter Ordnung).

Das bedeutet, daß es möglich, ist in einem konkreten Fall anders zu entscheiden, als es dem Modell entsprechen würde, ohne deshalb schon das Modell zu verändern. Andererseits kann das Ergebnis des Vergleiches auch eine Veränderung des Modells sein.

2.5.2 Anwendung auf das Rollenkonzept

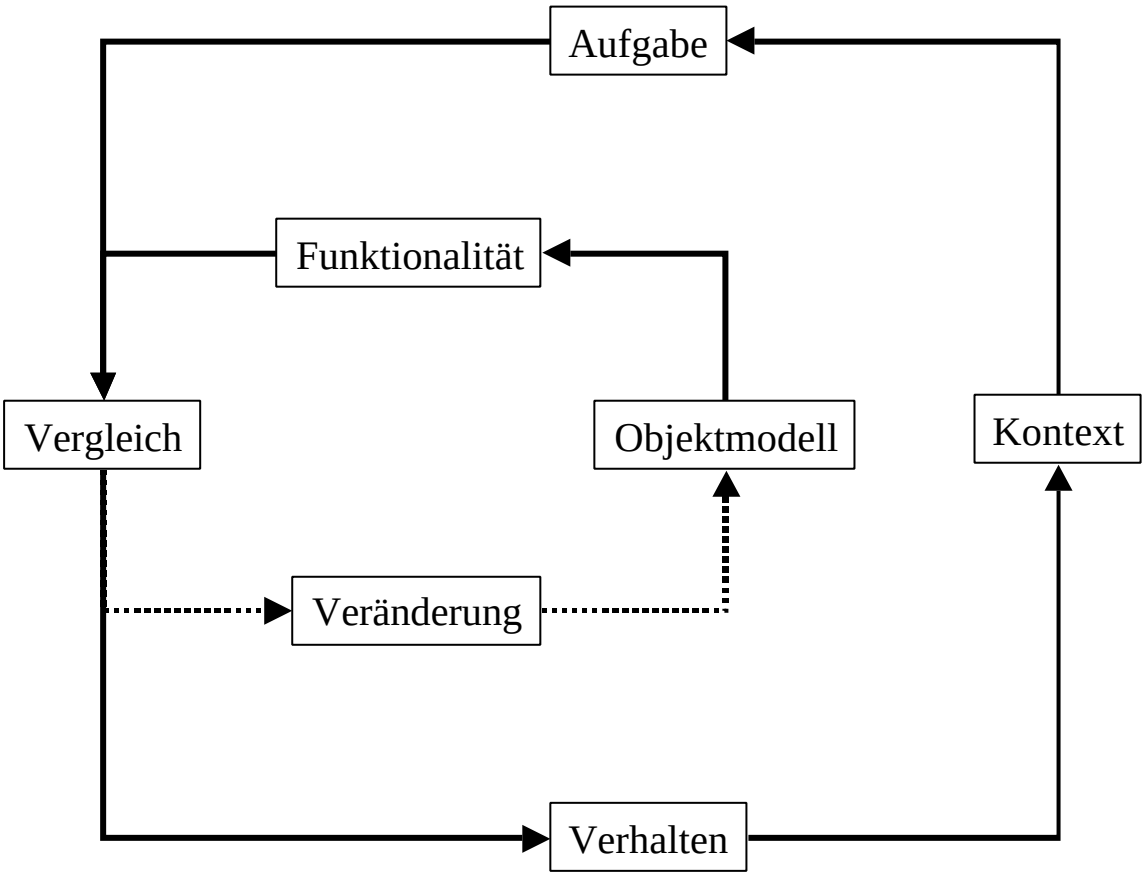

Abbildung 17: Lernprozesse im Rollenkonzept

In Abbildung 17 wird dargestellt, daß eine Aufgabe an ein Objekt aus einem bestimmten Kontext gestellt wird; das Objekt verhält sich in diesem Kontext nach Maßgabe der Funktionalität, die durch das Objektmodell definiert wird.

Reicht diese Funktionalität nicht zur Erledigung einer konkreten Aufgabe aus, so können extrinsic Komponenten im Rahmen einer Rolle hinzugefügt werden, ohne dadurch schon das eigentliche Objektmodell zu verändern (Lernen erster Ordnung).

Erst in einer zweiten Stufe wird die erlernte Funktionalität von der Rolle in das Objektmodell transferiert. Extrinsic Komponenten werden in intrinsic Komponenten umgewandelt und daraus resultiert dann auch eine Veränderung des Objektmodells (Lernen zweiter Ordnung).

Beide Stufen des Lernprozesses sind unverzichtbar: Die erste Stufe garantiert Dynamik und Flexibilität in der Reaktion auf eine konkrete Aufgabe, während die zweite Stufe die Persistenz erlernter Funktionalität über zeitliche und kontextliche Grenzen hinweg garantiert. Wann und ob diese Umwandlung erfolgt, ist abhängig vom jeweiligen Kontext.

3 Verteilte Systeme

3.1 Motivation

Der folgende Teil der Arbeit beschäftigt sich mit Überlegungen zu Objekten in verteilten Systemen. Unter den verbreiteten Standards zur Objektkommunikation in verteilten Systemen wurde die CORBA Spezifikation 2.0 [CORBA] ausgewählt, weil sie einen offenen, plattformunabhängigen Industriestandard darstellt.

CORBA beinhaltet aufbauend auf die grundlegenden Kommunikationsmechanismen sogenannte „Common Object Services“: Dienste, die allen Objekten zur Verfügung stehen und jeweils Lösungen für Standardprobleme anbieten.

Es wurden zwei der wesentlichsten Services herausgegriffen, die für die weitere Betrachtung im Zusammenhang mit dem Rollenkonzept von besonderem Interesse sein werden:

- Property-Service
- Relationship-Service

Für diese beiden Services wurde die Konzeption untersucht und in Form eines Objekt-Modells in OMT-Notation dargestellt, die internen Mechanismen, mit denen die einzelnen Teile der Konzepte zusammenwirken, beschrieben und eine kurze Anleitung zum Einsatz gegeben. Schließlich wurde versucht, jeweils ein repräsentatives Beispiel für die Verwendung der beiden Services zusammenzustellen.

Diese Betrachtung wurde in einer „Smalltalk“-basierten Entwicklungsumgebung [DST], [IDLReference96], [Programmer´sReference96], [User´sGuide96] durchgeführt, um die Objektorientierten Mechanismen in einer unverfälschten Form untersuchen zu können.

Im nächsten Kapitel schließt sich eine kurze Einführung in die Common Object Request Broker Architecture (CORBA) an; danach folgen einige Bemerkungen zur konkreten Umsetzung der CORBA-Spezifikationen in Distributed Smalltalk (DST). In Kapitel 3.4 wird das Surrogatkonzept, mit dessen Hilfe die Adressierung von verteilten Objekten in DST ermöglicht wird, erläutert. Kapitel 3.5 und 3.6 enthalten die Beschreibungen der beiden ausgewählten Services; Kapitel 3.7 enthält eine Zusammenfassung dieses Abschnittes.

3.2 Einführung in CORBA

Die Common Object Request Broker Architecture (CORBA) ist ein offener Standard der Object Management Group (OMG). Sie beschäftigt sich mit der Thematik von

verteilten Objekten und stellt Konzepte zur plattform-unabhängigen Kommunikation zur Verfügung.

Das grundlegende Kommunikationskonzept hierfür ist der ORB, der die Anfrage und die Antwort der kommunizierenden Objekte selbständig zum jeweiligen Adressaten transportiert. Es handelt sich also um ein Client-Server-Konzept, bei dem die Rollen der Objekte in jedem Anlaßfall neu vergeben werden.

Damit diese Kommunikation plattform-unabhängig funktioniert, müssen alle teilnehmenden Objekte ein Interface zur Verfügung stellen, das in der eigens dafür entwickelten Interface Definition Language (IDL) geschrieben sein muß. Im Falle der Kommunikation zwischen zwei Objekten in getrennten Objekt-Räumen wird die Anfrage vom ORB gemäß diesem Interface in ein allgemeines IDL-Statement übersetzt. Im Objekt-Raum des Empfänger-Objektes übersetzt der ORB dieses IDL-Statement wieder gemäß dem dort definierten Interface in die jeweilige „Raum-Sprache“. Das Empfänger-Objekt reagiert entsprechend dieser Anfrage und mit der Antwort wird analog vorgegangen.

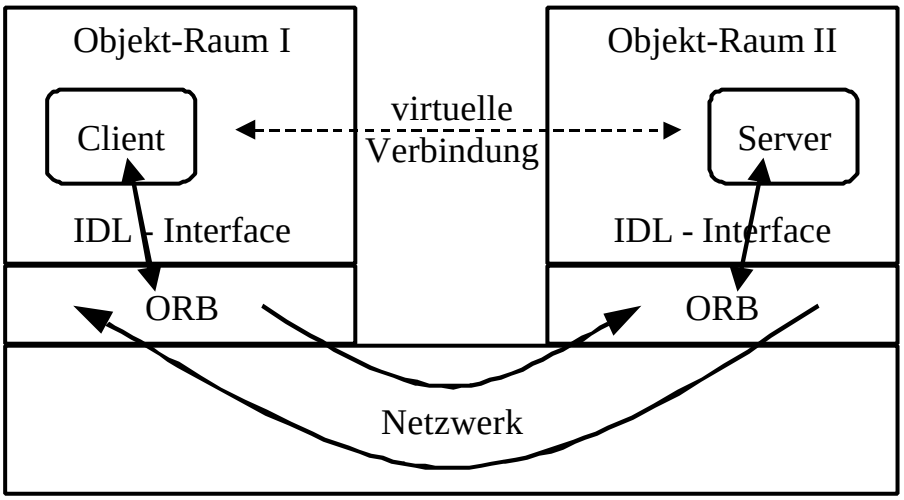

Abbildung 18: Das grundlegende Kommunikationskonzept von CORBA

Da diese Mechanismen der Kommunikation zwischen verteilten Objekten verhältnismäßig großen Overhead erzeugen, ist die Kommunikation über den ORB nur zwischen größeren Komponenten der Applikationen sinnvoll. Die Entscheidung, welche Objekte tatsächlich verteilt bleiben und wo mit Replikationen gearbeitet wird, stellt also bereits eine wesentliche Design-Entscheidung beim Einsatz von CORBA dar.

Aufbauend auf diesem grundlegenden Kommunikationskonzept wurden von der OMG Common Object Services (COS) spezifiziert, die der Erfüllung der grundlegenden Anforderungen an verteilte Systeme dienen sollen:

- Sicherheit
- Persistenz
- Transaktionskontrolle
- Verwaltung von Objekt-Lebenszyklen
- Benachrichtigung von Ereignissen
- Bereitstellung von Information
- Verwaltung von Beziehungen
- Konfigurierbarkeit

Diese COS sind auf System-Ebene spezifiziert und stehen allen Objekten zur Verfügung.

Zusätzlich existieren auf Applikations-Ebene Common Object Facilities wie User Interface, Install Shield, u.a.

3.3 Implementation in DST

Distributed Smalltalk erweitert die Grundversion von Visual Works um eine Menge von Klassen, die eine Entwicklungsumgebung schaffen, die den CORBA-Spezifikationen der OMG entspricht.

Es werden eigene Tools für die Definition der IDL-Statements (Editor, Compiler, etc.) und die Administration des ORBs angeboten. Es ist also möglich, wie gewohnt Software-Entwicklung unter Smalltalk zu betreiben; will man aber den eigenen Objekt-Raum verlassen und mit Objekten anderer Räume kommunizieren, so ist zusätzlich die Erstellung der IDL-Interfaces notwendig. Die Kommunikations-Mechanismen können auch lokal getestet werden.

In der Klassenhierarchie stehen abstrakte Klassen zur Verfügung, die die grundlegenden CORBA-Konzepte implementieren und als Wurzel der Klassenhierarchie der Applikations-Objekte dienen. Damit werden künftigen Objekten die Funktionalitäten, die zur verteilten Kommunikation notwendig sind, durch Vererbung implementiert.

Wenn gewisse Konventionen eingehalten werden (Bereitstellung der Methoden „abstractClassID“ und „CORBAName“) sind auch schon durch die Vererbung folgende COS verwendbar:

- Naming Service
- Event Notification Service
- Life Cycle Service
- Concurrency Control Service
- Transaction Service
- Relationship Service
- Property Service

Während diese COS volle Implementationen der OMG-Spezifikationen darstellen, wird daneben auch wieder das Smalltalk-spezifische Konzept der Trennung zwischen Model und View verwendet (Presentation / Semantic Split). Diese Trennung des eigentlichen Objekts von seiner Darstellung dient vor allem der Performance-Verbesserung, da nicht alle Veränderungen im User-Interface über das Netzwerk zu dem entfernt liegenden Objekt transportiert werden müssen, sondern nur genau definierte Zustands-Veränderungen. Die Definition dieser Trennung stellt eine weitere wesentliche Design-Entscheidung beim Einsatz von DST dar.

Die Adressierung von Objekten in anderen Objekt-Räumen wird in DST über ein Konzept von Stellvertreter-Objekten (Surrogate) realisiert, die die entfernten Objekte im eigenen Objekt-Raum repräsentieren und an sie geschickte Messages an den ORB weiterleiten. Im Gegensatz zum oben erwähnten Presentation / Semantic Split ist das Surrogatekonzept unumgänglicher Systembestandteil.

Die besondere Implementation des Relationship-Services in DST ermöglicht durch den Einsatz spezieller Container-Objekte (Building, Office, etc.) und die Ausnützung der Containment-Beziehungen der Applikations-Objekte zu diesen Container-Objekten eine de facto Umgehung des Naming Services.

Außerdem stehen verschiedene Administrations-Objekte (User, Session, Clipboard, Wastebasket, Orphanage) und einige Beispiel-Applikationen zur Verfügung.

In den folgenden Kapitel wird zunächst das Surrogatkonzept und dann jeweils ein COS näher beschrieben.

3.4 Surrogatkonzept

3.4.1 Motivation

Bei der Koppelung mehrerer unabhängig verwalteter Objekträume R_n zu einem virtuellen gemeinsamen Objektraum R entsteht die Notwendigkeit, alle Objekte im gesamten Objektraum R einheitlich ansprechen zu können. Dabei sollte es keinen

Unterschied machen, aus welchem Objektraum ein bestimmtes Objekt angesprochen wird. Der so entstandene Gesamtraum R soll den Objekten ermöglichen, über die real existierenden Verteilungsgrenzen hinweg zu kommunizieren.

Tatsächlich ist jedes Objekt in genau einem Objektraum R_h beheimatet. Innerhalb dieses Raumes wird das Objekt nach den in R_h üblichen Mechanismen angesprochen. Objekte aus einem fremden Objektraum R_f sollen dieses Objekt aber auch genauso ansprechen können, als wäre es in deren eigenem Objektraum beheimatet – also nach den in R_f üblichen Mechanismen.

Um also die oben gestellte Forderung nach einem gemeinsamen Objektraum R ohne Grenzen erfüllen zu können, muß jeder teilnehmende Objektraum R_n einen Mechanismus bereitstellen, der es den eigenen Objekten ermöglicht, Objekte fremder Objekträume so anzusprechen, als wären sie im eigenen Objektraum beheimatet. In DST wird dieser Mechanismus über Stellvertreter-Objekte (Surrogates) realisiert, die die entfernten Objekte im eigenen Objekt-Raum repräsentieren.

Die korrekte Weiterleitung der Anfragen an das tatsächlich angesprochene Objekt bleibt in der Verantwortung dieser Surrogates und damit in der Verantwortung des Systems und nicht in der der einzelnen Objekte.

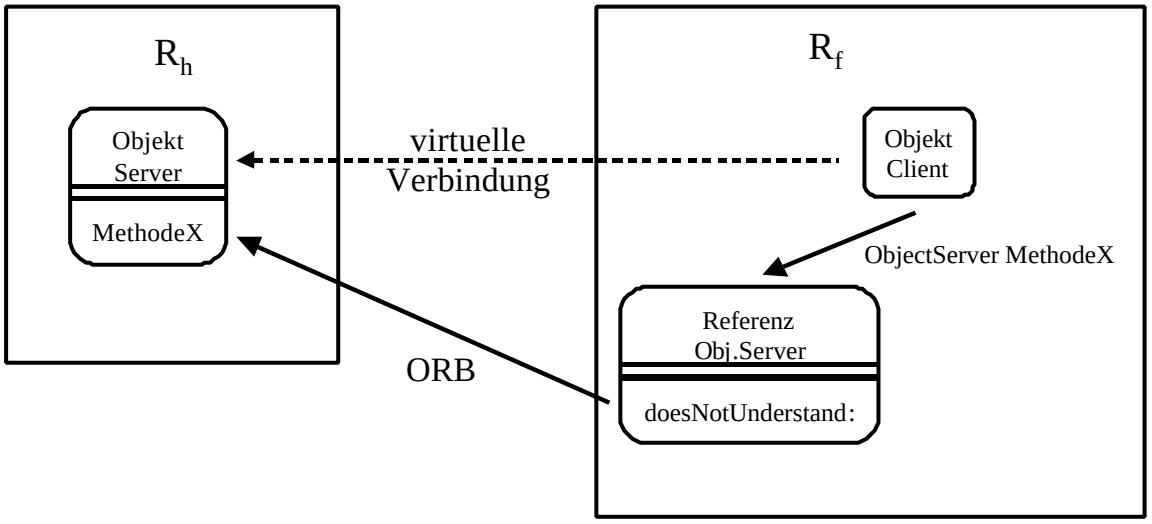

Abbildung 19: Das Surrogatkonzept

3.4.2 Objekt-Modell

In DST stehen eigene Klassen zur Verfügung, die diese Stellvertreter-Objekte bereitstellen und die vom System automatisch eingesetzt werden. An dieser Stelle sind auch eigene Klassen definiert, die es dem System ermöglichen, die Mechanismen in gekoppelten Objekträumen auch in einem einzelnen Objektraum zu testen.

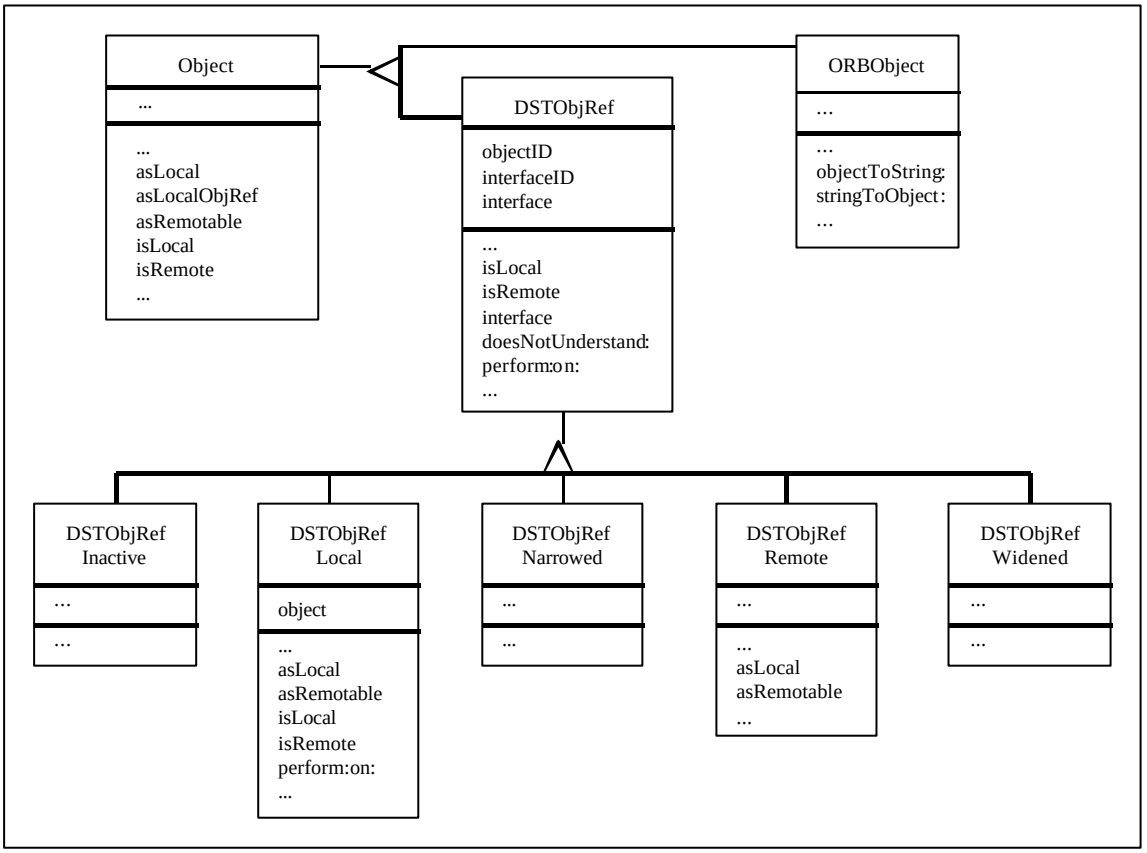

Abbildung 20: Objekt-Modell des Surrogatekonzeptes

Es gibt fünf verschiedene Arten von Objekt-Referenzen, die alle von der Klasse DSTObjRef abgeleitet werden; von diesen wurden zwei zur näheren Betrachtung ausgewählt:

- **DSTObjRefRemote**
  Diese Klasse erzeugt Stellvertreter-Objekte im eigenen Objektraum für Objekte in fremden Objekträumen; diese Surrogate dienen der eigentlichen Koppelung mehrerer verteilter Objekträume zu einem virtuellen gemeinsamen Objektraum.
- **DSTObjRefLocal**
  Diese Klasse erzeugt Stellvertreter-Objekte im eigenen Objektraum für Objekte im eigenen Objektraum; diese Surrogate dienen der Simulation der Mechanismen zur Koppelung mehrerer verteilter Objekträume in einem lokalen Objektraum.

Jede Instanz der Klasse DSTObjRef (bzw. einer ihrer Subklassen) hält in der Instanzvariable objectID den Universal Unique Identifier (UUID) des Objektes, welches sie referenziert. Da die Instanzen der Klasse DSTObjRefLocal Objekte im eigenen Objektraum referenzieren, haben sie zusätzlich auf der Instanzvariable object einen direkten Verweis auf dieses Objekt.

Bereits in der Wurzel-Klasse Object wird eine Menge von Methoden definiert, die also im Interface aller Objekte zu finden sind, aber in diesem Bereich der Klassenhierarchie redefiniert werden. Die in den relevanten Klassen definierten Rückgabewerte der einzelnen Methoden sind wie folgt:

Methode	RefRemote	RefLocal	Object
asLocal	RefRemote	Object asLocal	Object
asRemotable	RefRemote	RefLocal	Object / RefLocal
isLocal	False	Object isLocal	True
isRemote	True	True	False

Diese vier Methoden können also an alle Objekte im Objektraum gesandt werden; jedes Objekt antwortet darauf gemäß der Werte in der letzten Spalte – nur Instanzen der Referenz-Klassen antworten gemäß eigener Semantik:

- **DSTObjRefRemote**
  Die Referenz verweist auf ein Objekt in einem fremden Objektraum und kann daher nur sich selbst als Referenz retournieren
- **DSTObjRefLocal**
  Die Referenz verweist auf ein Objekt im eigenen Objektraum und retourniert daher bei expliziter Aufforderung dieses Objekt; sonst retourniert sie sich selbst als Referenz und erlaubt so die Simulation.
- **Object**
  Jedes Objekt retourniert grundsätzlich sich selbst – nur auf explizite Aufforderung retourniert sie bei Aktivierung der Option „Local RPC-Testing“ der Entwicklungsumgebung eine lokale Referenz auf sich selbst.

3.4.3 Interne Mechanismen

Das Programmier-Muster, das dem hier beschriebenen Konzept zugrundeliegt, ist das Proxy-Pattern aus [Gamma95]. Die Surrogate erfüllen die Aufgaben der Proxy-Objekte und sind Stellvertreter für die Objekte, auf die sie verweisen.

In diesen Surrogaten liegt keine eigene Funktionalität, außer der Zugriffssteuerung auf das referenzierte Objekt; in diesem Fall bedeutet dies, daß alle Methoden, die an ein

Surrogat gerichtet werden, über die Objektraum-Grenzen hinweg an dieses Objekt geschickt werden.

Diese Aufgabe erfüllen die Methoden doesNotUnderstand: bzw. perform:on: in der Klasse DSTObjRef: Dort wird die Anfrage in eine Form übersetzt (marshalling), die über den ORB transportiert werden kann, und die so übersetzte Anfrage über den ORB an das referenzierte Objekt geschickt.

Im Falle einer lokalen Referenz ist die Methode perform:on: so redefiniert, daß diese Art des Transportes nur gewählt wird, wenn die bereits oben angeführte Option aktiviert ist; andernfalls wird die Anfrage direkt (innerhalb des Objektraumes) an das Objekt geschickt.

3.4.4 Anleitung zum Einsatz

Die CORBA Spezifikation 2.0 [CORBA] enthält auch einen Standard, gemäß dem Objekte über einen String adressiert werden können, der u.a. ihre UUID und Interface-ID enthält.

Um für ein bestimmtes Objekt einen repräsentativen String gemäß dieser Spezifikation zu erzeugen, steht in der Klasse ORBObject die Klassenmethode objectToString: bereit, der das zu repräsentierende Objekt übergeben werden muß; diese Klassenmethode liefert dann einen String zurück, mit dem das repräsentierte Objekt aus allen Objekträumen gesucht werden kann.

Für diese Suche steht in der Klasse ORBObject ebenfalls die Klassenmethode stringToObject: zur Verfügung, der ein solcherart erzeugter String übergeben werden muß, um das von diesem String repräsentierte Objekt auffinden zu können. Diese beiden Methoden werden also nicht an das Objekt bzw. den String, der umgewandelt werden soll, gerichtet, sondern an die Klasse ORBObject.

Diese Implementation entspricht dem Singleton-Pattern aus [Gamma95], da die Umwandlungs-Funktionalität nicht Teil des Interfaces der Instanzen der Klassen Object bzw. String, sondern ausgelagert in die Klasse ORBObject selbst ist. Beim Aufruf der einzelnen Klassenmethoden müssen die jeweils notwendigen Informationen als Parameter übergeben werden – die Umwandlung führt dann in jedem Fall die Klasse ORBObject durch.

Wurde ein Objekt aus dem eigenen Objektraum gesucht, so liefert die Methode stringToObject: das Objekt selbst zurück[5]; wurde ein Objekt aber aus einem fremden Objektraum gesucht, so kann dort nicht das Objekt selbst zurück geliefert werden, da es

[5] Die Entwicklungsumgebung stellt die Option „Local RPC Testing“ zur Verfügung; ist diese Option aktiviert, wird auch bei der Suche eines Objektes aus dem eigenen Objektraum nicht das Objekt selbst, sondern eine Referenz zurückgeliefert – in diesem Fall ist dies eine Instanz der Klasse DSTObjRefLocal, also eine lokale Objekt-Referenz.

Diese Option erlaubt es, während der Entwicklung die Mechanismen der Kommunikation über den ORB zu testen, ohne tatsächlich über verteilte Objekte verfügen zu müssen.

in diesem Objektraum ja nicht existiert – vielmehr wird eine Objekt-Referenz, die in diesem Fall eine Instanz der Klasse DSTObjRefRemote ist, zurück geliefert.

Diese Objekt-Referenz steht als Surrogat stellvertretend für das Objekt aus dem Objektraum R_h im Objektraum R_f; an sie können alle Anfragen gerichtet werden, die eigentlich an das Objekt gerichtet werden sollten – das Surrogatkonzept sorgt dann selbständig dafür, daß die Anfragen nach den oben beschriebenen Mechanismen über den ORB in den Objektraum R_h zum eigentlich angesprochenen Objekt transportiert werden, und die Antworten auf dem gleichen Weg zurückkehren.

3.4.5 Beispiel

Im folgenden sollen die oben beschriebenen Mechanismen kurz durch ein Beispiel illustriert werden:

```
ziel := ShapeSO new.
ziel title: 'Ziel-Objekt'.
string := ORBObject objectToString: ziel
```

Das Ergebnis dieser Umwandlung der zuerst lokal erzeugten Instanz der Klasse ShapeSO in einen String nach der CORBA Spezifikation 2.0 ist:

'IOR:000000000000001749444c3a53686170652f536861706553656d3a312e300000000000
020000000100000074000000000000000050000000a000000107**ff47c4bec51000002000000**
000000000000000b000000107e25de3cf8ac0000020000000000000000810400000000000c44
53543a352e303a0499cf08081040030000000c302e302e302e303b31303236081040040000
000c302e302e302e303b3130323500000000000000028000100000000008302e302e302e3
000040200000000010**7ff47c4bec5100000200000000000000**'[6]

[6] In diesem String fett markiert ist die UUID des umgewandelten Objektes.

Wird nun dieser String beim folgenden Methoden-Aufruf als Parameter übergeben, so ist das Ergebnis dieser Umwandlung wieder das ursprüngliche Objekt:

```
ORBObject stringToObject: string
```

Abbildung 21: Inspector auf einem ShapeSO

Wird nun die Option „Local RPC Testing“ aktiviert, so ist das Ergebnis des selben Methoden-Aufrufes aber eine lokale Referenz auf das ursprüngliche Objekt:

```
ORBObject stringToObject: string
```

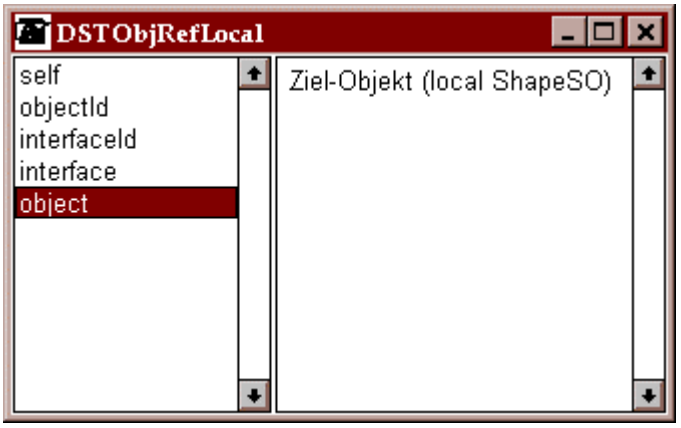

Abbildung 22: Inspector auf einem DSTObjRefLocal

An diese lokale Referenz kann nun – wie an jede Referenz – jeder Methoden-Aufruf geschickt werden, damit dieser dann über den ORB an das repräsentierte Objekt weitergeleitet wird, ohne weitere Benutzer-Interaktionen erforderlich zu machen:

```
(ORBObject stringToObject: string) title
```

Dieser Methoden-Aufruf liefert den Titel des Objektes zurück, obwohl die Methode title an die (lokale) Referenz des Objektes geschickt wurde:

```
'Ziel-Objekt'
```

3.5 Property-Service

3.5.1 Motivation

In vielen Fällen ist es sinnvoll, Informationen über ein Objekt (Meta-Informationen) von den Information des Objektes selbst getrennt aufzubewahren und zugänglich zu machen.

Gerade im Falle von verteilten Objekten kann der Ort der Meta-Information unter Umständen ein anderer sein als der Ort des Objektes selbst. Dann ist es zum Beispiel bei Kommunikation über ein Objekt (im Gegensatz zur Kommunikation mit einem Objekt) nicht mehr notwendig, an diesen Ort zu gehen.

Einige dieser Meta-Informationen sind nach ihrer Natur beeinflußbar (Title, Owner, Access Control List, etc.) – andere ergeben sich aus ihrer Natur automatisch (Creator, Creation Date, Modification Date, etc.)

Diese Überlegungen veranlaßten die OMG, ein COS in ihre Spezifikation aufzunehmen, das diese Aufgaben erfüllen soll.

3.5.2 Objekt-Modell

In DST steht ein Mechanismus zur Implementation dieses COS zur Verfügung, den ein Applikations-Objekt automatisch erbt, wenn es an der richtigen Stelle der Klassenhierarchie eingefügt wird (von DSTapplicationObject abgeleitet).

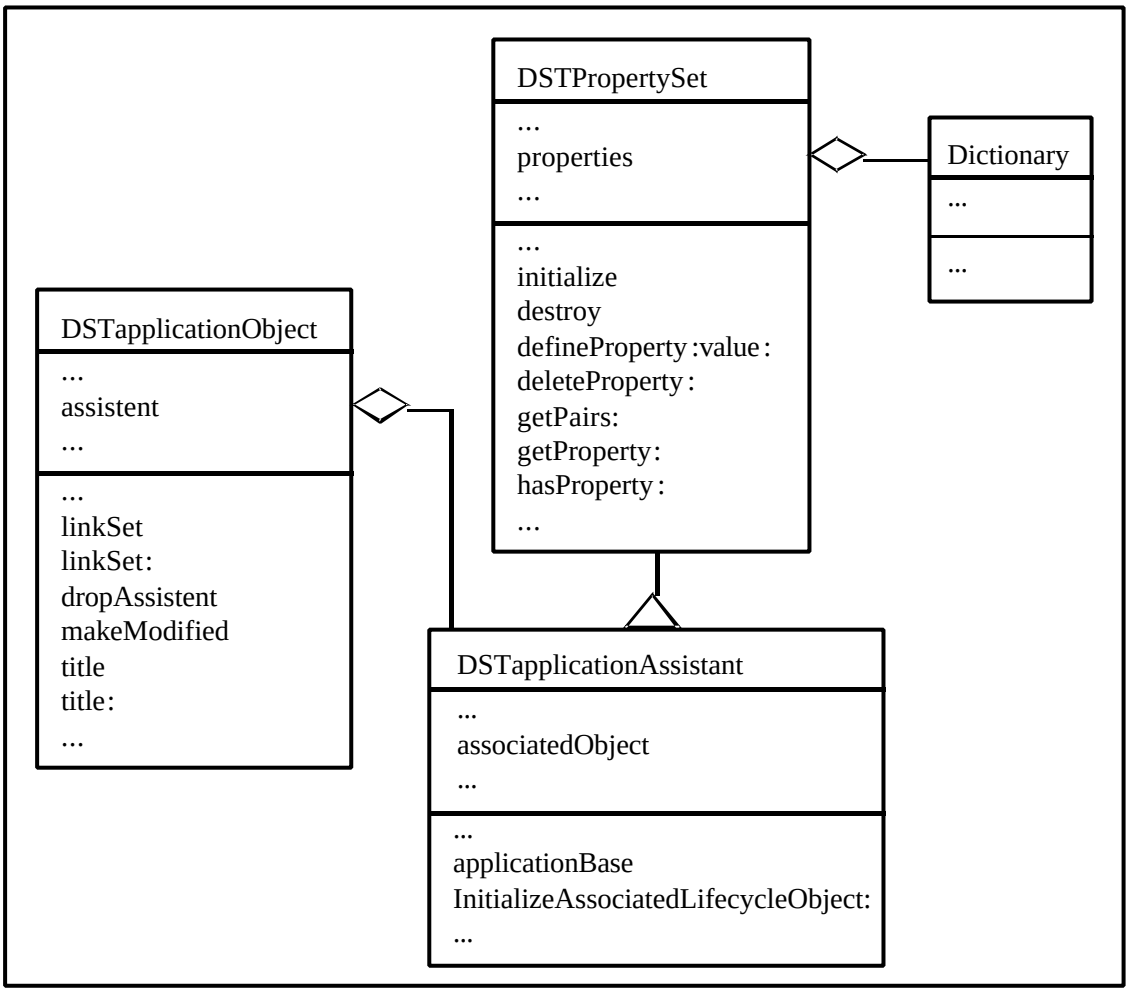

Abbildung 23: Objekt-Modell des Property-Services

Jede Instanz der Klasse DSTapplicationObject bzw. einer ihrer Subklassen enthält eine Instanzvariable assistant, auf der eine Instanz der Klasse DSTapplicationAssistant aggregiert liegt. Dieses Assistenz-Objekt ist über die Methode linkSet zugänglich und wird beim ersten Aufruf dieser Methode mit Default-Werten angelegt. Durch linkSet: kann ein anderer Assistent zugewiesen werden und durch dropAssistent kann er zerstört werden.

Die Klasse DSTapplicationAssistant erbt den Großteil der für das Property-Service notwendigen Funktionalität von der Klasse DSTPropertySet und fügt u.a. die Instanzvariable associatedObject und deren lesende Zugriffs-Methode applicationBase hinzu: Auf dieser Instanzvariable wird das eigentliche Applikations-Objekt referenziert

(„der Assistent kennt seinen Meister“). Dadurch wird der Assistent zu einem zugeordneten Property-Set.

In der Klasse PropertySet ist die Instanzvariable properties definiert, auf der beim Initialisieren ein Dictionary abgelegt wird. Mittels der in Abbildung 23 dargestellten Zugriffs-Methoden können in dieses Dictionary Einträge eingefügt, gelöscht und abgerufen werden.

3.5.3 Interne Mechanismen

Der Assistent wird nicht bei der Instanziierung eines Applikations-Objektes angelegt, sondern erst beim ersten Zugriff über die Methode linkSet des Applikations-Objektes; im Auslieferungs-Zustand von DST werden dabei die folgenden Eigenschaften an- und mit den folgen Werten belegt:

- Title ?
- Creation Date Jetzt
- Modification Date Jetzt
- Product Name der in der Klasse vereinbarte Produktname

Zu beachten ist dabei, daß die Eigenschaft Creation Date also in der Regel nicht mit dem tatsächlichen Instanziierungs-Zeitpunkt des Applikations-Objektes belegt wird, sondern mit jenem des Assistenten, der davon abweichen kann. Wird das in DST empfohlene Containment-Konzept verwendet, tritt dieses Problem nicht auf, da beim Einrichten der Containment-Beziehung auch der Assistent für das Relationship-Service notwendig ist. Weicht man jedoch von diesem empfohlenen Weg ab, ist die Konsistenz der Belegung dieser Eigenschaft nicht mehr garantiert!

Das Anlegen der Eigenschaften im Dictionary properties erfolgt durch Aufruf der Methode defineProperty:value: des Assistenten, die eine bestehende Eigenschaft mit dem neuen Wert überschreibt und eine neue Eigenschaft mit dem übergebenen Wert neu anlegt.

Für die Eigenschaft Title sind im Applikations-Objekt eigene schreibende bzw. lesende Zugriffs-Methoden definiert, da diese Eigenschaft beeinflußbar ist. Die Eigenschaft Modification Date wird durch die Methode makeModified des Applikations-Objektes aktualisiert. Die anderen beiden Eigenschaften können nicht beeinflußt werden.

3.5.4 Anleitung zum Einsatz

Neue Eigenschaften müssen in der Methode linkSet definiert und mit Default-Werten ausgestattet werden; ist die Möglichkeit zur Beeinflussung gewünscht, müssen eigene Zugriffs-Methoden im Applikations-Objekt definiert werden.

In der mitgelieferten Entwicklungs-Umgebung können alle Eigenschaften über den Menü-Befehl Properties eingesehen werden.

3.5.5 Verteilungskonzept

In diesem Abschnitt soll kurz darauf eingegangen werden, wie die einzelnen Teile des oben beschriebenen Objekt-Modells verteilt werden können.

Kommen die Standard-Mechanismen von DST zum Einsatz, wird der Assistent natürlich im selben Objekt-Raum R_h angelegt, in dem auch das Applikations-Objekt selbst angelegt wurde; soll dieses Standard-Verhalten nun bewußt umgangen werden, um die eingangs erwähnten Vorteile nutzen zu können, so besteht die Möglichkeit, eine Instanz der Klasse DSTapplicationAssistant in einem anderen Objekt-Raum R_f anzulegen und dem Applikations-Objekt zuzuweisen.

Das in Kapitel 3.4 beschriebene Surrogatkonzept sorgt dann automatisch dafür, daß im Objekt-Raum R_f eine Referenz auf das Applikations-Objekt im Objekt-Raum R_h und im Objekt-Raum R_h eine Referenz auf den Assistenten im Objekt-Raum R_f angelegt wird.

Alle in diesem Kapitel beschriebenen Mechanismen funktionieren dann unter Verwendung der in Kapitel 3.4 beschriebenen Mechanismen, ohne weitere Benutzer-Interaktionen erforderlich zu machen.

3.5.6 Beispiel

Im folgenden wird in einem kurzen Beispiel eine Anleitung zur Erweiterung des bestehenden Systems um die Eigenschaft Geschlecht und deren Verwendung geben:

```
linkSet
    "return the LinkSet of the receiver"
    assistant isNil
        ifTrue:
            [self linkSet: DSTapplicationAssistant new.
            assistant defineProperty: #modified value: Timestamp now formattedString.
            assistant defineProperty: #title value: '?'.
            assistant defineProperty: #geschlecht value: '?'.
            assistant defineProperty: #created value: Timestamp now formattedString.
            assistant defineProperty: #productName value: self productName].
    ^assistant asRemotable

geschlecht
    "Answer the value of geschlecht."
    ^self linkSet asLocal getProperty: #geschlecht

geschlecht: aValue
    "Set the value of geschlecht."
    self linkSet defineProperty: #geschlecht value: aValue
```

Um diese beiden neuen Methoden auch über den ORB ansprechbar zu machen, muß das Interface **::PSSplit::ApplicationSem** um den folgenden Ausdruck erweitert werden:

attribute string geschlecht;

3.6 Relationship-Service

3.6.1 Motivation

In vielen Fällen ist es notwendig, Beziehungen zwischen Objekten darzustellen und zu verwalten:

- Gruppen von Objekten sollen nach außen als Einheit angesprochen werden können.
- Objekte sollen von der Existenz von verbundenen Objekten abhängig sein.
- Die Verbindung zwischen Objekten soll unabhängig vom aktuellen Ort der Objekte bestehen.
- Die Verbindungen sollen mit Eigenschaften versehen werden können.
- u.s.w.

Diese Überlegungen veranlaßten die OMG, ein COS in ihre Spezifikation aufzunehmen, das diese Aufgaben erfüllen soll.

3.6.2 Objekt-Modell

Die Verbindungen (Links) sind in DST selbst Objekte mit wechselseitigen Verweisen und anderen Eigenschaften gemäß dem folgenden Objekt-Modell:

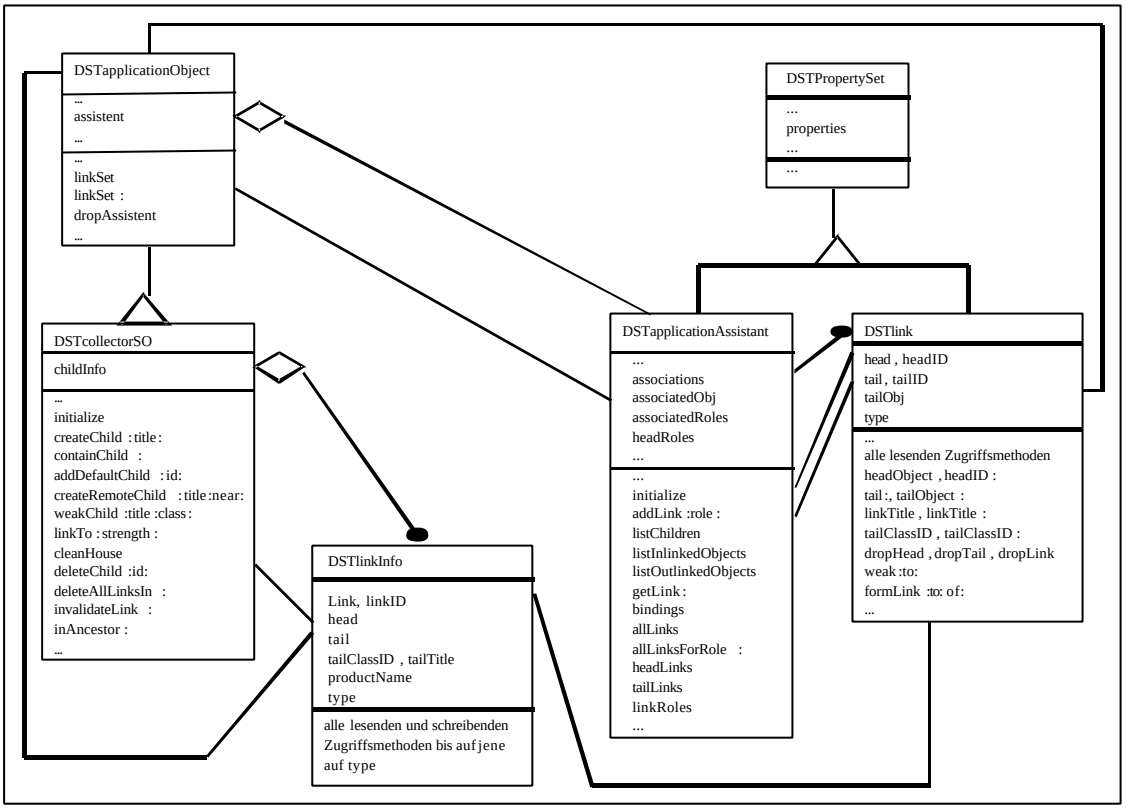

Abbildung 24: Objekt-Modell des Relationship-Services

Das Relationship-Service bedient sich ebenfalls des in Kapitel 3.5 beschriebenen Assistenten, der neben der Zuordnungs-Instanz-Variablen associatedObject auch noch mehrere Dictionaries zur Verwaltung der Beziehungen aufweist: associations, associatedRoles, headRoles. In der Variable lastKey wird der jeweils aktuelle Index-Stand geführt.

Im Dictionary associations ist unter jedem Rollen-Typ wieder ein Dictionary angelegt, das die Beziehungen von diesem Typ enthält. Jede Beziehung erhält durch Wartung der Variable lastKey einen über alle Dictionaries dieses Assistenten eindeutigen Indexwert, unter dem sie im jeweils richtigen Dictionary abgelegt wird.

Eine Beziehung ist eine Instanz der Klasse DSTLink, die wie die Klasse DSTApplicationAssistant eine Subklasse der Klasse DSTPropertySet ist. Sie fügt dieser die Instanz-Variablen head, headID, tail, tailID, tailObject und type hinzu: In den Variablen head und tail sind die Assistenten der beiden an der Beziehung teilnehmenden Objekte abgelegt, wohingegen in den Variablen headID und tailID der Index, den die Beziehung im jeweiligen Assistenten erhalten hat, gespeichert wird.

Zusätzlich enthält die Variable tailObject auch noch eine direkte Referenz auf das Objekt, wobei als head- bzw. tail-Objekt immer (auch bei an sich symmetrischen Beziehungen) das anfordernde bzw. das angeforderte Objekt betrachtet wird.

Da die Entwicklungsumgebung in DST sehr stark auf das Konzept der Containment-Links aufgebaut ist, werden im folgenden kurz die besonderen Erweiterungen betrachtet, die nur bei Beziehungen, die von Instanzen der Klasse DSTCollectorSO ausgehen, hinzukommen: Die Klasse DSTCollectorSO ist eine Subklasse der Klasse DSTApplicationObject und erweitert diese im wesentlichen um die Instanz-Variable childInfo. Dort wird ein Dictionary geführt, in das alle Beziehungen, die von diesem Collector ausgehen, in der Form von Beziehungsinformationen unter ihrem Indexwert eingetragen werden. Beziehungsinformationen sind Instanzen der Klasse DSTLinkInfo und stellen quasi statische Sichten auf die wesentlichen Informationen über eine Beziehung dar. Insbesondere kennen sie neben dem eigentlichen Link-Objekt die beiden an der Beziehung beteiligten Objekte direkt.

3.6.3 Interne Mechanismen

Eine neue Beziehung wird definiert, indem die Methode createLink:tail: an den Assistenten eines Objektes geschickt wird; dieses Objekt wird dadurch zum Kopf der neuen Beziehung – der Typ der Beziehung und der Assistent des Ziel-Objektes müssen der Methode als Parameter mitgegeben werden.

Dort wird dann eine neue Instanz der Klasse DSTLink erzeugt und an diese die Methode formLink:to:of: mit entsprechender Parameter-Weiterleitung geschickt.

Die Instanz der Klasse DSTLink setzt dadurch ihre Instanzvariablen, definiert auf diesem Weg eine neue gültige Beziehung und läßt diese Beziehung durch Aufruf der Methode addLink:role: in beiden teilnehmenden Assistenten eintragen. Als Parameter gibt sie dabei sich selbst und den jeweiligen Rollentyp (aus Sicht des Kopfes und des Zieles sind das unterschiedliche!) mit.

Der Assistent trägt die neue Beziehung im jeweils richtigen Dictionary unter Verwendung eines neuen Indexwertes (Variable lastKey) ein und löst ein entsprechendes Ereignis „linkAdded“ aus.

Dieses Ereignis wird im oben beschriebenen Fall von Containment-Beziehungen in der Entwicklungsumgebung von DST dazu verwendet, um die neue Beziehung durch Aufruf der Methode addDefaultChild:id: im Kopf-Objekt (das in diesem Fall eine Instanz der Klasse DSTCollectorSO oder einer ihrer Subklassen sein muß) in das Dictionary childInfo einzutragen. Eingetragen wird aber nicht eine Referenz auf das Beziehungs-Objekt selbst, sondern auf ein Beziehungsinformations-Objekt, das zu diesem Zeitpunkt vom Beziehungs-Objekt durch Aufruf der Methode asDSTLinkInfo angefordert wird.

Das Beziehungs-Objekt erzeugt dabei eine neue Instanz der Klasse DSTLinkInfo und versieht diese mit den notwendigen Informationen über sich selbst zu diesem Zeitpunkt.

3.6.4 Anleitung zum Einsatz

Dieses Service wird bei Navigation und Manipulation in der Entwicklungumgebung von DST ständig verwendet und gewartet. Für Eigenimplementationen stehen geeignete Methoden wie oben beschrieben zur Verfügung.

Es können Beziehungen unterschiedlichen Typs definiert werden. Im Auslieferungs-Zustand von DST sind die folgenden Typen möglich:

- Containment
- Reference
- Designation
- Weak

Die einzelnen Typen werden durch jeweils eindeutige Schlüsselwerte festgelegt, die in der globalen Variable CORBAConstants abgelegt sind. In der Klasse DSTLink stehen Klassen-Methoden zum einfacheren Zugriff auf diese Schlüsselwerte zur Verfügung.

Auch die Head- bzw. Tail-Rolle bei jedem Typ stellt jeweils einen eigenen Typ dar; in DST wird dies dadurch ausgedrückt, daß die Schlüsselwerte der Head-Rollen der einzelen Typen ungerade Werte aufweisen und die jeweilige Tail-Rolle immer um 1 höher verschlüsselt wird als die Head-Rolle gleichen Typs.

Diese Beziehungstypen werden vom hier behandelten Relationship-Service nur angeboten, um von anderen Services interpretiert werden zu können; wie im Falle von Containment-Beziehungen beschrieben, können andere Teile des Systems bzw. andere Services abhängig vom im Relationship-Service definierten Typ einer Beziehung unterschiedlich reagieren.

Im folgenden soll kurz auf die wesentlichen Bedeutungen der vier verfügbaren Typen eingegangen werden:

3.6.4.1 Containment-Beziehung (Head 1 / Tail 2)

- Beide Teilnehmer sind sich ihrer Teilnahme an der Beziehung bewußt.
- Jedes Objekt kann nur einmal in der Tail-Rolle an einer Beziehung diesen Typs teilnehmen.
- Solange die Beziehung existiert, ist die Existenz beider teilnehmenden Objekte gesichert.

Dies ist die Beziehung mit der stärksten Bindungsstärke – sie kann als „beinhaltet“ / „ist Inhalt von“ interpretiert werden; jeder Container kennt seinen Inhalt und jedes Objekt weiß, worin es enthalten ist, kann aber nur in einem Container enthalten sein.

3.6.4.2 Reference-Beziehung (Head 3 / Tail 4)

- Beide Teilnehmer sind sich ihrer Teilnahme an der Beziehung bewußt.
- Solange die Beziehung existiert, ist die Existenz beider teilnehmenden Objekte gesichert.

Dies ist die Beziehung mit der zweit-stärksten Bindungsstärke – sie kann als „referenziert“ / „referenziert“ interpretiert werden; die referentielle Integrität wird dabei vom Life-Cycle-Service überwacht.

3.6.4.3 Designation-Beziehung (Head 5 / Tail 6)

- Beide Teilnehmer sind sich ihrer Teilnahme an der Beziehung bewußt.
- Das in der Head-Rolle an der Beziehung teilnehmende Objekt wird informiert, wenn das in der Tail-Rolle an der Beziehung teilnehmende Objekt nicht mehr verfügbar ist.

Dies ist die Beziehung mit der dritt-stärksten Bindungsstärke – sie kann als „ist interessiert an“ / „kennt“ interpretiert werden; die referentielle Integrität wird dabei vom Life-Cycle-Service nur einseitig überwacht.

3.6.4.4 Weak-Beziehung (Head 7 / Tail 8)

- Nur das in der Head-Rolle an der Beziehung teilnehmende Objekt kennt die Beziehung.

Dies ist die Beziehung mit der schwächsten Bindungsstärke – sie kann als „kennt“ / „“ interpretiert werden; die referentielle Integrität wird dabei vom Life-Cycle-Service nicht überwacht.

3.6.5 Verteilungskonzept

In diesem Abschnitt soll kurz darauf eingegangen werden, wie die einzelnen Teile des oben beschriebenen Objekt-Modells verteilt werden können.

Im allgemeinen kann davon ausgegangen werden, daß die beiden an einer Beziehung teilnehmenden Applikations-Objekte nicht im selben Objekt-Raum beheimatet sind, denn gerade in der Verknüpfung von Objekten verschiedener Objekt-Räume liegt eine der Haupt-Motivationen des Relationship-Services.

Die Möglichkeiten der Verteilung der Assistenten der beiden an der Beziehung teilnehmenden Applikations-Objekte wurden bereits in Kapitel 3.5 ausführlich behandelt, daher soll es im folgenden besonders um die Wahl des Ortes gehen, an dem das eigentliche Beziehungs-Objekt angesiedelt wird.

Kommen die Standard-Mechanismen von DST zum Einsatz, wird diese Instanz der Klasse DSTLink in jenem Objekt-Raum R_h angelegt, in dem auch jenes Applikations-Objekt beheimatet ist, das die Beziehung angefordert hat – also das Kopf-Objekt der

Beziehung. Soll dieses Standard-Verhalten nun bewußt umgangen werden, kann in einem beliebigen Objekt-Raum R_b eine neue Instanz der Klasse DSTLink angelegt werden und an diese die Methode formLink:to:of: geschickt werden.

Als Parameter müssen dieser Methode dann jeweils Referenzen im Objekt-Raum R_b auf die Assistenten der beiden an der Beziehung teilnehmenden Applikations-Objekte übergeben werden.

Alle in diesem Kapitel beschriebenen Mechanismen funktionieren dann unter Verwendung der in Kapitel 3.4 beschriebenen Mechanismen, ohne weitere Benutzer-Interaktionen erforderlich zu machen.

3.6.6 Beispiel

Das folgende kurze Beispiel erzeugt eine neue Containment-Beziehung zwischen einem Kopf- und einem Ziel-Objekt. Wird dieses Code-Stück in einem Workspace ausgeführt, dann erzeugt es einen neuen Folder, in dem ein neues Shape enthalten ist:

```
kopf := FolderSO new.
kopf title: 'Neuer Kollektor'.
ziel := ShapeSO new.
ziel title: 'Ziel-Objekt'.
kopf linkSet createLink: 1 tail: (ziel linkSet).
```

Um den neu erzeugten Folder auch in der Entwicklungsumgebung von DST sichtbar zu machen, kann mit Hilfe der folgenden Zeilen eine Referenz-Beziehung in das Building (IP-Adresse 0.0.0.0) aufgenommen werden:

```
ns := ORBObject namingService.
building := (ns contextResolve: ('0.0.0.0' asDSTName)) asLocal.
building linkSet createLink: 3 tail: (kopf linkSet).
```

3.7 Zusammenfassung

Abschließend kommt man zu der Ansicht, daß die Implementation der CORBA-Spezifikationen in DST stark auf den Einsatz im Rahmen der mitgelieferten Entwicklungsumgebung zugeschnitten ist. Hält man sich im Umgang mit verteilten Objekten an die vordefinierten Strukturen und Werkzeuge, laufen alle internen Mechanismen in Einklang mit den Spezifikationen.

Versucht man jedoch die Mechanismen für Eigenentwicklungen außerhalb der Entwicklungsumgebung einzusetzen, stößt man teilweise auf nicht ganz nachvollziehbare Struktur-Entscheidungen, die wahrscheinlich vor allem auf Redundanzen zur Steigerung der Leistungsfähigkeit zurückzuführen sind.

Die in der CORBA Spezifikation 2.0 geforderte einheitliche Adressierung von Objekten, die auf verschiedene Objekt-Räume verteilt sind, wird in DST über ein

Surrogatekonzept umgesetzt. Der Einsatz von eigenen Stellvertreter-Objekten (Objekt-Referenzen) erlaubt es, Objekte in jedem der über den ORB verbundenen Objekt-Räume so anzusprechen, als wären sie im eigenen Objekt-Raum beheimatet.

Allgemein stellt das Relationship-Service eine Infrastruktur zur Realisierung von Verknüpfungen von Objekten mit einem Mindestmaß an Semantik zur Verfügung, wohingegen das Property-Service eine Infrastruktur zur Verwaltung von Zusatz-Informationen über Objekte zur Verfügung stellt.

Beide Services werden über das zentrale Konzept des Assistenten realisiert, der Information über das Objekt und seine Verknüpfungen von den Informationen des Objektes selbst trennt.

Beide Services gehen nicht über das Maß einer grundlegenden Infrastruktur hinaus und behandeln nicht oder nur geringfügig die Semantik der abgebildeten Beziehungen bzw. erlauben keine Definition von speziellen Zugriffsmethoden auf die ausgelagerten Informationen.

4 Spezifikation des Roleification-Services

4.1 Zielsetzung

In Kapitel 2 wurde das Rollenkonzept des Objektorientierten Paradigmas ausführlich behandelt. Das Rollenkonzept hat die Trennung von intrinsic und extrinsic Komponenten der Objekte zum Ziel, um einen höheren Grad an Flexibilität bei der Entwicklung von Applikationen zu erreichen.

In Kapitel 3 wurden einige Mechanismen des CORBA-Standards vorgestellt, die eine plattformunabhängige Koppelung von verteilten Objekträumen zum Ziel hat.

Die vorliegende Arbeit beschäftigt sich mit der Kombination dieser beiden Ideen: Die Flexibilität, die das Rollenkonzept durch das Zuordnen und Abgeben von Rollen bietet, ermöglicht die Wiederverwendung von Objektinstanzen in neuem Kontext; die CORBA-Mechanismen ermöglichen eine solche Wiederverwendung über Objektraumgrenzen hinweg.

Im folgenden Kapitel wird das Konzept vorgestellt, das im Rahmen dieser Arbeit als ein Service für CORBA in DST implementiert wurde.

4.2 Konzept

In diesem Kapitel wird allgemein spezifiziert, wie das Roleification Service in den CORBA-Standard [CORBA] eingebettet werden soll. Jede Implementation muß auf der jeweiligen Plattform dieser Spezifikation entsprechen.

4.2.1 Grundlegende Mechanismen im Objektdesign

Es soll möglich sein intrinsic und extrinsic Komponenten eines Subjektes von einander zu trennen. Die intrinsic Komponenten sollen dabei in einem „Kern-Objekt“ angesiedelt werden, die extrinsic Komponenten eines Kontextes begründen jeweils ein „Rollen-Objekt“ mit eigenständiger Identität.

Alle Objekte sollen unabhängig von einander in Klassenhierarchien eingeordnet werden können. In jeder dieser Klassenhierarchien sollen die gewohnten Vererbungsmechanismen des Objektorientierten Paradigmas Anwendung finden.

Zwischen einem „Kern-Objekt“ und beliebig vielen „Rollen-Objekten“ sollen Beziehungen eines eigenen Typs („Roleification“) definiert werden können. Auf dieser Ebene ist es also möglich, die einzelnen Klassenhierarchien orthogonal mit einander zu verknüpfen.

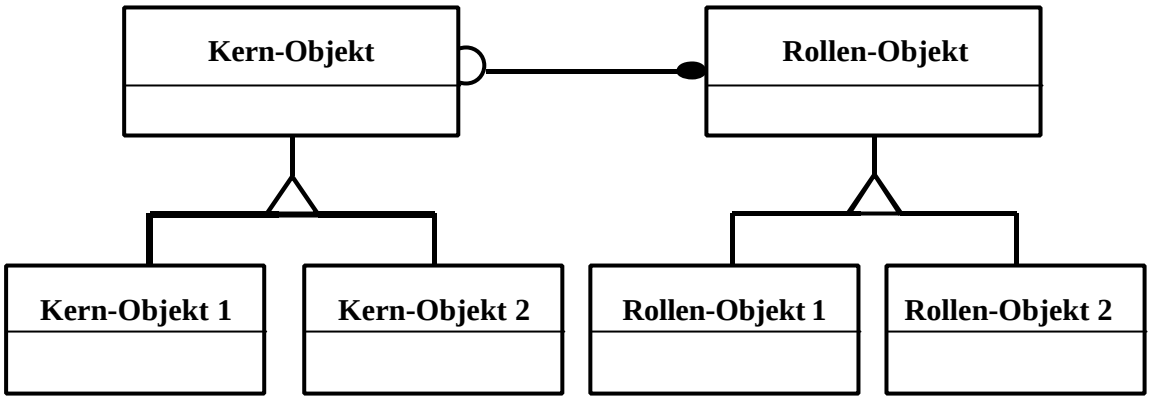

Abbildung 25: Trennung in unterschiedliche Klassenhierarchien

Um diese Beziehungen über Objektraumgrenzen hinweg definieren zu können, sollen sie auf Basis des Relationship-Services der CORBA-Spezification 2.0 [CORBA] realisiert werden.

4.2.2 Mechanismus zum Annehmen von Rollen

Entsprechend dem Objektorientierten Paradigma von selbständig agierenden Objekten soll die Zuordnung von „Rollen-Objekten“ zu einem „Kern-Objekt“ durch Annahme im „Kern-Objekt“ begründet werden („add_role“). Das „Kern-Objekt“ fungiert also immer als „head“ der Beziehung und ist daher für diese verantwortlich.

Außerdem soll es aber möglich sein, daß „Rollen-Objekte“ ihrerseits wieder Rollen annehmen. So entsteht ein mehrschichtiges Subjekt.

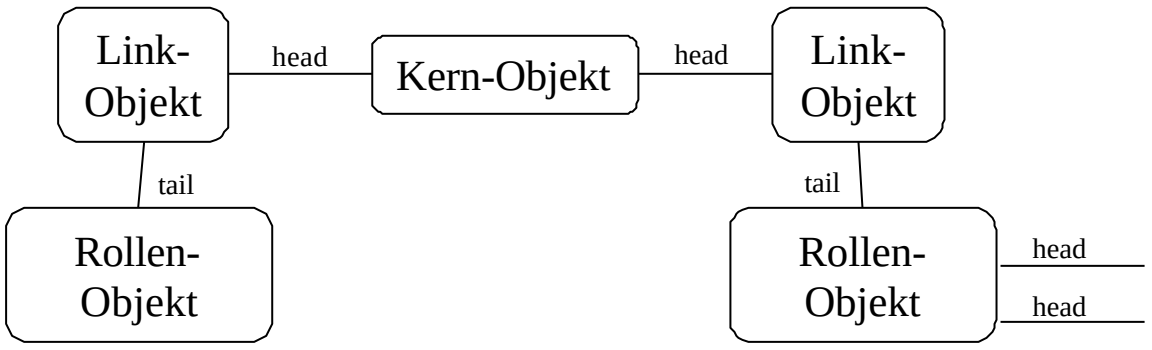

Abbildung 26: Reale Subjektstruktur

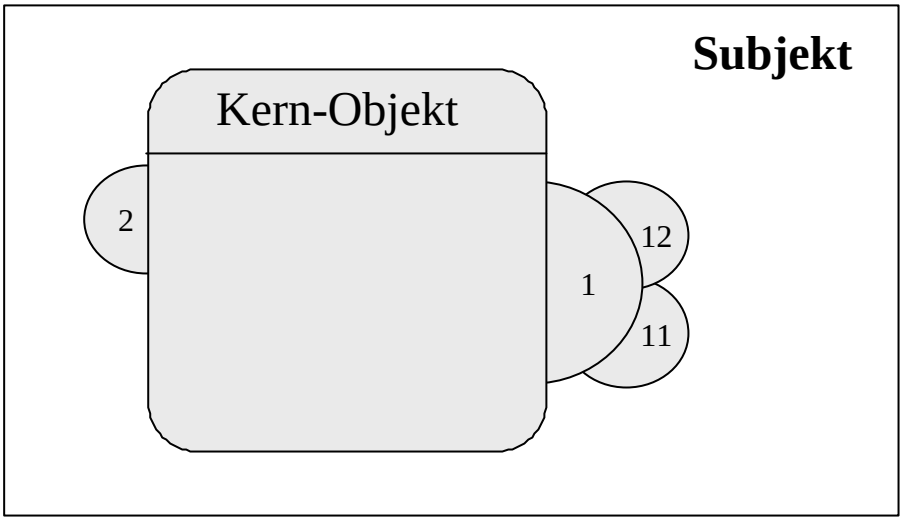

Abbildung 27: Virtuelle Subjektstruktur

Es soll möglich sein, in jedem Objekt gewisse Voraussetzungen zu definieren, die erfüllt sein müssen, um eine bestimmte Rolle annehmen zu dürfen („role_is_allowed“).

4.2.3 Verwaltungsmechanismen innerhalb des Subjektes

Jeder Teil des Subjektes soll direkten Zugriff auf das Zentrum des Subjektes unterstützen („root“). Außerdem soll jedes „Rollen-Objekt“ auf den unmittelbaren Vorgänger zugreifen können („role_of“).

Jeder Teil des Subjektes soll alle Rollen, die ihm gerade zugeordnet sind, auflisten können („list_all_roles“); dabei sollen alle nachfolgenden Schichten berücksichtigt werden.

Außerdem soll überprüft werden können, ob einem bestimmten Teil des Subjektes gerade eine bestimmte Rolle zugeordnet ist („exists_as“). Von jedem Teil des Subjektes soll zu einer Rolle navigiert werden können, die ihm gerade zugeordnet ist („as“).

Mit Hilfe dieser Mechanismen ist es möglich, verschiedene Vergleichsoperatoren für Subjekte zu definieren:

- Gleichheit des „Kern-Objektes“
- Gleichheit des unmittelbaren Vorgängers
- Gleichheit der jeweiligen Klasse
- Gleichheit der zugeordneten Rollen
- etc.

Welcher dieser Vergleichsoperatoren zur Anwendung kommt, ist nicht Teil dieser Spezifikation, sondern muß mit Hilfe der oben angeführten Mechanismen kontextabhängig realisiert werden.

4.2.4 Vererbungsmechanismen innerhalb des Subjektes

Jedes „Rollen-Objekt“ soll ihm unbekannte Methoden an den jeweils unmittelbaren Vorgänger weiterleiten (Delegation). Durch diesen Mechanismus wird eine Vererbungslinie geschaffen, die von den spezialisierteren Rollen über die weiter innen gelegenen Rollen schließlich zum „Kern-Objekt“ führt.

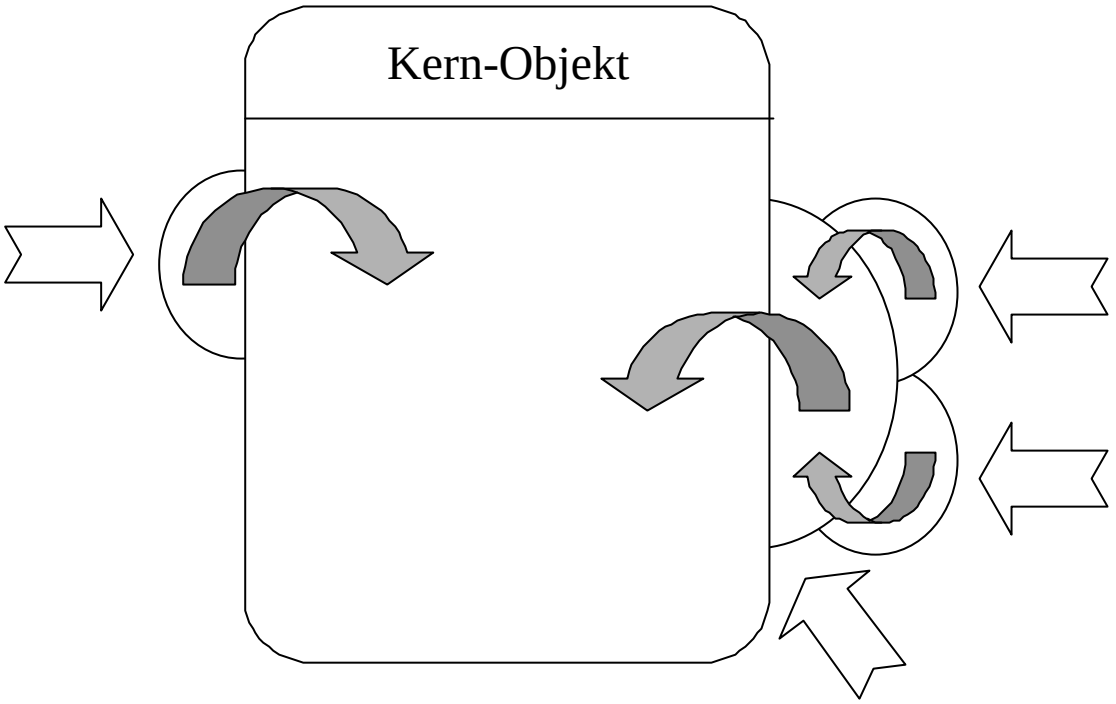

Abbildung 28: Vererbung von außen nach innen

Zusätzlich soll im „Kern-Objekt“ ein Mechanismus realisiert werden, der ermöglicht, daß eine bestimmte nicht verstandene Methode aus dem „Kern-Objekt“ an ein bestimmtes „Rollen-Objekt“ weitergeleitet wird, welches diese Methode versteht. Durch diesen Mechanismus wird eine Vererbungslinie geschaffen, die in einer Stufe vom „Kern-Objekt“ zu einem jeweils ausgewählten „Rollen-Objekt“ führt.

Diese Auswahl soll im „Kern-Objekt“ abhängig vom jeweiligen Kontext und der jeweiligen Methode erfolgen können („role_select_with_message“).

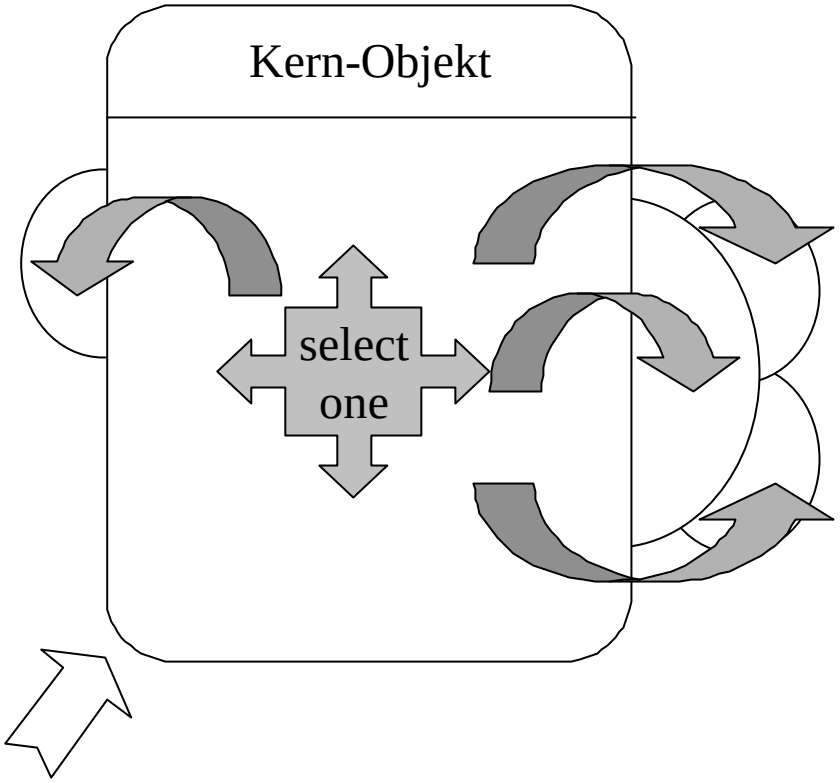

Abbildung 29: Vererbung von innen nach außen

Da diese Vererbungslinien auf der Zuordnung von Rollen beruhen, sind sie für jedes Subjekt individuell, sodaß man von Vererbung auf Instanzenebene sprechen kann.

4.2.5 Mechanismen zur Steuerung der Lebenszyklen

Jedes „Rollen-Objekt“ soll an Beziehungen vom Typ „Roleification“ nur genau einmal als „tail“ teilnehmen können. Zu einem Zeitpunkt, zu dem sie an keiner Beziehung vom Typ „Roleification“ teilnimmt, soll sie als ungültig gekennzeichnet werden.

Umgekehrt soll es möglich sein, daß ein Objekt eine Rolle auch mehrmals annehmen kann.

Durch geeignete Mechanismen soll es möglich sein, Roleification-Beziehungen zu lösen, indem die Methode „abandon“ an das „Rollen-Objekt“ geschickt wird. Dadurch und durch die Möglichkeit, wieder andere Rollen anzunehmen, kann ein Subjekt im Laufe seines Lebenszyklus unterschiedliche Strukturen aufweisen und so Klassenmigrationen auf flexible Art und Weise vermeiden.

4.3 Einordnung

Der „Treaty of Orlando“ [Lieberman88] schlägt drei Dimensionen zur Einordnung von Vererbungsmechanismen in Programmiersprachen vor:

- **Statische oder Dynamische Vererbung**
  Kann ein Objekt seine Vererbungslinie während der Laufzeit ändern, so handelt es sich um dynamische Vererbung. Ist dies nicht möglich, ist statische Vererbung gegeben.
- **Implizite oder Explizite Vererbung**
  Kann ein Objekt seine Vererbungslinie während der Laufzeit nennen, so handelt es sich um explizite Vererbung. Ist dies nicht möglich, ist implizite Vererbung gegeben.
- **Vererbung auf Instanzen- oder Klassenebene**
  Können zwei Instanzen der selben Klasse unterschiedliche Vererbungslinien besitzen, so handelt es sich um Vererbung auf Instanzenebene. Können Vererbungslinien definiert werden, die für alle Instanzen einer Klassen gelten, so handelt es sich um Vererbung auf Klassenebene.

Das vorliegende Konzept kann nach diesen Kriterien wie folgt eingeordnet werden:

- **Dynamische Vererbung**
  Ein Subjekt kann durch annehmen und abgeben von Rollen seine Struktur und damit sein Verhalten während der Laufzeit ändern.
- **Explizite Vererbung**
  Innerhalb eines Subjektes sind alle Roleification-Beziehungen, die die Vererbungslinien auf Instanzenebene definieren, während der Laufzeit bekannt.
- **Vererbung auf Instanzen- und auf Klassenebene möglich**
  Sowohl Kernobjekte als auch Rollenobjekte können in getrennten Klassenhierarchien definiert werden. Die Zuordnung auf Subjektebene erlaubt die zusätzliche Definition einer Vererbungslinie auf Instanzenebene.

4.4 IDL Interface

```
// Roleification
// This module defines the interfaces which form the Roleification-Service.
//
module Roleification  {

    //
    interface ObjectWithRolesInterface : ApplicationSem {

        // Adds a new Role to the Receiver-Object.
        SmalltalkObject addRole (in SmalltalkObject role);

        // Returns the current existents of the Receiver-Object in a particular Role.
        OrderedCollection as (in string roleTitle);

        // Returns a Boolean whether the Receiver-Object is currently playing this Role.
        boolean existsAs (in string roleTitle);

        // Returns a Dictionary with all Roles currently played by the Receiver-Object.
        Dictionary listAllRoles ();

        // Adds the currently played Roles to the signature of the Receiver-Object.
        SmalltalkObject printOn (in SmalltalkObject aStream);

        // Answer a Boolean as to whether the method dictionary of the receiver's class
        // contains aSymbol as a message selector.
        boolean respondsIntrinsicTo (in Symbol aSymbol);

        // Decides wether a Role is currently allowed to be bound to the Receiver-Object or not.
        boolean roleIsAllowed (in SmalltalkObject aRole);

        // Returns the root of the Subject.
        SmalltalkObject root ();
    };


    //
    interface RoleInterface : ObjectWithRolesInterface {

        // Removes the link between the Receiver-Role and its Ancestor.
        SmalltalkObject abandon ();

        // Returns the Ancestor of the Receiver-Role.
        SmalltalkObject roleOf ();

        // Returns the Title of the Receiver-Role.
        string roleTitle ();

    };
};

module CompoundLifecycles {
    ...
    const LinkType roleification_link = 9;
    ...
};
```

5 Implementation des Roleification-Services

Im folgenden Kapitel wird eine Implementation des in Kapitel 4 spezifizierten Roleification-Services für [DST] beschrieben. Es handelt sich um die Implementation einer Entwicklungsumgebung, die die Konfiguration und Verwendung dieses Services unterstützt.

5.1 Objekt-Modell

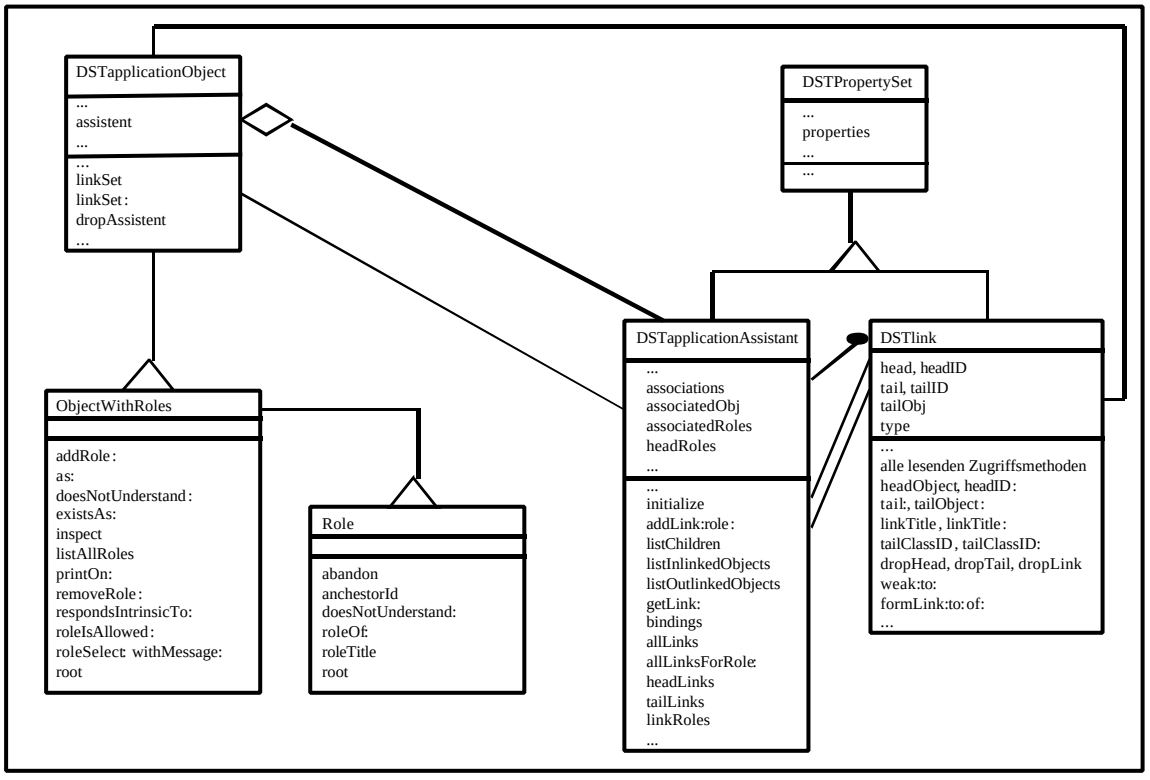

Abbildung 30: Objektmodell des Roleification-Services

Abbildung 30 zeigt das Objektmodell des implementieren Services:

Um auf das Relationship-Service der CORBA-Spezifikation 2.0 [CORBA] aufbauen und alle anderen Mechanismen verwenden zu können, ist die Wurzel-Klasse des Services (ObjectWithRoles) von der Klasse DSTapplicationObject abgeleitet. Die Mechanismen des Relationship-Services wurden schon eingehend in Kapitel 3.6 erläutert.

Jede Klasse, deren Instanzen die Fähigkeit besitzen sollen, Rollen anzunehmen, muß von der Klasse ObjectWithRoles abgeleitet werden.

Jede Klasse, deren Instanzen anderen Objekten als Rollen zugeordnet werden sollen, muß von der Klasse Role abgeleitet werden.

Damit Rollen-Objekte – wie in der Spezifikation gefordert - wieder Rollen annehmen können, ist die Wurzel-Klasse aller Rollen-Klassen von der Klasse ObjectWithRoles abgeleitet.

Die Klassen ObjectWithRoles und Role sind beide als abstrakte Klassen konzipiert, die nur der Implementation der Mechanismen dienen.

5.2 Interne Mechanismen

Im folgenden werden die einzelnen Methoden des Services beschrieben. Die Klassenzuordnung ist jeweils dem Objektmodell in Abbildung 30 zu entnehmen.

5.2.1 addRole: aRole

5.2.1.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.1.2 Übergabeparameter

Dieser Methode wird ein Rollenobjekt übergeben, das dem Empfänger zugeordnet werden soll.

5.2.1.3 Funktionsweise

Die Methode überprüft zunächst, ob das übergebene Rollenobjekt bereits als „tail“ an einer Beziehung vom Typ „Roleification“ teilnimmt – ist dies der Fall, kann es nicht noch einmal zugeordnet werden und ein Fehler wird ausgegeben.

Danach wird über die Methode roleIsAllowed überprüft, ob alle Voraussetzungen für die Annahme dieser Rolle erfüllt sind – ist dies nicht der Fall, wird eine Fehlermeldung ausgegeben.

Schließlich wird eine neue Beziehung vom Typ „Roleification“ zum übergebenen Rollenobjekt eingetragen.

5.2.1.4 Fehlermeldungen

'Role is already bound to another Object.' – Das übergebene Rollenobjekt nimmt bereits einmal als „tail“ an einer Beziehung vom Typ „Roleification“ teil, und kann daher nicht noch einmal zugeordnet werden.

'Role is currently not allowed to be bound to this Object.' – Nicht alle Voraussetzungen für die Annahme des übergebenen Rollenobjektes sind derzeit erfüllt.

5.2.1.5 Rückgabewert

Das Empfängerobjekt wird zurückgegeben.

5.2.1.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.2 abandon

5.2.2.1 Empfänger

Rollenobjekte, die von ihrem unmittelbaren Vorgänger gelöst werden sollen.

5.2.2.2 Übergabeparameter

Keine.

5.2.2.3 Funktionsweise

Die Methode sucht über die Methode ancestorId die Beziehung zum unmittelbaren Vorgänger und löst diese.

5.2.2.4 Fehlermeldungen

Keine.

5.2.2.5 Rückgabewert

Die gerade gelöste Rolle wird zurückgegeben.

5.2.2.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.3 listAllRoles

5.2.3.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.3.2 Übergabeparameter

Keine.

5.2.3.3 Funktionsweise

Diese Methode sucht alle Beziehungen vom Typ Roleification, an denen das Empfängerobjekt als head teilnimmt, und trägt diese unter ihrem roleTitle in ein Dictionary ein.

Danach wird diese Methode für alle gefundenen Rollen rekursiv aufgerufen und die gefundenen Rollen der Rollen ebenfalls in das gleiche Dictionary eingetragen.

5.2.3.4 Fehlermeldungen

Keine.

5.2.3.5 Rückgabewert

Ein Dictionary, das alle im gesamten Subjekt verfügbaren Rollen enthält.

5.2.3.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.4 existsAs: aRoleTitle

5.2.4.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.4.2 Übergabeparameter

Ein String, der eine Rollenklasse spezifiziert.

5.2.4.3 Funktionsweise

Überprüft mit Hilfe der Methode listAllRoles, ob das Empfängerobjekt ein Rollenobjekt mit dem übergebenen roleTitle zugeordnet hat.

5.2.4.4 Fehlermeldungen

Keine.

5.2.4.5 Rückgabewert

True oder False.

5.2.4.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.5 as: aRoleTitle

5.2.5.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.5.2 Übergabeparameter

Ein String, der eine Rollenklasse spezifiziert.

5.2.5.3 Funktionsweise

Sucht mit Hilfe der Methode listAllRoles dem Empfängerobjekt zugeordnete Rollenobjekte mit dem übergebenen roleTitle.

5.2.5.4 Fehlermeldungen

Keine.

5.2.5.5 Rückgabewert

Eine OrderedCollection aller dem Empfänger zugeordneten Rollen mit dem übergebenen roleTitle.

5.2.5.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.6 root

5.2.6.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.6.2 Übergabeparameter

Keine.

5.2.6.3 Funktionsweise

Diese Methode wird für Rollenobjekte redefiniert.

5.2.6.4 Fehlermeldungen

Keine.

5.2.6.5 Rückgabewert

Es wird immer das Kernobjekt des Subjektes, dem das Empfängerobjekt angehört, zurückgegeben.

5.2.6.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.7 roleOf

5.2.7.1 Empfänger

Rollenobjekte.

5.2.7.2 Übergabeparameter

Keine.

5.2.7.3 Funktionsweise

Diese Methode sucht mit Hilfe der Methode ancestorId die Beziehung zum unmittelbaren Vorgänger und ermittelt jenes Objekt, das an dieser Beziehung als head teilnimmt.

5.2.7.4 Fehlermeldungen

Keine.

5.2.7.5 Rückgabewert

Das unmittelbare Vorgängerobjekt.

5.2.7.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.8 ancestorId

5.2.8.1 Empfänger

Rollenobjekte.

5.2.8.2 Übergabeparameter

Keine.

5.2.8.3 Funktionsweise

Die Methode sucht alle Beziehungen vom Typ Roleification, an denen das Empfängerobjekt als tail teilnimmt – werden keine solche Beziehungen gefunden, wird eine Fehlermeldung ausgegeben.

Da ohnehin sichergestellt ist, daß es höchstens eine solche Beziehung geben kann, wird aus der ermittelten OrderedCollection die erste und einzige ausgewählt.

5.2.8.4 Fehlermeldungen

'This is not a Role of a valid Object.' – Das Empfängerobjekt befindet sich in einem ungültigen Zustand, da es keinem Vorgängerobjekt zugeordnet ist.

5.2.8.5 Rückgabewert

Der Indexwert, unter dem die Beziehung zum unmittelbaren Vorgänger im Assistenten des Empfängerobjektes eingetragen ist.

5.2.8.6 Interface

Keines.

5.2.9 roleTitle

5.2.9.1 Empfänger

Rollenobjekte.

5.2.9.2 Übergabeparameter

Keine.

5.2.9.3 Funktionsweise

Keine.

5.2.9.4 Fehlermeldungen

Keine.

5.2.9.5 Rückgabewert

Klassenname des Empfängerobjektes als String.

5.2.9.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.10 doesNotUnderstand: aMessage

5.2.10.1 Empfänger

Objekte, die Rollen annehmen können.

5.2.10.2 Übergabeparameter

Eine nicht verstandene Methode.

5.2.10.3 Funktionsweise

Die Methode sucht mit Hilfe der Methode respondsIntrinsicTo nach allen zugeordneten Rollenobjekten, die die übergebene Methode verstehen.

Falls solche Rollenobjekte gefunden werden, so wird mit Hilfe der Methode roleSelect:withMessage: eine davon ausgewählt und an diese die übergebene Methode weitergeleitet.

Andernfalls wird eine Fehlermeldung ausgegeben.

5.2.10.4 Fehlermeldungen

'Message not understood: ' – Im gesamten Subjekt wurde kein Teil gefunden, der die übergebene Methode versteht.

5.2.10.5 Rückgabewert

Der Rückgabewert des weitergeleiteten Methodenaufrufes wird zurückgegeben.

5.2.10.6 Interface

Keines.

5.2.11 doesNotUnderstand: aMessage

5.2.11.1 Empfänger

Rollenobjekte.

5.2.11.2 Übergabeparameter

Eine nicht verstandene Methode.

5.2.11.3 Funktionsweise

Die übergebene Methode wird an den unmittelbaren Vorgänger weitergeleitet.

5.2.11.4 Fehlermeldungen

Keine.

5.2.11.5 Rückgabewert

Der Rückgabewert des weitergeleiteten Methodenaufrufes wird zurückgegeben.

5.2.11.6 Interface

Keines.

5.2.12 respondsIntrinsicTo: aSymbol

5.2.12.1 Empfänger

Objekte, denen Rollen zugeordnet werden können.

5.2.12.2 Übergabeparameter

Der Selector einer Methode.

5.2.12.3 Funktionsweise

Es wird überprüft, ob die übergebene Methode im intrinsic Interface der Klasse des Empfängerobjektes enthalten ist.

5.2.12.4 Fehlermeldungen

Keine.

5.2.12.5 Rückgabewert

True oder False.

5.2.12.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.13 roleSelect: aCollection withMessage: aMessage

5.2.13.1 Empfänger

Objekte, denen Rollen zugeordnet werden können.

5.2.13.2 Übergabeparameter

Eine OrderedCollection von Rollen, die dem Empfängerobjekt zugeordnet sind und die die übergebene Methode verstehen.

5.2.13.3 Funktionsweise

Diese Methode soll aus der übergebenen OrderedCollection eine Rolle auswählen, an die die übergebene Methode in der Folge weitergeleitet werden soll.

Diese Methode muß zu diesem Zweck geeignet redefiniert werden. Wird sie nicht redefiniert, so wird keine Rolle ausgewählt, sodaß die Vererbungslinie an dieser Stelle endet.

5.2.13.4 Fehlermeldungen

Keine.

5.2.13.5 Rückgabewert

Eine ausgewählte Rolle oder NIL.

5.2.13.6 Interface

Keines.

5.2.14 roleIsAllowed: aRole

5.2.14.1 Empfänger

Objekte, denen Rollen zugeordnet werden können.

5.2.14.2 Übergabeparameter

Ein Rollenobjekt.

5.2.14.3 Funktionsweise

Diese Methode soll überprüfen, ob alle Voraussetzungen dafür erfüllt sind, daß die übergebene Rolle dem Empfängerobjekt zugeordnet werden darf.

Diese Methode muß zu diesem Zweck geeignet redefiniert werden. Wird sie nicht redefiniert, so werden keine Voraussetzungen überprüft, sodaß die Zuordnung immer zugelassen wird.

5.2.14.4 Fehlermeldungen

Keine.

5.2.14.5 Rückgabewert

True oder False.

5.2.14.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.15 printOn: aStream

5.2.15.1 Empfänger

Objekte, denen Rollen zugeordnet werden können.

5.2.15.2 Übergabeparameter

Ein Stream.

5.2.15.3 Funktionsweise

Diese Methode fügt der Signatur eines Objektes die roleTitle der gerade zugeordneten Rollen hinzu.

5.2.15.4 Fehlermeldungen

Keine.

5.2.15.5 Rückgabewert

Ein Stream.

5.2.15.6 Interface

Diese Methode kann über ein IDL-Interface aufgerufen werden.

5.2.16 inspect

5.2.16.1 Empfänger

Objekte, denen Rollen zugeordnet werden können.

5.2.16.2 Übergabeparameter

Keine.

5.2.16.3 Funktionsweise

Diese Methode öffnet ein geeignetes Testwerkzeug, in dem neben den Instanzvariablen des Empfängerobjektes auch ein Zeiger auf das Kernobjekt, sowie ein Dictionary der zugeordneten Rollen aufgelistet werden.

5.2.16.4 Fehlermeldungen

Keine.

5.2.16.5 Rückgabewert

Das Empfängerobjekt wird zurückgegeben.

5.2.16.6 Interface

Keines.

5.3 Entwicklungsumgebung

5.3.1 Testwerkzeug

Um in der Entwicklungsumgebung einen leichteren Überblick über ein gesamtes Subjekt zu erhalten, wurde auf der Klasse ObjektWithRoles ein eigener SubjectInspector installiert.

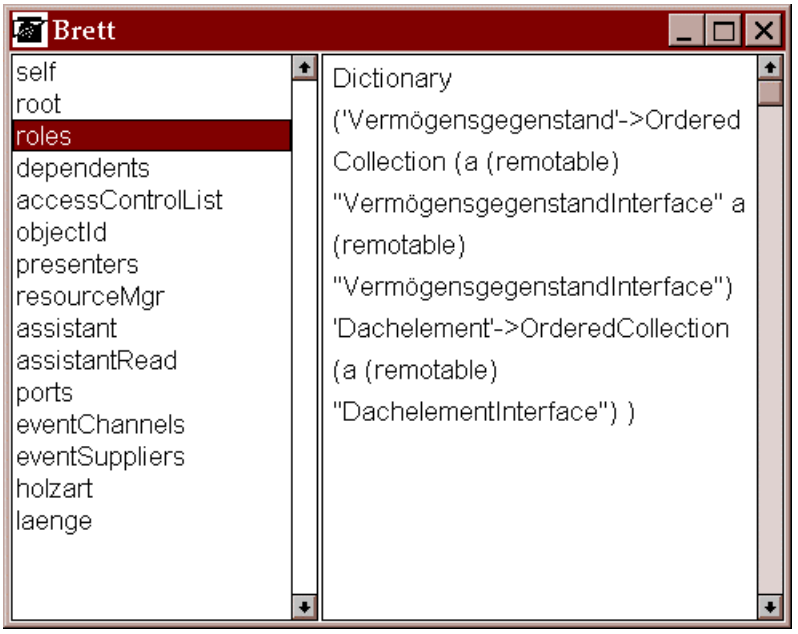

Abbildung 31: „SubjectInspector“ auf einem „ObjectWithRoles“

Die Klasse SubjectInspector wurde von der Klasse Inspector abgeleitet und folgende Methoden wurden redefiniert:

- fieldList
- fieldValue

Durch diese beiden Methoden werden zusätzlich zu den bereits vorhandenen Einträgen im Inspector zwei weitere aufgenommen und deren Inhalt spezifiziert:

- root: Ein Zeiger auf das Kernobjekt des Subjektes.
- roles: Ein Dictionary der gerade zugeordneten Rollen.

5.3.2 Erweiterung des Relationship-Services

Bei den Verknüpfungen zwischen Objekten und deren Rollen handelt es sich um Beziehungen eines neuen Typs: Roleification. Da die CORBA-Spezifikation 2.0 [CORBA] ein Service enthält, das Beziehungen unterschiedlichen Typs verwalten soll, baut die Implementation des Roleification-Services in DST auf diesem Service auf.

Zu diesem Zweck muß das Relationship-Service allerdings um einige Methoden zur Verwaltung des neuen Beziehungstyps erweitert werden:

- roleification Instanzmethode
- isRoleification Instanzmethode
- roleification Klassenmethode
- isRoleification: Klassenmethode
- roleificationHeadRole Klassenmethode
- roleificationTailRole Klassenmethode

Weiters müssen die folgenden Methoden erweitert werden, um den neuen Beziehungstyp verwalten zu können:

- printOn: Instanzmethode
- typeString Instanzmethode
- headRole Instanzmethode
- tailRole Instanzmethode

Die einzige Veränderung, die im IDL-Interface vorzunehmen ist, ist die Definition einer neuen CORBA Konstante im Modul „CompoundLifecycles“:

```
const LinkType roleification_link = 9;
```

Der Wert der Konstante muß dabei – entsprechend der Konvention des Relationship-Services – ein beliebiger ungerader Wert sein, der noch keinen anderen Beziehungstyp beschreibt. Zu beachten ist, daß dieser Wert in allen Implementationen, zwischen denen eine Beziehung dieses Typs verwaltet werden soll, gleich eingestellt sein muß.

5.4 Anleitung zum Einsatz

5.4.1 Konfiguration des Roleification-Services

5.4.1.1 Konfiguration der Vererbungslinie

Benutzerklassen, die von der abstrakten Klasse ObjectWithRoles abgeleitet werden, können die folgende Methode redefinieren: „roleSelect:withMessage:“.

Diese Methode wird immer dann aufgerufen, wenn das Objekt eine Methode nicht versteht, aber Rollenobjekte zugeordnet sind, die diese Methode verstehen. Sie erhält als Parameter eine OrderedCollection von Rollenobjekten, die die ebenfalls übergebene Methode verstehen.

Als Rückgabewert wird eine dieser Rollen oder NIL erwartet – an dieser Stelle kann also spezifiziert werden, an welche der Rollen die nicht verstandene Methode weitergeleitet wird. Für diese Auswahl stehen sowohl Informationen über das Objekt

(self), die einzelnen Rollenobjekte (OrderedCollection) als auch über die Methode und ihre Parameter (Message) zur Verfügung.

Ohne Redefinition wird NIL zurückgegeben und die Vererbungslinie an dieser Stelle abgebrochen.

5.4.1.2 Konfiguration der Voraussetzungen

Benutzerklassen, die von der abstrakten Klasse ObjectWithRoles abgeleitet werden, können die folgende Methode redefinieren: „roleIsAllowed:“.

Diese Methode wird immer dann aufgerufen, wenn dem Objekt eine neue Rolle zugeordnet werden soll. Sie erhält als Parameter jenes Rollenobjekt, das zugeordnet werden soll.

Als Rückgabewert wird True oder False erwartet – an dieser Stelle kann also spezifiziert werden, welche Voraussetzungen erfüllt sein müssen, um eine bestimmte Rolle annehmen zu können. Diese Voraussetzungen können sowohl Bedingungen enthalten, die erfüllt sein müssen, als auch solche, die nicht erfüllt sein dürfen. Für diese Voraussetzungen stehen sowohl Informationen über das Objekt (self) als auch über das Rollenobjekt (Role) zur Verfügung.

Ohne Redefinition wird True zurückgegeben und die Zuordnung immer zugelassen.

5.4.1.3 Konfiguration der Signatur

Benutzerklassen, die von der abstrakten Klasse Role abgeleitet werden, können die folgende Methode redefinieren: „roleTitle“.

Diese Methode wird immer dann aufgerufen, wenn eine Signatur der Rolle angefordert wird.

Als Rückgabewert wird ein für die Rolle repräsentativer String erwartet – an dieser Stelle kann also spezifiziert werden, unter welchem Titel eine Rolle dem Objekt bekannt gemacht wird.

Ohne Redefinition wird der Klassenname der Rolle zurückgegeben.

5.4.2 Verwendung des Roleification-Services

5.4.2.1 Lebenszyklus

Ein Rollenobjekt kann einem Objekt zugeordnet werden, wenn dessen Klasse von der abstrakten Klasse ObjectWithRoles abgeleitet ist, indem an dieses Objekt die Methode addRole: geschickt wird und das Rollenobjekt als Parameter übergeben wird.

Diese Zuordnung kann wieder gelöst werden, indem an eine zugeordnete Rolle die Methode abandon geschickt wird.

5.4.2.2 Navigation

Über die Methode root kann von jedem Teil eines Subjektes aus das Kernobjekt erhalten werden; jedes Rollenobjekt kann zusätzlich über die Methode roleOf seinen unmittelbaren Vorgänger nennen.

Für jedes Objekt, das Rollen annehmen kann, kann mittels der Methode existsAs: überprüft werden, ob dem Objekt eine bestimmte Rolle zugeordnet ist; mittels der Methode as: kann eine bestimmte zugeordnete Rolle aufgesucht werden – da es möglich ist, daß diese Rolle mehrfach zugeordnet ist, liefert die Methode as: eine OrderedCollection aller zugeordneten Rollenobjekte dieses Typs zurück.

5.4.3 Erstellung der IDL-Interfaces

Die für den Einsatz des Services notwendigen Klassen verfügen über ein eigenes Interface, das alle relevanten Methoden beinhaltet:

- ::Roleification::ObjectWithRoles
- ::Roleification::Role

Für die von diesen Klassen abgeleiteten Benutzerklassen müssen ebenfalls Interfaces definiert werden. Damit die Vererbungslinien auf Instanzenebene auch über den ORB eingesetzt werden können, müssen alle Teile des Subjektes über alle Interfaces des Subjektes verfügen können, da andernfalls die Methoden von den Surrogat-Objekten gar nicht an die eigentlichen Objekte weitergeleitet werden würden.

Da die IDL die Möglichkeit der Mehrfachvererbung bietet, ist dies grundsätzlich sehr einfach realisierbar. Dabei muß aber beachtet werden, daß keine Zyklen in der Vererbungslinie der Interfaces entstehen.

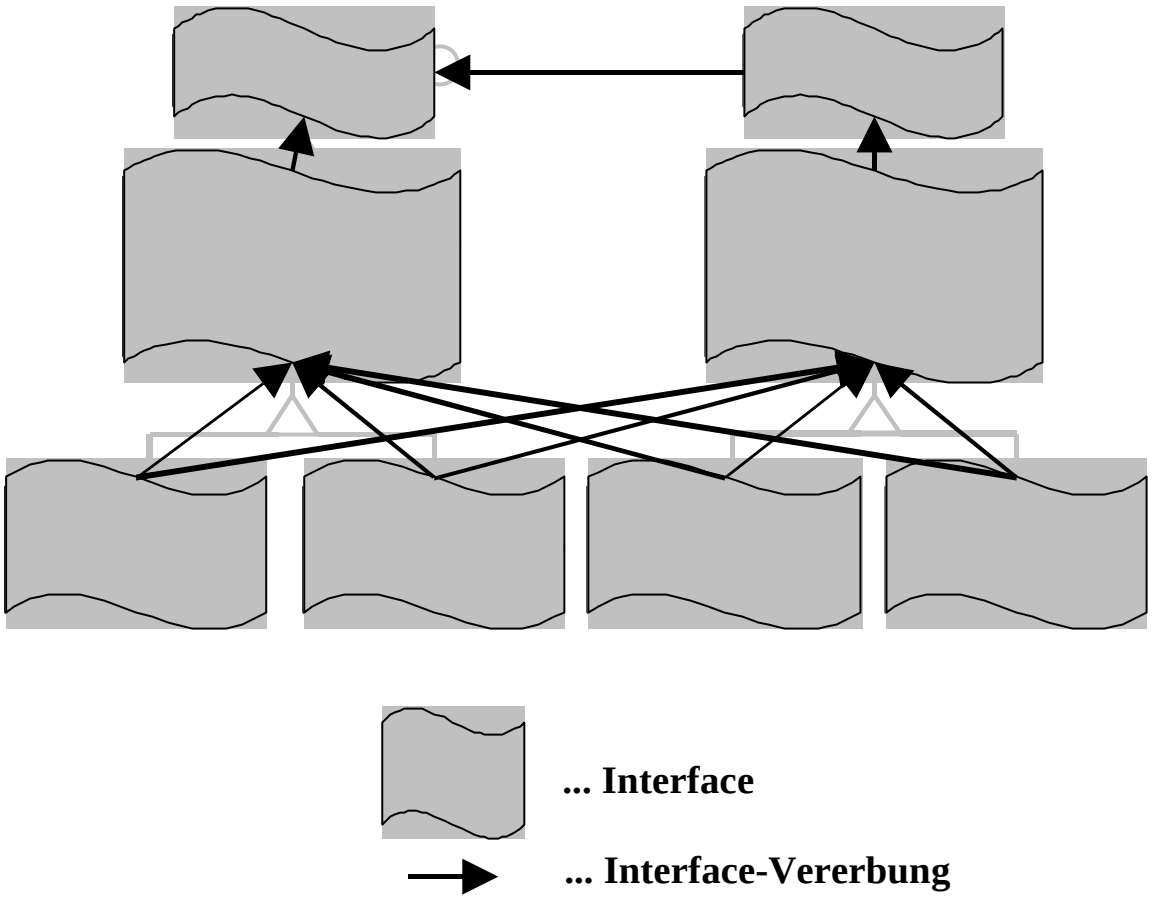

Abbildung 32: Zyklenfreie Interface Vererbung

Abbildung 32 zeigt anhand des Beispiels aus Kapitel 5.5.1, wie die Interfaces der Benutzerklassen definiert werden können, um allen Teilen des Subjektes alle Interfaces zugänglich zu machen und dennoch Zyklen zu vermeiden.

5.5 Beispiele

Zwei konkrete Beispiele sollen die Funktionalität des Roleification-Services demonstrieren. Sie sollen die Umsetzung der theoretisch diskutierten Modellierungsaspekte verdeutlichen und Möglichkeiten zur Verwendung des Services anleiten.

5.5.1 „Person“

5.5.1.1 Objektmodell

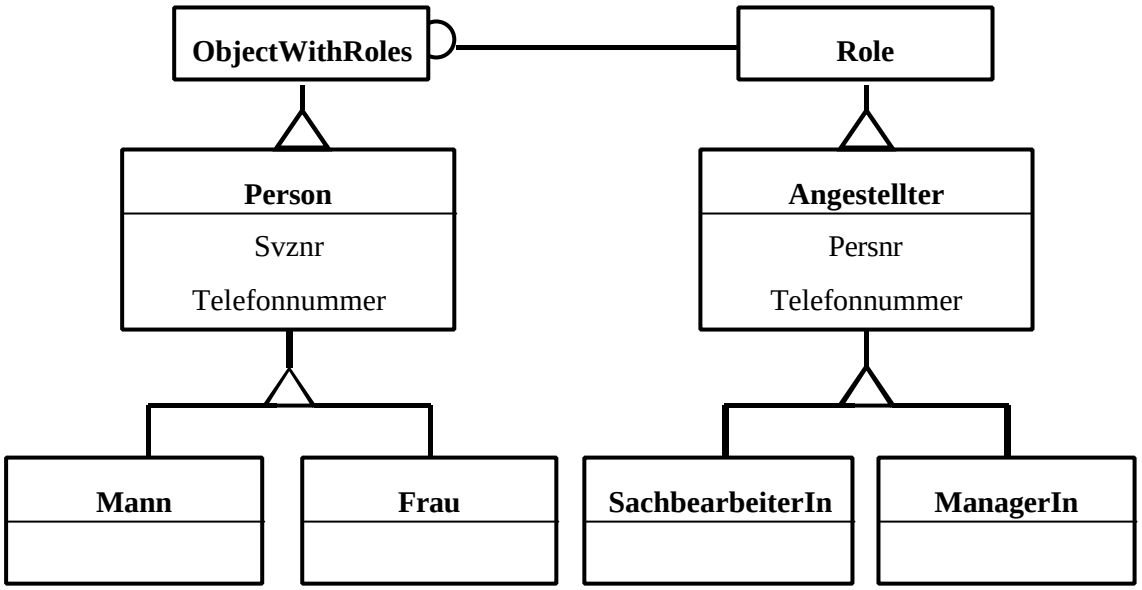

Abbildung 33: Objektmodell des Beispiels „Person“

Die Klasse Person wird von der abstrakten Klasse ObjektWithRoles abgeleitet; die Klasse Angestellter wird von der abstrakten Klasse Role abgeleitet. Beide Klassen sind wieder abstrakte Klassen und stellen ihren konkreten Subklassen Mann und Frau bzw. SachbearbeiterIn und ManagerIn lesende und schreibende Zugriffsmethoden auf ihre Instanzvariablen zur Verfügung.

Jede Person wird über ihre Sozialversicherungsnummer identifiziert, jeder Angestellte über seine Personalnummer; beide Klassen verfügen über eine eigene Telefonnummer.

Jeder Mann und jede Frau kann prinzipiell Rollen der Klassen SachbearbeiterIn und ManagerIn annehmen.

5.5.1.2 Konfiguration

Die Methode roleIsAllowed ist in der Klasse Person so redefiniert, daß eine neue Angestellten-Rolle nur dann angenommen werden darf, wenn der Person gerade keine andere Angestellten-Rolle zugeordnet ist.

Weil dadurch jeder Person zu jedem Zeitpunkt nur maximal eine Angestellten-Rolle zugeordnet sein kann, ist die Methode roleSelect:withMessage: in der Klasse Person auf sehr einfache Art und Weise redefiniert: Immer die erste Rolle aus der OrderedCollection wird ausgewählt.

5.5.1.3 Mechanismen

Der Beispielcode enthält ein kurzes Testprogramm, das die folgenden Mechanismen demonstriert:

- Eine Person nimmt eine Angestelltenrolle an.

```
Chris := Mann new: 1744.
Chris asRemotable name: 'Christian'.
Chris asRemotable telefonnummer: 7124412.
```

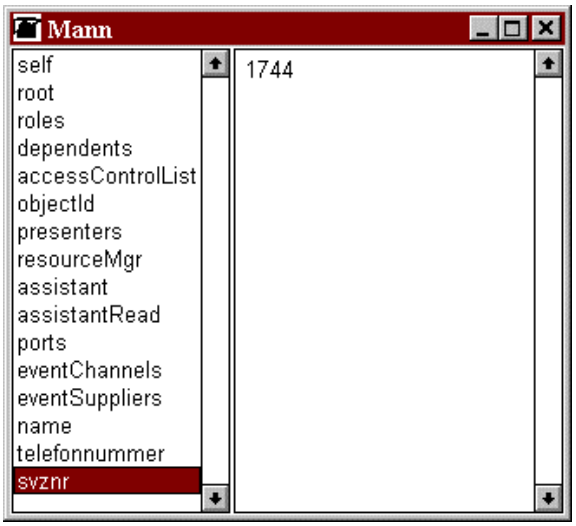

Abbildung 34: Inspector auf einer Instanz der Klasse Mann

```
S1 := SachbearbeiterIn new.
S1 asRemotable telefonnummer: 3566.
S1 asRemotable persnr: 78719.
```

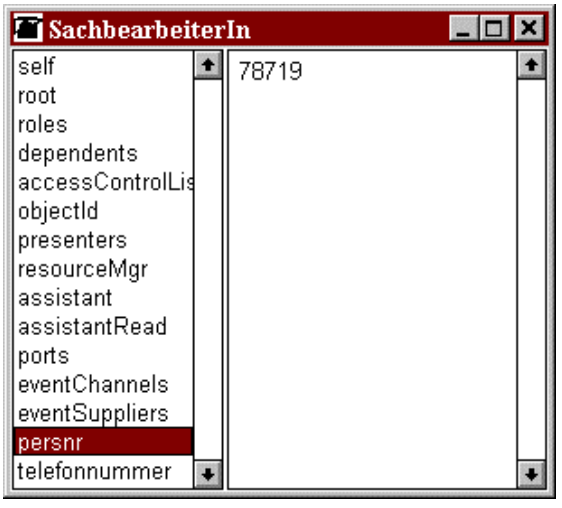

Abbildung 35: Inspector auf einer Instanz der Klasse SachbearbeiterIn

```
Chris asRemotable addRole: S1.
```

- Zu dieser Rolle kann navigiert werden.

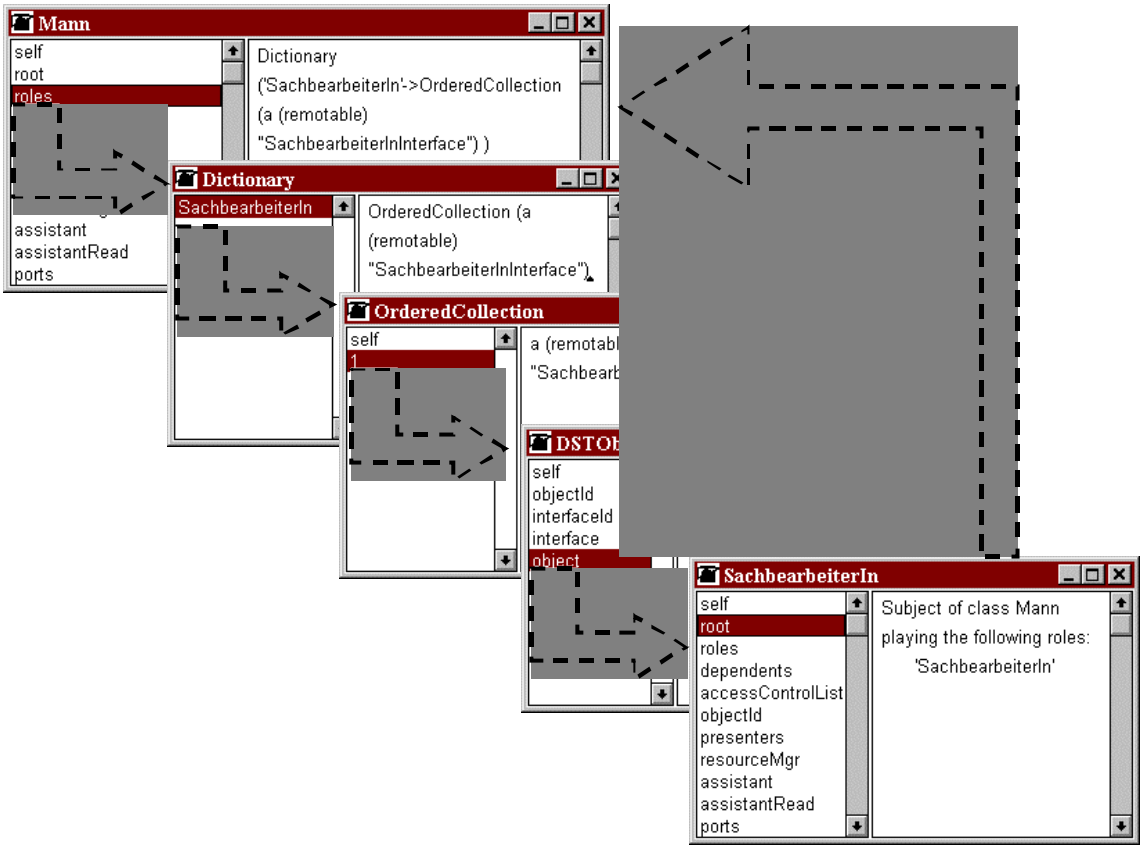

Abbildung 36: Der Weg von einem Objekt zu einer seiner Rollen

- Die Methode telefonnummer ist im Subjekt zweimal vorhanden und liefert unterschiedliche Ergebnisse.

```
(Chris asRemotable as: 'SachbearbeiterIn') first telefonnummer. <printIt> 3566
Chris asRemotable telefonnummer. <printIt> 7124412
```

- Die Person kann keine zweite Angestelltenrolle annehmen.

```
M1 := ManagerIn new.
M1 asRemotable telefonnummer: 06763566.
M1 asRemotable persnr: 78719.
```

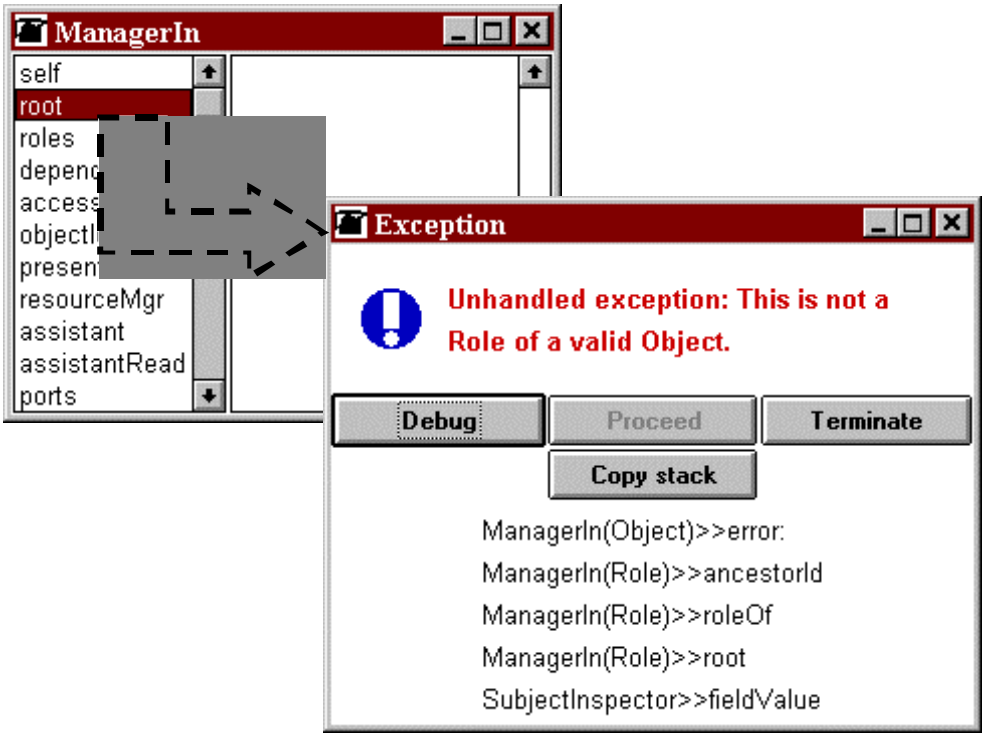

Abbildung 37: Inspector auf einem ungültigen Rollenobjekt

```
Chris asRemotable addRole: M1.
```

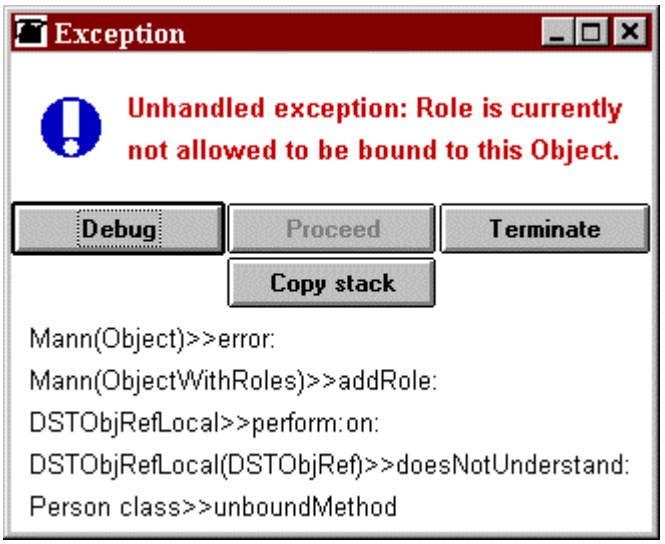

Abbildung 38: Eine Person kann keine zweite Angestellten-Rolle annehmen

- Die Person kann die erste Angestelltenrolle abgeben und danach eine neue Angestelltenrolle annehmen.

```
S1 asRemotable abandon.
Chris addRole: M1.
```

- Methoden der Angestelltenrolle können auch an das Personenobjekt geschickt werden und umgekehrt.

```
M1 asRemotable svznr. <printIt> 1744
Chris asRemotable persnr. <printIt> 78719
```

5.5.2 „Brett“

5.5.2.1 Objektmodell

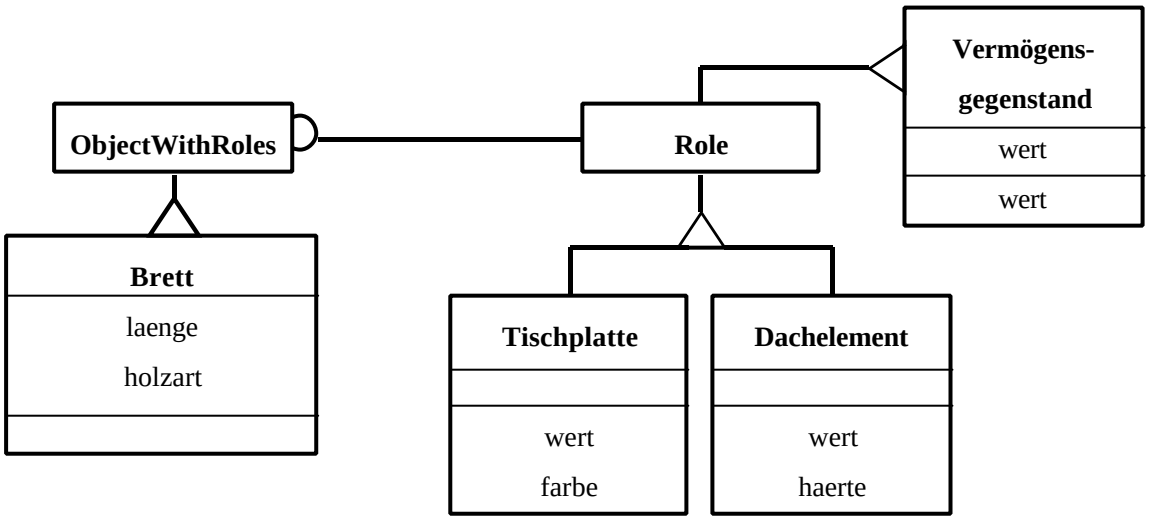

Abbildung 39: Objektmodell des Beispiels „Brett“

Die Klasse Brett wird von der abstrakten Klasse ObjektWithRoles abgeleitet; die Klassen Dachelement, Tischplatte und Vermögensgegenstand werden von der abstrakten Klasse Role abgeleitet.

Die Klasse Brett stellt lesende und schreibende Zugriffsmethoden auf ihre Instanzvariablen „laenge“ und „holzart“ zur Verfügung; diese Instanzvariablen werden von den Klassen Tischplatte und Dachelement jeweils in unterschiedlicher Art und Weise interpretiert.

Die Klasse Vermögensgegenstand fügt dem Subjekt eine eigene Instanzvariable „wert“ hinzu. Dadurch kann jede der Rollenarten einen eigenen Wert anbieten, das Brett selbst hingegen hat keinen Wert.

5.5.2.2 Konfiguration

Die Methode roleSelect:withMessage: ist in der Klasse Brett so redefiniert, daß wenn die Methode „wert“ an ein Brett gerichtet wird, aus allen Rollen, die Werte anbieten, jene Rolle ausgewählt wird, die den höchsten Wert darstellt.

5.5.2.3 Mechanismen

Der Beispielcode enthält ein kurzes Testprogramm, das die folgenden Mechanismen demonstriert:

- Ein Brett kann die Rollen Tischplatte und Dachelement annehmen.
- Die Rolle Vermögensgegenstand kann sowohl dem Brett als auch jeder seiner Rollen zugeordnet werden.

```
B1 := Brett new.
B1 asRemotable holzart: 'Mahagony'.
B1 asRemotable laenge: 5.
T1 := Tischplatte new.
D1 := Dachelement new.
V1 := Vermögensgegenstand new.
V2 := Vermögensgegenstand new.
V3 := Vermögensgegenstand new.
V1 asRemotable wert: 10.
V2 asRemotable wert: 50.
V3 asRemotable wert: 30.
B1 asRemotable addRole: T1.
B1 asRemotable addRole: D1.
B1 asRemotable addRole: V1.
T1 asRemotable addRole: V2.
D1 asRemotable addRole: V3.
```

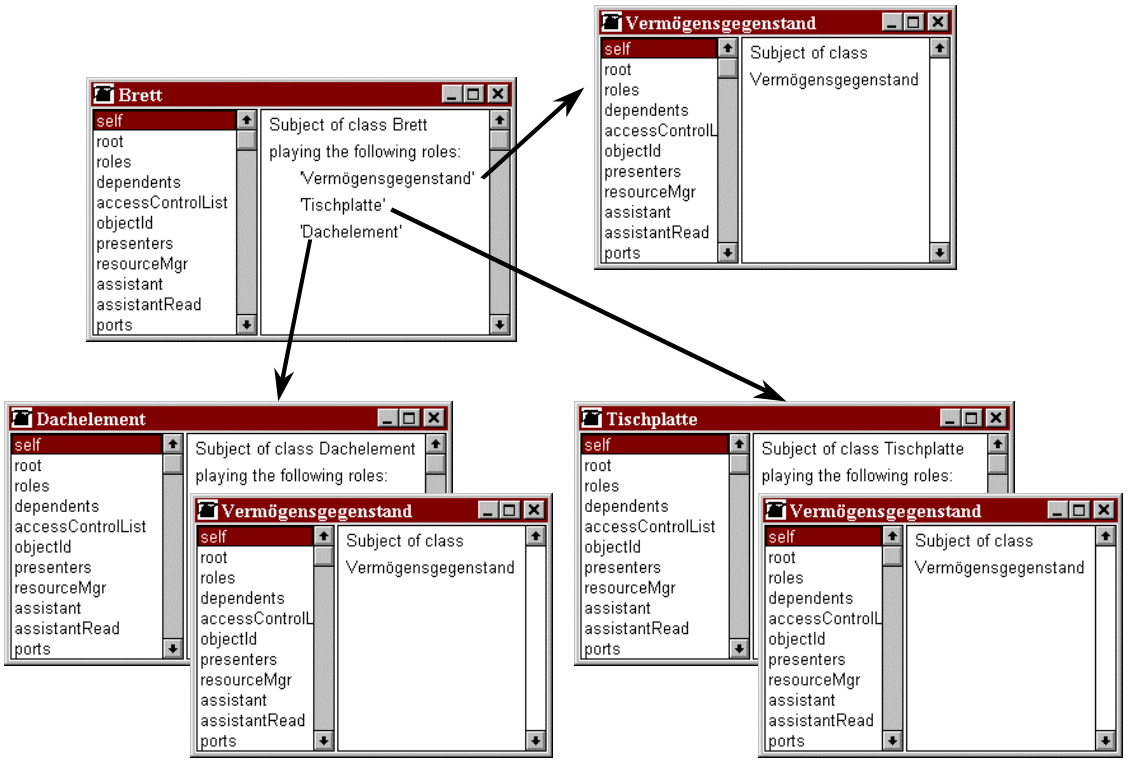

Abbildung 40: Ein Subjekt mit mehrstufig zugeordneten Rollen

- Jede Rolle liefert einen anderen Wert des Subjektes zurück.

```
V1 wert. <printIt> 10
V2 wert. <printIt>  50
V3 wert. <printIt> 30
D1 wert. <printIt> 25
T1 wert. <printIt> 22
```

- Wird das Brett nach seinem Wert gefragt, so liefert es den höchsten der Einzelwerte zurück.

```
B1 wert. <printIt> 50
```

- Die Instanzvariablen des Bretts können über jede Rolle gesetzt werden.

```
V1 holzart: 'Fichte'.
V3 haerte. <printIt> 'hart'
```

- Wird durch Veränderung der Instanzvariablen ein anderer Einzelwert zum höchsten, so wird dieser als Wert des Brettes zurückgeliefert.

```
V2 laenge: 12.
B1 wert. <printIt> 60
```

6 Zusammenfassung

Die vorliegende Arbeit versuchte die in der Literatur bisher vorhandenen Überlegungen zum Thema „Rollen von Objekten“ zusammenzufassen und gegenüberzustellen. Aus der Kritik der einzelnen Konzepte sollte ein Ansatz resultieren, der möglichst viele der dargelegten Ideen beinhaltet und möglichst viele der erkannten Schwächen der bisherigen Ansätze vermeidet.

Nach den in [Lieberman88] vorgeschlagenen Dimensionen unterstützt dieser Ansatz neben statischer auch dynamische Vererbung, neben Vererbung auf Klassenebene auch Vererbung auf Instanzenebene und ist in der Lage, alle diese Vererbungslinien explizit auszudrücken. So können diverse Probleme der klassischen Objektorientierten Programmierung – wie Klassenmigrationen wärend der Lebenszyklen von Objekten und Mehrfachvererbung bei Überschneidungen in der Klassenhierarchie – vermieden werden.

Die Kombination dieses Ansatzes mit den Überlegungen zur Verteilung von Objekten, führte zur Spezifikation eines Common Object Services für [CORBA]. Dieses Roleification-Service soll die Wiederverwendung von Objekt-Instanzen in mehreren Kontexten ermöglichen, indem Objekten dynamisch Rollen zugeordnet werden können.

Da dieses Service in die Mechanismen von [CORBA] eingebettet wurde, ist es möglich, daß die einzelnen Teile eines auf diese Art definierten Subjektes in unterschiedlichen Systemen beheimatet sind und verwaltet werden. Da die Spezifikation des Roleification-Services unabhängig von der Umgebung der konkreten Implementation – also von Betriebssystem, Programmiersprache, etc. – ist, können auch heterogene Systeme gekoppelt werden.

Außerdem wurden die Mechanismen der CORBA-Implementation in Distributed Smalltalk untersucht, um festzustellen, wie es am effizientesten möglich ist, ein Roleification-Service gemäß der obigen Spezifikation in Distributed Smalltalk umzusetzen.

Aufbauend auf diese Erkenntnisse wurde eine Implementation dieses Services vorgestellt: Diese Implementation hatte das Ziel, die kontextübergreifende Wiederverwendung von Objekt-Instanzen mit Hilfe des Rollenkonzeptes der Objektorientierten Programmierung in verteilten Systemen zu ermöglichen.

Es handelt sich um eine Entwicklungsumgebung, die alle notwendigen Mechanismen unterstützt, Werkzeuge zur Verwaltung und Konfiguration zur Verfügung stellt und den Einsatz des Roleification-Services durch einige Beispiele illustriert.

Da sich die konkrete Implementation in Distributed Smalltalk auf die Standard-Mechanismen von CORBA 2.0 stützt, ist es auf einfache Art möglich, die Koppelung mit anderen Objekträumen herbeizuführen, wenn in diesen Objekträumen eine

Implementation des Roleification-Services verfügbar ist, die ebenfalls der plattformunabhängigen Spezifikation genügt.

7 Anhang

7.1 Install.st

```
Transcript cr.
Transcript cr.
Transcript show: 'Das Roleification-Service fuer CORBA 2.0 wird installiert ...'.
Transcript cr.

Transcript cr.
Transcript show: 'Datei DSTRepository-CompoundLifecycles.st enthaelt das erweiterte IDL-
Interface des Relationship-Services ...'.
(Filename named: 'A:\Roleification\DSTRepository-CompoundLifecycles.st') fileIn.

Transcript cr.
Transcript show: 'Datei DSTlink.st enthaelt die erweiterte Klasse DSTlink ...'.
(Filename named: 'A:\Roleification\DSTlink.st') fileIn.

Transcript cr.
Transcript show: 'Datei Roleification.st enthaelt die Service-Klassen und Beispiel-Klassen ...'.
(Filename named: 'A:\Roleification\Roleification.st') fileIn.

Transcript cr.
Transcript show: 'Datei DSTRepository-custom IFs.st enthaelt die notwendigen IDL-Interfaces
...'.
(Filename named: 'A:\Roleification\DSTRepository-custom IFs.st') fileIn.

Transcript cr.
Transcript cr.
Transcript show: 'Das Roleification-Service fuer CORBA 2.0 wurde erfolgreich installiert.'.
Transcript cr.
Transcript cr.
```

7.2 DSTRepository-CompoundLifecycles.st

```
!DSTRepository methodsFor: 'policy IFs'!

// CompoundLifecycles
// This module defines the types and interfaces used to manage
// object lifecycles, and to define basic links between objects.
// (Compound Lifecycle and Links must be merged in the absence of
// support for cross-module forward references.)
//
module CompoundLifecycles  {

    // -- This section is for compound lifecycle --
    typedef sequence<LinkSet> LinkSets;
```

```
typedef sequence<CORBA::ORBId> ORBIds;
exception SemanticError {string reason; };

//
// This interface defines the operations on a Tour
//
interface Tour {

    #pragma  selector add_to_tour addToTour:
    // This operation adds a LinkSet object to the tour of the
    // receiver
    void add_to_tour (in LinkSet to_visit);

    #pragma  selector add_to_copy_map addCopyMap:is:
    // This operation adds a LinkSet object to the copy map of the
    // receiver
    void add_to_copy_map (in CORBA::ORBId id, in LinkSet link_set);

    #pragma  selector add_to_destroy_map addDestroyMap:is:dependsOn:
    // This operation adds a LinkSet object and the IDs of the
    // LinkSets which hold existence ensuring links to it to the
    // destroy map of the receiver
    void add_to_destroy_map (
                                in CORBA::ORBId id,
                                in LinkSet link_set,
                                in ORBIds depends_on);

    #pragma  selector lookup_copy_map lookupCopyMap:
    // This operation returns the LinkSet which has the given ID,
    // or null if there is no LinkSet in the map with the given ID.
    LinkSet lookup_copy_map (in CORBA::ORBId id);
};

//
// This interface defines the operations on the lifecycle traversal
// object. Note that a SessionContext may be provided in the
// initialization of the receiver to allow for graceful interaction
// with the user during the course of compound lifecycle
// operations.
//
interface Traversal : Tour {

    #pragma  selector set_session session:
    // This operation is used to set the SessionContext of the
    // receiver
    // so that subsequent user interactions may be handled
    // gracefully
    void set_session (in UIContexts::SessionContext session);

    #pragma  selector copy copy:to:
    // This operation coordinates a copy of the given LinkSet
    // object to the given location.
    LinkSet copy (in LinkSet what, in CosLifeCycle::FactoryFinder where)
                        raises (SemanticError);

    #pragma  selector deep_copy deepCopy:to:
    // This operation coordinates a copy of the given LinkSet
```

```
    // object to the given location.
    LinkSet deep_copy (in LinkSet what, in CosLifeCycle::FactoryFinder where)
                        raises (SemanticError);

    #pragma selector move move:to:
    // This operation coordinates a move of the given LinkSet
    // object to the given location
    void move (in LinkSet what, in CosLifeCycle::FactoryFinder where)
                        raises (SemanticError);

    #pragma  selector destroy destroy:orphans:
    // This operation coordinates a destroy of the given LinkSet
    // object. If any objects cannot be destroyed (eg due to
    // existance-ensuring links) then these will be reparented to
    // the orphans parameter. The operation will return true
    // unless aborted by the user.
    boolean destroy (in LinkSet what, in LinkSet orphans)
                                raises (SemanticError);

    #pragma  selector externalize externalize:to:
    // This operation coordinates an externalize of the given
    // LinkSet object to the given Stream
    void externalize (in LinkSet what, in SmalltalkTypes::Stream data)
                        raises (SemanticError);

    #pragma  selector internalize internalizeAt:from:
    // This operation coordinates an internalize of a compound
    // Lifecycle object from the given Stream
    LinkSet internalize (
                                in CosLifeCycle::FactoryFinder where,
                                in SmalltalkTypes::Stream data)
                        raises (SemanticError);

    // This operation indicates to the receiver that it is done (with
    // the transaction)
    void destroy_traversal ();
};

//
// This interface defines the operations required to administer
// lifecycle operations on compound object structures
//
interface LifecycleAdmin {

    #pragma  access build_copy_tour read
    // This operation is used to build a compound copy tour
    void build_copy_tour (in Tour the_tour);

    #pragma  selector do_copy copyTo:tour:
    // This operation is used to make a compound copy of the
    // receiver in a tour
    void do_copy (in CosLifeCycle::FactoryFinder where, in Tour the_tour);

    #pragma  selector post_copy_fixup postCopyFixup:
    // This operation is used to fixup the receiver after a copy
    // operation
    void post_copy_fixup (in Tour the_tour);
```

```
#pragma  access build_deep_copy_tour read
// This operation is used to build a compound deep copy tour
void build_deep_copy_tour (in Tour the_tour);

#pragma  selector do_deep_copy deepCopyTo:tour:
// This operation is used to make a compound deep copy of
// the receiver in a tour
void do_deep_copy (
                        in CosLifeCycle::FactoryFinder where,
                        in Tour the_tour);

#pragma  selector post_deep_copy_fixup postDeepCopyFixup:
// This operation is used to fixup the receiver after a deep
// copy operation
void post_deep_copy_fixup (in Tour the_tour);

#pragma  access build_move_tour write
// This operation is used to build a compound move tour
void build_move_tour (in Tour the_tour);

#pragma  selector do_move moveTo:tour:
// This operation is used to perform a compound move of the
// receiver in a tour
void do_move (in CosLifeCycle::FactoryFinder where, in Tour the_tour);

#pragma  selector post_move_fixup postMoveFixup:
// This operation is used to fixup the receiver after a move
// operation
void post_move_fixup (in Tour the_tour);

// This operation is used to build a compound destroy tour
void build_destroy_tour (in Tour the_tour);

#pragma  selector do_destroy destroyInTour:
// This operation is used to perform a compound destroy of
// the receiver in a tour
void do_destroy (in Tour the_tour);

// This operation is used to build a compound externalize
// tour
void build_externalize_tour (in Tour the_tour);

#pragma  selector do_externalize externalizeTo:tour:
// This operation is used to externalize the receiver to a
// stream in a tour
void do_externalize (in SmalltalkTypes::Stream data, in Tour the_tour);

#pragma  selector do_internalize internalizeAt:from:tour:
// This operation is used to internalize the receiver in a tour
void do_internalize (
                        in CosLifeCycle::FactoryFinder where,
                        in SmalltalkTypes::Stream data,
                        in Tour the_tour);

#pragma  selector post_internalize_fixup postInternalizeFixup:
// This operation is used to fixup the receiver after an
// internalize operation
void post_internalize_fixup (in Tour the_tour);
```

```
};

//
// This interface defines the Lifecycle operations required by
// clients of the Traversal service
//
interface Lifecycle :
                        CosLifeCycle::LifeCycleObject,
                        CosLifeCycle::FactoryRepresentative
{

    attribute LinkSet link_set;
};

// ---------- This section is for links ----------
typedef unsigned long LinkId;

typedef unsigned short LinkType;
typedef LinkType LinkRole;
typedef sequence<LinkRole> LinkRoles;

#pragma  class LinkState ByteArray
typedef sequence<octet> LinkState;

#pragma  class LinkBinding Association
struct  LinkBinding {LinkId key; LinkRead value; };

typedef sequence<LinkBinding> LinksRead;

// link types are odd, and the headRole is always the same value
// as the link type. The tailRole is always equal to the headRole +
// 1.
const LinkType containment_link = 1;

const LinkType reference_link = 3;
const LinkType designation_link = 5;
const LinkType weak_link = 7;
const LinkType roleification_link = 9;
exception UnknownId {};

//
// This interface defines types and readable attributes which
// allow navigation within a network of links to be performed by
// clients without allowing them modification rights.
//
interface LinkRead {

    // link attributes which allow readonly navigation
    readonly attribute LinkSet head;

    readonly attribute LinkSet tail;
    readonly attribute LinkId head_id;
    readonly attribute LinkId tail_id;
    readonly attribute LinkRole head_role;
    readonly attribute LinkRole tail_role;
    readonly attribute Lifecycle head_object;
    readonly attribute Lifecycle tail_object;
    readonly attribute LinkType type;
```

```
    readonly attribute boolean is_existence_ensuring;
    readonly attribute boolean propagate_deep_copy;
};

//
// This interface describes the data structures and operations
// used to manage and modify Link information.
//
interface Link : LinkRead, LifecycleAdmin {

    #pragma  selector initialize_link formLink:to:of:
    // This operation is used to initialize a new link
    void initialize_link (in LinkSet head, in LinkSet tail, in LinkType type);

    #pragma selector destroy_link dropLink
    // This operation is used to drop the link from the head and
    // tail LinkSets and to destroy the link
    void destroy_link ();

    #pragma  selector get_link_state linkState
    // This operation is used to return state information which
    // can be used by LinkSet::createAndInitializeLink
    LinkState get_link_state ();

    #pragma  selector set_propagate_deep_copy setPropagateDeepCopy:
    // This operation is used to set the state of the
    // propagate_deep_copy attribute
    void set_propagate_deep_copy (in boolean propagate);
};

//
// This interface defines the LinkSet read-only functionality
//
interface LinkSetRead {

    readonly attribute CORBA::ORBId unique_id;
    readonly attribute Lifecycle associated_lifecycle_object;

    // This operation returns all of the links of the receiver
    LinksRead all_links ();

    // This operation returns all of the links for which the receiver
    // is the head
    LinksRead head_links ();

    // This operation returns all of the links for which the receiver
    // is the tail
    LinksRead tail_links ();

    #pragma  selector all_links_for_role allLinksForRole:
    // This operation returns all of the links in a given role of the
    // receiver
    LinksRead all_links_for_role (in LinkRole role);

    #pragma  selector get_link_read getLinkRead:
    // This operation returns a particular link in the receiver
    LinkRead get_link_read (in LinkId id)
                                raises (UnknownId);
```

```
        #pragma  selector get_link_roles linkRoles
        // This operation returns the link roles which are currently
        // managed by the receiver.
        LinkRoles get_link_roles ();
    };

    //
    // This interface defines the LinkSet object assistant functionality
    //
    interface LinkSet : LinkSetRead, LifecycleAdmin, CosLifeCycle::LifeCycleObject {

        #pragma  selector initialize_associated_lifecycle_object
initializeAssociatedLifecycleObject:
        // This operation is used to initialize the
        // associated_lifecycle_object readonly attribute declared in
        // LinkSetRead.
        void initialize_associated_lifecycle_object (in Lifecycle the_object);

        #pragma  selector create_link createLink:tail:
        // This operation is used to create and add a new link in the
        // same location as the receiver
        Link create_link (in LinkType type, in LinkSet tail)
                        raises (SemanticError);

        #pragma  selector create_and_initialize_link createAndInitializeLink:tail:
        // This operation is used to create, initialize, and add a new
        // link in the same location as the receiver
        Link create_and_initialize_link (in LinkState state, in LinkSet tail)
                        raises (SemanticError);

        #pragma  selector add_link addLink:role:
        // This operation is used to add a link to the receiver in a
        // given role
        LinkId add_link (in Link the_link, in LinkRole role)
                        raises (SemanticError);

        #pragma  selector drop_link dropLink:
        // This operation is used to drop an link from the receiver
        Link drop_link (in LinkId id)
                        raises (UnknownId);

        #pragma  selector get_link getLink:
        // This operation augments the LinkSetRead::getLink method
        // to return a writable Link objref.
        Link get_link (in LinkId id)
                        raises (UnknownId);
    };

    //
    // ---------------------------------------------
    // This section defines interfaces for LinkSet and Traversal
    // factories
    //
    // This is the interface of LinkSetFactory objects
    //
    interface LinkSetFactory : CosLifeCycle::GenericFactory {};
```

```
    //
    // This is the interface of TraversalFactory objects
    //
    interface TraversalFactory : CosLifeCycle::GenericFactory {};
};! !
```

7.3 DSTlink.st

```
DSTPropertySet subclass: #DSTlink
    instanceVariableNames: 'head headId tail tailId tailObject type '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DST-Policy'!
DSTlink comment:
'                               (c) Copyright 1996 ParcPlace-Digitalk, Inc.  All Rights Reserved
                                (c) Copyright 1993-1995 Hewlett-Packard Company.  All Rights
Reserved

DSTlinks are used to represent containment and link relationships between consenting
DSTapplicationObject instances.  They work in conjunction with DSTapplicationAssistant
objects which are attached to the application object and implement the CompoundLifecycle
operations on networks of application objects.  Links are directed relationships between a
holder (parent) and a holdee (child).  Links inherit from DSTpropertySet and thus support
attributed, directed relationships.

Instance Variables:
    head   <DSTapplicationAssistant> which is the holder of the link.
    headId          <DSTObjRef> the LinkID by which the head object identifies the link.
    tail            <DSTapplicationAssistant> which is the holdee of the link.
    tailId   <DSTObjRef> the LinkID by which the tail object identifies the link.
    tailObject  <DSTapplicationObject> the associated application object of the tail of the link.
    type    <Integer> the LinkType of the link

Class variables:
    Map    <>

Subclasses must implement:
    LifeCycleAdmin
        moveTo:tour:
'!


!DSTlink methodsFor: 'DesktopLink'!

asLinkInfo
    "build a LinkInfo collection. If I am WEAK, I require no interaction
    from the tail object"

    | linkInfo |
    linkInfo := DSTlinkInfo new.
    linkInfo link: self; linkId: headId; head: self headObject; tail: self tailObject; type: type;
tailClassID: self tailClassID; linkTitle: self linkTitle; product: ((CORBAConstants at:
#'INV_OBJREF')
            handle:
                    [:ex |
```

```
                    linkInfo tail: nil.
                    '* Bad link *']
            do: [(self tailObject linkSet hasProperty: 'productName')
                        ifTrue: [self tailObject linkSet getProperty: 'productName']
                        ifFalse: [self tailObject productName]]).
    ^linkInfo!

copyTo: aContainer loc: aLocAssist in: aSession
    "produce a copy of the tail of the receiver and its contained
    sub-structure. Return the new object which is contained by
    aContainer. Create all links and objects in the domain of
    aContainer"

    | traversal dta |
    traversal := DSTTraversal new.
    traversal session: aSession.
    dta := traversal copy: tail to: aLocAssist factoryFinder.
    traversal destroyTraversal.
    ^aContainer containChild: dta!

deepCopyTo: aContainer loc: aLocAssist in: aSession
    "produce a deepCopy of the tail of the receiver and its contained
    sub-structure. Return the new object which is contained by
    aContainer. Create all links and objects in the domain of
    aContainer"

    | traversal dta |
    traversal := DSTTraversal new.
    traversal session: aSession.
    dta := traversal deepCopy: tail to: aLocAssist factoryFinder.
    traversal destroyTraversal.
    ^aContainer containChild: dta!

disposeIn: aSession
    "dispose of self and of destination if no other
    existence-guaranteeing
    links exist to it."

    | traversal ok |
    ok := true.
    self dropHead.
    self dropTail.
    self isContainment
        ifTrue:
            [traversal := DSTTraversal new.
            traversal session: aSession.
            ok := traversal destroy: tail orphans: aSession orphanage linkSet.
            traversal destroyTraversal].
    ok
        ifTrue: [self destroy]
        ifFalse: [self
                    formLink: head
                    to: tail
                    of: type]!

externalizeTo: stream in: aSession
    "externalize the tail of the receiver and return the stream"
```

```
    | traversal result |
    traversal := tail createObject: DSTTraversal getInstanceACL.
    traversal session: aSession.
    result := traversal externalize: tail to: stream.
    traversal destroyTraversal.
    ^result!

linkTitle
    "return the title of the link"

    | lt |
    ^(self hasProperty: #linkTitle)
        ifFalse:
            [lt := self tailObject title.
            self isWeak ifTrue: [self defineProperty: #linkTitle value: lt].
            lt]
        ifTrue: [self getProperty: #linkTitle]!

linkTitle: aStr
    "set the title of the link"

    self isContainment
        ifTrue: [self tailObject title: aStr]
        ifFalse: [self defineProperty: #linkTitle value: aStr]!

moveTo: newContainer loc: aLocAssist in: aSession
    "move the receiver to a new container. If this is a containment link
    and the new container is not local, create a transaction to move the
    contained substructure as well"

    | traversal dta link state |
    self checkContainmentConsistency: newContainer.
    self isContainment & newContainer isLocal not
        ifTrue:
            [self dropHead.
            self dropTail.
            state := self linkState.
            traversal := DSTTraversal new.
            traversal session: aSession.
            ORBObject noPermissionSignal
                    handle:
                            [:x |
                            "a no permission error occured while trying to build the
                            tour"
                            headId := head addLink: self role: self headRole.
                            tailId := self isWeak
                                                    ifTrue: [0]
                                                    ifFalse: [tail addLink: self role: self tailRole].
                            ^x reject]
                    do: [dta := traversal move: tail to: aLocAssist factoryFinder].
            traversal destroyTraversal.
            link := newContainer linkSet createAndInitializeLink: state tail: dta.
            self destroy.
            ^link]
        ifFalse:
            [self dropHead.
            self defineProperty: #linkTime value: Timestamp now formattedString.
            head := newContainer linkSet.
```

```
            headId := head addLink: self role: self headRole.
            ^self]!

promoteToContainment
    "promote the receiver to containment link status, as it may have been
    responsible for the creation of an orphan copy during a deepcopy
    operation. There must be a better way to update its childInfo..."

    self dropHead.
    self dropTail.
    self
        formLink: head
        to: tail
        of: self containment!

shareTo: aContainer in: aSession
    "returns a new link to the tail object with downgraded link
    strength, and linked as a child of aContainer"

    ^aContainer linkSet createLink: (self isContainment
            ifTrue: [self reference]
            ifFalse: [self isReference
                                ifTrue: [self designation]
                                ifFalse: [self weak]])
        tail: tail!

tailClassID
    "return the tailClassID of the receiver"

    | cl |
    ^(self hasProperty: #tailClassID)
        ifFalse:
            [self defineProperty: #tailClassID value: (cl := self tailObject ACL).
            cl]
        ifTrue: [self getProperty: #tailClassID]!

tailClassID: aClass
    "set the class of the tailObject of the link"

    self defineProperty: #tailClassID value: aClass! !

!DSTlink methodsFor: 'LifeCycleAdmin'!

buildCopyTour: aTour
    "build the copy tour for the receiver"

    self isContainment ifTrue: [aTour asRemotable addToTour: tail]!

buildDeepCopyTour: aTour
    "build the deep copy tour for the receiver"

    self propagateDeepCopy ifTrue: [aTour asRemotable addToTour: tail]!

buildDestroyTour: aTour
    "build the destroy tour for the receiver"

    self isContainment ifTrue: [aTour asRemotable addToTour: tail]!
```

```
buildExternalizeTour: aTour
    "build the externalize tour for the receiver"

    self isContainment ifTrue: [aTour asRemotable addToTour: tail]!

buildMoveTour: aTour
    "build the move tour for the receiver"

    self isContainment ifTrue: [aTour asRemotable addToTour: tail]!

copyTo: aFactoryFinder tour: aTour
    "do nothing in this pass"!

deepCopyTo: aFactoryFinder tour: aTour
    "do nothing in this pass"!

destroyInTour: aTour
    "destroy the receiver in the context of the tour"

    self dropLink.!

externalizeTo: aStream tour: aTour
    "should never get here"

    self shouldNotImplement!

internalizeAt: aFinder from: aStream tour: aTour
    "internalize an instance of the receiver"

    self shouldNotImplement!

moveTo: aFactoryFinder tour: aTour
    "move the receiver to the domain of the factory in the context of the tour"

    self subclassResponsibility!

postCopyFixup: aTour
    "add the receiver's copy to the mapped copy as indicated. The
    default is to do a shallow copy unless the tail object is in the
    tourMap, and nothing if the head object is not in the map."

    | hcopy tcopy |
    hcopy := aTour asRemotable lookupCopyMap: head uniqueIdentifier.
    hcopy isNil ifTrue: [^nil].
    tcopy := aTour asRemotable lookupCopyMap: tail uniqueIdentifier.
    tcopy isNil ifTrue: [tcopy := tail].
    hcopy asRemotable createAndInitializeLink: self linkState tail: tcopy!

postDeepCopyFixup: aTour
    "add the receiver's copy to the mapped copy as indicated. The
    default is to do a shallow copy unless the tail object is in the
    tourMap, and nothing if the head object is not in the map."

    | hcopy tcopy |
    hcopy := aTour asRemotable lookupCopyMap: head uniqueIdentifier.
    hcopy isNil ifTrue: [^nil].
    tcopy := aTour asRemotable lookupCopyMap: tail uniqueIdentifier.
    tcopy isNil ifTrue: [tcopy := tail].
```

```
    hcopy asRemotable createAndInitializeLink: self linkState tail: tcopy!

postInternalizeFixup: aTour
    "Reattach to the tail object, if it exists. Otherwise, drop the
    association"

    self shouldNotImplement!

postMoveFixup: aTour
    "link the copy so as to replace the receiver in the network"

    | hcopy tcopy |
    hcopy := aTour asRemotable lookupCopyMap: head uniqueIdentifier.
    hcopy isNil ifTrue: [hcopy := head].
    tcopy := aTour asRemotable lookupCopyMap: tail uniqueIdentifier.
    tcopy isNil ifTrue: [tcopy := tail].
    hcopy asRemotable createAndInitializeLink: self linkState tail: tcopy.
    self destroyInTour: aTour! !

!DSTlink methodsFor: 'printing'!

printOn: aStream
    "print a representation of the receiver onto the stream"

    self isContainment ifTrue: [^aStream nextPutAll: 'a Containment Link'].
    self isReference ifTrue: [^aStream nextPutAll: 'a Reference Link'].
    self isDesignation ifTrue: [^aStream nextPutAll: 'a Designation Link'].
    self isWeak ifTrue: [^aStream nextPutAll: 'a Weak Link'].
    self isRoleification ifTrue: [^aStream nextPutAll: 'a Roleification Link'].
    aStream nextPutAll: 'an Invalid Link'! !

!DSTlink methodsFor: 'PortRepresentative'!

dataViewAs: dataType
    "return a data view of the appropriate type by forwarding it to the
    viewport"

    | viewport |
    viewport := self viewPort.
    ^viewport isNil
        ifTrue: [self tailObject dataViewAs: dataType under: nil]
        ifFalse: [viewport dataViewAs: dataType]!

viewPort
    "return the viewport of the receiver"

    | viewport |
    (self hasProperty: #viewport)
        ifTrue: [^self getProperty: #viewport]
        ifFalse:
            [viewport := self tailObject createPort.
            self viewPort: viewport.
            ^viewport]!

viewPort: aPort
    "set the viewport of the receiver"

    self makeModified.
```

```
    self defineProperty: #viewport value: aPort! !

!DSTlink methodsFor: 'accessing'!

allContentsInto: aSet
    "add the receiver and all of its containment relatives to the set"

    super allContentsInto: aSet.
    self isContainment ifTrue: [tail allContentsInto: aSet]!

asLinkRead
    "return the receiver widened to a read-only interface"

    ^self widenTo: #LinkRead!

checkContainmentConsistency: newHead
    "verify that the proposed change does not violate the containment
    assumptions"

    self isContainment ifTrue: [(newHead isAncestor: tail uniqueIdentifier)
            ifTrue: [^(CORBAConstants at: #'::CompoundLifecycles::SemanticError')
                            raiseWith: (Dictionary with: #reason -> 'containment consistency')
                            errorString: ' containment semantics violation']]!

tail: aLinkSet
    "Set the value of tail."

    tail := aLinkSet!

tailObject: anObject
    "Set the associated object of the tail associate"

    tailObject := anObject! !

!DSTlink methodsFor: 'link attributes'!

containment
    "return the attributes of a containment link"

    ^self class containment!

designation
    "return the attributes of a designation link"

    ^self class designation!

isContainment
    "return if the receiver is a containment link"

    ^type = self containment!

isDesignation
    "return if the receiver is a designation link"

    ^type = self designation!

isReference
    "return if the receiver is a reference link"
```

```
    ^type = self reference!

isRoleification
    "return if the receiver is a roleification link"

    ^type = self roleification!

isWeak
    "return if the receiver is a weak link"

    ^type = self weak!

reference
    "return the attributes of a reference link"

    ^self class reference!

roleification
    "return the attributes of a roleification link"

    ^self class roleification!

typeString
    "return the string representation of the link type"

    self isContainment ifTrue: [^'containment'].
    self isReference ifTrue: [^'reference'].
    self isDesignation ifTrue: [^'designation'].
    self isRoleification ifTrue: [^'roleification'].
    ^type printStringRadix: 16!

weak
    "return the attributes of a weak link"

    ^self class weak! !

!DSTlink methodsFor: 'persistence'!

copyPersistentState: anArray
    "copy my persistent state"

    self makeClean.
    super copyPersistentState: (anArray at: 1).
    (self hasProperty: #linkTitle) ifTrue: [self linkTitle: '%' , self linkTitle].
    type := (anArray at: 4).!

persistentState
    "return an information array about the receiver"

    ^Array
        with: super persistentState
        with: head
        with: tail asInactiveObjRef
        with: type!

restorePersistentState: anArray
    "restore my state from the persistent state array"
```

```
    self makeClean.
    super restorePersistentState: (anArray at: 1).
    head := (anArray at: 2).
    tail := (anArray at: 3).
    type := (anArray at: 4).! !

!DSTlink methodsFor: 'private'!

dropHead
    "drop the head link. If my head is remote, ignore any errors"

    Object errorSignal handle: [:y | head isLocal ifTrue: [y reject]]
        do: [head dropLink: headId]!

dropTail
    "drop the tail link if I am not weak. Ignore any errors"

    self isWeak ifFalse: [Object errorSignal handle: [:y | ]
            do: [tail dropLink: tailId]]!

weak: headAssoc to: tailObj
    "establish a new weak link. Used by the tester to create links to
    arbitrary objects. Do whatever is necessary to make this work,
    recognizing that errors may result if these limits are exceeded"

    self
        formLink: headAssoc
        to: (DSTapplicationAssistant new initializeAssociatedLifecycleObject: tailObj)
        of: self weak! !

!DSTlink methodsFor: 'ReportingLink'!

disableEvent: event
    "disable the given event"

    (self hasProperty: #linkEvents)
        ifTrue: [(self getProperty: #linkEvents)
                remove: event ifAbsent: []]!

enableEvent: event
    "enable the given event"

    (self hasProperty: #linkEvents)
        ifTrue: [(self getProperty: #linkEvents)
                add: event]
        ifFalse: [self defineProperty: #linkEvents value: (Set with: event)]!

raiseEvent: event withAny: any
    "propogate the event to the headObject of the receiver if it has been
    enabled"

    Object errorSignal handle: [:ex | type class == ORBDeadObject
            ifTrue: ["ignore dead links"
                ^nil]
            ifFalse: [ex reject]]
        do: [((self hasProperty: #linkEvents)
                and: [(self getProperty: #linkEvents)
```

```
                                includes: event])
                        ifTrue:
                                [(event asString = 'propertyChanged' and: [(any at: 2) asString = 'title'])
                                        ifTrue: [((self hasProperty: #linkTitle)
                                                        and: [self isContainment not])
                                                        ifTrue: [^nil]
                                                        ifFalse: [^self headObject propertyChanged:
any onLink: headId]].
                                (event asString = 'propertyChanged' and: [(any at: 2) asString =
'classId'])
                                        ifTrue:
                                                [self tailClassID: (any at: 3).
                                                ^self headObject propertyChanged: any onLink:
headId].
                                ^self headObject
                                        perform: (event asString , ':onLink:') asSymbol
                                        with: any
                                        with: headId]]! !

!DSTlink methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"

    ^'58fbcff1-a32b-0000-020f-1c6813000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"

    ^#'::DTOLinks::DesktopLink'! !

!DSTlink methodsFor: 'Link'!

dropLink
    "drop links to src & dest.  Destroy the receiver"

    self dropHead.
    self dropTail.
    self destroy!

formLink: src to: dest of: typ
    "set links to src & dest"

    self makeModified.
    type := typ.
    head := src.
    tail := dest.
    self defineProperty: #linkTime value: Timestamp now formattedString.
    self checkContainmentConsistency: self headObject.
    tailObject := dest applicationBase.
    headId := head addLink: self role: self headRole.
    tailId := self isWeak
                    ifTrue: [0]
                    ifFalse: [tail addLink: self role: self tailRole]!

linkState
    "return a serialized form of the receiver"
```

```
    ^self externalize contents!

setPropagateDeepCopy: bool
    "set the deepCopy property to the given value"

    self defineProperty: #deepCopy value: bool! !

!DSTlink methodsFor: 'LinkRead'!

head
    "Answer the value of head."

    ^head!

headId
    "Answer the value of headId."

    ^headId!

headId: anId
    "Set the value of headId."

    ^headId := anId!

headObject
    "return the associated object of the head associate"

    ^head applicationBase!

headRole
    "Answer the role to be used for the head associate."

    self isContainment ifTrue: [^self class containmentHeadRole].
    self isReference ifTrue: [^self class referenceHeadRole].
    self isDesignation ifTrue: [^self class designationHeadRole].
    self isRoleification ifTrue: [^self class roleificationHeadRole].
    ^self class weakHeadRole!

isExistenceEnsuring
    "return if the receiver is an existence ensuring link"

    ^self isContainment | self isReference!

propagateDeepCopy
    "return if the receiver has the deepCopyPropagation property and
    it is true"

    ^(self hasProperty: #deepCopy)
        ifTrue: [self getProperty: #deepCopy]
        ifFalse: [self isContainment]!

tail
    "Answer the value of tail."

    ^tail asRemotable!

tailId
    "Answer the value of tailId."
```

```
    ^tailId!

tailObject
    "Return the associated object of the tail associate"

    ^tailObject asRemotable!

tailRole
    "Answer the role to be used for the tail associate."

    self isContainment ifTrue: [^self class containmentTailRole].
    self isReference ifTrue: [^self class referenceTailRole].
    self isDesignation ifTrue: [^self class designationTailRole].
    self isRoleification ifTrue: [^self class roleificationTailRole].
    ^self class weakTailRole!

type
    "Answer the value of type."

    ^type! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

DSTlink class
    instanceVariableNames: ''!


!DSTlink class methodsFor: 'link roles'!

containmentHeadRole
    "return the head role of a containment link"

    ^self containment!

containmentTailRole
    "return the tail role of a containment link"

    ^self containment + 1!

designationHeadRole
    "return the head role of a designation link"

    ^self designation!

designationTailRole
    "return the tail role of a designation link"

    ^self designation + 1!

referenceHeadRole
    "return the head role of a reference link"

    ^self reference!

referenceTailRole
    "return the tail role of a reference link"

    ^self reference + 1!
```

```
roleificationHeadRole
    "return the head role of a roleification link"

    ^self roleification!

roleificationTailRole
    "return the tail role of a roleification link"

    ^self roleification + 1!

weakHeadRole
    "return the head role of a weak link"

    ^self weak!

weakTailRole
    "return the tail role of a weak link"

    ^self weak + 1! !

!DSTlink class methodsFor: 'link attributes'!

containment
    "return the attributes of a containment link"

    ^CORBAConstants at: #'::CompoundLifecycles::containment_link'!

designation
    "return the attributes of a designation link"

    ^CORBAConstants at: #'::CompoundLifecycles::designation_link'!

isContainment: attr
    "return if the attributes denote a containment link"

    ^attr = self containment!

isDesignation: attr
    "return if the attributes denote a designation link"

    ^attr = self designation!

isReference: attr
    "return if the attributes denote a reference link"

    ^attr = self reference!

isRoleification: attr
    "return if the attributes denote a roleification link"

    ^attr = self roleification!

isWeak: attr
    "return if the attributes denote a weak link"

    ^attr = self weak!
```

```
reference
    "return the attributes of a reference link"

    ^CORBAConstants at: #'::CompoundLifecycles::reference_link'!

roleification
    "return the attributes of a roleification link"

    ^CORBAConstants at: #'::CompoundLifecycles::roleification_link'!

weak
    "return the attributes of a weak link"

    ^CORBAConstants at: #'::CompoundLifecycles::weak_link'! !

!DSTlink class methodsFor: 'creating'!

new
    "create an initialized instance of the receiver"

    ^super basicNew initialize! !
```

7.4 Roleification.st

```
DSTapplicationObject subclass: #ObjectWithRoles
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!ObjectWithRoles methodsFor: 'printing'!

inspect

    SubjectInspector  openOn: self.!

printOn: aStream
    "Adds the currently played Roles to the signature of the Receiver-Object."

    aStream nextPutAll: 'Subject of class ', (self class printString).
    (self listAllRoles) isEmpty ifFalse: [aStream nextPutAll: ' playing the following roles: ';
cr.
                                          (self listAllRoles) keys do: [:x| aStream tab;
nextPutAll: x printString; cr].
                                          aStream cr.]! !

!ObjectWithRoles methodsFor: 'accessing'!

listAllRoles
    "Returns a Dictonary with all Roles currently played by the Receiver-Object."

    |collection roleSet tail tailTitle nextRoleSet|
    roleSet := Dictionary new.
    collection := (self linkSet allLinksForRole: (DSTlink roleificationHeadRole)).
```

```
    collection do: [:v | tail:= v value tailObject. tailTitle:= tail roleTitle.
                                    (roleSet at: (tailTitle) ifAbsentPut: (OrderedCollection
new)) add: tail.
                                    nextRoleSet := (tail listAllRoles).
                                    nextRoleSet keysAndValuesDo: [:x :y| y do: [:z | (roleSet at:
x ifAbsentPut: [OrderedCollection new]) add: z] ]
                                ].
    ^roleSet!

root
    "The root of an Object is always the Object itself."


    ^self! !

!ObjectWithRoles methodsFor: 'roleification'!

addRole:role
    "Adds a new Role to the Receiver-Object."

    | collection |
    collection := (role linkSet allLinksForRole: (DSTlink roleificationTailRole)).
    collection isEmpty ifFalse: [^self error: 'Role is already bound to another Object.']
                                        ifTrue: [(self roleIsAllowed: role)
                                                                    ifTrue: [self linkSet
createLink: (DSTlink roleification) tail: (role linkSet)]
                                                                    ifFalse: [^self error: 'Role is
currently not allowed to be bound to this Object.' ] ]!

as: roleTitle
    "Returns the current existents of the Receiver-Object in a particular Role."

    ^self listAllRoles at: roleTitle!

existsAs: roleTitle
    "Returns a Boolean whether the Receiver-Object is currently playing this Role."

    ^self listAllRoles keys includes: roleTitle! !

!ObjectWithRoles methodsFor: 'private'!

roleIsAllowed: aRole
    "Decides wether a Role is currently allowed to be bound to the Receiver-Object or
not."
    "--- If not redefined binding is always allowed ---"

    ^true.!

roleSelect: aCollection withMessage: aMessage
    "Decides which of the Roles in aCollection is selected to perform aMessage."
    "--- If not redefined there is no selection ---"

    ^nil! !

!ObjectWithRoles methodsFor: 'inheritance'!

doesNotUnderstand: aMessage
    "Delegates an unkown Message to a played Role understanding the Message."
```

```
    |collection instance|
    collection := OrderedCollection new.
    self listAllRoles values do: [:x| x do: [ :y | (y respondsIntrinsicTo: (aMessage selector))
                                                    ifTrue: [collection add: y ]]].
    collection isEmpty  ifTrue: [instance:= nil.]
        ifFalse: [ instance := (self roleSelect: collection withMessage: aMessage)].
    instance isNil ifTrue:
                [       | selectorString |
                        selectorString :=
                        Object errorSignal
                                handle: [:ex | ex returnWith: '** unprintable selector **']
                                do: [aMessage selector printString].
                        Object messageNotUnderstoodSignal
                                raiseRequestWith: aMessage
                                errorString: 'Message not understood: ' , selectorString.
                ^self perform: aMessage selector withArguments: aMessage arguments]
                        ifFalse:
                [^instance perform: aMessage selector withArguments: aMessage
arguments]!

respondsIntrinsicTo: aSymbol
    "Answer a Boolean as to whether the method dictionary of the receiver's class
    contains aSymbol as a message selector."

    ^self class canUnderstand: aSymbol! !

!ObjectWithRoles methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81237317-40e5-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Roleification::ObjectWithRolesInterface'! !

Inspector subclass: #SubjectInspector
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!SubjectInspector methodsFor: 'field list'!

fieldList

    | max |
    max := object basicSize.
    ^((Array with: 'self'), (Array with: 'root'), (Array with: 'roles')
        , object class allInstVarNames , ((max <= 40
            ifTrue: [1 to: max]
            ifFalse: [(1 to: 30)
                        , (max - 10 to: max)])
            collect: [:i | i printString]))!

fieldValue
```

```
    "Answer the value of the currently selected item."

    field = 'self' ifTrue: [^object].
    field = 'root' ifTrue: [^object root].
    field = 'roles' ifTrue: [^object listAllRoles].
    field first isDigit
        ifTrue: [^object basicAt: self fieldIndex]
        ifFalse: [^object instVarAt: self fieldIndex]! !

ObjectWithRoles subclass: #Brett
    instanceVariableNames: 'holzart laenge '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Brett methodsFor: 'private'!

roleSelect: aCollection withMessage: aMessage

    |temp instance max|
    temp := Dictionary new.
    max := 0.
    instance := aCollection first.

    ((aMessage selector) = #wert) ifTrue: [
                aCollection do: [ :x | temp at: x put: (x perform: (aMessage selector)
withArguments: (aMessage arguments))].
                temp keysAndValuesDo: [ :x :y | (y>max) ifTrue: [instance := x. max := y]]
                        ].

    ^instance! !

!Brett methodsFor: 'accessing'!

holzart
    ^holzart!

holzart: aString
    ^holzart := aString!

laenge
    ^laenge!

laenge: anInteger
    ^laenge := anInteger! !

!Brett methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'80d1fd87-a602-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::BrettInterface'! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
```

```
Brett class
    instanceVariableNames: ''!


!Brett class methodsFor: 'testing'!

test
    B1 := Brett new.
    B1 asRemotable holzart: 'Mahagony'.
    B1 asRemotable laenge: 5.
    T1 := Tischplatte new.
    D1 := Dachelement new.
    V1 := Vermoegensgegenstand new.
    V2 := Vermoegensgegenstand new.
    V3 := Vermoegensgegenstand new.
    V1 asRemotable wert: 10.
    V2 asRemotable wert: 50.
    V3 asRemotable wert: 30.
    B1 asRemotable addRole: T1.
    B1 asRemotable addRole: D1.
    B1 asRemotable addRole: V1.
    T1 asRemotable addRole: V2.
    D1 asRemotable addRole: V3.
    B1 asRemotable haerte.
    B1 asRemotable farbe.
    V1 wert.
    V2 wert.
    V3 wert.
    D1 wert.
    T1 wert.
    B1 wert.
    V1 holzart: 'Fichte'.
    V3 haerte.
    V2 laenge: 12.
    B1 wert.! !

ObjectWithRoles subclass: #TestObject
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!TestObject methodsFor: 'testing'!

objectMethod

    ^'Hello world. This is the Object speaking.'! !

!TestObject methodsFor: 'private'!

roleIsAllowed: aRole

    (aRole roleTitle = 'TestRole1') ifTrue: [(self existsAs: 'TestRole') ifFalse: [^false]].
    ^true!

roleSelect: aCollection withMessage: aMessage
```

```
    ^aCollection first! !

!TestObject methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'8123776b-945d-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Tests::TestObjectInterface'! !

ObjectWithRoles subclass: #Person
    instanceVariableNames: 'name telefonnummer svznr '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Person methodsFor: 'accessing'!

name

    ^name!

name: aName

    name := aName.
    ^name!

svznr

    ^svznr!

telefonnummer

    ^telefonnummer!

telefonnummer: aNumber

    telefonnummer := aNumber.
    ^telefonnummer! !

!Person methodsFor: 'initialize'!

initialize: aSVZNR

    svznr := aSVZNR! !

!Person methodsFor: 'private'!

roleIsAllowed: aRole

    Angestellter subclasses do: [ :x| ((self root) existsAs: (x printString)) ifTrue: [^false] ].
    ^true.!

roleSelect: aCollection withMessage: aMessage
```

```
    ^aCollection first! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Person class
    instanceVariableNames: "!


!Person class methodsFor: 'instance creation'!

new
    self shouldNotImplement!

new: aSVZNR

    ^super new initialize: aSVZNR.! !

!Person class methodsFor: 'testing'!

test
    Chris := Mann new: 1744.
    Chris asRemotable name: 'Christian'.
    Chris asRemotable telefonnummer: 7124412.
    S1 := SachbearbeiterIn new.
    S1 asRemotable telefonnummer: 3566.
    S1 asRemotable persnr: 78719.
    Chris asRemotable addRole: S1.
    (Chris asRemotable as: 'SachbearbeiterIn') first telefonnummer.
    Chris asRemotable telefonnummer.
    M1 := ManagerIn new.
    Chris asRemotable addRole: M1.
    M1 asRemotable telefonnummer: 06763566.
    M1 asRemotable persnr: 78719.
    S1 asRemotable abandon.
    Chris addRole: M1.
    (Chris asRemotable as: 'ManagerIn') first telefonnummer.
    Chris asRemotable telefonnummer.
    M1 asRemotable svznr.
    Chris asRemotable persnr.! !

Person subclass: #Frau
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Roleification'!


!Frau methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81322fb3-e1fc-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::FrauInterface'! !

Person subclass: #Mann
    instanceVariableNames: "
```

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Mann methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81322fb3-e1fb-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::MannInterface'! !

ObjectWithRoles subclass: #Role
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Role methodsFor: 'inheritance'!

doesNotUnderstand: aMessage
    "Performs not known Messages to the Ancestor of the Receiver-Role."

    ^self roleOf perform: (aMessage selector) withArguments: (aMessage arguments)! !

!Role methodsFor: 'roleification'!

abandon
    "Removes the link between the Receiver-Role and its Ancestor."

    |linkId|
    linkId := (self ancestorId).
    ((self linkSet) getLink: linkId) dropLink.! !

!Role methodsFor: 'accessing'!

ancestorId
    "Returns the ID of the link to the Ancestor of the Receiver-Role."

    |collection|
    collection := (self linkSet allLinksForRole: (DSTlink roleificationTailRole)).
    collection isEmpty ifTrue: [^self error: 'This is not a Role of a valid Object.']
                               ifFalse: [^collection first key]!

roleOf
    "Returns the Ancestor of the Receiver-Role."

    |linkId link|
    linkId := self ancestorId.
    link := self linkSet getLink: linkId.
    ^link headObject.!

roleTitle
    "Returns the Title of the Receiver-Role."
```

```
    "--- If not redefined the class-name is considered the Role-Title ---"

    ^self class printString!

root
    "The root of the Receiver-Role is the Root of its Ancestor."

    ^self roleOf root.! !

!Role methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81237317-40e6-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Roleification::RoleInterface'! !

Role subclass: #Angestellter
    instanceVariableNames: 'persnr telefonnummer '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Angestellter methodsFor: 'accessing'!

persnr
    ^persnr!

persnr: aNumber

    ^persnr := aNumber!

telefonnummer

    ^telefonnummer!

telefonnummer: aNumber

    ^telefonnummer := aNumber! !

!Angestellter methodsFor: 'private'!

roleIsAllowed: aRole

    ^false.! !

Role subclass: #Tischplatte
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Tischplatte methodsFor: 'accessing'!
```

```
farbe
    ^ ((self root holzart)='Mahagony') ifTrue: [ ^'dunkelbraun']
                                                    ifFalse: [^'hellbraun']!

wert
    ^ ((self root holzart)='Mahagony') ifTrue: [ ^22]
                                                    ifFalse: [^17]! !

!Tischplatte methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'80d1fd87-a604-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::TischplatteInterface'! !

Angestellter subclass: #ManagerIn
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!ManagerIn methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81321acb-55a4-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::ManagerInInterface'! !

Role subclass: #TestRole
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!TestRole methodsFor: 'testing'!

roleMethod

    ^'Hello world. This is the Role speaking.'! !

!TestRole methodsFor: 'private'!

roleIsAllowed: aRole

    (aRole roleTitle = 'TestRole1') ifTrue: [(self root existsAs: 'TestRole') ifFalse: [^false]].
    ^true! !

!TestRole methodsFor: 'repository'!

abstractClassId
```

```
    "return the abstract class Id of the receiver"
    ^'8123776b-945e-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Tests::TestRoleInterface'! !

Role subclass: #Vermoegensgegenstand
    instanceVariableNames: 'wert '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Vermoegensgegenstand methodsFor: 'accessing'!

wert
    ^wert!

wert: anInteger

    ^wert := anInteger! !

!Vermoegensgegenstand methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'80d1fd87-a605-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::VermoegensgegenstandInterface'! !

Angestellter subclass: #SachbearbeiterIn
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!SachbearbeiterIn methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'81321acb-55a6-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::SachbearbeiterInInterface'! !

Role subclass: #Dachelement
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!Dachelement methodsFor: 'accessing'!
```

```
haerte
    ^ ((self root holzart)='Fichte') ifTrue: [ ^'hart']
                                                    ifFalse: [^'weich']!

wert
    ^ (self root laenge)*5! !

!Dachelement methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'80d1fd87-a603-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Samples::DachelementInterface'! !

Role subclass: #TestRole1
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roleification'!


!TestRole1 methodsFor: 'testing'!

roleMethod

    ^'Hello world. This is the other Role speaking.'!

roleMethod1

    ^'Hello world. This is the right message.'! !

!TestRole1 methodsFor: 'private'!

roleIsAllowed: aRole

    (self root existsAs: aRole roleTitle) ifTrue: [^false].
    ^true! !

!TestRole1 methodsFor: 'repository'!

abstractClassId
    "return the abstract class Id of the receiver"
    ^'8123776b-945c-0000-027f-000001000000' asUUID!

CORBAName
    "return the name of my CORBA interface in the repository"
    ^#'::Tests::TestRole1Interface'! !
```

7.5 DSTRepository-custom IFs.st

```
!DSTRepository methodsFor: 'custom IFs'!
```

```
// Roleification
// This module defines the interfaces which form the Roleification-Service.
//
module Roleification  {

    //
    interface ObjectWithRolesInterface : ApplicationSem {

        // Adds a new Role to the Receiver-Object.
        SmalltalkObject addRole (in SmalltalkObject role);

        // Returns the current existents of the Receiver-Object in a particular Role.
        OrderedCollection as (in string roleTitle);

        // Returns a Boolean whether the Receiver-Object is currently playing this Role.
        boolean existsAs (in string roleTitle);

        // Returns a Dictonary with all Roles currently played by the Receiver-Object.
        Dictionary listAllRoles ();

        // Adds the currently played Roles to the signature of the Receiver-Object.
        SmalltalkObject printOn (in SmalltalkObject aStream);

        // Answer a Boolean as to whether the method dictionary of the receiver's class
        // contains aSymbol as a message selector.
        boolean respondsIntrinsicTo (in Symbol aSymbol);

        // Decides wether a Role is currently allowed to be bound to the Receiver-Object or
not.
        boolean roleIsAllowed (in SmalltalkObject aRole);

        // Returns the root of the Subject.
        SmalltalkObject root ();
    };


    //
    interface RoleInterface : ObjectWithRolesInterface {

        // Removes the link between the Receiver-Role and its Ancestor.
        SmalltalkObject abandon ();

        // Returns the Ancestor of the Receiver-Role.
        SmalltalkObject roleOf ();

        // Returns the Title of the Receiver-Role.
        string roleTitle ();

    };
};!

// Samples
// This module defines the interfaces for the Sample-Classes.
//
module Samples  {

    //
```

```
    interface AngestellterInterface : RoleInterface {

        attribute long persnr;

        attribute long telefonnummer;
    };

    interface ManagerInInterface : AngestellterInterface, PersonInterface {};

    interface SachbearbeiterInInterface : AngestellterInterface, PersonInterface {};

    //
    interface PersonInterface : ObjectWithRolesInterface {

        attribute string name;

        readonly attribute long svznr;

        attribute long telefonnummer;
    };

    interface MannInterface : AngestellterInterface, PersonInterface  {};

    interface FrauInterface : AngestellterInterface, PersonInterface  {};


    //
    interface Beispiel2ObjInterface : ObjectWithRolesInterface {

        attribute string holzart;

        attribute short laenge;
    };


    //
    interface Beispiel2RoleInterface : RoleInterface {

        readonly attribute string farbe;

        readonly attribute string haerte;

        readonly attribute short wert;
    };

    interface BrettInterface : Beispiel2RoleInterface, Beispiel2ObjInterface {};

    interface DachelementInterface : Beispiel2RoleInterface, Beispiel2ObjInterface {};

    interface TischplatteInterface : Beispiel2RoleInterface, Beispiel2ObjInterface {};

    interface VermoegensgegenstandInterface : Beispiel2RoleInterface,
Beispiel2ObjInterface {

        attribute short wert;
    };

};!
```

```
// Tests
// This module defines the interfaces for the Test-Classes.
//
module Tests  {

    //
    interface TestObjectInterface : ObjectWithRolesInterface {

        string objectMethod ();
    };

    //
    interface TestRoleInterface : RoleInterface {

        string roleMethod ();
    };

    interface TestRole1Interface : RoleInterface {

        string roleMethod ();

        string roleMethod1 ();
    };
};! !
```

8 Literatur

[Bachman77]
„*The Role Concept in Data Models*“, **Charles W. Bachman, Manilal Daya**, Proceedings of 3rd International Conference on Very Large Databases (VLDB), 1977

[Bergamini93]
„*An Object Data Model with Roles*“, **Albano, R. Bergamini, G. Ghelli, R. Orsini**, Proceedings of the International Conference on Very Large Databases (VLDB), 1993

[Booch94]
„*Object Oriented Analysis and Design with Applications*“, **Grady Booch**, Benjamin/Cummings Publishing Company Inc., 1994 Second Edition

[Brockhaus]
LexiROM © 1995 Microsoft Corporation und Bibliographisches Institut & F.A. Brockhaus AG

[Chen76]
„*The Entity Relatioship Model – Toward a Unified View of Data*”, **P. P-S. Chen**, ACM Transactions on Database Systems (TODS), Vol. 1, No. 1, 1976

[CORBA]
http://www.omg.org, CORBA 2.0 – CORBAservices Volumes 1 & 2

[DST]
http://www.parcplace.com, Visual Works 2.5.1

[Gamma95]
„*Design Patterns – Elements of Reusable OO-Software*“, **E. Gamma, R. Helm, R. Johnson, J. Vlissides**, Addison-Wesley, 1995

[Goldberg89]
„*Smalltalk 80 - The Language and it´s Implementation*“, **A. Goldberg, Robson**, Addison-Wesley, 1989

[Harrison93]
„*Subject-oriented Programming (a Critique of Pure Objects)*“, **William Harrison, Harold Ossher**, Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), 1993

[IDLReference96]
Visual Works - Distributed Smalltalk, *IDL Programmer´s Reference*, ParcPlace-Digitalk Inc. 1996

[Info&Ges95]
„Informatik und Gesellschaft", **Jürgen Friedrich, Thomas Herrmann, Max Peschek, Arno Rolf**, Spektrum Akademischer Verlag, 1995

[Kristensen95]
„Roles: Conceptual Abstraction Theory & Practical Language Issues", **Bent Bruun Kristensen, Kasper Osterbye**, Technical Report, Institute for Electronic Systems, Aalborg University, 1995

[Lieberman86]
„Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", **Henry Lieberman**, Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), 1986

[Lieberman88]
„Of Types and Prototypes: The Treaty of Orlando", **Henry Lieberman, Lynn Andrea Stein, D. Unger**, SIGMOD Not. (ACM), 1988

[Milling95]
„Organisationales Lernen und seine Unterstützung durch Managementsimulatoren", **Peter Milling**, Zeitung für Betriebswirtschaft Ergänzungsheft 3, Gabler Verlag, 1995

[Odell92]
„Object Oriented Analysis & Design", **James Martin, James J. Odell**, Prentice Hall, 1992

[Papazoglou91]
„Roles: A Methodology for Representing Multifaceted Objects", **M. P. Papazoglou**, Proceedings of the International Conference on Database and Expert Systems Applications, 1991

[Pernici90]
„Objects with Roles", **Barbara Pernici**, Proceedings of the ACM Conference on Office Information Systems, 1990

[Programmer´sReference96]
Visual Works - Distributed Smalltalk, *Programmer´s Reference*, ParcPlace-Digitalk Inc. 1996

🕮 [Richardson91]
„*Aspects: Extending Objects to Support Multiple, Independent Roles*“, **Joel Richardson, Peter Schwarz**, Proceedings of the International Conference on Management of Data (SIGMOD), 1991

🕮 [Rumbaugh91]
„*Object-Oriented Modelling and Design*“, **J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen**, Prentice Hall, 1991

🕮 [Schrefl96]
„*Extending Object-Oriented Systems with Roles*“, **Georg Gottlob, Michael Schrefl, Brigitte Roeck**, ACM Transactions on Information Systems (TOIS), Vol. 14, No. 3, 1996

🕮 [Sciore89]
„*Object Specialization*“, **Edward Sciore**, ACM Transactions on Information Systems (TOIS), Vol. 7, No. 2, 1989

🕮 [Shilling89]
„*Three Steps to View: Extending the Object-Oriented Paradigm*“, **J.J. Shilling, P.F. Sweeney**, Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), 1989

🕮 [Shlaer88]
„*Object-Oriented Systems Analysis – Modeling the World in Data*“, **S. Shlaer, S. J. Mellor**, Yourdon Press Computing Series, 1988

🕮 [Stein87]
„*Delegation is Inheritance*“, **Lynn Andrea Stein**, Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), 1987

🕮 [Stein90]
„*Clovers: The Dynamic Behavior of Types and Instances*“, **Lynn Andrea Stein, Stanley B. Zdonik**, Technical Report, Brown University, Providence, 1990

🕮 [Su91]
„*Dynamic Constraints and Object Migration*“, **Jianwen Su**, Proceedings of the 17th International Conference on Very Large Databases (VLDB), 1991

🕮 [UML]
http://www.rational.com, UML Spezification 1.1 Glossary

🕮 [User´sGuide96]
Visual Works - Distributed Smalltalk, *User´s Guide*, ParcPlace-Digitalk Inc. 1996

🕮 [Wieringa91]
„The Identification of Objects and Roles - Object Identifiers Revisited", **Roel Wieringa, Wiebren de Jonge**, Technical Report, Faculty of Mathematics and Computer Science, Vrije Universiteit De Boelelaan, Amsterdam, 1991

🕮 [Wirfs-Brock90]
„Designing Object-Oriented Software", **Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener**, Prentice Hall, Englewood Cliffs, New Jersey, 1990