Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik

Diplomarbeit

Evaluierung modularer Softwareentwicklung mit "Object Teams" am Beispiel eines Projektmanagementsystems

Matthias Veit
MatrNr.:172221
<mveit@cs.tu-berlin.de>

Berlin, den 28. Dezember 2002

Betreut von Dr. S. Herrmann
am Lehrstuhl von Prof. Dr. S. Jähnichen

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Die Anforderungen an Softwarelösungen im Allgemeinen werden immer umfangreicher, spezifischer und komplexer. Die benötigte Zeit für eine Realisierung soll dabei eher sinken und nicht anwachsen. Auch eine bestehende Softwarelösung soll bestimmte Anforderungen erfüllen. Sie soll einfach zu warten sein, leicht an veränderte Anforderungen angepasst oder um bestimmte Funktionalitäten erweitert werden können.

Software entsteht durch die Hand von Menschen. Um den wachsenden Anforderungen gerecht zu werden, bedarf es erweiterter Techniken für die Softwareentwicklung, die den Entwickler bei seiner Tätigkeit unterstützen und auf ihn zugeschnitten sind. Dies schließt den gesamten Prozess der Softwareentwicklung von der Analyse bis zur Implementierung ein. Das objektorientierte Paradigma hat hier Großes beigetragen und stellt heute den Standard dar. Hier wird der Strukturierungsmechanismus *Klasse* eingeführt, der eine Modellierung von Entitäten einer Domäne erlaubt. Die Technik der Vererbung ermöglicht verschiedene Abstraktionsebenen, wobei eine Wiederverwendung von allgemeinen Klassen erzielt werden kann.

Das Paradigma der aspektorientierten Programmierung führt eine zweite Form der Strukturierung ein. Sie erlaubt die Kapselung von Verhalten in einem *Aspekt*. Auch Klassen kapseln Verhalten, verknüpfen dieses aber an eine bestimmte Struktur: das in einer Klasse gekapselte Verhalten kann nur über Vererbung oder Delegation einer anderen Klasse verfügbar gemacht werden. Ein Aspekt ist dagegen eine Teilfunktionalität einer Klasse. Der Aspekt kann von einer Klasse genutzt werden, ohne damit eine strukturelle Aussage zu treffen. Die Definition eines Aspektes sollte viel verständlicher und lesbarer sein, da alle zusammenhängenden Belange an einer Stelle aufgeschrieben werden können, die eine spezifische Funktionalität betreffen. Man muß nur diese eine Funktionalität bedenken und nicht das Zusammenspiel dieser in einer Klasse. Ein Aspekt sollte viel wartbarer sein, da nur der Aspekt gewartet werden muss und nicht alle Klassen die diesen Aspekt benutzen. Außerdem sollte ein Aspekt viel wiederverwendbarer sein als eine Klasse, da nur das gekapselte Verhalten unabhängig von der jeweiligen Domäne definiert wurde.

Das Paradigma der aspektorientierten Programmierung ist noch sehr jung und so existiert noch kein Standard, der definiert, was ein Aspekt ist, wie dieser aufgeschrieben werden kann und wie er an die benutzende Klasse gebunden wird. Im

Rahmen dieser Diplomarbeit soll eine neue Form der aspektorientierten Programmierung evaluiert werden, die Object Teams [Herrmann 2002b] heißt. Object Teams führt die Metapher *Rolle* für einen Aspekt ein. Eine Rolle kann separat definiert und an eine Domänenklasse gebunden werden. Eine Klasse verhält sich auf diese Weise im Sinne des definierten Rollenverhaltens. Im Laufe dieser Diplomarbeit soll evaluiert werden, wie intuitiv solch eine Rollendefinition und -bindung ist, welche Stärken und Schwächen das Modell aufweist, bzw. welche Möglichkeiten damit existieren. Der Fokus soll hierbei auf der *Modularität* der Aspekte liegen. Folgende Fragen werden dabei diskutiert:

- Wie abgeschlossen ist das gekapselte Verhalten?
- Welche Möglichkeiten der Adaption ist nötig, bzw. möglich?
- Kann eine Rolle an spezifische Bedürfnisse angepasst werden?
- Wie wiederverwendbar sind Rollen?

Um diese Fragen an einem praktischen Beispiel erörtern zu können, sollte im Verlauf dieser Arbeit eine Software zur organisatorischen Unterstützung von Projektabläufen (Projektmanagement) entwickelt werden. Die Anforderungen an diese Software wurden von der Firma ITSO [Glöckner und Storl 2001] definiert. Sie sind ausreichend komplex und umfangreich, so dass hier durchaus von einem *'real world scenario'* gesprochen werden kann. Aus der Modellierung der Domäne mussten die Aspekte erst einmal identifiziert werden. Für die Realisierung dieser Software wurden drei Aspekte gefunden. Diese Aspekte wurden separat als Rollen modelliert, implementiert und wieder an die Domänenklassen gebunden. Die gefundenen Aspekte stehen stellvertretend für allgemeine Lösungsstrategien und sollen in Bezug auf ihre Modularität und Anwendbarkeit ausführlich besprochen werden.

Struktur der Arbeit

Diese Arbeit ist unterteilt in drei große Kapitel:
Kapitel 2.1 gibt eine Einführung in das Paradigma von Object Teams. Hier werden die generellen Ideen, das Vorgehen und die Arbeitsweise von Object Teams dargestellt. Kapitel 2.2 führt eine UML Erweiterung ein, mit der es möglich ist, Aspekte zu modellieren. Von dieser Modellierungstechnik wird im Realisierungsteil intensiv Gebrauch gemacht. Beide Kapitel stellen die theoretische Grundlage für die weitere Arbeit dar.

Kapitel 3 stellt eine Realisierung von Object Teams für die Programmiersprache Ruby vor. Um mit diesen Eigenschaften vertraut zu werden, wird Ruby in Kapitel 3.1 kurz vorgestellt. Kapitel 3.2 beschreibt die Implementierung von Ruby Object Teams

und zeigt, wie die einzelnen Funktionalitäten vom Entwickler genutzt werden können. Um diese Spracherweiterung syntaktisch zu demonstrieren, wird in Kapitel 3.3 ein Beispiel demonstriert.

In Kapitel 4 wird die Realisierung einer Projektmanagementsoftware mit Hilfe von Object Teams beschrieben. Die Anforderungen an solch eine Software werden in Kapitel 4.1 erläutert. In dem Domänenmodell wurden drei Aspekte identifiziert. Diese drei Aspekte werden in den folgenden Kapiteln näher erläutert: Die Funktionalität einer graphischen Benutzeroberfläche wird in Kapitel 4.2, die dauerhafte Speicherung von Objekten in Kapitel 4.3 und die Funktionalität der Zugriffskontrolle auf Basis von Privilegien in Kapitel 4.4 beschrieben.

2 Object Teams - eine Einführung

2.1 Die Idee von ObjectTeams

Einer der größten Gewinne bei Anwendung des objektorientierten Paradigmas ist die Möglichkeit, Entitäten einer Domäne zu modellieren und strukturiert zu definieren. Die modellierten Entitäten ermöglichen einen relativ natürlichen Umgang und fördern auf diese Weise Einfachheit und Übersichtlichkeit. Die starke Gewichtung auf Struktur birgt aber auch Probleme. Betrachtet man einen Problemkreis aus funktionaler Sicht, kann man möglicherweise beobachten, dass sich eine bestimmte Funktionalität auf verschiedene Entitäten verteilt (*'scattering'*), oder dass sich eine bestimmte Funktionalität in verschiedenen Entitäten wiederholt. Die Entscheidung für eine bestimmte Struktur erfordert möglicherweise die Implementierung verschiedener Funktionalitäten in einer Methode (*'tangling'*). Haben solche Funktionalitäten, strukturell betrachtet, keinen Zusammenhang, so sind sie auch nicht mit den Werkzeugen der Objektorientierung modellierbar bzw. kapselbar. Man bezeichnet diese Funktionalitäten auch als *'Crosscutting Concerns'* [Kiczales u. a. 2001; Herrmann 2002b], also Verantwortlichkeiten, die sich in einer bestimmten Art und Weise gestreut wiederfinden.

Das Paradigma der *Aspektorientierten Programmierung* [AOSD 2002] setzt an genau diesem Punkt der streuenden Verantwortlichkeiten an. Auch hier wird ein Strukturierungsmechanismus eingeführt: der *Aspekt*. Ein Aspekt kapselt Funktionalitäten die im Domänenmodell nicht objektorientiert erfasst werden können. Schafft man es, ein objektorientiertes Domänenmodell unabhängig von bestimmen Aspekten zu entwerfen, führt dies zu einem einfacheren, saubereren und besseren Design, das ja Grundlage für eine gute Implementierung ist. Das Problem der Vermischung verschiedener Zuständigkeiten (*'tangling code'*) an einzelnen Punkten kann man mit solchen Techniken bewältigen. Die Funktionalitäten, die in einem Aspekt gekapselt werden, unterliegen keiner Beschränkung und können durchaus komplex sein. Verteilt sich eine bestimmte Funktionalität auf verschiedene Klassen, so müssen die einzelnen Teilfunktionalitäten als Aspekte modelliert werden. Man spricht dann von *kollaborierenden Aspekten*. Es bedarf erweiterter Konzepte, um solche Kollaborationen zu modellieren bzw. zu implementieren.

Den Zusammenhang zwischen Kollaborationen, einer Menge zusammengehöriger Aspekte, und Klassen sieht man in Abbildung 2.1. Das Beobachtermuster [Gam-

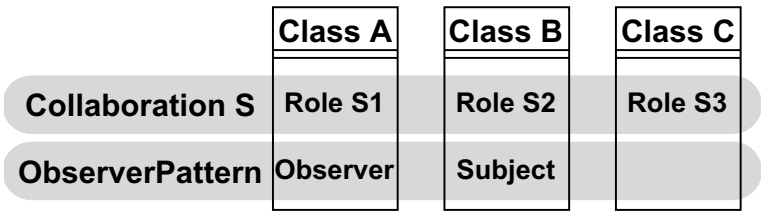

Abbildung 2.1: Zusammenhang zwischen Klassen und Kollaborationen

ma u. a. 1996, *'Observerpattern'*] beispielsweise besteht aus zwei Teilen: dem Teil des beobachteten Subjektes und dem Teil des Beobachters selbst. Die Interaktion zwischen Beobachter und Subjekt ist spezifisch für das Beobachtermuster, aber nicht spezifisch für Klasse A oder Klasse B. Spricht man von Beobachtermuster meint man immer Beobachter und Subjekt auch wenn sich das Muster immer in mindestens zwei Klassen wiederfindet. Das Muster kann in unterschiedlichsten Klassen auftauchen und ist wiederverwendbar. Über die Wiederverwendbarkeit von Klasse A, B oder C kann an dieser Stelle keine Aussage gemacht werden. Klassen bestehen unter Umständen aus einer Menge von einzelnen Aspekten unterschiedlicher Kollaborationen, so dass man hier von einer Orthogonalität zwischen Klassen und Kollaborationen sprechen kann.

Die Konzepte der Objektorientierung erlauben es Klassen und Schnittstellen zu modellieren und zu implementieren. Im Verlauf dieser Arbeit soll herausgestellt werden, dass es in vielen Situation wünschenswert und sinnvoll wäre, zusätzlich zu den bekannten objektorientierten Strukturierungsmechanismen auch Aspekte und Kollaborationen modellieren und implementieren zu können. Kollaborationen als eine erweiterte Form des Baukastenprinzips: eine adaptierbare vorgefertigte Funktionalität, die den jeweiligen spezifischen Domänenklassen beigefügt werden kann. Diese Funktionalität hat dabei keinen Einfluss auf die jeweilige Struktur – diese wird ja von der jeweiligen Domäne festgelegt. Auf den folgenden Seiten sollen die Lösungsstrategien vorgestellt werden, die das Object Teams Paradigma für den Umgang mit den so genannten *'Crosscutting Concerns'* bereit hält.

2.1.1 Die Rollendefinition

Aspekte einer Kollaboration kapseln Verhalten. Dieses Verhalten ist stereotyp und spezifisch für die jeweilige Kollaboration, nicht aber spezifisch für die jeweilige Klasse, die dieses Verhalten implementiert. Man nennt das Verhalten einer Entität, die sich stereotyp in einem bestimmten *Kontext* verhält auch Rollenverhalten. Der einzelne Aspekt einer Kollaboration wird in Object Teams aus diesem Grund *Rolle* genannt.

Die Rollenfunktionalität wird als Klasse definiert. Da sich Object Teams in eine objektorientierte Struktur einfügt, erscheint der Umgang mit dem bekannten Konstrukt Klasse sehr natürlich, wenn auch erstaunlich. Der Aspekt, der sich gerade als eine Teilfunktionalität einer Domänenklasse kristallisiert hat, wird nun selbst als Klasse implementiert. Der Vorteil dieser Herangehensweise ist die getrennte Modellierung, Definition und Wartung von getrennten Funktionalitäten. Syntaktisch unterscheidet sich die Definition einer Rolle kaum von der einer „normalen" Klasse, dafür gibt es zahlreiche semantische Unterschiede:

2.1.1.1 Deklarative Vollständigkeit

Eine Rollenklasse kapselt ein bestimmtes Verhalten. Da die Verhaltensbeschreibung losgelöst ist von der jeweils ausführenden Entität, ergeben sich eine Reihe von offenen Funktionalitäten, die nicht definiert, sondern nur deklariert werden können (*'open spots'*). Dies hat wenig mit einer nicht abgeschlossenen Kapselung zu tun, als vielmehr mit einer kompletten und abstrakten Beschreibung eines Verhaltens – einer deklarativ vollständigen Beschreibung. So ist z.B. in dem oben genannten Beobachtermuster definiert, dass der Beobachter benachrichtigt wird, wenn sich das Subjekt verändert. Es ist aber nicht definiert, was „Benachrichtigung" bedeutet! Die Klasse, die den Beobachter realisiert, muss durch Implementierung einer abstrakten Methode (`update`) definieren, was sie unter Benachrichtigung versteht. Auch in einer Rollenklasse werden alle offenen Funktionalitäten als abstrakte Methoden deklariert und nicht definiert. Eine Rolle ist erst dann nutzbar, wenn alle abstrakten Methoden definiert sind.

2.1.1.2 Rollen sind nicht autark

Eine Rollenklasse ist für sich alleine nicht wirklich sinnvoll. Sie kapselt ein bestimmtes Verhalten, welches einer Domänenklassen beigefügt werden kann. Das Verhalten der Rolle ist ja nur ein funktionaler Ausschnitt einer Domänenklasse. Die selbe Rollenklasse kann aber in ganz verschiedenen Domänenklassen benutzt werden. Rollen existieren immer nur im Zusammenhang mit Objekten der Domäne, denen das spezifische Verhalten zu eigen ist.

2.1.1.3 Explizite Methodenspezialisierung

Object Teams bietet mit der Definition einer Rollenklasse die Möglichkeit Methoden vorzusehen, welche bestimmte Methoden der Domänenklasse spezialisieren. Man kennt diesen Mechanismus von Klassen in einer Vererbungsrelation. Methoden einer erbenden Klasse, die den gleichen Namen und die gleiche Signatur wie die

der Basisklasse tragen, ersetzen die ursprüngliche Methode (*'overriding'*). Die überschriebene Methode ist von der überschreibenden Methode noch mit einem Schlüsselwort (in Java beispielsweise `super()`) zugreifbar. Die überschreibende Methode kann also die Implementierung der Superklasse weiter benutzen und spezialisieren. Ganz in diesem Sinne kann auch eine Rolle solche Methoden deklarieren. Da die Rolle aus Gründen der Wiederverwendbarkeit komplett unabhängig von einer Domänenklasse implementiert wird, ist die Gleichheit von Methodennamen und Signatur nicht entscheidend um bestimmte Methoden der Domänenklasse zu spezialisieren. Dies muss explizit deklariert werden. Eine Rollenmethode, welche so deklariert wurde, darf innerhalb des Methodenkörpers dass Schlüsselwort `base()` benutzen, um die ursprüngliche Funktionalität zu benutzen. Solch deklarierte Methoden werden im folgenden *Replacements* genannt.

2.1.2 Der Teambegriff

Das Object Teams Paradigma führt ein neues Modulkonzept mit dem Namen *Team* ein. Ein Team ist ein Paket, in dem Rollen zu einer Kollaboration zusammengefasst werden können. Das Konstrukt Paket ist bekannt aus der *UML* [OMG 2002]. Ein Team kann als solches Paket angesehen werden, hat aber weitergehende semantische Eigenschaften. Erst einmal ist ein Team ein abgeschlossenes Konstrukt. Die Pakete aus der UML spiegeln sich meist als einfache Namensraumabgrenzung wieder, welche beliebig groß sein können. Die Definition der einzelnen Rollenklassen erfolgt immer innerhalb eines speziellen Teams. Die Gesamtheit der Rollenklassen eines Teams bilden dessen Rahmen. Rollen kapseln ein bestimmtes spezifisches Verhalten, ein Team dagegen kapselt das Verhalten und die Interaktion einzelner Rollen untereinander. So könnte es z.B. das Team Beobachtermuster geben, welches zwei Rollen definiert: Beobachter und Subjekt.

Ein Team kann spezialisiert werden. Es ist also möglich, eine Vererbungsrelation zu realisieren. Erbt ein Team von einem anderen Team, werden automatisch alle Rollen, die im Basisteam verfügbar sind, in das zu erbende Team kopiert, so dass alle Rollen des Basisteams im erbenden Team enthalten sind. Diese Art der Vererbung heißt *implizite Vererbung* (*'implicit inheritance'*). Die einzelnen Rollen können nun im spezialisierten Team verfeinert werden. Wird ein Team ohne Vererbungsrelation deklariert, so erbt dieses Team implizit von einem Basisteam namens `Team`. Hier wird dieselbe Technik verwendet, die man aus Programmiersprachen mit einer einheitlichen Wurzelklasse kennt. Wird eine Klasse ohne Vererbungsrelation deklariert, so wird implizit eine Vererbungsrelation zum Basistyp (z.B. `Object` in Java, CORBA und Ruby) eingefügt. Ein Team muss als abstrakt deklariert werden, solange es eine oder mehrere Rollen im Team gibt, die abstrakt sind[1]. Sind alle abstrakten Rollen-

[1] Es kann eine effektive Rollenklasse geben, die eine abstrakte Rollenklasse im selben Team verfeinert. In diesem Fall ist das Team natürlich nicht abstrakt. Für die Abstraktheit ausschlaggebend sind also

methoden aufgelöst, wird das Team *effektiv*. Ein effektives Team kann instantiiert werden.

Jedes instantiierte Team definiert einen Kontext, der aktiviert bzw. deaktiviert werden kann. Das definierte Rollenverhalten wird nur in einem aktiven Kontext ausgeführt. Rollenobjekte werden vom Team erzeugt und verlassen niemals[2] ihr umschließendes Team. Das Team hält Referenzen auf alle erzeugten Rollenobjekte in einer Kompositionsrelation. Jede Teaminstanz definiert einen eigenen Subtyp. Rollenobjekte eines bestimmten Teams sind auf diese Weise niemals konform zu anderen Rollenobjekten eines anderen Teams – selbst wenn die Teams vom gleichen Typ sind. Es ist also nicht möglich, Rollenobjekte unterschiedlicher Teams auszutauschen.

2.1.3 Konnektor

Eine goldene Regel in der Informatik heißt *'Divide et Impera!'* – teile und herrsche, welche ziemlich alte Wurzeln hat. Mit der Definition von Rollen ist es möglich, einzelne Aspekte einer Klasse selbst als Klasse zu kapseln. Die Gesamtheit aller kollaborierenden Rollen manifestieren sich in einem Team. Die goldene Regel wird hiermit optimal unterstützt – es bleibt nun die Frage, wie die extrahierten Aspekte wieder in das Domänenmodell gewoben werden können, wie also die einzelnen Teile wieder zusammengesetzt werden können. Die Softwaretechnik kennt eine ganze Reihe Möglichkeiten, ein Problem in kleine Teile zu separieren, aber nur wenige Mechanismen, diese Teile wieder „zusammen zu kleben".

Object Teams definiert ein Konstrukt mit dem Namen *Konnektor*. Ein Konnektor ist eine spezielle Form des Teams, ist also eine direkte oder indirekte Spezialisierung des Basisteams `Team`. Im Konnektor findet weder Definition noch Spezialisierung von Rollen statt, hier werden einzelne Rollen des Basisteams an Domänenklassen gebunden. Bindung bedeutet: eine bestimmte Domänenklasse *spielt* eine bestimmte Rolle (z.B. die Klasse `Nachbar` spielt die Rolle des Beobachters, d.h. die Klasse verhält sich wie ein Beobachter). Die in einer Rolle gekapselte Funktionalität entspricht einer Teilfunktionalität einer Domänenklasse. Sie muss also an die Stellen der Domänenklasse gebunden werden, wo diese Teilfunktionalität benutzt werden soll. Die Bindung von Rollenklasse und Domänenklasse wird auf Methodenebene definiert und nutzt die Technik der einfachen *Delegation* (*'forwarding'*). Ein Methodenaufruf an einem Objekt wird also an ein anderes Objekt delegiert.

Zwei Arten der Delegation werden in Object Teams unterschieden und zwar aus der Sicht eines Teams. Die Delegation eines Methodenaufrufes an einem Objekt einer Domänenklasse zu einer Rolle wird als *Callin* bezeichnet, die Delegation eines

alle Rollenklassen eines Teams, für die es keine weitere Verfeinerung mehr gibt (*'most sepcific role'*).

[2] Eine Ausnahme gibt es von dieser Regel: externalisierte Rollen, Kapitel 2.1.4.2

Methodenaufrufes von einer Rolle zum Objekt der Domänenklasse bezeichnet man als *Callout*.

2.1.3.1 Callin Bindung

Ein Aufruf an einem Objekt der Domänenklasse – im folgenden als Basis bezeichnet – soll an eine Methode einer Rolle delegiert werden. Voraussetzung für solche Delegation ist die Definition einer *spielt*-Beziehung. Die Rollenklasse wird also von einer bestimmten Basisklasse gespielt, zu der delegiert werden soll. Es gibt drei Arten der Delegation, die man in Abbildung 2.2 sehen kann. Ein Methodenaufruf wird an

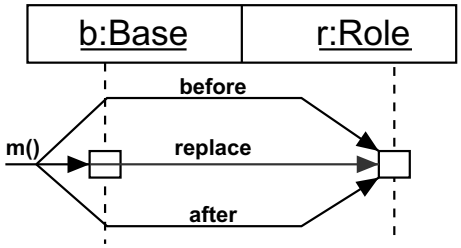

Abbildung 2.2: Die drei Arten der *'Callin'*-Delegation

eine definierte Methode der Rolle delegiert:

- bevor es zur Ausführung der Basismethode kommt (*before*).
- nachdem die Basismethode ausgeführt wurde (*after*).
- ohne die Basismethode auszuführen, was einem Überschreiben der Basismethode gleichkommt (*replace*).

In natürlicher Sprache würden solche Konstrukte so lauten: nachdem (oder bevor) eine bestimmte Basismethode aufgerufen wurde, rufe eine bestimmte Methode der Rolle auf, bzw.: ersetze eine bestimmte Methode der Basis durch eine Methode der Rolle.

Diese Delegation ist nicht in der Basisklasse definiert, sondern im Konnektor. Der Konnektor muss also die Möglichkeit haben, die Basisklasse zu *adaptieren*, um diese Delegation zuzulassen. Es ist keine Namensgleichheit der Basismethode und der Rollenmethode für diese Technik erforderlich. Unter Umständen unterscheidet sich aber die Signatur der beiden Methoden. In diesem Fall muss die Signatur angepasst werden, was in Kapitel 2.1.3.4 beschrieben wird.

2.1.3.2 Basecall

In einer Rollenklasse können einzelne Methoden derart deklariert werden, dass sie Methoden der Basisklasse überschreiben (siehe *'Replacement'* in Kapitel 2.1.1). In diesem Fall kann die Rollenmethode, die überschriebene Methode der Basis mit dem Schlüsselwort `base()` weiterbenutzen. Diese Methoden der Rolle müssen über ein *'Callin'* an die Basis gebunden, welche eine Methode der Basis ersetzen (*'replace'*). Man kann diesen Mechanismus in Abbildung 2.3 sehen: Der Methodenaufruf an

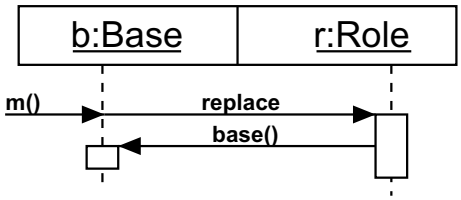

Abbildung 2.3: Spezialisierung einer Basismethode durch die Rolle

der Basis wird an die Rolle delegiert. Die Rollenmethode ersetzt die Methode der Basis. Die Rollenmethode kann nun die Implementierung der Basis über `base()` benutzen und verfeinern. Prinzipiell kann jede Methode der Rolle eine Methode der Basis ersetzen, dazu bedarf es keiner besonderen Deklaration in der Rollenmethode[3]. Diese Deklaration wird dann notwendig, wenn die Rollenmethode das Schlüsselwort `base()` benutzt. Da `base()` nur während einer Methodenersetzung eine valide Bindung hat, kann diese Methode nicht einfach von der Rolle selbst aufgerufen werden und hat deshalb eine Sonderstellung.

Die Rollenklasse verfeinert in diesem Fall das Verhalten der Basisklasse und kann in dieser Konstellation als Subtyp der Basisklasse angesehen werden. Man beachte, dass auch die Rollenklasse selber in eine Vererbungshierarchie eingebunden sein kann. Es kann also in einer Methodenersetzung nicht nur `base()` sondern auch `super()` verwendet werden - die Rolle hat hier zwei Vererbungshierarchien!

2.1.3.3 Callout Bindung

Die Richtung der Delegation entgegengesetzt zum *'Callin'*-Mechanismus, nennt man *'Callout'* – eine Delegation der Rolle an ihre Basis. Wie in Kapitel 2.1.1 beschrieben, kann eine Rolle abstrakte Methoden (*'open spots'*) deklarieren. Diese abstrakten Methoden können auf zwei Arten effektiv werden. Es besteht die bekannte Möglichkeit der Implementierung solcher Methoden in einer abgeleiteten Klasse. Dazu ist es

[3] Es ist kein guter Stil, Methoden der Basis zu ersetzen ohne die ursprüngliche Implementierung zu benutzen. Da Rollen einen Aspekt kapseln und unabhängig von der jeweiligen Basis modelliert und implementiert werden, gibt es eigentlich keine Anwendung für diese Möglichkeit.

nötig ein Team zu spezialisieren, es werden alle Rollen implizit vererbt. Nun kann eine Definition der abstrakten Methoden erfolgen, so dass die Rollenklasse effektiv wird. Die zweite Möglichkeit eine abstrakte Methode zu definieren, ist die Bindung dieser Methode an eine Methode der Basis. Immer wenn die abstrakte Methode der Rolle aufgerufen wird, wird dieser Methodenaufruf an eine definierte Methode der Basis delegiert. Die Basisklasse implementiert in diesem Fall also die abstrakte Methode der Rollenklasse. Dies kommt einer Vererbungsbeziehung zwischen Rollenklasse und Basisklasse gleich - die Basisklasse kann in diesem Fall als Subtyp der Rollenklasse angesehen werden.

2.1.3.4 Signaturanpassung

Einzelne Teams sind autark zu modellieren, zu implementieren und zu warten. Diese Unabhängigkeit trägt zu einem strukturierteren Gesamtsystem bei. Ein Konnektor bindet die einzelnen Rollen an Domänenklassen. Die Bindung erfolgt auf Methodenebene, also einer Ebene der kleinsten Granularität. Die Unabhängigkeit der beiden Strukturen, des Teams und der Domänenklasse, kann dazu führen, dass die Signatur von Methoden der Rolle, die an die Basis gebunden werden sollen, nicht übereinstimmen. Da bei einer Delegation auch alle Parameter an die Rolle, bzw. die Basis übergeben werden, kann dies zu Unwegsamkeiten und damit zur Abhängigkeit von Team und Domänenklasse führen. Aus diesem Grund besteht in Object Teams die Möglichkeit, bei einer Bindung auch Anweisungen zu definieren, wie die Signatur angepasst werden soll. So können Parameter z.B. vertauscht oder weggelassen werden. Es ist sogar möglich einzelne Parameter zu transformieren. Wichtig ist dabei, dass die ursprüngliche Parameterliste erhalten bleibt. Es ist möglich, mehrere Bindungen unterschiedlicher Konnektoren an eine Methode der Basis zu deklarieren. In diesem Fall geht jede einzelne Rolle von der Signatur aus, wie sie in der Domänenklasse deklariert ist. Signaturanpassungen können an allen Bindungen, ob Callin oder Callout, deklariert werden. Ist eine Signaturanpassung deklariert, so wird die Parameterliste vor der Delegation automatisch angepasst. Da zur Signatur auch der Rückgabewert zählt, kann natürlich auch das Resultat transformiert werden. Es existiert eine automatische Signaturanpassung für Methoden ohne Parameter. Hier werden alle Parameter automatisch versteckt, ohne dass dies deklariert werden muss.

2.1.3.5 Konnektoraktivierung

In einem Konnektor sind Bindungen von Rollenklassen zu Domänenklassen definiert. Die Definition der einzelnen Bindungen muss nicht vollständig sein, in diesem Fall bleibt der Konnektor abstrakt. Ein Konnektor wird *effektiv*, wenn alle abstrakten Methoden der einzelnen Rollenklassen entweder implementiert oder an Methoden

der Domänenklassen gebunden wurden. Ein Konnektor der effektiv ist, kann instantiiert werden.

Die einzelnen Bindungen, die im Konnektor definiert wurden, sind nicht a priori verfügbar und müssen explizit aktiviert werden. Die Aktivierung wird über die Konnektorinstanz gesteuert. Alle Bindungen werden aktiv, wenn der Konnektor aktiviert wird und inaktiv, wenn der Konnektor deaktiviert wird. Die Funktionalität zum aktivieren und deaktivieren ist im Basisteam `Team` definiert und somit in jedem Team, also auch in jedem Konnektor, verfügbar. Wird eine Konnektorinstanz aktiviert, verändert sich das Verhalten aller involvierten Domänenklassen. Sie werden angereichert um die Funktionalität der Rollenklassen, zu denen Bindungen existieren. Dies bedeutet, dass die existierende Definition der Domänenklasse dynamisch (zur Laufzeit der Anwendung) um die Delegation zu der definierten Rollenfunktionalität angereichert wird. Man bezeichnet diesen Prozess auch als *dynamisches Weben* (‘*runtime weaving*’). Die Technik des dynamischen Webens ist nicht invasiv, da der Quellcode der Klasse weder erforderlich ist, noch verändert wird. Wird ein Konnektor deaktiviert, werden alle von diesem Konnektor gewobenen Erweiterungen aus der Klassendefinition entfernt.

Statische Aktivierung eines Konnektors Im Gegensatz zu der vorgestellten dynamischen Aktivierung, existiert auch die Möglichkeit ein konkretes Rollenverhalten statisch an die Domänenklassen zu binden. In diesem Fall muss der Konnektor statisch deklariert werden. Alle Bindungen werden vor dem Start der Anwendung in die Domänenklassen gewoben. Ein statisch deklarierter Konnektor kann nicht instantiiert werden und ist immer aktiv.

2.1.3.6 Der Konnektor als Kontextdefinition

Da die Aktivierung der im Konnektor definierten Bindungen an die Aktivierung der Konnektorinstanz gekoppelt ist, kann diese Instanz als Kontextdefinition angesehen werden. Wird ein Kontext betreten (der Konnektor wird aktiviert), verändert sich das Verhalten aller involvierten Domänenklassen. Dies lässt sich sehr schön mit der Rollenmetapher in Einklang bringen. Das spezifische Rollenverhalten tritt nur in einem spezifischen Kontext auf. Ist der Kontext nicht gegeben, existiert das Rollenverhalten nicht. Wird also der Kontext verlassen (der Konnektor wird deaktiviert), verhalten sich die einzelnen Domänenklassen wieder so, als ob die Bindung zu einem Rollenverhalten nie existiert hätte. Diese Kopplung ist äußerst mächtig, da sie die Modellierung dynamischer bzw. kontextändernder Aspekte ermöglicht, wobei die Art des Kontextes durch die Domäne festgelegt wird.

2.1.4 Translationspolymorphismus

Domänenklassen wie auch Teamklassen werden unabhängig voneinander modelliert und entwickelt. Keine der beiden trägt Wissen über die Existenz des jeweils anderen. Somit kennen die Rollenklassen keinen Typ der Domäne, Domänenklassen keinen Typ aus einem Team. Die Bindung einer Rollenklasse an eine Domänenklasse hat eine weitere semantische Eigenschaft. Beide Typen sind innerhalb eines aktiven Kontextes ersetzbar. Mit der Definition einer *spielt*-Beziehung (eine Rollenklasse wird von einer bestimmten Domänenklasse gespielt) bringt man zwei Klassen bzw. Typen in Beziehung. Instanzen vom Typ der Domänenklasse genügen mit solch einer Definition auch dem Typ der definierten Rollenklasse. Umgedreht gilt dies natürlich genauso: Instanzen einer bestimmten Rollenklasse sind ersetzbar zu der definierten Domänenklasse. Die Beziehung gilt nur in dem jeweiligen aktiven Kontext. Da es mehrere Teamdefinitionen geben kann, die an die gleiche Klasse der Domäne gebunden werden können, genügt diese Klasse einer Menge von Typen - man spricht dann von Polymorphie. Object Teams führt eine neue Form des *'Castings'* ein, mit der es möglich ist, eine Basis zu einer bestimmten Rolle zu transformieren. Diese Transformation heißt in Object Teams *lifting*. Die Transformation der Rolle zurück zu ihrer Basis heißt *lowering*. Eine dynamische Transformation (*'dynamic cast'*) in einer objektorientierten Sprache ändert nicht das Objekt, sondern nur den dynamischen Typ mit der dieses Objekt referenziert wird. Ein Lifting einer Basis zu einer Rolle resultiert in einer Referenz auf ein anderes Objekt - der Rolle der Basis (Rollen sind eigene Objekte). So ergibt ein Lowering dieser Rolle wieder die Referenz auf das Basisobjekt. Da hier zwei Objekte unterschiedlichen Typs, die Basis und die Rolle, das gleiche Objekt repräsentieren, heißt diese Form der Polymorphie *Translationspolymorphie*.

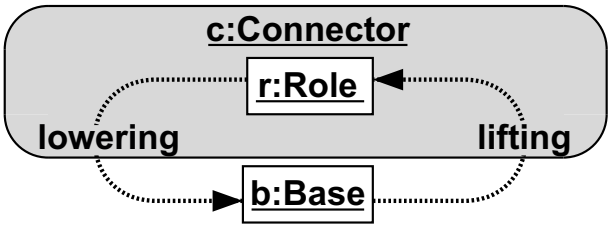

Abbildung 2.4: Ein *'Casting'* in Object Teams

Lifting bzw. Lowering findet immer im Kontext eines Konnektors statt. Nur mit Hilfe dieses Kontextes ist überhaupt definiert, zu welcher Rolle geliftet werden soll. Da es mehrere Konnektoren zu der gleichen Basis geben kann, ist es auch möglich zu verschiedenen Rollen zu liften. Das Lowering einer bestimmten Rolle zurück zur Basis ist wiederum nur an genau der Teaminstanz des erzeugenden Teams definiert, da alle anderen Teams diese Rolle nicht kennen. Ein Lifting einer Basis zu einer Rolle an einem Konnektor resultiert immer in dem gleichen Rollenobjekt - gleiches gilt

natürlich für das Lowering einer Rolle. Soll eine Basis geliftet werden, dessen Rolle nicht existiert, so wird automatisch ein neues Rollenobjekt erzeugt und zu der Basis assoziiert.

2.1.4.1 Zwei Welten - ein Objekt

Als Resultat der Unabhängigkeit zwischen Domänenklassen und Teamklassen, wird weder das Lifting noch das Lowering explizit benutzt. Die Definition einer *spielt*-Beziehung führt dazu, dass bei einer aktiven Callin-Bindung, die Basis zu ihrer Rolle geliftet wird und der Methodenaufruf zu dieser Rolle delegiert wird. Diese spezielle Rolle repräsentiert also das Basisobjekt innerhalb des speziellen Teams. Bei dieser Delegation wird nicht nur das Basisobjekt transformiert, sondern auch alle Parameterobjekte, dessen Klassen eine Rolle in diesem Team spielen. Es gibt also keine Möglichkeit, Objekte der Domäne, die eine Rolle in einem Team spielen, in dieses Team zu reichen, da diese Objekte *automatisch* in ihre spezielle Rolle geliftet werden. Umgedreht gilt dies natürlich genauso und beinhaltet die interessantere Aussage: Eine Rolle wird automatisch zu ihrer Basis transformiert, sobald der Kontext des umschließenden Teams verlassen wird. Eine Rolle kann also unter dieser Voraussetzung ihr umschließendes Team nie verlassen.

Die automatische Transformation von Instanzen der Domänenklasse zu einer Rolle und umgekehrt stellt sicher, dass sich die Menge aller Objekte der Domäne und die Menge aller Objekte eines Teams, niemals untereinander mischen. Es sind zwei disjunkte Mengen, die über eine Relation miteinander verbunden sind:

$$\mathbf{Domain} = \{d \mid d \in domain\} \tag{2.1}$$

$$\mathbf{Team} = \{t \mid t \in T \wedge T\ instanceof\ Teamclass\} \tag{2.2}$$

$$\mathbf{Domain} \cap \mathbf{Team} = \oslash \tag{2.3}$$

$$\begin{aligned} \mathbf{Konnektor} = \{&(d,t) \in \mathbf{Domain} \times \mathbf{Team} \mid \\ &\forall d \in \mathbf{Domain}\ \exists t \in \mathbf{Team} \wedge \\ &\forall t \in \mathbf{Team}\ \exists d \in \mathbf{Domain}\} \end{aligned} \tag{2.4}$$

Abbildung 2.5: Die Relation zwischen Basis- und Rollenobjekten

Ein typisches Szenario einer Callin Bindung wäre also: die Basis wird an dem aktiven Konnektor zu ihrer Rolle geliftet. Alle Parameter der Callin-Bindung werden zu ihrer Rolle geliftet, falls sie Instanzen einer Klasse sind, die in dem aktiven Kontext eine Rolle spielen. Die gebundene Methode wird mit diesen neuen Parametern aufgerufen. Ist der Rückgabewert selbst eine Rolle, wird dieser zu ihrer Basis transformiert (*'lowering'*). Wird während der Ausführung innerhalb der Rolle eine Funk-

tionalität an der Basis wahrgenommen (Callout-Bindung oder `base`), wird die Rolle zu ihrer Basis transformiert (*'lowering'*), mit allen Parametern, falls es Rollen des aktiven Kontextes sind. Der Rückgabewert wird geliftet, wenn das Ergebnis eine Rolle im aktiven Team spielt (siehe Abbildung 2.4).

2.1.4.2 Externalisierte Rollenobjekte

Jede Regel ist eine gute Regel, wenn es Ausnahmen derselben gibt. So darf es in der Domäne Klassen geben, die sich einer Bindung einer bestimmten Rollenklasse an eine bestimmte Basisklasse bewusst sind und die die zusätzliche Rollenfunktionalität explizit nutzen möchten. Diese Funktionalität kann natürlich nur am Objekt, also an der Rolle selbst wahrgenommen werden. Es ist also notwendig, dass eine Referenz auf eine Rolle aus ihrem Team herausgereicht werden kann. Man nennt diese Referenzen *externalisierte Rollenobjekte*. Ein Rollenobjekt ist an ihr erzeugendes Team gebunden. Der Prozess des Lifting und des Lowering ist nur in diesem Team definiert. Um dies sicher zu stellen, bildet jede Instanz eines Teams einen eigenen Subtyp. Der Typ der Rolle ist an diesen Subtyp gebunden. Auf diese Weise kann eine Rolleninstanz nur konform zu Rollen des gleichen Teams sein und wird als Rolle nur im gleichen Team akzeptiert.

2.2 Modellierung mit ObjectTeams

UML [OMG 2002] ist der Standard in der Modellierungstechnik. Der große Vorteil des objektorientierten Designs ist die einfache und geradlinige Umsetzung des Entwurfs in der Phase der Implementierung. Im Zuge dieser Arbeit soll eine Softwarelösung vorgestellt werden, die maßgeblich auf der Modellierung und Implementierung von Aspekten als Rollen in Teams beruht. In den nächsten Kapiteln soll deshalb eine Erweiterung der UML vorgestellt werden, mit der es möglich ist, Rollen, Teams und Konnektoren darzustellen. Die Erweiterung trägt den Namen: *UFA - UML for Aspects* und wurde von Herrmann [2002a] vorgeschlagen.

2.2.1 Modellierung von Aspekten

Die Modellierung in der Phase des objektorientierten Designs lässt verschiedene Ebenen der Granularität zu. Im Interesse der Übersichtlichkeit und Verständlichkeit muss das Modell nicht vollständig beschrieben werden, oder kann ein bestimmtes Designziel fokussieren. Systeme können auf einem sehr hohen Abstraktionsniveau modelliert werden, wohingegen sie in der Phase der Implementierung sehr konkret beschrieben werden müssen. Dem Begriff des Aspektes haftet eine starke Relation zu der Ebene von Quellcode an. Kristallisieren sich Aspekte also erst in der Phase

der Implementierung? Herrmann [2002a] gibt zwei Begründungen, warum Aspekte in der Domänenmodellierung eine untergeordnete Rolle spielen und erst in der technischen Modellierung auftauchen:

1. In einem Domänenmodell können Funktionalitäten existieren, die durch ein einzelnes Wort hinreichend beschrieben werden. Ein Beispiel für solch eine Funktionalität wäre z.B. Persistenz. In der Applikation kann damit ein ganzes Subsystem mit weit verteilten Funktionalitäten gemeint sein. In der Modellierung reicht die Markierung *„persistent“* völlig aus. In der UML werden für solche Zwecke Stereotypen eingeführt, die für diese Phase des Designs völlig ausreichen auch wenn sie zu einem späteren Zeitpunkt genau definiert werden müssen
2. Die UML erlaubt die Modellierung von Funktionalitäten, z.B. in Sequenzdiagrammen. Es besteht die Möglichkeit, Konzepte zu modellieren, welche zu verschiedenen Klassen im Modell assoziiert werden können. Dies kommt einer Aspektmodellierung gleich.

Das Paradigma der Object Teams erlaubt es, ein komplexes Verhalten abstrakt zu beschreiben, stückweise zu verfeinern und durch aussagekräftige Bindetechniken in die Domänenklassen zu weben. Dies ermöglicht einen strukturierten Umgang mit Aspekten in der Phase der technischen Modellierung. Um die Konzepte auch im Modell optimal nutzen zu können, sind Adaptionen bekannter Konstrukte, bzw. völlig neue Konstrukte notwendig, die im folgenden näher erläutert werden sollen.

2.2.2 Teammodellierung mit UFA

Es liegt nahe, eine Team aus Object Teams als Paket in UML darzustellen. Da die einzelnen Rollen selbst Klassen innerhalb eines Teams sind, ist auch hier kein neues Konstrukt in der Modellierung nötig. Das Beobachtermuster [Gamma u. a. 1996, Observerpattern] könnte beispielsweise wie in Abbildung 2.6 zu sehen modelliert werden. Es wird ein Team mit dem Namen des Musters definiert, welches zwei Rol-

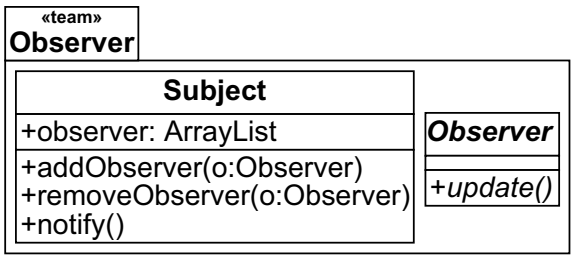

Abbildung 2.6: Das Beobachtermuster in UFA

len enthält: das Subjekt und den Beobachter. Die Klasse des Beobachters ist abstrakt

deklariert, da es eine abstrakte Methode (`update()`) enthält. Diese Methode muss entweder implementiert werden oder an eine Domänenklasse gebunden werden. Da dieser Punkt hier explizit offen gelassen wurde, ist das gesamte Team abstrakt deklariert.

Es besteht nun die Möglichkeit, eine abstrakte Teamdefinition zu verfeinern. Dies muss nicht notwendigerweise bedeuten, dass das erbende Team effektiv ausmodelliert werden soll — es ist möglich, verschiedene Abstraktionsebenen in Form konkretisierender Teams zu modellieren. Eine einfache Erweiterung der Funktionalität beispielsweise kann man in Abbildung 2.7 sehen. Eine spezielle Form des

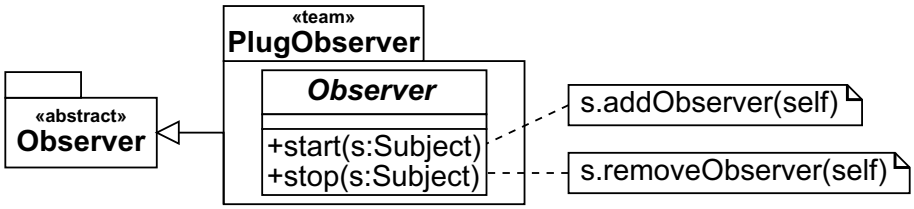

Abbildung 2.7: Teamvererbung in UFA: Ein spezieller Observertyp

Beobachtermusters wird hier modelliert. Sie ermöglicht dem einzelnen Beobachter sich an einem speziellen Subjekt an- bzw. abzumelden. Es ist zu beachten, dass in dem Paket `PlugObserver` zwei Rollen definiert sind, auch wenn man nur eine Klassendefinition sieht. Durch implizite Vererbung (siehe Kapitel 2.1.2) sind in der Vererbungshierarchie eines Teams all jene Rollen implizit verfügbar, die das jeweils beerbte Team schon deklariert hat. Die Klasse `Observer` wurde im beerbten Team schon beschrieben. Die Modellierung dieser Klasse in `PlugObserver` ist also eine implizite Verfeinerung. Es ist kein expliziter Spezialisierungspfeil nötig, da dieser auf Teamebene schon definiert wurde. In der Beobachterklasse des spezialisierenden Teams sind also drei Methoden verfügbar (`start()`, `stop()` und `update()`). Da es keine Implementierung der abstrakten Methode `update()` gibt, ist die Klasse und damit das ganze Team abstrakt.

In einem Team können, wie in einer Klasse, Attribute und Methoden modelliert werden. Das Paket aus UML wird um den Stil einer Klassenmodellierung erweitert. So kann ein Paket um zwei Kästen (*'compartments'*) erweitert werden, in denen die Attribute und Methoden in bekannter Syntax aufgeschrieben werden können.

2.2.3 Kompositionsmodellierung

Ein Konnektor ist eine Spezialisierung eines schon definierten Teams. Da er also selbst ein Team ist, wird auch das gleiche Symbol benutzt: ein Paket. Dieses wird aber besonders gekennzeichnet. UFA führt zu diesem Zweck einen neuen Stereotypen «`connector`» ein.

Ein Konnektor ist, wie in Kapitel 2.1.3 beschrieben, in der Lage Rollenklassen eines spezialisierenden Teams an Klassen der Domäne zu binden. Die jeweiligen Domänenklassen werden verändert — adaptiert. UFA führt aus diesem Grund eine neue Art der Assoziationsbeziehung ein: die Adaptierungsbeziehung <<adapts>>. Diese Assoziation geht immer von einem Konnektor aus und zeigt auf das zu adaptierende Paket, bzw. die zu adaptierende Klasse.

In dem Paket des Konnektors können Bindungen definiert werden. Der Konnektor darf nur Rollen binden, die in dem spezialisierenden Team oder einem Superteam definiert wurden. Die einzelnen Rollen können nur an Klassen der Domäne gebunden werden, zu der es eine Adaptierungsbeziehung gibt. Die Bindung auf der Ebene der Klassen wird mit einer besonderen Form des Klassendiagramms ausgedrückt. Der Klassenname ist in diesem Fall der Name der Bindung in der Form: „Rollenklasse = Basisklasse“. Die Bindung auf der Ebene der Methoden erweitert das bekannte Klassendiagramm. Herkömmlicherweise besteht es aus drei Boxen. In der ersten steht der Klassenname, in der zweiten sind alle Attribute definiert und in der dritten Box sind alle Methoden deklariert. UFA definiert eine vierte Box, in der einzelne Bindungen von Methoden definiert werden können. Eine Bindung hat immer die Form „Rollenmethode Bindungstyp Basismethode“. Die einzelnen Bindungstypen werden folgendermaßen dargestellt:

Bindungstyp	Rollenmethode	Darstellung	Basismethode
Callout		→	
Callin, davor		← **before**	
Callin, danach		← **after**	
Callin, ersetzend		←	

Abbildung 2.8: UFA: Darstellung von Bindungstypen.

2.2.4 Die UFA Elemente in einem Beispiel

Um die einzelnen Elemente in einen Zusammenhang zu bringen, soll hier ein einfaches Beispiel gezeigt werden. In Abbildung 2.9 ist ein vereinfachtes Domänenmo-

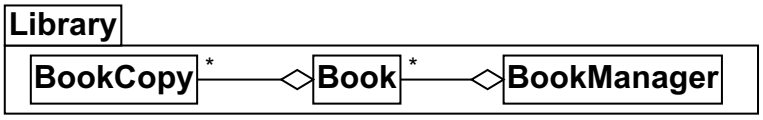

Abbildung 2.9: Einfaches Beispiel einer Domäne: eine Bibliothek

dell einer Bibliothek modelliert. Die Bibliothek besteht aus einer Reihe von Büchern

(`Book`), die von einem Buchmanager (`BookManager`) verwaltet werden. Die physikalisch vorhandenen Bücher der Bibliothek werden als Buchkopien (`BookCopy`) im System dargestellt. Jedes Buch kann also in verschiedener Stückzahl in der Bibliothek vorhanden sein

Die Funktionalität des Buchmanagers soll nun um die Fähigkeit erweitert werden, alle ausgeliehenen Bücher aufzulisten. Jedesmal wenn ein Buch ausgeliehen oder zurückgegeben wird, soll der Buchmanager über diese Aktion benachrichtigt werden, um diese Liste aktuell zu halten. Hier soll das Beobachtermuster Anwendung finden, um die gewünschte Funktionalität zu erbringen.

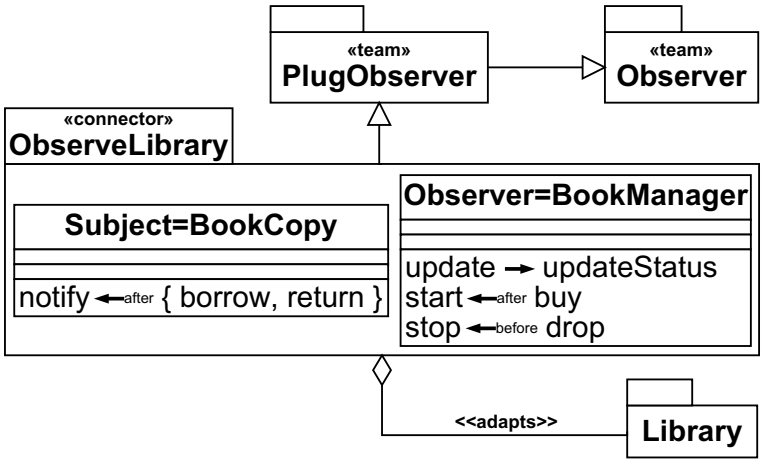

Abbildung 2.10: Anwendung des Beobachtermusters in der Bibliothek

Der Buchmanager spielt in diesem Beispiel den Beobachter. Alle in der Bibliothek vorhandenen Exemplare aller Buchtitel sollen beobachtet werden. Die Buchkopie spielt also die Rolle des Subjektes in diesem Muster. Abbildung 2.10 zeigt die Bindung des speziellen Beobachtermusters `PlugObserver` an die Bibliothek. Die Klassen der Bibliothek werden *adaptiert* - die Rollen des Teams werden an die Klassen der Bibliothek gebunden. Die Bindung auf Klassenebene umfasst nur zwei Deklarationen: die Buchkopie (`Subject=BookCopy`) spielt die Rolle des Subjektes und der Buchmanager (`Observer=BookManager`) spielt die Rolle des Beobachters. Die Methodenbindungen haben folgende Aussagen:

- `notify` $\leftarrow_{\text{after}}$ `{borrow, return}`: wenn immer eine Buchkopie ausgeliehen, bzw. zurückgegeben wurde, sollen alle Beobachter benachrichtigt werden.

- `update` $\rightarrow$ `updateStatus`: die abstrakte Methode `update` in `Observer` wird durch die Methode `updateStatus` in `BookManager` implementiert.

- `start` ←after `buy`: nachdem ein Buch vom Buchmanager gekauft wurde, meldet sich dieser Buchmanager bei der Buchkopie als Beobachter an.
- `stop` ←before `drop`: bevor ein Buch aus dem Bestand entfernt wird, meldet sich der Buchmanager als Beobachter wieder ab.

Die Klassendefinitionen im Konnektor definieren hier nur Methodenbindungen, Attribut- und Methodenboxen sind leer. Die beiden Rollendefinitionen sind vollständig — dieser Konnektor ist instantiierbar und ermöglicht das geforderte Verhalten nach einer Aktivierung.

3 Ruby Object Teams

Die Frage, welche Programmiersprache man benutzt, kommt der Gretchenfrage gleich. Grundlage der hier vorgestellten Lösung ist die Programmiersprache *Ruby* [Matsumoto 2002]. Gründe für diese Wahl sind weniger Neigungen für unbekannte Sprachen oder die Ablehnung strenger Typsysteme, als vielmehr der Fakt, das zum Zeitpunkt der Entstehung dieser Arbeit, die Implementierung von Object Teams für Java [Binder und Hundt 2002] noch mitten in der Entwicklung steckte und nicht wirklich nutzbar war. *Ruby* brachte viele Voraussetzungen mit, die eine schnelle Implementierung der Konzepte von Object Teams möglich machten. Um die Implementierung dieser Konzepte zu verstehen, soll kurz auf die Sprache selbst eingegangen werden (Kapitel 3.1), um danach zu zeigen wie die Konzepte von Object Teams in dieser Programmiersprache realisiert wurden (Kapitel 3.2). An einem kleinen einführenden Beispiel soll dann die Funktionalität erläutert werden (Kapitel 3.3).

3.1 Ruby

Ruby ist eine sehr mächtige objektorientierte Skriptsprache mit einer einfachen Syntax. Es vereint verschiedene Konzepte aus Eiffel, Smalltalk, Lisp und Java ohne schwerwiegende syntaktische Konstrukte. Rubycode wird interpretiert, wobei verschiedenste Möglichkeiten vorgesehen wurden, in das Verhalten des Interpreters einzugreifen.

Im folgenden Text sollen nur die wesentlichen Merkmale genannt werden, die wichtig für die Implementierung von Ruby Object Teams waren. Für eine detaillierte Beschreibung von Ruby sei auf Thomas und Hunt [2000]; Fulton [2001]; Matsumoto [2002] verwiesen.

Typisierung

Ruby benutzt keine starke Typisierung wie etwa C++, da diese nur im Zusammenspiel mit einem Compiler Sinn machen würde. Die Art der unterstützten Typisierung heißt hier „*dynamische Typisierung*“. Das bedeutet, dass Variablen in Ruby mit keinem Typ deklariert werden und Initial auch keinem Typ entsprechen. Objekte hingegen entsprechen einem bestimmten Typ, sie sind ja Instanzen einer bestimmten Klasse. Der Typ einer Variablen entspricht somit dem Typ des referenzierten

Objektes, wobei jede Variable jedes Objekt referenzieren kann. Der dynamische Typ einer Variablen kann zur Laufzeit erfragt werden und zur Typprüfung genutzt werden.

Ein Methodenaufruf wird in Ruby als Nachricht verstanden, die dem referenzierten Objekt einer Variablen gesendet wird. Jedes Objekt bietet die Möglichkeit zu erfragen, ob eine bestimmte Nachricht entgegengenommen werden kann. Ein Konstrukt wie Schnittstelle (*'Interface'*) oder abstrakte Klasse ist in Ruby nicht definiert. An ihre Stelle tritt die Definition von Nachrichten, welche das Objekt verstehen muss, egal welchem Typ diese Instanz entspricht.

Vererbung

Ruby unterstützt einfache Vererbung (*'single inheritance'*), wie das auch in Java der Fall ist, in Kombination mit einem Objektmodell. Eine Klasse ohne Vererbungsrelation erbt automatisch von der Basisklasse `Object`. Jede Klassenhierarchie hat also ihre Wurzel in dieser Klasse.

In Ruby gibt es eine Dualität zwischen Klassen und Objekten. Jede Klasse ist im Laufzeitsystem eine Instanz der Klasse `Class` und somit selbst wiederum ein Objekt. Methoden, die statisch deklariert wurden, sind auf diesem Wege Instanzmethoden der Klassenobjekte — mit allen Vorzügen der Objektorientierung. So ist beispielsweise in der Klasse `Class` die Methode `new` implementiert, welche eine neue Instanz des jeweiligen Typs erzeugt (vgl. Smalltalk). Sogar die Objekterzeugung braucht also kein neues Schlüsselwort: `myObject = MyClass.new` erzeugt ein neues Objekt vom Typ `MyClass`. Eine Klassendefinition kann außerdem Anweisungen enthalten, die als statische Methoden in Superklassen definiert wurden. Wird solch eine Klasse geladen, wird dieser Code ausgeführt (statischer Block).

Möchte man zusätzlich Funktionalität in verschiedenen Klassen wiederverwenden, die in keiner Vererbungsrelation stehen, so besteht die Möglichkeit *Module* zu definieren, welche in die bestimmten Klassen importiert werden können. Solch ein Import macht alle im Modul definierten Methoden der jeweiligen Klasse verfügbar. Die Kombination dieser beiden Techniken, einfache Vererbung plus Import von Funktionalität ist sehr mächtig, aussagekräftig und umgeht geschickt die Probleme, die sich mit multipler Vererbung verbinden.

Referenzierung

In Ruby ist jede Entität ein Objekt. Es existieren keine einfachen Basistypen. Auf die einzelnen Objekte wird immer per Referenz zugegriffen. Die Allokation von Speicher für ein Objekt übernimmt das Laufzeitsystem, nicht referenzierte Objekte werden von einem *'Garbage Collector'* finalisiert.

Spezielle Eigenschaften

Ruby unterstützt das Prinzip des einheitlichen Zugriffs [Meyer 1997, *'Uniform Access principle'*] auf Attribute einer Klasse. Die Deklaration von Attributen umfasst das generieren der nötigen Zugriffsmethoden (*'getter/setter'*). Diese Zugriffsmethoden heißen genauso wie das deklarierte Attribut. Da man bei einem Methodenaufruf die Klammern weglassen kann, ist der Zugriff auf ein Attribut von einem Methodenaufruf nicht zu unterscheiden.

Es besteht die Möglichkeit Codeblöcke als Objekt zu kapseln. Solche Objekte können einer Methode als normaler Parameter übergeben werden. Ruby definiert dazu eine eigene Syntax in folgendem Stil: `o.method(){<code>}`. Die Empfängermethode kann mit einem Schlüsselwort diesen Block ausführen. Die Möglichkeit Codeblöcke als Objekt zu kapseln, liefert die syntaktische Grundlage für die Erweiterungen, die für Ruby Object Teams nötig waren.

Das Konzept der *'Inner Classes'*, bekannt aus java, ist hier nicht definiert. Es besteht die Möglichkeit Klassen zu schachteln (*'nested classes'*), also Klassen innerhalb von umschließenden Klassen zu definieren. Diese Möglichkeit wird in Ruby nur zur Namensraumabgrenzung genutzt und hat keine weitere semantische Bedeutung. Das Konstrukt der geschachtelten Klasse wurde für die Realisierung von Teams aufgegriffen und um Techniken angereichert, die für das Object Teams Paradigma nötig waren.

Ruby ist eine reflexive Sprache. Es ist möglich Typinformationen, definierte Konstanten, implementierte Methoden und Instanzvariablen zur Laufzeit an einem Objekt zu erkunden. Es besteht die Möglichkeit von Adaptionen und Erweiterungen von Klassen, auch nachdem eine Klasse schon geladen oder gar benutzt wird. So ist es z.B. möglich, nachträglich Module zu importieren oder Methoden zu definieren. Es existiert sogar eine Technik, Methoden umzubenennen. Es dürfen für eine Methode alias-Namen vergeben werden, so dass eine Methode über mehrere Namen zugreifbar ist. Anstelle der ursprünglichen Methode kann nun eine neue definiert werden.

Beispiel

Die Syntax von Ruby ist sehr intuitiv und einfach. Aus diesem Grund zeigt diese Arbeit an einigen Stellen kurze Beispiele im Quellcode. Um mit der Syntax vertraut zu werden, soll ein kleines Beispiel gezeigt werden: Eine Klasse wird mit dem Schlüsselwort `class` deklariert. Ruby macht von der `begin-end` Schreibweise Gebrauch, so dass Klassen, Methoden, Bedingungen etc. mit einem `end` abgeschlossen werden müssen. Methoden werden mit dem Schlüsselwort `def` und einem Namen eingeleitet. Die Attribute wie auch die Parameter der Methoden werden ohne Typ deklariert, ihr Typ ergibt sich aus dem assoziierten Objekt. In einer Klasse können Attribute mit `attr` deklariert werden. Solch eine Deklaration erzeugt eine Methode mit dem Namen des Attributes, so dass von außen auf diese Variable völlig transparent

```ruby
class Person #Klassendeklaration
  attr :name #Attributdeklaration mit Lesezugriff
  def initialize(name) #Konstruktor
    @name = name #Attributzuweisung
  end
end
```

Listing 3.1: Ruby Syntax am Beispiel

zugegriffen werden kann. Mit `attr_accessor` entsteht zusätzlich noch eine Methode `<attribut>=`, so dass die Variable auch gesetzt werden kann. Diese Syntax ermöglicht einen einheitlichen Zugriff (*'uniform access'*) auf Attribute einer Klasse. Innerhalb der Klasse kann mit vorangestelltem `@` direkt auf das Attribut zugegriffen werden. Die üblichen Deklarationen für die Sichtbarkeit: `public`, `protected` und `private` können entweder direkt als Modifizierer einer Methode benutzt werden (wie in Java) oder allein stehen (wie das in C++ üblich ist), dann gelten sie bis zum nächsten Modifizierer.

```ruby
class Student < Person #Vererbungsdeklaration
  attr_accessor :matrikelnummer #Lese– und Schreibzugriff
  def initialize(name, matrikelnummer)
    super(name) #Konstruktor der Superklasse
    @matrikelnummer = matrikelnummer
  end
  def lese(text) #Methodendeklaration mit Parameter
    print(text+" fertig gelesen!\n")
  end
end
hm = Student.new("Herbert Mustermann", 1234567890)
hm.lese("Ruby in a Nutshell")
```

Listing 3.2: Ruby in einer Nussschale

Wird eine Klasse verfeinert, geschieht dies mit dem $<$ Operator. So ist die Klasse `Student` ein Subtyp der Klasse `Person`. Die Methode `initialize` hat eine besondere Bedeutung. Immer wenn ein neues Objekt von einer Klasse instantiiert wird, wird diese Methode automatisch aufgerufen. Sie stellt also eine Art Konstruktor dar, in der das Objekt initialisiert werden kann. Methoden können überschrieben werden, wobei die Implementierung der Basisklasse über das Schlüsselwort `super` verfügbar ist. Ein überladen von Methoden ist nicht möglich.

3.2 Implementierung von Object Teams in Ruby

Dieses Kapitel beschreibt die Synthese der Konzepte von Object Teams in die Programmiersprache Ruby. Ruby unterscheidet sich von anderen Programmiersprachen in vielerlei Hinsicht in Bezug auf unterstützte Techniken und Konzepte. Eine Umsetzung allgemeiner Konzepte in eine spezielle Umgebung sollte sich möglichst natürlich in diese einpassen. Das hier beschriebene Resultat ist also eine spezifische Ruby-Lösung, mit ihren ganz eigenen Vor- und Nachteilen.

Ruby ist eine interpretierte Sprache. Das heißt vor allem: es gibt keinen Kompiler und keine Kompilierzeit. Ein Kompiler könnte, abgesehen von statischer Typprüfung, viele Transformationen schon in dieser Phase durchführen, die notwendig für die Konzepte von Object Teams sind. Alle notwendigen Erweiterungen und Adaptionen müssen also zur Laufzeit in die für sie vorgesehenen Strukturen eingebracht werden. Wie in Kapitel 3.1 beschrieben, gibt es zahlreiche Möglichkeiten in das Verhalten des Interpreters einzugreifen, also das Verhalten bestimmter Klassen und Objekte zu manipulieren, die im folgenden näher beschrieben werden sollen.

Kapitel 3.2.1 erläutert, wie ein Team und die enthaltenen Rollen definiert werden und zeigt die einzelnen Eigenschaften an einem Beispiel. Kapitel 3.2.2 geht auf die Definition eines Konnektors ein und erklärt wie Bindungen deklariert, bzw ausgewertet werden. In Kapitel 3.2.3 wird der dynamische Teil von Ruby Object Teams beleuchtet: was passiert mit den einzelnen beteiligten Klassen, wenn ein Konnektor aktiviert bzw. deaktiviert wird, wie sich also die einzelnen Bindungstypen manifestieren.

3.2.1 Die Erstellung eines Teams

Das Team in Object Teams (siehe 2.1.2) wird in Ruby als einfache Klasse umgesetzt. Ein Team umschließt eine Menge von Rollen und hat in dieser Hinsicht den Charakter eines Paketes. Eine Klasse erlaubt die Definition von Klassen im Kontext derselben zu schachteln. Diese geschachtelten Klassen sind Rollen des umschließenden Teams - Rollen werden also auch als einfache Klassen implementiert.

Die Definition von Teams und Rollen unterscheidet sich syntaktisch nicht von „normalen“ Klassen, wohl aber semantisch. Ruby Object Teams definiert die Klasse `Team`. Jede Teamdefinition muss diese Klasse verfeinern, entweder direkt oder durch Verfeinerung einer bestehenden Teamdefinition. Die Klasse `Team` hat somit für Teams eine äquivalente Rolle, wie die Klasse `Object` für Klassen - sie definiert die Wurzel aller Teamhierarchien. Das Basisteam `Team` implementiert eine Reihe von Funktionalitäten, die somit jedem Team zur Verfügung stehen.

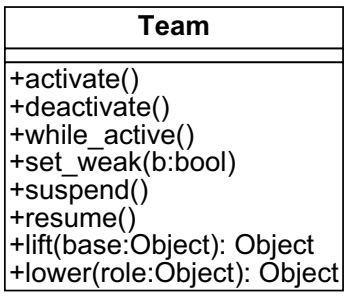

Abbildung 3.1: Das Basisteam `Team`

In Abbildung 3.1 sieht man einen Ausschnitt der Funktionalität, die ein Team erbringt. Jede einzelne Methode steht für einen Mechanismus oder eine Technik in Ruby Object Teams, die in den folgenden Abschnitten näher erläutert werden soll.

3.2.1.1 Implizite Vererbung

Bei einer Verfeinerung eines Teams, des Basisteams `Team` oder eines schon definierten Teams, kommt eine spezielle Form der Vererbung zum Tragen, welche die Technik der Klassenvererbung erweitert. Die Klassenvererbung sorgt dafür, dass das neue Team eine Subklasse des beerbten Teams ist und macht dem erbenden Team alle öffentlichen Merkmale der Superklasse verfügbar. Dieser Mechanismus wird derart erweitert, dass auch alle inneren Klassen dem neuen Team verfügbar gemacht werden. Das Verfeinern eines Teams bedeutet also auch die implizite Definition aller inneren Klassen des beerbten Teams im erbenden Team. Die Deklaration einer Klasse in dem erbenden Team mit einem Namen wie er auch schon im beerbten Team verwendet wird, hat automatisch die Semantik einer Verfeinerung, auch wenn diese nicht explizit deklariert wurde. D.h. das alle Merkmale der inneren Klasse des beerbten Teams in der inneren Klasse des erbenden Teams verfügbar sind. Da diese Vererbungsrelation nicht explizit deklariert werden muss, heißt diese Art der Vererbung *Implizite Vererbung* (*'implicit inheritance'*).

In Ruby Object Teams wird hierzu die Klassenvererbung genutzt. Die Klasse `Team` implementiert statisch einen Steuermechanismus der aufgerufen wird, wenn immer die Klasse `Team` verfeinert wird, egal ob auf direktem oder indirektem Wege. Dieser Mechanismus listet sich alle inneren Klassen des beerbten Teams auf und definiert für jede einzelne eine neue Klasse ohne Implementierung im erbenden Team mit einer Vererbungsrelation zu der gleichnamigen Klasse im beerbten Team. Eine Vererbung auf Teamebene definiert also $x + 1$ neue Klassen, wobei x die Anzahl der im beerbten Team definierten Klassen ist. Enthält das erbende Team eine Klassendefinition für solch eine generierte Klasse, so ist diese Definition automatisch eine Erweiterung dieser generierten Klasse.

In Ruby werden innere Klassen zur Namensraumabgrenzung genutzt. Eine innere Klasse `R` des umschließenden Teams `A` heißt voll qualifiziert `A::R`. Wird das Team `A` von Team `B` verfeinert, wird durch implizite Vererbung der Typ `B::R` eingeführt, der die Klasse `A::R` verfeinert. Eine Definition von `R` in `B` erweitert automatisch die Klasse `B::R`. Die Referenzierung eines Typs innerhalb eines Teams wird vom Interpreter automatisch in den voll qualifizierten Typ transformiert. Ist beispielsweise in einer imaginären Methode in `A::R` die Instantiierung eines neuen Objektes vom Typ `R` deklariert, so erzeugt die Methode immer Objekte vom Typ `A::R`. Auch die Klasse `B::R` würde bei Aufruf dieser Methode Objekte vom Typ `A::R` erzeugen — die Methode wurde im Kontext von `A` ausgewertet, so dass `R` automatisch zu `A::R` expandiert wird. Da durch implizite Vererbung sichergestellt wird, dass auch der spezifischere Typ `B::R` existiert und konform zu `A::R` ist, bietet Ruby Object Teams die Möglichkeit den spezifischeren Typ zu deklarieren. Der spezifischste Typ ergibt sich aus dem jeweils umschließenden Team: in `A` ist der spezifischste Typ von `R` gerade `A::R`, in `B` ist es `B::R`. Ruby Object Teams führt einen *Team-Scope-Modifikator* ein, der auf den Laufzeittyp des aktuellen Teams zeigt und `myTeam` heißt. Wird in der imaginären Methode in `A::R` also die Erzeugung eines Objektes vom Typ `myTeam::R` definiert, so erzeugt diese Methode unterschiedliche Objekte, abhängig vom umschließenden Team: in Team `A` werden Objekte vom Typ `A::R` erzeugt, in Team `B` Objekte vom Typ `B::R`. Dieses Prinzip funktioniert ganz analog zu virtuellen Methoden, wo je nach Laufzeittyp die speziellste Methode in einer Vererbungshierarchie aufgerufen wird. Aus diesem Grund heißt die Technik der Auswahl der spezifischsten Klasse *virtuelle Klasse* (*'virtual class'*).

Die Implementierung von Object Teams für `java` [Binder und Hundt 2002] nutzt eine erweiterte Form der impliziten Vererbung. Hier werden die einzelnen Rollenklassen in das erbende Team kopiert (*'copy inheritance'*) ohne eine Subtypbeziehung zu bilden. Diese Form der Vererbung ermöglicht es einer Rollenklasse eine andere Klasse zu verfeinern. Diese Möglichkeit besteht in Ruby Object Teams nicht, da Rollenklassen in einer Teamverfeinerung automatisch eine Vererbungsrelation haben.

3.2.1.2 Implementierung von Rollen

Rollen werden als innere Klassen von umschließenden Teamklassen implementiert. Da die Möglichkeit bestehen muss, Instanzen von Rollenklassen automatisch erzeugen zu können, ist es nötig, dass alle Rollenklassen einen Standardkonstruktor implementieren. Ist eine spezielle Form der Initialisierung nötig, muss diese Funktionalität in eine eigene Methode ausgelagert und gesondert aufgerufen werden[1].

[1] Es besteht außerdem die Möglichkeit in den Prozess der Rollenerzeugung einzugreifen (siehe 3.2.3.3). Die Initialisierung mit Parametern ist trotzdem nicht möglich, da Rollen immer implizit erzeugt werden. Eine explizite Erzeugung würde das Wissen über die Existenz der Rolle in der Domäne voraussetzen.

Wie im folgenden noch genauer erläutert wird, kennt jede Rolle ihr umschließendes Team und ihre zugehörige Basis. Diese Information wird vom Laufzeitsystem gesetzt, so dass eine Rolle immer von deren Existenz ausgehen kann.

Ein wesentlicher Bestandteil von Rollen ist die Definition von offenen Funktionalitäten als abstrakte Methoden. Diese abstrakten Methoden definieren einen Vertrag, der erfüllt werden muss, damit die Rolle funktionieren kann. In Ruby gibt es das Konstrukt der abstrakten Methode nicht. Object Teams für Ruby führt aus diesem Grund ein neues Schlüsselwort ein: `expected`. Eine Rollenklasse kann also Methoden deklarieren, von denen sie erwartet, dass sie entweder implementiert oder durch einen Konnektor an eine Basis gebunden werden.

3.2.2 Konnektordefinition àla Ruby Object Teams

Ein Konnektor in Object Teams verfeinert ein Team und bietet die Möglichkeit Rollenklassen des beerbten Teams an Domänenklassen zu binden. Ein Konnektor ist also selbst auch ein Team. Die Bindungen von Rollenklassen an Domänenklassen heißen *Spielerklauseln*. Eine bestimmte Rollenklasse wird also von einer Domänenklasse *gespielt*. Die definierten Bindungen müssen nicht vollständig sein. Das heißt, dass ein Konnektor bestimmte Bindungen definieren kann, die in einem abgeleiteten Konnektor verfeinert werden. Da in einer Teamverfeinerung auch die Verfeinerung von Rollen vorgenommen werden können, gibt es keine harte Abgrenzung zwischen einem Konnektor und einem Team. Jedes Team kann also Rollen, Teammerkmale und Spielerklauseln definieren. Es ist trotzdem ein guter Programmierstil ein Team abstrakt zu definieren und in einer Teamverfeinerung die einzelnen Rollen an Domänenklassen zu binden. Erst dies ermöglicht die Wiederverwendbarkeit eines Teams.

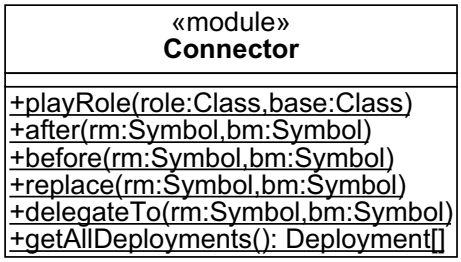

Abbildung 3.2: Statische Funktionalität zur Rollenbindung.

Die Funktionalität eines Konnektors ist in Ruby Object Teams im Modul `Connector` definiert, das man in Abbildung 3.2 sehen kann. Dieses Modul definiert statische Methoden, die den Bindungsmöglichkeiten in Object Teams entsprechen. Die Klasse `Team`, die Wurzel aller Teamklassen, importiert die Funktionalität aus diesem Modul. Die statische Funktionalität ist somit in jedem Team verfügbar. Wie in

Kapitel 3.1 beschrieben, können statische Methoden einer Klasse im statischen Block einer erbenden Klasse verwendet werden. So ist es in jeder Teamdefinition möglich, zusätzlich zu Definition von Rollen und Teammerkmalen, Bindungen zu deklarieren, indem die statischen Methoden aus `Connector` verwendet werden.

Eine Spielerklausel bindet eine Rolle an eine Domänenklasse. Diese Funktionalität übernimmt die Methode `playRole`, die als Parameter zwei Klassen bekommt: die Rollenklasse und die Domänenklasse die verbunden werden sollen. Zu jeder Spielerklausel können Bindungen auf Methodenebene definiert werden. Die Definition von Callin-Bindungen hat drei Ausprägungen: der Aufruf der Rollenmethode erfolgt: vor (*'before'*), nach (*'after'*) oder ohne (*'replace'*) Aufruf der Basismethode. Das Modul `Connector` definiert aus diesem Grund drei Methoden, die der jeweiligen Ausprägung entsprechen: `before`, `after` und `replace`. Alle drei Methoden bekommen als Parameter zwei Objekte vom Typ `Symbol`. Das erste Symbol kennzeichnet die Methode der Rolle die gebunden werden soll. Der zweite Parameter definiert die Methode der Basis, die zu der definierten Rollenmethode delegieren soll. Die Definition der Basismethode muss nicht eindeutig sein. Da die gleiche Rollenmethode an mehrere Methoden der Basis gebunden werden kann, darf als zweiter Parameter auch ein regulärer Ausdruck oder eine Liste aus Symbolen, bzw. regulären Ausdrücken verwendet werden. Diese Art der Spezifikation erlaubt die Auswahl einer Menge von Basismethoden. Die Callout-Bindung einer abstrakten Methode (*'expected method'*) der Rolle an eine Methode der Basis wird mit Hilfe von `delegateTo` ausgedrückt. Auch diese Methode bekommt als Parameter zwei Symbole. Hier muss die Basismethode aber eindeutig deklariert sein - es darf ja nicht mehrere Implementierungen einer abstrakten Methode in der selben Klasse geben. Das Listing 3.3 zeigt, wie die Definition eines Konnektors aussehen könnte.

```
class MyConnector < MyTeam                                          1
   playRole(RoleClass, BaseClass) {                                 2
      replace(:roleMethod1, :baseMethod1)                           3
      before(:roleMethod2, /set.*/)                                 4
      after(:roleMethod3, [:getThis, :getThat])                     5
      delegateTo(:expectedMethod, :baseMethod2)                     6
   }                                                                7
end                                                                 8
```

Listing 3.3: Ein Beispielkonnektor

Der Konnektor verfeinert ein Team (Zeile 1) und bindet die dort definierte Rollenklasse an eine Klasse der Domäne (Zeile 2) mit Hilfe von `playRole`. Diese Methode bekommt zusätzlich einen ausführbaren Block übergeben, welcher alle Callin- und Callout-Bindungen dieser spielt-Beziehung enthält (Zeile 3-6). Die einzelnen Methodenbindungen haben dabei folgende Bedeutung: In Zeile 3 wird deklariert, dass die Basismethode `baseMethod1` durch die Rollenmethode `roleMethod1`

ersetzt wird. Die Bindung in Zeile 4 deklariert einen regulären Ausdruck. Alle Methoden der Basis die etwas setzen, deren Name also mit „set“ beginnt, rufen vorher die Rollenmethode `roleMethod2` auf. Zeile 5 stellt sicher, dass nachdem `getThis` oder `getThat` gerufen wurde auch die Rollenmethode `roleMethod3` gerufen wird. Die abstrakte Rollenmethode `expectedMethod` wird durch die Methode `baseMethod2` implementiert (Zeile 6).

3.2.2.1 Deploymentdefinition

Wird eine Teamklasse geladen, welche Spielerklauseln enthält, so werden die entsprechenden statischen Methoden im Modul `Connector` automatisch aufgerufen. Jede einzelne Methode generiert einen Deskriptor, welche die deklarierte Funktionalität als Objekt kapselt. In Abbildung 3.3 sieht man die Struktur der einzelnen Deskriptoren. Jede Definition einer Spielerklausel erzeugt ein Objekt vom Typ `DeploymentDescriptor`. Objekte dieser Klasse speichern Referenzen auf die de-

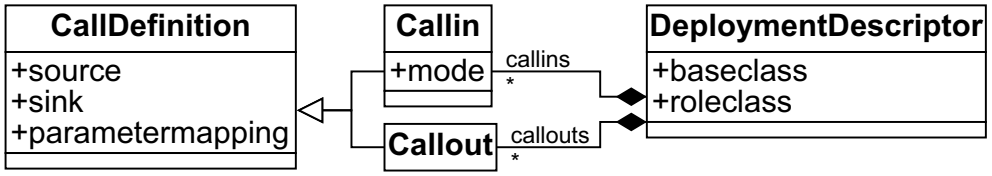

Abbildung 3.3: Kapselung der Spielerdefinitionen.

klarierten Klassen: Rollenklasse und Basisklasse. Zu solch einem Deskriptor gehört auch eine Liste von Methodenbindungen, also Callin- bzw. Callout-Bindungen. Eine Methodenbindung wird als Objekt der Klasse `CallDefinition` modelliert. Die Klasse hält Referenzen auf zwei Symbole — genau den Symbolen, die bei einer Methodenbindung deklariert werden. Die Semantik dieser beiden Symbole ist erst durch die Verfeinerung durch die Klassen `Callin` und `Callout` gegeben. Sie geben der Definition von Quelle und Senke erst eine Richtung und somit eine Bedeutung. Die Methoden `before`, `after` und `replace` erzeugen also Objekte vom Typ `Callin`. Um die jeweilige Ausprägung dieser Callin-Definition zu unterscheiden, kapselt diese Klasse den Typ in einem eigenen Attribut (mode). Das erzeugte Objekt wird in dem aktiven Deskriptor als Callin-Definition registriert. Die Methode `delegateTo` erzeugt Objekte vom Typ `Callout` und macht diese dem aktiven Deskriptor verfügbar. Sind alle Methodenbindungen ausgewertet, wird der aktive Deskriptor in einer statischen Variable in `Connector` registriert. Die Registrierung erfolgt mit der Typinformation des aktuell ausgewerteten Teams.

Rollenklassen und Domänenklassen werden in der Regel unabhängig voneinander modelliert und entwickelt. Dies kann zur Folge haben, dass die Signatur bei einer

Delegation angepasst werden muss (siehe 2.1.3.4). Alle Methoden mit der man eine Callin bzw. Callout Bindung deklarieren kann, akzeptieren aus diesem Grund einen weiteren Block, mit der die Parameter angepasst werden können. Eine Definition wie z.B. `delegateTo(:r, :b){|p1,p2| [p2]}` sorgt dafür, dass bei Aufruf der abstrakten Rollenmethode `r`, von den beiden übergebenen Parametern p1 und p2 nur der Parameter p2 an die Basismethode `b` delegiert wird. In diesem Beispiel wird nur ein Parameter weggelassen. Dieser Mechanismus kann aber auch für komplexe Transformationen genutzt werden, indem die Transformation einfach im Block deklariert wird. Dieser Block wird als Objekt in die jeweilige Methode gereicht und in der erzeugten Instanz von `Callin` bzw. `Callout` gespeichert (parametermapping).

Während eine Konnektorklasse geladen wird, werden also alle Spielerklauseln und Methodenbindungen in Objekte transformiert und in einer statischen Variable verfügbar gemacht. Nachdem die Klasse geladen ist, kann mit Hilfe der statischen Methode `MyTeam.getAllDeployments` auf eine Liste mit allen Deskriptoren zugegriffen werden, die für `MyTeam` definiert wurden. Ein Team, welches ein anderes Team verfeinert, erbt nicht nur dessen Merkmale und Rollen, sondern auch alle definierten Spielerklauseln und Methodenbindungen. Die Menge der Deskriptoren eines Teams ergibt sich also aus allen Deskriptoren in der Klassenhierarchie eines bestimmten Teams bis zu ihrer Wurzel, welche durch das Basisteam `Team` definiert ist.

3.2.2.2 Interzeption von Basismethoden

Die Definition von Spielerklauseln in einem Team führen dazu, dass sich die deklarierten Basisklassen *unter Umständen* anders verhalten, als die eigentliche Implementierung besagt. Dieser Umstand ist definiert durch die Aktivierung einer Instanz eines Teams, welche Rollen an die jeweilige Klasse bindet, die Callin-Definitionen enthält. Eine Callin-Definition verändert ja das Verhalten einer bestimmten Methode, indem Funktionalitäten hinzugefügt werden. Es besteht also eine Abhängigkeit zwischen dem Verhalten bestimmter Methoden einer Basisklasse und einem Team. Die betroffenen Methoden ergeben sich aus den definierten Callin-Bindungen dieses Teams.

Ruby Object Teams nutzt die Technik der Interzeption, um diese Abhängigkeit auszudrücken. Zur Erklärung dieser Technik ist es günstig, einen Methodenaufruf als Nachricht zu verstehen, die von einem Sender zu einem bestimmten Empfänger gesendet wird. Der Nachrichtenfluß ist immer direkt, der Sender kennt den Empfänger. Wird ein Team aktiviert, verändert sich das Verhalten des Empfängers. Dieselbe Nachricht resultiert in einem veränderten Verhalten. Dies wird realisiert, indem bestimmte Nachrichten am Empfänger abgefangen werden. Eine gesendete Nachricht wird zu einer anderen Methode umgeleitet. In der umgeleiteten Methode, kann die

eigentliche Empfängermethode aufgerufen werden. Es ist aber auch zusätzlich möglich, weitere Funktionalität zu implementieren. In Object Teams wäre solche zusätzliche Funktionalität das liften der Basis zu ihrer Rolle und die Delegation zu einer spezifizierten Rollenmethode. Die Reihenfolge der Aufrufe, ist durch den Typ der Delegation spezifiziert:

- `before`:
  - rufe Rollenmethode
  - rufe ursprüngliche Nachricht
- `after`:
  - rufe ursprüngliche Nachricht
  - rufe Rollenmethode
- `replace`:
  - rufe Rollenmethode

Die Technik der Interzeption wird als Umleitung realisiert und verändert nicht die ursprüngliche Methode. Die Umleitung erlaubt das hinzufügen von Funktionalität. Bei der Deaktivierung eines Teams muss nur die Umleitung entfernt werden, so dass sich der Empfänger wie gewohnt verhält.

Dieser Mechanismus wird in Object Teams für Ruby durch Aliasing (siehe alias in Kapitel 3.1) realisiert. Die in einer Callin-Bindung spezifizierte Methode einer Domänenklasse wird umbenannt und unter einem neuen Namen verfügbar gemacht. Sie ist fortan von außen nicht mehr zugreifbar. Unter dem ursprünglichen Namen wird eine neue Methode definiert, deren Inhalt sehr prototypisch ist und generiert werden kann. Der implementierte Algorithmus hat ein Schema, wie in Abbildung 3.4 zu sehen: Ein Aufruf der Methode findet nun die neue Implementierung.

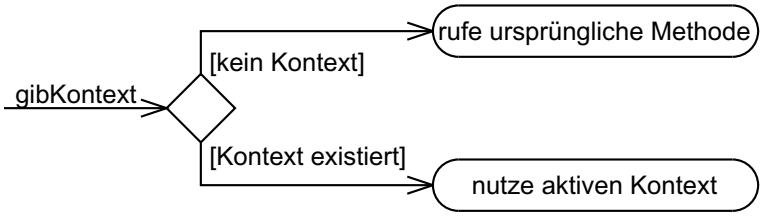

Abbildung 3.4: Interzeptionsmechanismus für Methoden.

Diese sucht nach einem Kontext. Ist ein Kontext aktiv, regelt dieser das weitere Verhalten. Wird kein Kontext gefunden, wird die ursprüngliche Methode, die nun unter einem anderen Namen zugreifbar ist, aufgerufen. Die Existenz eines Kontextes entspricht also einer Umleitung. Ist es möglich, einen Kontext für eine bestimmte

Methode zu hinterlegen, so ändert sich deren Verhalten. Eine Klasse mit solch präparierten Methoden ist in ihrem Verhalten ohne einen aktiven Kontext nicht von der ursprünglichen Definition zu unterscheiden.

Die Technik der Methodenadaption wird pro Methode und Klasse höchstens einmal durchgeführt — eine schon ersetzte Methode wird nicht noch einmal ersetzt. Wird ein Team geladen, welches Spielerklauseln mit Methodenbindungen enthält, werden die einzelnen Bindungen also nicht nur in Objekte transformiert, sondern zusätzlich auch alle nicht adaptierten Methoden der jeweiligen Basisklassen präpariert, zu denen eine Callin-Bindung definiert wurde. Ist das Team geladen, sind alle Methoden der Basisklassen verändert, zu denen dieses Team eine Callin-Bindung definiert. Diese Methodenveränderung bewirkt keine Verhaltensänderung sondern erweitert die Methode um einen Kontextmechanismus. Dieser Kontextmechanismus kann zur Verhaltensänderung genutzt werden.

3.2.3 Aktivierung der definierten Adaptionen

Ein Team wird effektiv, wenn alle abstrakten Teammerkmale implementiert sind und alle enthaltenen Rollen effektiv sind. Eine Rolle heißt effektiv, wenn alle abstrakten (*'expected'*) Methoden entweder implementiert sind oder an Methoden der Basis gebunden wurden. Ein effektives Team kann instantiiert werden. Zu einem bestimmten Team gehören alle erzeugten Deskriptor-Objekte, die zu dieser Teamklasse und allen Superklassen definiert wurden. Jedes Deskriptor-Objekt repräsentiert eine Bindung eines bestimmten Rollenverhaltens an eine Klasse der Domäne. Die Aktivierung dieser Bindung ist an die Aktivierung der Teaminstanz gebunden.

Object Teams für Ruby definiert zwei Möglichkeiten der expliziten Aktivierung einer Teaminstanz durch Teammerkmale im Basisteam. Den syntaktisch strukturierteren Mechanismus kann man in Listing 3.4 sehen. Die Methode `while_active`

```
team = MyTeam.new() # Erzeuge ein neues Team.
team.while_active {
  # team ist in diesem Block aktiv.
}
# team ist inaktiv.
```

Listing 3.4: Strukturierte Teamaktivierung

definiert klare Kontextgrenzen. Wird der Block betreten, wird das Team aktiviert. Innerhalb des definierten Blocks sind alle definierten Rollen- und Methodenbindungen aktiv. Beim verlassen des Blocks wird das Team deaktiviert - alle definierten Bindungen werden aus den Basisklassen entfernt. Die syntaktische Kopplung der Teamaktivierung an einen Block erleichtert die Übersichtlichkeit.

Es kann ein Rollenverhalten gekapselt werden, dessen Aktivierung und Deaktivierung von externen Ereignissen abhängt. In diesem Fall kann der strukturierte Weg über `while_active` nicht genommen werden. Die Funktionalität zerfällt in zwei Einzelaktionen, die separat voneinander genutzt werden können: `activate` aktiviert eine Instanz eines Teams, `deactivate` deaktiviert es.

Eine Aktivierung eines Teams hat die Synthese aller Callin-Bindungen aller definierten Spielerklauseln zur Folge. Diese Synthese erfolgt auf Klassenebene! Das heißt, das Verhalten der definierten Domänenklassen wird adaptiert. Diese Adaption wirkt sich also implizit auf alle Instanzen dieser Klassen aus. Jede einzelne Instanz solcher Klasse wird in dem aktuellen Team durch eine eigene Rolle repräsentiert. Der Typ der Rolle ergibt sich aus der zugehörigen Spielerklausel. Die definierten Callout-Bindungen manifestieren sich in der Rolle und sind von der Aktivierung/Deaktivierung nicht betroffen. Ein kontextabhängiges Rollenverhalten bezieht sich immer auf die Domänenklassen. Dort wird ein Verhalten adaptiert. Die Funktionalität der Rollenklassen ist dabei von statischer Natur.

3.2.3.1 Realisierung von Callin-Bindungen

Zu einer Teaminstanz gehören eine Menge von Deskriptoren, die ein bestimmtes Rollenverhalten für Klassen der Domäne spezifizieren. Die Deskriptoren fungieren als Vorlagen oder Schablonen, da es mehrere Instanzen der gleichen Teamklasse und somit desselben Rollenverhaltens geben kann. Ein Bindungsdeskriptor entspricht außerdem nicht notwendigerweise genau einer Bindung, da auch Listen von Methoden oder sogar reguläre Ausdrücke zur Bindungsspezifikation genutzt werden können. Bei einer Aktivierung eines Teams müssen alle Callin-Deskriptoren ausgewertet werden und Kontextobjekte für jede einzelne Bindung erzeugt werden. Diese Kontextobjekte müssen bei der Domänenklasse für eine bestimmte Methode hinterlegt werden, so dass sich diese Methode für die Zeit der Aktivierung anders verhält.

Zur Modellierung dieser Kontexte führt Ruby Object Teams die Klasse `Call` und ihre Spezialisierungen ein (siehe Abbildung 3.5). Die Klasse `Call` fungiert als *Adapter* [Gamma u. a. 1996, Adapter] und kapselt eine Methode. Die gekapselte Methode kann über eine einheitliche Schnittstelle (`call`) aufgerufen werden und erhält den Empfänger und die übergebenen Parameter der Methode als Argumente. Sie ist abstrakt deklariert, da an dieser Stelle keine Aussage gemacht werden kann, was bei einem Aufruf geschehen soll.

Die Klasse `MethodCall` ist ein einfacher Wrapper für eine Methode der Basis. Die abstrakte Methode `call` wird implementiert, indem die gekapselte Nachricht an das übergebene Objekt gesendet wird. Um eine Bindung einer Rollenmethode an eine Basismethode zu modellieren, gibt es die Klasse `WeavedCall`. Diese Klasse definiert die Methode `callRole`. Der Empfänger der gekapselten Nachricht ist

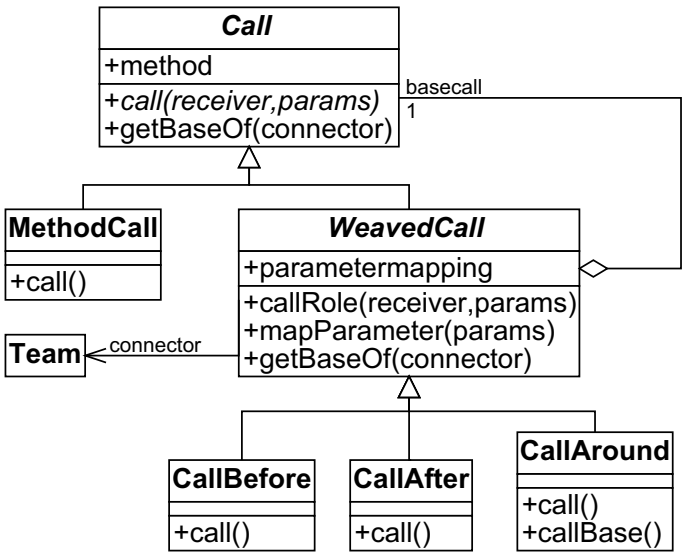

Abbildung 3.5: Call Hierarchie

nicht das übergebene Basisobjekt, sondern die Rolle dieser Basis. Die Basis muss zu ihrer Rolle geliftet werden, bevor die Nachricht versendet wird. Das Lifting ist aber nur im Kontext eines bestimmten Konnektors definiert, nämlich genau jenem Konnektor, der diese Callin-Bindung aktiviert hat. Aus diesem Grund gibt es eine explizite Referenz auf das Team, welche verantwortlich für die Methodenbindung ist. Eine Rollenmethode wird immer an eine Methode der Basis gebunden. Da Methoden der Basis auch als `Call`-Objekte modelliert werden, gibt es eine Referenz auf ein weiteres `Call`-Objekt, welches die Basismethode repräsentiert. Die Klasse `WeavedCall` lässt die Bindungsart der Rollenmethode an die Basismethode offen und bleibt aus diesem Grunde abstrakt. Die drei Callin-Bindungstypen werden durch einzelne Klassen repräsentiert, die die Klasse `WeavedCall` verfeinern. Sie implementieren die abstrakte Methode `call`, indem sie die Aufrufreihenfolge von `callRole` und `basecall.call` variieren, um dem jeweiligen Bindungstyp zu entsprechen.

Bei der Aktivierung eines Teams erzeugt jeder Callin-Deskriptor mindestens ein `Call`-Objekt, welches der jeweilig deklarierten Bindung entspricht. Wurde als Definition der Basismethode eine Liste von Methoden oder ein regulärer Ausdruck angegeben, entspricht die Anzahl der passenden Methoden der Zahl der erzeugten `Call`-Objekte. Jedes dieser Objekte wird bei der betroffenen Klasse für die jeweilige Methode hinterlegt. Das Laufzeitsystem hat sichergestellt, dass die jeweilige Methode abgefangen und umdefiniert wurde. Ein Aufruf an einem Objekt der Basisklasse würde nun einen aktiven Kontext, nämlich das hinterlegte `Call`-Objekt, finden. Die umdefinierte Methode würde an dieser Stelle nicht die ursprüngliche Methode aufrufen, sondern das `Call`-Objekt benutzen und `call` aufrufen. Als Ar-

gumente werden dabei immer das aufgerufene Objekt (self) und die übergebenen Parameter der Methode mitgeliefert.

Object Teams definiert keine Schranken für die Definition von Bindungen. Es ist möglich eine Rollenmethode an verschiedene Basismethoden zu binden, es ist aber auch möglich unterschiedliche Rollenmethoden an die gleiche Basismethode zu binden. Im ersten Fall wird jeweils ein `Call`-Objekt generiert und bei den definierten Methoden hinterlegt. Im zweiten Fall ist aber nur eine Methode betroffen. Was passiert mit den einzelnen Bindungen? Und was bedeutet in diesem Kontext davor oder danach? Das referenzierte Basisobjekt von `WeavedCall` in Abbildung 3.5 ist vom abstrakten Typ `Call`. Es muss also nicht notwendigerweise die Basismethode direkt sein. Eine Referenz auf das Methodenobjekt liefert die betroffene Basisklasse. Ist für die jeweilige Methode bereits ein `Call`-Objekt hinterlegt, wird dieses Objekt anstatt der Methode zurückgegeben. Ist kein `Call`-Objekt hinterlegt, wird ein neues vom Typ `MethodCall` erzeugt. Die Referenz auf die Basismethode wird in dem neuen `Call`-Objekt gesetzt und für die jeweilige Methode registriert. Sie stellt damit für alle weiteren Bindungen dieser Methode die Basismethode dar. Auf diese Art und Weise können bei der Aktivierung von Teams Ketten von `Call`-Objekten für eine Methode entstehen, die die gewünschte Bindung repräsentieren. Die Sicht einer Callin-Bindung auf die Basismethode ist dabei also sehr lokal, bezogen auf den jeweiligen Bindungszustand.

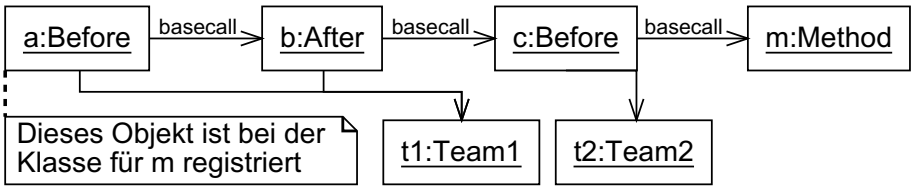

Abbildung 3.6: Mehrere Bindungen für die Methode „m“

Abbildung 3.6 zeigt ein Beispiel solch einer Kette für eine Methode `m` einer Domänenklasse. Die Instanz vom Typ `Team2` wurde zuerst aktiviert und definiert eine Callin-Bindung, deren Basismethode die ursprüngliche Methode m ist. Das Team vom Typ `Team1` definiert zwei Callin-Bindungen, die in der Reihenfolge der Definition in der Teamklasse aktiviert werden. Ein Aufruf der Methode `m` würde in dieser Konstellation folgende Aufrufreihenfolge zur Folge haben: `a` → `c` → `m` → `b`. Beide aktivierten Teams definieren eine `before`-Bindung. Da `t1` später als `t2` aktiviert wurde, hat die dort definierte Bindung Vorrang und wird zuerst ausgeführt. (Eine Aktivierung von `t1` vor `t2` hätte die Aufrufreihenfolge `c` → `a` → `m` → `b` zur Folge.)

3.2.3.2 Aufrufdelegation der Rolle an die Basis

Domänenklassen können verschiedene Rollen in verschiedenen Teams spielen. Die einzelne Domänenklasse ist über `Call`-Objekte an ihre Rollen gebunden. Die Funktionalität zur Verwaltung solcher `Call`-Objekte ist in dem Modul `Base` definiert. Dieses Modul erweitert die Metaklasse `Class`, so dass die Modulfunktionalität in allen Klassen verfügbar ist. Da die Methoden statisch deklariert sind, kann auch nur statisch auf diese in den einzelnen Klassen zugegriffen werden (z.B. `MyClass.getCallin(:test)`). Jedes einzelne `Call`-Objekt hält eine Referenz auf das Team, welches für diese Bindung verantwortlich ist.

Ein einzelnes Rollenobjekt hat immer genau eine Basis. Jede Klasse, die eine Rolle in einem Team spielt wird automatisch um das Modul `Role` erweitert. In diesem Modul ist ein verstecktes Attribut (`__base`) definiert, welches vom Laufzeitsystem auf das aktuelle Basisobjekt gesetzt wird. Somit kennt jedes Rollenobjekt sein zugehöriges Basisobjekt. Aus Gründen die in Kapitel 3.2.3.3 besprochen werden, läuft die Kommunikation zwischen Rolle und Basis nicht direkt, sondern über einen Vermittler [Gamma u. a. 1996, Proxy]. Dieser Vermittler übernimmt das Lifting und Lowering, der Rolle bzw. der Parameter. Die Zuordnung von Basisobjekt zu Rollenobjekt übernimmt dabei das jeweilige aktive Team. Es ergibt sich somit eine Struktur, die in Abbildung 3.7 zu sehen ist.

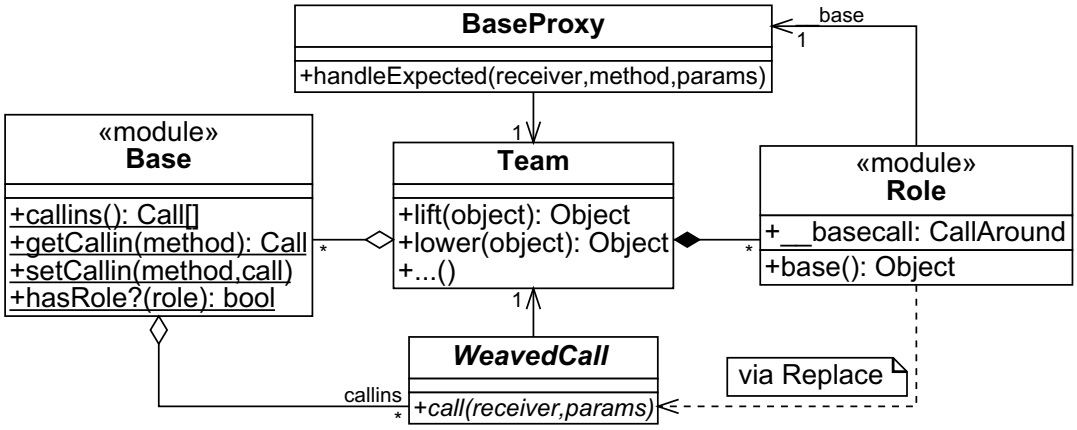

Abbildung 3.7: Beziehung zwischen Basis und Rolle

Callout Eine Rolle kann Methoden abstrakt deklarieren. Die Implementierung solcher abstrakten Methode kann in einer abgeleiteten Klasse erfolgen, oder an eine Methode der Basis gebunden werden. Abstrakte Methoden werden in Object Teams für Ruby mit dem Schlüsselwort `expected` deklariert. Dieses Schlüsselwort sorgt dafür, dass beim Laden der Klasse die deklarierte Methode implementiert wird. Die

Nachricht wird an ein Stellvertreterobjekt der Basis, welches über ein verstecktes Attribut zugreifbar ist, weitergeleitet. Im Stellvertreterobjekt können nun Adaptionen vorgenommen werden, die im zugehörigen Konnektor definiert wurden. Die Methode der Basis kann beispielsweise einen anderen Namen oder eine andere Signatur tragen, die angepasst werden müssen.

Replacement Die drei Typen von möglichen Callin-Bindungen unterscheiden sich in der Definition des Zeitpunktes, in dem die Rollen- bzw. Basismethode ausgeführt wird. Bei einer `before`-Bindung, wird die Rollenmethode vor der Basismethode ausgeführt, bei einer `after`-Bindung ist die Reihenfolge genau vertauscht und bei einer `replace`-Bindung ist der Zeitpunkt völlig offen gelassen. Die Basismethode wird durch eine Rollenmethode ersetzt — diese entscheidet, ob und wann die Basismethode zur Ausführung kommt. Die Rolle kann innerhalb solch einer ersetzten Methode das Schlüsselwort `base` benutzen, welches die jeweils überschriebene Methode, also die Basismethode, ausführt. Da die gleiche Rollenmethode an mehrere Methoden der Basis gebunden werden kann, ist zur Erzeugungszeit der Rolle nicht eindeutig definiert, was mit `base` gemeint ist. Eine valide Bindung von `base` ist also immer nur während der Ausführung einer `replace`-Bindung gegeben, indem die jeweils ersetzte Methode zur Ausführung kommt. Methoden die das Schlüsselwort `base` benutzen, haben eine gesonderte Stellung, da diese Methoden von der Rolle selbst nicht aufgerufen werden können.

Jede Rollenklasse wird um die Methode `base` und ein weiteres verstecktes Attribut `__basecall` erweitert (Modul `Role`). Die Klasse `CallAround` lässt dieses Attribut auf sich selbst zeigen (siehe Abb. 3.5), bevor es an die jeweilige Rollenmethode delegiert. Nachdem es zur Ausführung der Rollenmethode kam, wird diese Referenz wieder gelöscht. Wird in einer Rollenmethode die Methode `base` aufgerufen, wird erst einmal kontrolliert, ob das versteckte Attribut überhaupt eine gültige Referenz hält. Ist dies nicht der Fall, kommt es zu einem Fehler. Solch ein Fehler kann also nur auftreten, wenn die jeweilige Methode nicht über eine `replace`-Bindung aufgerufen wurde. Wird eine gültige Referenz gefunden, wird die Methode `callBase` an der jeweilig aktiven Instanz von `CallAround` aufgerufen, welche die jeweilige Basismethode ausführt und das Ergebnis an die Rollenmethode reicht (siehe Abb. 3.7).

3.2.3.3 Lifting/Lowering

Die Bindung einer Rolle an eine Basis findet immer im Kontext eines bestimmten Konnektors statt. In einem Team können Spielerklauseln definiert werden, die eine Relation einer Domänenklasse zu einer Rollenklasse beschreiben. Ein Team stellt die Relation zwischen Instanzen der Rolle und der Basis sicher und definiert Transformationsmechanismen, die als Lifting und Lowering (siehe Kapitel 2.1.4) beschrieben wurden. Um diese Transformation nutzbar zu machen, stellt die Klasse Team

die Methoden `lift` und `lower` zur Verfügung. Ist für eine Klasse der Domäne eine Spielerklausel definiert, so liefert die Methode `lift` das zugehörige Rollenobjekt. Man erhält das jeweilige Basisobjekt mit der Methode `lower`, wenn das Rollenobjekt übergeben wird.

Für jede Callin- bzw. Callout-Delegation ist eine Instanz eines bestimmten Teams verantwortlich. Bei einer Callin-Delegation muss das Basisobjekt plus allen übergebenen Parametern geliftet werden, wenn die Klassen der einzelnen Objekte eine Rolle im aktuellen Team spielen. Der Rückgabewert muss gegebenenfalls gelowered werden. Bei einer Callout-Delegation oder einem `base`-Aufruf muss das Rollenobjekt und alle Parameter, die Rollen des aktuellen Teams sind, gelowered, der Rückgabewert unter Umständen gelifted werden. Da Ruby eine ungetypte Sprache ist, kann aufgrund der ungetypten Methodensignaturen nicht überprüft werden, ob für die jeweiligen Parameter eine Spielerklausel vorliegt. Aus diesem Grund gibt es keine direkte Kommunikation zwischen Basis und Rolle. Für beide Kommunikationspfade gibt es ein Vermittlerobjekt, welche die Trennung der beiden Welten sicher stellt. Bei einer Delegation der Basis an die Rolle ist ein `Call`-Objekt verantwortlich, in der umgekehrten Richtung wird diese Aufgabe von der Klasse `BaseProxy` übernommen. Sie stellen das Lifting aller Basisobjekte, bzw. das Lowering aller Rollen sicher.

Der Lifting-Mechanismus läuft in drei Stufen ab: (1) es wird in einem assoziativen Speicher nachgeschaut, ob für das gegebene Basisobjekt ein Rollenobjekt existiert. Ist dies der Fall, wird die gefundene Rolle zurückgeliefert. War die Suche nicht erfolgreich, wird nach einer passenden Spielerklausel gesucht, die auf den Typ des jeweiligen Objektes passt (2). Die Typinformation wird dabei polymorph behandelt. Gibt es mehrere passende Spielerklauseln, wird diejenige gewählt, die in der spezialisiertesten Teamdefinition gefunden wurde. Es ist möglich, dass als Ergebnis dieser Suche keine Klasse gefunden wird. In diesem Fall wird das ursprüngliche Objekt zurückgeliefert, da für diesen Typ keine Rollen definiert wurden. Wurde eine Klasse gefunden (3), wird eine neues Objekt dieser Klasse instantiiert. Dieses neue Objekt wird zu dem Basisobjekt assoziiert, so dass eine erneute Suche bei Schritt (1) endet. Die Instantiierung eines neuen Rollenobjektes ist also immer die Folge einer Lifting-Operation, welche üblicherweise bei einer Methodendelegation implizit ausgeführt wird. Das Rollenobjekt zu einem Objekt der Domäne wird also erst kreiert, wenn eine Methode aufgerufen wird, für die eine Callin-Bindung vorliegt.

Der Benutzer eines Teams hat verschiedene Möglichkeiten, in den Prozess der Rolleninstantiierung einzugreifen, um eine spezifischere Logik zu implementieren. Eine Teamaktivierung bedeutet immer eine Adaption auf Klassenebene. Das heißt, dass alle Instanzen einer bestimmten Klasse bei einer Callin-Bindung automatisch eine Rolle assoziiert bekommen. Dies ist unter Umständen nicht unbedingt erwünscht. So bietet die Klasse `Team` die Möglichkeit, den Mechanismus des Liftings für ein bestimmtes Team abzuschwächen (*'weak activation'*). Wird an einem Objekt

einer Domänenklasse eine Callin-Definition mit einem schwachen Konnektor vorgefunden, wird das Objekt nur noch geliftet, wenn eine Rolle schon erzeugt wurde. Ist dies nicht der Fall, wird keine neue Rolle erzeugt — das Rollenverhalten wird für dieses Objekt ausgesetzt. Es ist mit dieser Technik möglich ein System in einen initialen Zustand zu versetzen und den Prozess der Rollenerzeugung für den eingeschwungenen Zustand abzuschalten. Das definierte Rollenverhalten wird so nur für eine Teilmenge der Objekte einer Klasse wirksam. Möchte man den Prozess der Rollenerzeugung noch genauer kontrollieren und z.B. am Zustand des Basisobjektes entscheiden, ob eine Rolle erzeugt werden soll oder nicht, kann dies nicht mehr mit Hilfe von Teammerkmalen unterstützt werden. Der Programmierer hat dann die Möglichkeit in der Teamdefinition die Methode `createRole` zu spezialisieren, die als Argumente das Basisobjekt und die Rollenklasse bekommt, für die eine Rolle erzeugt werden soll. Hier kann nun die spezifische Logik für spezifische Basisklassen implementiert werden.

Das Lowering einer Rolle ist nur in dem Team definiert, indem die Rolle erzeugt wurde. Jede erzeugte Rolle in einem Team wird in einen assoziativen Speicher geschrieben (siehe Abbildung 3.7). Dies ist für die Funktionalität des Liftings unerlässlich und kann auch zum Lowering benutzt werden. Eine Suche nach der passenden Basis besteht im Gegensatz zum Lifting nur aus einem Schritt. Das Rollenobjekt wird im assoziativen Speicher gesucht und zurückgegeben. Wird keine passende Basis für die Rolle gefunden, so ist das jeweilige Objekt keine Rolle des Teams und kann nicht gelowered werden.

3.2.4 Deaktivierung eines Konnektors

Aktivierung und Deaktivierung eines Teams beziehen sich immer auf Callin-Bindungen, die in diesem Team definiert sind. Die Aktivierung eines Teams verläuft sehr strukturiert. Die einzelnen Methodenbindungen werden als `Call`-Objekte modelliert und bei der Klasse registriert. Es gibt immer genau ein `Call`-Objekt pro Methode, welches zu einer Liste verkettet werden kann. Diese Kette wird immer vom Kettenanfang an erweitert, so dass das jeweils aktuell aktivierte Team völlige Kontrolle über die Aufrufabfolge hat.

Wird ein Team deaktiviert, bedeutet dies das Entfernen aller registrierten `Call`-Objekte des jeweiligen Teams bei den einzelnen Basisklassen. Die Deaktivierung eines Teams ist zu jeder Zeit im Ausführungspfad möglich. Es spielt dabei keine Rolle, welches Team wann aktiviert wurde. Eine Aufrufkette muss also nicht genau so abgebaut werden, wie sie aufgebaut wurde (so wie es der Fall wäre, wenn man nur mit `while_active` operiert). Eine solche Kette muss unter Umständen vervollständigt werden, wenn zu dem deaktiviertem Team `Call`-Objekte gehören, die mitten in solch einer Kette stecken. Würde man das Teams `t2` der Abbildung 3.6 von S. 38 deaktivieren, müsste das Methodenobjekt `m` an Das Callobjekt `b` gebunden

werden, da dieses den neuen Basisaufruf darstellt. Um Zugriff auf das jeweilige Basisobjekt zu ermöglichen, definiert die Klasse `Call` die Methode `get_base_of`, die das nächste Aufrufobjekt liefert, welches nicht zum aktuell deaktivierten Team gehört.

Object Teams bietet eine zweite Möglichkeit ein Team kurzzeitig zu deaktivieren (via `suspend`). Diese kurzzeitige Deaktivierung veranlasst keine Veränderung von Aufrufketten, sondern unterdrückt jede weitere Delegation einer Basismethode an eine Rolle des aktuellen Teams. Wird solch ein deaktiviertes Team wieder aktiv (via `resume`), wird exakt das selbe Rollenverhalten wirksam. Diese Möglichkeit kann von Rollenklassen genutzt werden, um die gebundene Funktionalität für einen kurzen Zeitraum auszuschalten.

Hier wird ein Problem adressiert, dass bekannt geworden ist unter dem Namen '*Jumping Aspects*' [Brichau u. a. 2000]. Die Autoren argumentieren, dass sich die Bindungen in Abhängigkeit des Eintrittspunktes in den Kontext verändern, diese also „springen" zu scheinen. Als Beispiel wird die Implementierung einer Liste angeführt, die beobachtet werden soll. In der Listenimplementierung gibt es unter anderem die Methoden `addElement`, welche ein Element in die Liste einfügt und `addCollection`, welche für alle Elemente in der Liste `addElement` aufruft. Um diese Liste zu beobachten, bedarf es einer Bindung an `addElement`, wenn diese Methode direkt aufgerufen wird und einer Bindung an `addCollection`, wobei die Bindung zu `addElement` hier überflüssig ist, da nicht jede atomare Änderung von Bedeutung ist, sondern nur eine Änderung als ganzes. Die Bindungen scheinen sich also zu unterscheiden, je nachdem wie die Methode `addElement` aufgerufen wird – direkt oder von `addCollection`. Um dieses Problem zu lösen, kann in der Rollenklasse die Teamfunktionalität kurzzeitig ausgeschaltet werden. Es würde also in der Tat zwei Bindungen an `addElement` und `addCollection` geben. Die Bindung an `addCollection` könnte nun für den Zeitraum der Ausführung der Methode das Team deaktivieren (`suspend`). Alle Aufrufe der Methode `addElement` wären für diese Zeit von der Beobachterfunktionalität ausgenommen. Das Team wird nach Ausführung von `addCollection` wieder aktiviert (`resume`), so dass alle weiteren Aufrufe an `addElement` und `addCollection` beobachtet werden können.

Rollenklassen bilden nicht nur eine Teilfunktionalität einer Domänenklasse ab, sondern können auch einen Zustand kapseln. Wird ein Team aktiv, ist die Funktionalität und der Zustand der Rolle verfügbar. Wird ein Team deaktiviert, bleiben alle gelifteten Rollen der einzelnen Instanzen der Domänenklassen im Team erhalten. Bei einer erneuten Aktivierung des Teams, sind so die jeweiligen Rollen vorhanden, bleibt der gekapselte Zustand erhalten. Die Lebensdauer eines Rollenobjektes hängt von zwei Faktoren ab: der Lebenszeit des umschließenden Teams und der Lebenszeit des Basisobjektes. Wird eines der beiden finalisiert, wird auch die Rolle nicht mehr benötigt. Der erste Fall ist relativ einfach zu lösen. Da Rollen ihr umschließendes Team nicht verlassen können, müssen sie im Team gehalten werden. Wird das

Team finalisiert, werden automatisch ebenfalls alle Rollen finalisiert.

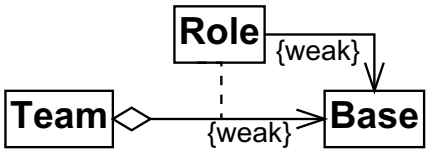

Abbildung 3.8: Basisobjekte dürfen nicht direkt referenziert werden

Bei einer automatischen Speicherbereinigung (*'Garbage Collector'*) bedarf der letztere Fall weiterer Automatismen. So darf kein Team ein Basisobjekt direkt referenzieren. Es muss dafür immer eine besondere Art der Referenzierung genutzt werden, die von der automatischen Speicherbereinigung nicht registriert wird. Solche Referenzen heißen schwache Referenzen (*weak references*). Ruby Object Teams hält die Relation von Basisobjekt zu Rollenobjekt in einer speziellen Form eines Hash, welche die Schlüsselobjekte mit solch einer schwachen Referenz hält (`WeakHash`). Wird ein Basisobjekt von der Applikation nicht mehr referenziert, kann es finalisiert werden, da es keine weiteren expliziten Referenzen mehr gibt. Existierende Einträge in einer Instanz von `WeakHash`, welche als Schlüssel das Basisobjekt referenzieren, werden automatisch mitgelöscht. Die Relation von Basis zu Rolle wird also automatisch entfernt, wenn das Basisobjekt finalisiert wird. Da es nun keine weiteren Referenzen auf das Rollenobjekt mehr gibt, kann auch dieses von der automatischen Speicherbereinigung finalisiert werden.

Eine Ausnahme dieser Regel bilden die externalisierten Rollenobjekte. Hier handelt es sich um Objekte, die direkt in der Domäne referenziert werden. Solange es eine Referenz auf ein externalisiertes Rollenobjekt gibt, muss auch das passende Basisobjekt dazu existieren, auch wenn dieses Objekt von der Anwendung nicht mehr referenziert wird. Um solche Objekte vor der Speicherbereinigung zu schützen, bedarf es einer direkten Referenz. Da hier eine Abhängigkeit von dem zugehörigen Rollenobjekt ausgeht, existiert ein zweiter Hash mit schwachen Referenzen (`WeakHash`). Als Schlüssel wird nun aber das externalisierte Rollenobjekt eingetragen und als Wert das Basisobjekt. Die Referenz wird automatisch aus dieser Liste ausgetragen, wenn es in der Anwendung keine Referenz mehr auf das externalisierte Rollenobjekt gibt. Ein externalisiertes Rollenobjekt muß über die Methode `asRole` angefordert werden, welche das Objekt in diesen Hash einträgt.

3.3 Syntax von Ruby Object Teams am Beispiel

In diesem Kapitel soll das Beispiel aus Kapitel 2.2.4 für Ruby Object Teams im Quellcode[2] gezeigt werden. Das Team `Observer` kann man in Listing 3.5 sehen. Diese

```
1  class Observer < Team
2     class Observer # Die Rolle des Beobachters.
3        expected :update # Diese abstrakte Methode soll gebunden werden.
4     end
5     class Subject # Die Rolle des Beobachteten.
6        attr :observer #Liste aller Beobachter
7        def initialize()
8           @observer=Array.new # Initialisiere eine neue Liste
9        end
10       def addObserver(observer)
11          @observer.push(observer)
12       end
13       def removeObserver(observer)
14          @observer.delete(observer)
15       end
16       def notify()
17          @observer.each { |observer| # Iterator
18             observer.update
19          }
20       end
21    end
22 end
```

Listing 3.5: Das Beobachtermuster in Ruby Object Teams

Klasse verfeinert kein vordefiniertes Team und erbt deshalb vom Basisteam `Team`. Diese Vererbung muß deklariert werden, sonst kann diese Klasse nicht als Team erkannt werden. In der Klasse `Observer` sind die beiden Rollen `Subject` und `Observer` definiert. Die Klasse `Subject` implementiert das bekannte Verhalten. Hier wird nur Ruby-typische Syntax benutzt. In der Methode `notify` (Zeile 16) wird über die Liste aller Beobachter iteriert (die Methode `each` [Zeile 17] bekommt einen Block übergeben, den sie mit jedem Element der Liste ausführt) und die Methode `update` der Klasse `Observer` aufgerufen. Diese Methode wurde in der Rolle mit dem Schlüsselwort `expected` abstrakt deklariert. Diese Methode muß in einem erbenden Team entweder verfeinert oder an eine Domänenklasse gebunden werden.

Listing 3.6 zeigt das Team `PlugObserver`, welches das Team `Observer` verfeinert. Da in dem beerbten Team 2 Rollen definiert wurden, sind diese durch impli-

[2]Dieses Beispiel ist in der Distribution von Ruby Object Teams [Veit 2002] enthalten

zite Vererbung auch automatisch im Team `PlugObserver` verfügbar. Die Definition der Klasse `PlugObserver::Observer` ist somit eine Verfeinerung der Klasse `Observer::Observer`. Da auch hier die Methode `update` nicht implementiert wurde, bleibt die Klasse und somit das gesamte Team abstrakt.

```
class PlugObserver < Observer # 2 Rollen werden geerbt                    1
   class Observer # Implizite Verfeinerung                                2
      def start(subject)                                                  3
         subject.addObserver(self)                                        4
      end                                                                 5
      def stop(subject)                                                   6
         subject.removeObserver(self)                                     7
      end                                                                 8
   end                                                                    9
end                                                                       10
```

Listing 3.6: Ein erweitertes Beobachtermuster

Die Implementierung der Klassen der Domäne `BookManager` und `BookCopy` sind sehr einfach und werden aus diesem Grund hier nicht gezeigt. Das Listing 3.7 zeigt die Bindung des Teams `PlugObserver` an die Domänenklassen. Es wird ein neues Team definiert, welches das Team `PlugObserver` verfeinert. Auch hier werden wieder zwei Rollen automatisch durch implizite Vererbung definiert, die aber nicht verfeinert werden. Eine Verfeinerung einer Rolle verbunden mit einer Bindung an eine Domänenklasse im selben Team ist prinzipiell möglich, wird in diesem Beispiel aber nicht benötigt.

Die Rolle `Subject` wird an die Domänenklasse `BookCopy` gebunden (Zeile 2). Bei dieser Bindung müssen keine abstrakten Methoden implementiert oder gebunden werden. Es muss aber sehr wohl deklariert werden, wann die Methode `notify` aufgerufen werden soll. Es gibt nur zwei Methoden, die den Zustand des Buches verändern `borrow` und `returnIt`. Hier wird von der Listenschreibweise Gebrauch gemacht: Die Methode `notify` wird an eine Liste von Methoden gebunden (Zeile 3). Methoden werden immer mit einem vorangestellten Doppelpunkt deklariert.

Die Rolle `Observer` wird von der Klasse `BookManager` gespielt. Hier muss die abstrakte Methode `update` gebunden werden. Die implementierende Methode in `BookManager` heißt `updateStatus` (Zeile 6). Für eine korrekte Funktionsweise der Beobachterstruktur muß sich der Beobachter auch bei dem Subjekt anmelden. Diese Funktionalität wird auch an Methoden der Basisklasse gebunden: sie wird gestartet, nachdem ein Buch gekauft wurde (Zeile 7) und beendet, bevor das Buch aus dem Bestand entfernt wird (Zeile 8). Alle Rolle im Team `ObserveLibrary` sind effektiv und somit auch das gesamte Team. Die hier definierte Bindung soll immer verfügbar sein. Aus diesem Grund wird sie statisch aktiviert (Zeile 11).

```
class ObserveLibrary < PlugObserver # 2 Rollen werden geerbt
   playRole(Subject, BookCopy) { # verhält sich wie Subject
      after(:notify, [:borrow, :returnIt])
   }
   playRole(Observer, BookManager) { # verhält sich wie Observer
      delegateTo(:update, :updateStatus)
      after(:start, :buy)
      before(:stop, :drop)
   }
end
Team.activate_static(ObserveLibrary)
```

Listing 3.7: Bindung an die Domäne Bibliothek

Kauft der Buchmanager nun ein neues Buch `buy(copy:BookCopy)` wird diese Methode an die Rolle vom Typ `Observer` delegiert. Die Instanz der Buchkopie wird dabei automatisch zur zugehörigen Rolle vom Typ `Subject` geliftet. Der Aufruf von `start` ist also Typsicher. In der Methode `start` meldet sich nun die `Observer`-Rolle des Buchmanagers bei der `Subject`-Rolle der Buchkopie als Beobachter an. Wird dieses Buch ausgeliehen oder zurückgegeben, wird die Methode `notify` der `Subject`-Rolle aufgerufen. Alle Beobachter werden nun benachrichtigt, indem die Methode `update` aufgerufen wird. Diese Methode wird an die Basis delegiert, so dass die Methode `updateStatus` am Basisobjekt vom Typ `BookManager` aufgerufen wird.

4 Object Teams am Beispiel eines Projektmanagementsystems

Dieses Kapitel evaluiert den praktische Nutzwert einer Object Teams basierten Lösung in der Modellierung und der daraus resultierenden Implementierung im Gegensatz zu einer rein objektorientierten Herangehensweise. Um eine angemessene Praxisrelevanz sicher zu stellen, wurde als Fallbeispiel eine Projektmanagementsoftware entwickelt. Diese Software steht stellvertretend für eine ganze Reihe von Problematiken, die sich in vielen Anwendungsentwicklungen wiederfinden. Drei Schwerpunkte sollen hierbei fokussiert werden:

- **Die Realisierung einer graphischen Schnittstelle:** In vielen modernen Anwendungen ist eine Interaktion zwischen Benutzer und System erforderlich. Die Möglichkeiten dieser Interaktion können dabei sehr komplex sein. Graphische Schnittstellen ermöglichen eine Abbildung der angebotenen Funktionalität auf graphische Standardelemente, die dem Benutzer vertraut sind.

  Für die Implementierung von graphischen Benutzerschnittstellen hat sich der Einsatz von Frameworks (so genannter graphischer *'Toolkits'*) durchgesetzt. Diese *'Toolkits'* beruhen auf einer funktionalen Trennung von Modell und Repräsentation. Diese Trennung ist Anlass für verschiedene konkurrierende Designziele, die konzeptionell objektorientiert nicht vereinigt werden können. Wie der Einsatz von Object Teams eine Brücke zwischen diesen Zielen schlägt, soll hier erörtert werden.

- **Persistenz von Objekten:** Ein oft genanntes Beispiel eines *'Crosscutting Concerns'* besteht in der Funktionalität, ein Objekt dauerhaft zu speichern. Man versteht darunter die Möglichkeit, den Zustand eines Objektes zu speichern und wiederherzustellen. Als Speichermedium hat sich der Einsatz von Datenbanken durchgesetzt. Die Möglichkeit der Persistenz von Objekten ist eine typische Systemfunktionalität, die nichts mit der Funktionalität der Domäne gemein hat. Beide sollten aus diesem Grunde auch nicht miteinander vermischt werden. Dies hat zur Folge, dass die Erbringung einer solchen Systemfunktionalität eine Grundregel der Objektorientierung verletzen muss: Daten werden von einer Klasse gekapselt und sind nur über Methoden der Klasse zugreifbar. Da auf diesen internen Zustand von außen zugegriffen werden muß, ist hier ein Konflikt gegeben. Es soll gezeigt werden, wie mit Hilfe von Object Teams

solch eine Systemfunktionalität erbracht werden kann, die weitestgehend unabhängig ist von der Funktionalität der Domäne.

- **Mechanismen der Zugriffskontrolle auf Basis von Privilegien** In einem Systemen mit mehreren Benutzern ist es üblich, sensitive Teile des Systems vor Zugriffen zu schützen. Ein Benutzer hat nur dann Zugriff auf solche sensitiven Bereiche, wenn er dazu berechtigt ist. Es gibt eine ganze Reihe von Herangehensweisen eine solche Zugriffskontrolle zu realisieren. Sie alle beruhen auf der Möglichkeit, anhand einer Identität bzw. anhand von Privilegien einen Zugriff zu gewähren oder zu verweigern. Es ist auch möglich, dass sich eine bestimmte Funktionalität durch die Zugriffskontrolle ändert, indem z.B. eine beschränkte Funktionalität ermöglicht wird. Verschiedene Benutzer haben auf diese Weise unterschiedliche Sichten auf das gleiche System. Das Schlagwort *personalisierte Software* sei an dieser Stelle genannt. Die unterschiedlichen Sichtweisen sollten dabei nicht Teil des Systems selber sein. Diese Funktionalität kann als Aspekt aufgefasst und separat gekapselt werden, wie versucht wird zu zeigen.

Diese drei Schwerpunkte adressieren drei ganz unterschiedliche Problemkreise, die sich durch den Einsatz von rein objektorientierten Lösungen ergeben. Diese Probleme sollen in den nun folgenden Kapiteln näher beschrieben werden, um daraufhin eine Object Teams basierte Lösung im Gegenzug vorzustellen. Es sei darauf hingewiesen, dass es hierbei nicht darum geht, Konzepte der Objektorientierung schlecht zu machen, oder dass es nicht auch etliche (rein objektorientierte) Möglichkeiten gibt, bestimmte Probleme zu umgehen. Es geht erstens darum, bestimmte Modellierungen im Design auf Schwächen zu prüfen, die ihren Ursprung nicht im Modell sondern in der Idee der unterstützten Strukturierungsmechanismen selbst haben. Es soll des weiteren gezeigt werden, wie der Einsatz von Object Teams eine Modellierung und Implementierung im Vergleich unter Umständen vereinfachen kann. Der dritte Punkt, der hier genannt werden soll: vielleicht werden einfach nur bessere Metaphern für Entitäten und Funktionalitäten gefunden, die eine Softwarelösung verständlicher machen.

Kapitel 4.1 beschreibt Anforderungen an eine Projektmanagementsoftware und erklärt das zugrunde liegende Domänenmodell. Es wird auf die Domänenmodellierung und technische Rahmenbedingungen eingegangen, die sich aus den Anforderungen ergeben. Kapitel 4.1.3 beschreibt im Überblick die realisierte Projektmanagementsoftware „Promote“. Dieses Kapitel ist knapp gehalten, da die einzelnen Problematiken dieser Modellierung und Implementierung in den darauf folgenden Kapiteln besprochen werden. Kapitel 4.2 geht auf die Implementierung graphischer Benutzeroberflächen ein, Kapitel 4.3 beschäftigt sich mit der Problematik der Persistenz von Objekten und in Kapitel 4.4 wird der Aspekt der Zugriffskontrolle behandelt.

4.1 Domänenmodell eines Projektmanagementsystems

Der Begriff „Projektmanagement“ ist ein Schlagwort von sehr allgemeiner Herkunft und keinesfalls auf die Domäne der Softwareentwicklung beschränkt. Er umfasst eine Vielzahl von Aufgaben und Themen. Eine einfache Definition von Projektmanagement ist die Summierung aller Aufgaben und Tätigkeiten, welche organisatorisch für den korrekten Ablauf eines Projektes notwendig sind. Der Fokus liegt auf dem organisatorischen Rahmen, der den Prozess einer Entwicklung in allen Phasen umfasst und begleitet. Dieser Rahmen kann je nach Art und Umfang eines bestimmten Projektes unterschiedlich beschaffen sein. Er umfasst in der Regel Aufgaben wie die Projekt- und Arbeitsplanung, der Verwaltung aller teilnehmenden Projektmitglieder, des Kundenstamms, der nötigen Ressourcen, des verfügbaren Budgets und der anfallenden Kosten, die Planung des Arbeitsablaufs usw.

Die Aufgaben des Projektmanagements sind nicht automatisierbar. Ein Softwaresystem kann die jeweiligen anfallenden Aufgaben nicht übernehmen, wohl aber unterstützen. Es kann dazu beitragen, alle für ein Projekt relevanten organisatorischen Daten zusammenzutragen und zur Verfügung zu stellen. Es liegt nahe, alle Verwaltungsaufgaben zu automatisieren und auf die wesentlichen Schritte für die einzelne Person zu reduzieren. Solch eine Software kann auf diese Weise als Werkzeug für alle Personen, die an einem Projekt beteiligt sind, angesehen werden.

Die Firma ITSO hat im Rahmen des Projektes Q-MEKKA (Fraunhofer FIRST, TU-Berlin 2001-2002) eine Fallstudie zum Thema „Projektmanagement-und Informationsportal“ [Glöckner und Storl 2001] erstellt. Dieses Dokument beschreibt Anforderungen an ein Projektmanagementsystem und spezifiziert die entstehenden Aktionen an und Interaktionen mit dem System. Der hier definierte Anforderungskatalog bildet die Grundlage der in dem Rahmen dieser Diplomarbeit entstandenen Projektmanagementsoftware. ITSO beschreibt vier Schwerpunkte die eine solche Software erfüllen muss und die im folgenden näher beschrieben werden sollen. Diese Beschreibung fokussiert keine bestimmte Domäne und benutzt deshalb sehr allgemeine Begriffe und Praktiken.

Personalverwaltung und Rollenverwaltung

Es existiert eine Personalverwaltung, die alle relevanten Daten eines Mitarbeiters speichert. Zu diesen Daten gehören neben Name und Anschrift auch die Möglichkeiten die Person zu erreichen. Eine bestimmte Teilmenge dieser Daten kann der der Allgemeinheit zur Verfügung gestellt werden, so dass dieser notwendige Teil die Funktionalität eines Telefon- und Adressbuches übernehmen kann. Die hierarchischen Organisationsformen einer Firma sollen als Rollen modelliert werden. Solche Rollen sind beispielsweise Projektleiter, Mitarbeiter, Geschäftsführer oder Sekretär. Eine bestimmte Person kann mehrere Rollen einnehmen. Es soll ein Rechtesystem

auf Grundlage dieser Rollen implementiert werden. So hat in aller Regel ein Geschäftsführer mehr Rechte im System, als beispielsweise ein einfacher Mitarbeiter. Er könnte z.B. neue Mitarbeiter einstellen, Projekte aquirieren etc. Um solche Sicherheitsmechanismen sicher zu stellen, ist eine Identifikation der Person und somit ihrer Rollen nötig.

Arbeits- und Projektplanung

Ein Projekt kann in Phasen aufgeteilt werden, die wiederum in einzelne Teilabschnitte zergliederbar sind. In der Geschäftswelt haben sich die Begriffe *Meilenstein* und *Aufgabe* etabliert, die diese Teilabschnitte kennzeichnen. Ein Projekt kann in eine Reihe von Meilensteinen aufgeteilt werden. Ein einzelner Meilenstein wird erreicht, wenn alle enthaltenen Aufgaben erfüllt sind. Eine Aufgabe kann zu einer beliebigen Anzahl neuer Aufgaben führen, was zu einer Schachtelung führt. Jedem Meilenstein und jeder Aufgabe werden geplante Aufwände, voraussichtliche Fertigstellungstermine und verantwortliche Projektteilnehmer zugeordnet. Die jeweils geplanten Zeit- und Budgetmittel sind auf diese Weise in jedem Teilabschnitt ersichtlich. Der einzelne Mitarbeiter hat einen Überblick, für welche Aufgaben er verantwortlich ist und bis wann diese erfüllt sein muss.

Mitarbeiter in einem Team führen einen digitalen Stundenzettel. Die aufgewendete Arbeit wird auf die jeweils bearbeitete Aufgabe kontiert. Nun können geplante und reale Zeitaufwände zueinander aufgezeigt werden - für jede einzelne Aufgabe, durch Addition der Aufgaben für jeden Meilenstein und somit für das gesamte Projekt. Es ist also relativ einfach ersichtlich, ob ein Fertigstellungstermin eingehalten werden kann, oder nicht. Auf Grundlage dieser Daten soll es jederzeit möglich sein, einen Report für ein Projekt erstellen zu lassen, welches den aktuellen Stand aller Meilensteine und Aufgaben aufzeigt. Solch ein Report kann der Geschäftsführung bzw. auch dem Kunden als Statusbericht vorgelegt werden.

Abrechnung- und Kostenverwaltung

Jede Tätigkeit für ein Projekt muss genau abgerechnet werden können. Um eine wohlstrukturierte Rechnung zu erzielen, bedarf es spezieller Abrechnungs- und Ergebnistypen. Der digitale Stundenzettel kann diese Aufgabe erfüllen, wenn jeder Mitarbeiter die Art der Tätigkeit in einem vorgegebenen Schema spezifiziert. Das System kann eine Rechnung erstellen, indem die genauen Arbeitsaufwände und Kosten der einzelnen Teilabschnitte genau dargelegt werden.

Kundenstammverwaltung

Es muss eine Kundenstammverwaltung existieren. Hier werden alle nötigen Daten des Kunden plus den einzelnen Kontakten zu Personen in dieser Firma hinterlegt.

Diese Information dient vor allem den einzelnen Projektmitgliedern zur Kontaktierung des Kunden. Es ist außerdem denkbar, dass auch der Kunde eine eingeschränkte Sicht auf den Stand des Projektes hat.

4.1.1 UML-Modellierung der Domäne

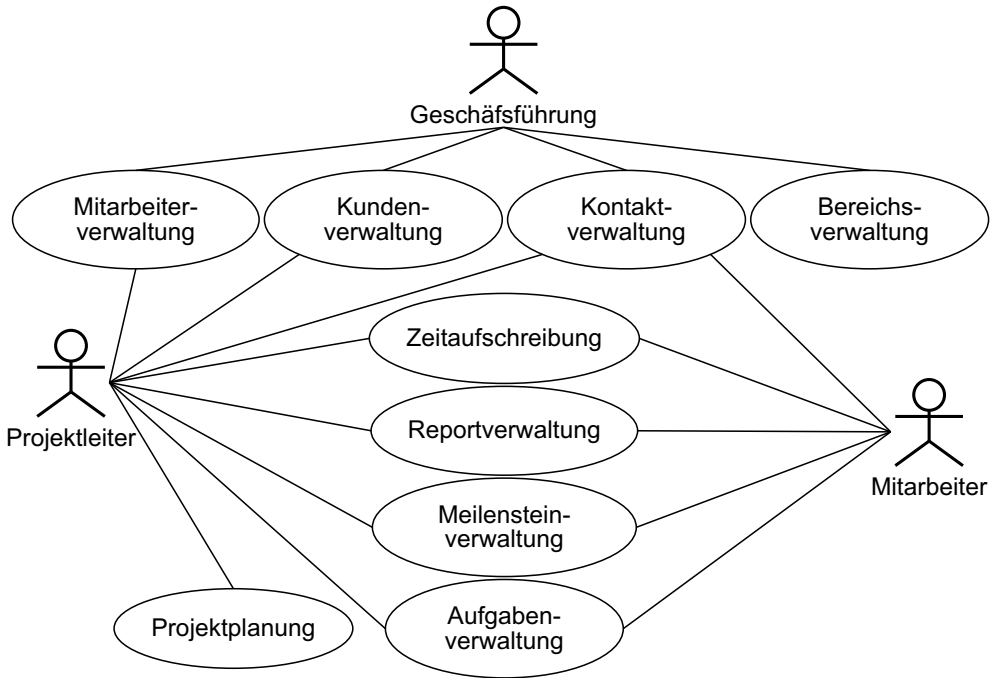

Abbildung 4.1: Sicht der einzelnen Rollen auf das System (Quelle ITSO)

ITSO definiert drei Akteure in ihrer Fallstudie für das System: Geschäftsführer, Projektleiter und Mitarbeiter. Diese drei Arten von Akteuren unterscheiden sich in ihrer Sicht auf das System. Es ergibt sich ein Anwendungsfalldiagramm, wie es in Abbildung 4.1 zu sehen ist. Ein Geschäftsführer hat administrative Aufgaben, die die gesamte Firma betreffen. Es existiert keine direkte Relation zu einem speziellen Projekt und der damit verbundenen Funktionalität. Ein Mitarbeiter arbeitet in verschiedenen Projekten und soll auch nur mit diesen konfrontiert werden. Für diese Projekte kann sich der Mitarbeiter um die einzelnen Aufgaben und Meilensteine kümmern und führt einen digitalen Stundenzettel über die geleistete Arbeit. Ein Projektleiter hat die gleichen Aufgaben wie ein Mitarbeiter in einem Projekt. Er ist zusätzlich für die Kontakte zum Kunden, die Mitarbeiter in seinem Team und die Projektplanung in seinem zu leitenden Projekt verantwortlich. Projektplanung bedeutet das erstellen von Meilensteinen und Aufgaben und die Zuweisung dieser an Mitglieder des Teams.

Die einzelnen Sichtweisen lassen sich als Rechtesystem subsumieren, die als Rollen modelliert werden können. Eine Rolle steht stellvertretend für eine bestimmte Menge von Rechten, die zu bestimmten Interaktionen mit dem System berechtigen. Jeder Mitarbeiter in einer Firma kann verschiedene Rollen haben. Dieser Rollenmechanismus ist erweiterbar, so dass weitere Rollen denkbar und möglich sind (z.B. Sekretär, Bereichsleiter etc.), je nach der vorhandenen Firmenstruktur.

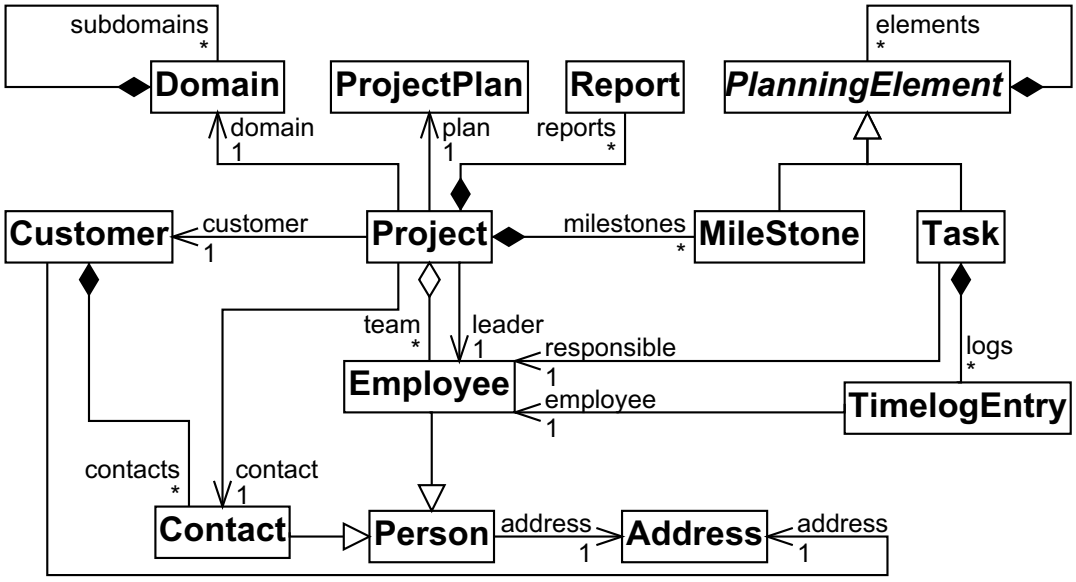

Abbildung 4.2: Domänenmodell eines Projektmanagementsystems

Abbildung 4.2 zeigt das Domänenmodell zu den definierten Anforderungen. Aus Gründen der Übersichtlichkeit werden nur die Klassen ohne Attribute und Methoden dargestellt. Eine Firmenstruktur besteht aus einzelnen Bereichen, welche hierarchisch organisiert sein können (`Domain`). In einem bestimmten Bereich sind Mitarbeiter (`Employee`) angestellt. Ein Mitarbeiter wird als Spezialisierung einer Person modelliert. Die Klasse `Person` hält alle relevanten Daten zu der Person, wie Name, Adresse etc. Für jeden Mitarbeiter werden weitere organisatorische Daten gekapselt, die für die Firmenstruktur notwendig sind. Jeder Mitarbeiter ist über einen eindeutigen Namen im System zugreifbar.

Die Logik der Projektplanung wird in der Klasse `PlanningElement` realisiert. Sie kapselt ein abstraktes Planungselement und kann beliebig verschachtelt sein. Ein Planungselement umfasst eine Beschreibung der Aufgabe, den geplanten Arbeitsaufwand, den geplanten Fertigstellungstermin und den derzeitigen Status (z.B. in Planung, in Bearbeitung, abgeschlossen). Ein Meilenstein (`MileStone`) speichert des weiteren den vereinbarten Fertigstellungstermin, welcher sich vom geplanten Termin unterscheiden kann. Einzelne Mitarbeiter werden für bestimmte Aufga-

ben (`Task`) verantwortlich gemacht. Arbeitsaufwände für eine bestimmte Aufgabe können auf diese kontiert werden. Die Zeitaufschreibung wird als `TimelogEntry` modelliert, welche der jeweiligen Aufgabe zugeordnet ist. Die Summierung aller Zeitaufschreibungen zu einer Aufgabe ergibt den bisher geleisteten Aufwand, der dem geplanten Aufwand gegenübergestellt werden kann. Diese Daten können auch für die Kostenstelle aufbereitet werden.

Die Modellierung des Kunden vollzieht sich in der Klasse `Customer`. Hier werden alle kundenspezifischen Daten gespeichert. Zu jedem Kunden werden eine Menge Kontakte (`Contact`) assoziiert. Eine große Firma hat beispielsweise einen Hauptsitz und mehrere Standorte. Kontakte hat man immer zu Personen in einem bestimmten Standort, deshalb werden diese separat modelliert.

Ein Projekt hat nun Verbindungen zu allen diesen Entitäten. Es besteht aus einem Team, welches sich aus einer Menge von Mitarbeitern zusammensetzt (`team`). Jedes Projekt hat einen Projektleiter (`leader`). Dieser Leiter hat für dieses Projekt besondere Rechte. Jedes Projekt hat einen Auftraggeber, also einen Kunden (`customer`), der dieses Projekt in Auftrag gegeben hat . Zu einem Kunden gehört ein spezieller Ansprechpartner in der Firma, der für Fragen und Vorschläge über dieses Projekt verantwortlich ist (`contact`). Das Dokument der Anforderungsbeschreibung wird von der Klasse `ProjectPlan` gekapselt und dem Projekt assoziiert. Zu einem Projekt können zu jeder Zeit Reporte erstellt werden, die die Geschäftsführung bzw. den Kunden über den derzeitigen Stand des Projektes informieren. Ein Report ist eine Momentaufnahme aller geleisteten Arbeiten zu allen Aufgaben im Vergleich zu den geplanten Aufwänden. Alle erzeugten Reporte werden mit einem Zeitstempel versehen und im jeweiligen Projekt gespeichert.

4.1.2 Technische Rahmenbedingungen

Die definierten Anforderungen für ein Projektmanagementsystem forcieren Architekturentscheidungen, die im folgenden evaluiert werden sollen. Jeder Benutzer muss vom System identifizierbar sein, wobei jeder einzelne unabhängig von anderen mit dem System interagieren kann. Es muss also ein Mehrbenutzersystem implementiert werden, welches die Zugriffe auf das System synchronisiert, so dass nur konsistente Zustände möglich sind. Jeder Benutzer muss sich am System anmelden, um Funktionalität zu nutzen.

Jeder Benutzer benötigt eine Schnittstelle zum System, welches die jeweilige Funktionalität ermöglicht. Um eine Akzeptanz bei den Mitarbeitern zu erzielen, ist eine graphische Benutzeroberfläche oder ein HTML-Frontend für diese Schnittstelle nötig. Der Funktionsumfang der Schnittstelle hängt von den Rollen des Akteurs ab. Es ist eine Rechtesystem notwendig, welches diese Anforderung erfüllt.

Die Interaktion mit dem System muss von verschiedenen Computern aus nutzbar sein. Die Verteilung der Anwendung erfordert eine Art Client/Server-Architektur. Jeder Benutzer (Client) agiert über eine Schnittstelle mit dem System (Server). Das System offeriert Funktionalität, welche vom Anwender genutzt werden kann. Das System muss die Konsistenz über alle Benutzer sicher stellen. Die funktionale Trennung dieser beiden Entitäten ist an dieser Stelle noch komplett unspezifiziert. Die Verteilung über Rechnergrenzen hinaus erfordert einen Kommunikationskanal zwischen Client und Server.

Alle im System verfügbaren Daten sollen gesichert werden. Um dies sicherzustellen, müssen die Daten persistent gemacht werden können. Hier bietet sich eine Datenbank bzw. ein Repository an.

4.1.3 Das Projektmanagementsystem „PromOTe“

Im Rahmen dieser Diplomarbeit entstand „PromOTe“ — ein Projektmanagementsystem, welches mit Hilfe von Object Teams modelliert und implementiert wurde (PromOTe → Projektmanagementsystem mit Hilfe von Object Teams). Grundlage dieses Systems bildet die Anforderungsbeschreibung der Firma ITSO. Als Zielarchitektur wurde von ITSO eine Web-Anwendung definiert: eine zentralisierte Anwendung, auf die mit einem Browser zugegriffen werden kann. Die anfallenden Daten sollten von dem Subsystem *bluedot* dauerhaft gespeichert werden.

Die bereitzustellende Schnittstelle zum System wurde für PromOTe umdefiniert: hier sollte eine graphische Benutzeroberfläche (GUI) realisiert werden. Im Gegensatz zu der Web basierten Lösung kommen hier weitere Anforderungen hinzu, da eine graphische Benutzeroberfläche mehr Interaktion mit dem System zulässt. Eine Web-Anwendung erlaubt nur das explizite Anfordern von Information (pull), eine GUI-Anwendung kann auch extern gesteuert werden (push & pull). Dargestellte Informationen sollten hier also automatisch aktualisiert werden, sobald sich diese ändern.

Alle Benutzer arbeiten auf einer gemeinsamen Datenquelle. Die im System verfügbaren Informationen und Daten werden hier zentral verwaltet. Diese gemeinsame Datenquelle stellt eine Konsistenz der Informationen über alle Benutzer sicher. Ändert ein Benutzer eine bestimmte Information, so wird die Änderung an alle betroffenen Stellen weitergeleitet. Die dargestellten Informationen der einzelnen Benutzer sind auf diese Weise immer aktuell. Die Datenquelle speichert alle Informationen und Daten dauerhaft in einer Datenbank, so dass diese auch beim nächsten Systemstart verfügbar sind. Da das System *bluedot* von ITSO nur für Java verfügbar ist, musste hier ein eigener Mechanismus programmiert werden, Daten dauerhaft in einer Datenbank zu speichern.

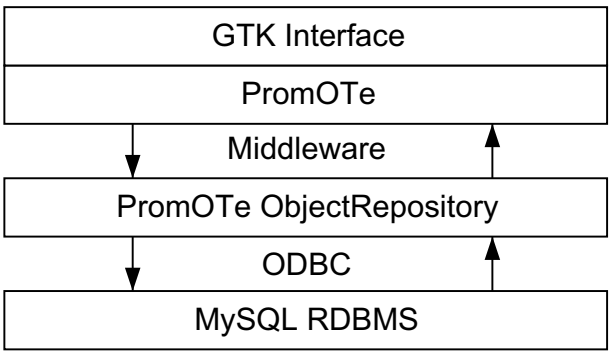

Abbildung 4.3: Die Architektur von PromOTe

In Abbildung 4.3 kann man die allgemeine Architektur von PromOTe sehen. Die Implementierung des Domänenmodells (Abbildung 4.2) liegt hier auf der ebene PromOTe. Hier ist die Logik manifestiert, die in der Anforderung von ITSO definiert wurde. Um diese Logik nutzen zu können, wurde eine graphische Schnittstelle programmiert, die auf den Entitäten der Domäne arbeitet. Alle Objekte der Domänenklassen werden in einem Objektrepository gehalten. Dieses Repository ist unabhängig von der Anwendung des Benutzers. Es ermöglicht das Arbeiten mehrerer Benutzer auf den gleichen Daten und sorgt für deren Aktualität. Um den Zugriff auf dieses Repository zu ermöglichen ist ein Kommunikationskanal zwischen den Anwendungen und dem Repository nötig. Dieser Kommunikationskanal wird durch eine Middleware bereit gestellt. Die Daten in dem Repository sollen permanent gespeichert werden, da alle Daten und Informationen auch nach einem Systemstart verfügbar sein müssen. Aus diesem Grund speichert das Objektrepository alle Daten in einer relationalen Datenbank. Um auf diese Datenbank zuzugreifen, ist wieder ein spezieller Kommunikationskanal notwendig.

Die graphische Schnittstelle für PromOTe[1] wurde mit Hilfe von GTK+ Team [2001] entwickelt. Um diese Funktionalität von Ruby aus zu nutzen, wurde das Paket Ruby-Gnome[Igarashi u. a. 2001] genutzt, welche eine Bindung für Ruby an diese Bibliothek bereitstellt. Die einzelnen Dialoge wurden mit dem Schnittstellendesigner Glade[Chaplin 2001] entworfen. Dieses Werkzeug enthält eine Bibliothek, die es erlaubt, die modellierten Dialoge direkt aus der erstellten Projektdatei erzeugen zu lassen. So muss kein Quellcode erzeugt oder gewartet werden, was zu einer hohen Flexibilität beiträgt.

Die Wahl der relationalen Datenbank fiel auf das MySQL[MySQL AB 2002] Datenbankmanagementsystem. Diese Datenbank existiert auf vielen Plattformen, ist sehr performant und sicher. Um auf eine Datenbank von Ruby aus zuzugreifen, wurde

[1] Alle für PromOTe benutzten Softwarepakete und -systeme sind frei verfügbar und unterliegen der *'GNU Public License'*

das Paket RubyDBI[Neumann 2002] genutzt, welches eine Abstraktionsebene über spezielle Datenbankverbindungen einführt. Das Paket benutzt spezielle Treiber um auf spezielle Datenbanken zuzugreifen. Es ist also leicht möglich, eine andere relationale Datenbank von dem Objektrepository aus anzusteuern.

In der Realisierung ging es vor allem um die Frage: Wie modular können spezielle Funktionalitäten gekapselt und angewendet werden. Es sollten Aspekte gefunden werden, die unabhängig von PromOTe aufgeschrieben werden können, dort aber Anwendung finden. Hier wurde das Object Teams Paradigma angewendet, und zwar in seiner speziellen Form der Ruby Object Teams. Im Verlauf dieser Arbeit wurden drei Aspekte gefunden, die auf den folgenden Seiten detailliert beschrieben werden.

Stand des Projektes Vornehmliches Ziel dieses Projektes war die Evaluierung des Object Teams Paradigmas bezüglich der Anwendbarkeit in einem *'Real World Scenario'*. Es sollte herausgefunden werden, ob die relativ junge Technik der Object Teams den dort auftauchenden Problemen gewachsen ist und ob sie den Programmierer wirklich unterstützt oder nur neue, bzw andere Probleme schafft. Aufgrund dieser Prämisse ging es also weniger um Vollständigkeit und Benutzbarkeit einer speziellen Projektmanagementsoftware auf Grundlage der Anforderungen von ITSO, sondern um Erfahrungen mit Object Teams.

Das Domänenmodell wurde für PromOTe implementiert. Es existiert ein Objektrepository, welches an eine relationale Datenbank gebunden werden kann. Alle vorkommenden Objekte der Domänenklassen können in der Datenbank gespeichert werden. Es existiert eine graphische Schnittstelle, um auf dem Domänenmodell zu arbeiten. Es entstand im Verlauf der Implementierung ein Rahmenwerk für graphische Sichten, welches sehr einfach erweiterbar ist. Im wesentlichen wurde die Komponente der Arbeits- und Projektplanung sowie der Kundenstammverwaltung weitestgehend implementiert. Die Funktionalität der Personal- und Rollenverwaltung wurde implementiert, hier fehlt es aber an der graphischen Visualisierung und Administration. ITSO definierte als vierten Punkt die Abrechnungs- und Kostenstelle. Da hier die funktionale Beschreibung in der Anforderung fehlte, wurde dieser Punkt nicht beachtet. Die Einführung einer Kommunikationsschicht zwischen der graphischen Schnittstelle und dem gemeinsam genutzten Objektrepository wurde vorgesehen, aber noch nicht vollzogen.

Für den praktischen Einsatz von PromOTe muss diese Kommunikationsschicht eingeführt werden. Des weiteren muß die Administration von Personen und Rollen implementiert werden, damit überhaupt mehrere Benutzer das System nutzen können. Alle nötigen funktionalen Voraussetzungen sind dafür geschaffen.

4.2 Programmierung graphischer Benutzeroberflächen

Die Realisierung einer graphischen Benutzerschnittstelle (GUI) für eine Anwendung birgt, dank mannigfaltig vorhandener graphischer Bibliotheken (so genannter *'toolkits'*), keine großen Schwierigkeiten. Oft sind sogar Werkzeuge vorhanden (so genannte *'GUI-Builder'*), mit denen man die gewünschte Schnittstelle interaktiv gestalten kann, ohne dabei auch nur eine Programmzeile selbst zu schreiben.

Die graphischen Bibliotheken sind in der Regel als Framework konzipiert und beruhen auf einer funktionalen Trennung, die schon seit den frühen Tagen der Objektorientierung bekannt ist als *Model-View-Controller* - Paradigma. Im Gegensatz zu der ungetypten Implementierung in Smalltalk-80 muss in statisch typisierten Sprachen zwischen verschiedenen konkurrierenden Designzielen abgewogen werden, die bis heute nicht geeint werden konnten. Dies führt in der praktischen Entwicklung von graphischen Benutzerschnittstellen zu Kompromisslösungen, welche die sehr klare MVC-Architektur vernebeln.

4.2.1 Model View Controller

Das Model-View-Controller Paradigma [Krasner und Pope 1988], im folgenden kurz MVC genannt, führt eine Trennung zwischen der darzustellenden Entität (dem Modell) und deren Repräsentation (der View) ein, die von einer dritten Instanz (dem Controller) gesteuert wird (siehe Abbildung 4.4).
Hinweis: Da die englischen Begriffe View und Controller das Wesen dieser Entitäten besser beschreiben als ihre deutschsprachigen Pendants, werden diese im folgenden Text benutzt.

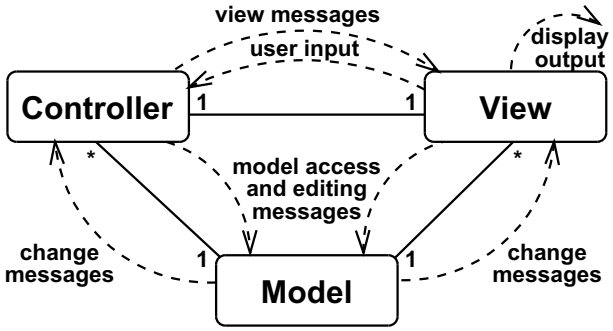

Abbildung 4.4: MVC - Kardinalität und Nachrichtenfluß

Das Modell ist eine Klasse der Domäne, welches angezeigt werden soll. Es entspricht einer domänenspezifischen Entität, welches kein Wissen von graphischen Benutzeroberflächen hat. Die Repräsentation des Modells als GUI-Element wird

View genannt. Die View ist in der Lage, die gekapselten Daten des Modells, bzw. eine Teilmenge davon, darzustellen. Sie kann als Wrapper [Gamma u. a. 1996, Wrapper] um das Modell aufgefasst werden. Jede View hat einen zugehörigen Controller. Ein Controller ist für alle Aktionen zuständig, die in der View definiert sind und die das zugehörige Modell betreffen. Ein Modell kann mehrere unterschiedliche Sichten haben. Dies erscheint naheliegend, da der selbe Satz von Daten sehr unterschiedlich repräsentiert werden kann. Jeder kennt beispielsweise verschiedene Diagrammtypen, die auf der gleichen Menge von Daten erstellt werden können.

Das Modell wird ohne konkretes Wissen über die Repräsentation modelliert und implementiert. Es existiert eine implizite Referenz auf die View bzw. den Controller über einen Beobachtermechanismus [Gamma u. a. 1996, Observer]. Das Modell implementiert die Funktionalität des Subjektes und teilt den Beobachtern jede Änderung des Zustandes mit. Die View, wie auch der Controller haben eine explizite Referenz auf das Modell, u.A. um den aktuellen Zustand zu erfragen. Die View kann Aktionen definieren und Ereignisse senden, falls eine Aktion ausgelöst wurde. Der Controller ist in der Lage, auf Ereignisse zu reagieren, die eine Relevanz für das Modell haben. Ein Controller „übersetzt“ also die spezifischen Ereignisse der View in die Logik der Anwendung. Er hat die Rolle des Vermittlers zwischen der darzustellenden Entität und deren Repräsentation.

4.2.1.1 Ein einfaches Beispiel

Um die einzelnen Designentscheidungen zu verdeutlichen, soll ein kleines Beispiel aufgeführt werden: eine Stoppuhr. Die Funktionalität einer Stoppuhr ist sehr einfach und ist mit drei Knöpfen zu kontrollieren Start, Stopp und Zurücksetzen. Ist die Stoppuhr gestartet, zeigt sie die verstrichene Zeit, die mit Stopp angehalten werden kann. Die Klasse `StopWatch` wird eingeführt, die solch ein Verhalten implementiert. Die Methode `start` startet einen Thread, welcher für eine Sekunde schläft und dann die Methode `advance` aufruft, solange bis die Methode `stop` aufgerufen wird. In der Methode `advance` wird ein Zähler inkrementiert, der die vergangenen Sekunden repräsentiert. Die Methode `stringValue` liefert die vergangene Zeit als Zeichenkette. Die Klasse `WatchDisplay` setzt sich aus einem Label, welches die vergangene Zeit zeigt, und drei Knöpfen zusammen: start, stop und clear. Der Controller ist dafür zuständig, die Ereignisse dieser Knöpfe mit den gleichnamigen Methoden des Modells zu verbinden. Um die View mit dem Modell konsistent zu halten, sind fünf Formen der Kommunikation nötig.

1. Die Anzeige muss sich als Observer bei der Stoppuhr registrieren (`View`→`Modell`).

2. Die Stoppuhr benachrichtigt alle Observer, wenn immer der Zustand des Objektes sich verändert. Eine Veränderung des Zustandes geschieht nur in der

Methode `advance` (`Modell→View`).

3. Die Anzeige muss die aktuell vergangene Zeit erfragen (`View→Modell`).
4. Wenn der Benutzer start, stop oder clear drückt, erzeugt die Anzeige Ereignisse, die der Controller behandelt (`View→Controller`).
5. Ereignisse am Controller haben eine Bedeutung im Sinne des Modells. Im Controller ist das Wissen manifestiert, wie auf ein bestimmtes Ereignis zu reagieren ist. Er agiert dabei nur als Mediator und delegiert die Aktion an das Modell (`Controller→Modell`).

Der Kommunikationspfad Modell→Controller und Controller→View sind in diesem Beispiel nicht von Bedeutung.

4.2.1.2 Schwachpunkte in einem MVC basierten Design

Die strikte Trennung von Verantwortlichkeiten in einzelne Klassen, die die gleiche Entität betreffen, führen zu einer sehr dichten Kopplung der einzelnen Funktionalitäten. Dies ist auch in der Abbildung 4.4 zu sehen. Die drei unabhängigen Entitäten sind durch drei bidirektionale Kommunikationspfade verbunden. Jeder einzelne Teil hat eine Referenz auf die jeweils anderen.

Das Modell entspringt einer spezifischen Domäne und sollte nichts über GUI-Funktionalität wissen. Damit eine Synchronisation trotzdem möglich ist, wird eine allgemeinere Form der Referenzierung gewählt, nämlich eine Observerinfrastruktur. Es ist nur ein kleiner Aufwand, diese Infrastruktur zur Verfügung zu stellen. Sie ist aber genau genommen keine Funktionalität der Domäne. Das Design dieser Infrastruktur erfordert eine exakte Planung im Modell. Im Modell wird entschieden, welche Ereignisse propagiert werden und auf welcher Granularitätsebene diese ausgelöst werden. Die Anforderung der Granularität der Benachrichtigungen erwächst aber nicht auf der Seite des Subjektes, sondern wird vom Beobachter definiert. Dies bedeutet im einfachen Fall nur, dass das Modell mehr Benachrichtigungen sendet, als überhaupt erforderlich sind. Dies kann aber auch zu einer Änderung im Modell führen, wenn bestimmte Benachrichtigungen für eine bestimmte View auf das Modell benötigt werden. In der Planung der Observerinfrastruktur ist eine Abhängigkeit des Modells von der View gegeben, die eigentlich nicht erwünscht ist. Die häufigste Lösung dieses Problems ist das Nebeneinanderentwickeln dieser Infrastruktur, während die View-Klassen entwickelt werden. Liegt das Modell nur in Binärform vor, ist diese Option nicht möglich.

View und Controller werden unter Berücksichtigung eines konkreten Modells entwickelt. Um statische Typprüfung zu ermöglichen, arbeiten diese also auf einem bestimmten Typ, der die erforderliche Interaktion spezifiziert. Aus diesem Grund ist eine Wiederverwendbarkeit dieser Klassen schwierig, da sie an einen speziellen

Typ einer speziellen Domäne gebunden sind. Eine gängige Möglichkeit dieses Problem zu umgehen, kann man in vielen graphischen Bibliotheken beobachten. Sie besteht darin, die Spezifikation des Modells als Schnittstelle oder abstrakte Klasse zu definieren. Um solche Views und Controller zu verwenden, muss die Domänenklasse adaptiert werden, damit sie der abstrakten Spezifikation des Modells genügt. Es existiert somit eine Abhängigkeit der Domänenklasse von ihrer Repräsentation.

Die angesprochenen Probleme einer MVC basierten Architektur werden von speziellen objektorientierten Techniken adressiert aber nicht gelöst. Um eine Lösung dieser Probleme zu erzielen, müssen die Abhängigkeiten der einzelnen Teilstrukturen noch stärker reduziert werden, als dies klassischerweise der Fall ist.

4.2.2 MVC à la Object Teams

Die klassische Objektorientierung kennt zwei Möglichkeiten der Bindung zwischen Klassen: Vererbung und einfache Delegation (*'forwarding'*). Mit dem Paradigma von Object Teams wird eine weitere Art der Bindung möglich, die so stark ist wie die Bindung der Vererbungsrelation, aber viel modularer und flexibler. Sie ist außerdem so vielseitig wie die Bindung der (manuell kodierten) Delegation, aber viel sicherer, da ein Laufzeitsystem die Koordination überwacht und kontrolliert. Die MVC-Architektur beruht auf der Bindung der Delegation. Diese ist zwar extrem flexibel, aber auch zu schwach, um alle Anforderungen zu erfüllen.

In einer Object Teams-basierten MVC-Architektur werden die einzelnen Delegationsbindungen durch Rollenbindungen ersetzt. Die View wird somit als Rolle des Modells aufgefasst. Die einzelnen Kommunikationsformen der jeweiligen Entitäten werden ersetzt[2].

View → Modell Die einzelnen View-Klassen arbeiten nicht mehr auf einem konkreten Modell, also nicht mehr auf einem speziellen Typ. Die benötigten Funktionalitäten des Modells werden als abstrakte Methoden (*'open spots'*) deklariert. Diese werden von einem Konnektor per Callout gebunden. View-Klassen bleiben somit immer abstrakt. Eine Anpassung an bestimmte Anforderungen einer speziellen Domäne kann im Konnektor vorgenommen werden. Die Möglichkeit der Wiederverwendung solcher abstrakten View-Klassen wird auf diese Weise sehr viel stärker unterstützt, als in der klassischen Variante möglich.

Modell → View Die Synchronisation von Modell und Repräsentation wird durch Callin-Bindungen realisiert, die auch im Konnektor definiert werden. Die explizite Observerinfrastruktur im Modell kann somit entfallen. Für jede einzelne View kann nun eine Synchronisation individuell definiert werden. So sind

[2] Die Kommunikationspfade zwischen Modell und Controller sind hier nicht aufgelistet. Sie werden in Kapitel 4.2.2.2 diskutiert.

beispielsweise verschiedene Repräsentationen denkbar, die jeweils eine unterschiedliche Synchronisationsstrategie definieren. Dafür ist keine Änderung im Modell nötig, es genügt die Definition unterschiedlicher Callin-Bindungen.

Das Object Teams Paradigma ermöglicht eine Zuweisung von Zuständigkeiten an die Stellen, an denen die jeweiligen Anforderungen dieser Zuständigkeiten erwachsen. Das Resultat dieser Verschiebung von Zuständigkeiten ist eine größere Unabhängigkeit zwischen den einzelnen Entitäten. Diese Unabhängigkeit führt zu einem klarer strukturierten System und einer höheren Wiederverwendbarkeit.

Ein weiteres Problem welches sich immer bei Benutzung von Delegation ergibt, ist die Frage der Identität. Aus einer abstrakten Sicht meint man bei Modell und Repräsentation die gleiche Entität. Aus einer strukturellen Sicht ist dies nicht der Fall. Object Teams löst dieses Problem: Die View und das Modell sind zwei unterschiedliche Entitäten, dessen Identitäten vom Laufzeitsystem durch die Technik des Lifting bzw Lowering in Verbindung gebracht werden — die View im Kontext des umschließenden Teams, das Modell in der Domäne. Der Programmierer muss sich um die Frage der Identität nicht mehr kümmern, die äußere Sicht ist eine gemeinsame Identität.

4.2.2.1 Eine Stoppuhr mit Object Teams

Das Design mit Object Teams am Beispiel der Stoppuhr (Kap. 4.2.1.1) kann man in Abbildung 4.5 sehen. Die Klasse `StopWatch` unterstützt nun keine explizite Obser-

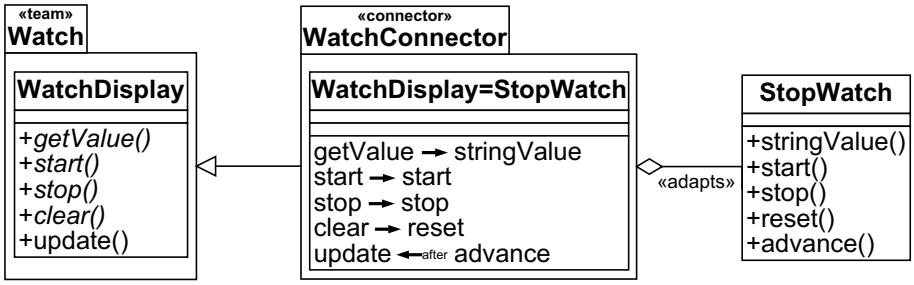

Abbildung 4.5: Modellierung der Stoppuhr mit Object Teams

verinfrastruktur mehr, unterscheidet sich ansonsten aber nicht in der Funktionalität. Die Klasse `WatchDisplay` wird in einem eignen Paket (`Watch`) definiert. Die Klasse deklariert abstrakte Methoden für alle Funktionalitäten, die erforderlich sind für ein korrektes Verhalten der Anzeige. So gibt es die abstrakten Methoden `start`, `stop` und `clear`, die dann aufgerufen werden, wenn die gleichnamigen Knöpfe gedrückt wurden. Das resultierende Verhalten der einzelnen Aktionen hat eine domänenspezifische Bedeutung, die an dieser Stelle nicht definiert werden kann. Der Wert, den die Uhr anzeigen soll liefert die Methode `getValue`. Wie dieser Wert sich

ergibt, ist eine Frage des darunter liegenden Modells und nicht die der Repräsentation. Die Anzeige hat dafür eine Methode `update` vorgesehen, die immer dann aufgerufen wird, wenn immer sich der Zustand des Modells verändert. Die Definition der Klasse `WatchDisplay` ist somit deklarativ vollständig. Möchte jemand dieses Paket benutzen, muss er alle abstrakten Methoden entweder implementieren oder einen Konnektor definieren, der diese Methoden an Methoden einer Domänenklasse bindet. Es existieren weiterhin keine Bindungen zu Klassen außerhalb des Paketes, welches die Wiederverwendbarkeit des gesamten Paketes ermöglicht.

Das Modell soll nichts von GUI-Funktionalität wissen und die View soll auf keinem speziellen Typ eines Modells arbeiten. Um diese Anforderung zu erfüllen, muss die Bindung dieser beiden Entitäten extern vorgenommen werden. Hier kommt nun der Konnektor zum Zug. Er kann die Rollen eines Teams an Klassen der Domäne binden. Er erfüllt zusätzlich eine Mediator-Funktionalität. Die Bindung der abstrakten Methoden einer Rollenklasse kann an unterschiedliche Methoden der Domänenklasse gebunden werden. Außerdem unterstützt ein Konnektor die Anpassung an eine spezielle Signatur (siehe Kapitel 2.1.3.4). In diesem Beispiel bindet der Konnektor `WatchConnector` die abstrakten Methoden `start` und `stop` an die gleichnamigen Methoden der Basis. Die Funktionalität der abstrakten Methode `clear`, wird in der Basis von der Methode `reset` erbracht. Um die aktuell vergangene Zeit darzustellen, wird die Methode `getValue` aufgerufen, die an die Methode `stringValue` der Basis gebunden wird. Mit der Bindung aller abstrakten Methoden der Rolle `WatchDisplay` an die Klasse `StopWatch` wird die Rolle und damit das gesamte Team *effektiv* und kann instantiiert werden. Um die Synchronisation der Repräsentation mit dem Modell zu gewährleisten, muss ein Triggermechanismus im Modell verfügbar sein. Dieser Mechanismus wird durch eine Callin-Bindung realisiert. Immer wenn die Methode `advance` aufgerufen wurde, soll auch die Methode `update` aufgerufen werden, damit die aktuell vergangene Zeit korrekt dargestellt wird. Gäbe es mehrere Methoden die den Zustand des Modells verändern, so müssten diese alle an die Methode `update` von `WatchDisplay` gebunden werden.

Es ist nun möglich diesen Konnektor zu instantiieren und zu aktivieren. Mit einer Aktivierung des Konnektors wird die Callin-Bindung in die Klasse `StopWatch` gewoben. Jede Instanz dieser DomänenKlasse bekommt nun eine Rolle vom Typ `WatchDisplay` assoziiert, der Repräsentation des darunter liegenden Modells. In Kapitel 4.2.1.1 wurden fünf Kommunikationsformen diskutiert. Die beschrieben Kommunikation findet auch hier statt, doch hat sie eine andere Form bekommen:

- Die Registrierung der View als Beobachter des Modells wird durch das Weben der Callin-Bindungen bei der Aktivierung des Konnektors realisiert (`update` ←**after** `advance`).
- Die Synchronisation von Modell und View wird durch diese Callin-Bindung sichergestellt. Sie ist solange gültig, solange der Konnektor aktiv ist.

- Das Erfragen des aktuellen Zustandes am zugehörigen Modell wird durch Callout-Bindung einer abstrakten Methoden realisiert
  (`getValue` → `stringValue`).

- Das Eintreffen von Ereignissen in der View durch Aktionen des Benutzers hängt von der jeweiligen benutzten graphischen Bibliothek ab und erfährt keine Änderung (klassisch).

- Die Methoden zur Behandlung der einzelnen Ereignisse werden abstrakt in der View deklariert und per Callout an das Modell gebunden
  (`start` → `start`, `stop` → `stop` und `clear` → `update`).

Die Kommunikationskanäle zwischen Modell und View wurden ersetzt durch Callin- bzw. Callout-Bindungen. Zwei Klassen wurden separat definiert und über eine dritte Instanz miteinander verbunden. Die beiden Klassen bleiben aus diesem Grund unabhängig voneinander und können wiederverwendet werden.

4.2.2.2 Designentscheidungen beim Controller

Ein Controller übernimmt die Funktion eines Mediators zwischen Modell und View. Ereignisse, welche durch Interaktion des Benutzers an der View entstehen und eine domänenspezifische Semantik haben, werden vom Controller in Operationen des Modells „übersetzt“. Der Controller kann auch das Verhalten der View beeinflussen, wenn diese vom Zustand des Modells abhängig ist. Diese funktionale Beschreibung eines Controllers kann sehr verschiedene Realisierungen haben, die im folgenden diskutiert werden sollen. Eine Eigenschaft wird von allen Realisierungen geteilt: der Controller benötigt direkten Zugriff auf die View. Da diese als Rolle in einem Team implementiert wird und der Zugriff auf Rollen immer nur innerhalb des selben Teams erlaubt sind, muss der Controller selbst auch als Teil des Teams begriffen werden. Diese Eigenschaft fügt sich nahtlos in das generelle Konzept von Object Teams, nämlich alle für eine bestimmte Funktionalität erforderlichen Strukturen in einem Team zu bündeln.

Es wurden vier Möglichkeiten evaluiert, die Funktionalität des Controllers zu implementieren. Der Controller als:

Teil der View Diese Form des Designs ist auch bekannt als *Document View* Muster, einer speziellen Form des MVC Paradigmas. Es existiert hier keine explizite Ausprägung des Controllers. Die zu erbringende Funktionalität wird in der View selbst definiert. Diese Möglichkeit ist für einfache Interaktionen völlig ausreichend. Die Ereignisse der View werden direkt auf abstrakte Methodendeklarationen abgebildet, die per Callout an das Modell gebunden werden

müssen. Werden für die einzelnen Abbildungen Anpassungen nötig, so können diese im Konnektor definiert werden. (Diese Form des Controllers wurde in dem vorangegangenen Beispiel, der Stoppuhr in Object Teams (Kapitel 4.2.2.1), angewendet. Die Interaktion beschränkte sich auf drei Knöpfe, dessen Verwaltung direkt in der View definiert wurden.)

Teammerkmal Diese Variante kann als direktes Pendant zum Controller im klassischen MVC-Paradigma betrachtet werden. Der Konnektor implementiert die Funktionalität aller Controller für die einzelnen Sichten dieses Teams. Auch hier besteht die Möglichkeit die Funktionalität nur zu deklarieren und nicht zu implementieren, was für die Wiederverwendbarkeit wichtig ist. Ein erbendes Team ist dann für die Implementierung der einzelnen abstrakten Methoden zuständig. Für die Aktivierung einer View ist sowieso eine Instanz des Konnektors notwendig. Diese Instanz kann nun auch dazu benutzt werden, die einzelnen Sichten explizit zu kontrollieren, so dies gewünscht ist. Sollen spezifische Methoden einer Domänenklasse im Konnektor genutzt werden, so müssen diese in der View als abstrakte Methoden deklariert werden und an die Domänenklasse gebunden werden.

Im Team enthaltenes Objekt Soll das Verhalten eines Controllers nicht von außen steuerbar sein, so besteht die Möglichkeit den Controller als Klasse im Team zu definieren, ihn aber an keine Klasse der Domäne zu binden. Solch eine Klasse muss also effektiv sein und ist so innerhalb des Teams instantiierbar. Die Instanzen sind von außen nicht zugreifbar – sie sind unabhängig. Spezielle Aufrufe am Modell müssen auch hier in der View deklariert und an das Modell gebunden werden.

Rollenobjekt Eine universale Variante liegt in der Möglichkeit, den Controller selbst als Rolle des Modells zu definieren. Der Controller wie auch die View sind somit Rollen des gleichen Basisobjektes. Die Definition von spezieller Funktionalität des Modells kann nun im Controller selbst deklariert und genutzt werden.

In der Implementierung von PromOTe hat sich die zweite Variante als anwendungsfreundlich erwiesen. Der Vorteil dieser Variante liegt in der expliziten Kontrolle der View über den Konnektor selber. Er bietet weiterhin den Vorteil, allgemeine Funktionalitäten in einem abstrakten Team zu definieren, von dem alle Teams erben, die GUI-Funktionalität binden. So wurde ein abstraktes Team `Controller` implementiert, welches das Team `Team` verfeinert. In diesem Team wurden Standardfunktionalitäten implementiert, die für alle Controller in PromOTe Gültigkeit haben.

4.2.3 Baukastenprinzip für graphische Elemente

Die Benutzung einer graphischen Bibliothek beruht auf den Konzepten der Vererbung und der Aggregation. Eine Menge von Anzeigeelementen (*'widgets'*) werden bereitgestellt, die vom Benutzer an die jeweiligen Bedürfnisse angepasst werden können. Es wird von der jeweiligen Klasse geerbt und fehlende Funktionalität implementiert, bzw. bestehende Funktionalität verfeinert. Eine Synthese zu einem neuen komplexen Anzeigeelement ist in der Regel durch Aggregation einzelner Elemente möglich. Die bereitgestellten Elemente moderner graphischer Bibliotheken sind in der Regel so komplex und weit entwickelt, dass man sich auf die Bindung derselben an das Domänenmodell konzentrieren kann. Es gibt wohl nur wenige Bereiche in der Softwareentwicklung, wo ein solch hoher Grad an Unabhängigkeit und Wiederverwendbarkeit erreicht wurde.

Mit Hilfe von Object Teams ist eine Erweiterung der bereitgestellten Technologien möglich. Der modulare Gedanke ist hier nicht nur auf einzelne Klassen reduziert, sondern umfasst eine Menge kollaborierender Funktionalitäten. Für die Darstellung komplexer Strukturen beispielsweise, sind in der Regel mehrere Klassen verantwortlich. Der Vertrag für die einzelnen Interaktionen der einzelnen Teilnehmer kann nun in einem Team separat definiert werden. Dieses Vorgehen hat mehrere Vorteile. Es ist möglich, für jede einzelne Rolle des Teams Funktionalität zu definieren. Dies ist in einer Programmiersprache mit einfacher Vererbung nicht möglich. Die erforderliche Funktionalität wird hier in der Regel „nur“ als Schnittstelle definiert und muss von der Domänenklasse implementiert werden. Die Bereitstellung von Standardfunktionalität kann hier einen großen Vorteil bringen. Weiterhin soll die Anpassungsfähigkeit genannt werden, mit der ein Team an verschiedene unterschiedliche Domänenklassen gebunden werden kann. Dies umfasst Methoden- und Signaturanpassungen, aber auch die Möglichkeit im Konnektor Funktionalität zu definieren, die spezifisch für die einzelne Bindung ist. Auf der einen Seite bleibt das Design der Domänenklassen auf diesem Wege unberührt, auf der anderen Seite können abstrakte, wiederverwendbare Teams für GUI-Funktionalitäten definiert werden.

Um eine rapide Anwendungsentwicklung zu ermöglichen, ist eine Menge von Standard-GUI-Funktionalität, im Sinne der graphischen Bibliotheken, als Team eine Voraussetzung. Als Grundlage bilden diese bestehende Bibliotheken eine ideale Voraussetzung. So kann mit Hilfe von Vererbung oder Aggregation die gewünschte Schnittstelle bereitgestellt werden. Für die Implementierung der graphischen Benutzeroberfläche von PromOTe, wurde dies für die dort vorkommenden Anzeigeelemente durchgeführt. So gibt es Teams für Listen, Bäume, etc. Um die Möglichkeiten und den Nutzen solcher Teams zu demonstrieren, soll dies für die Visualisierung von Baumstrukturen am Beispiel gezeigt werden.

4.2.3.1 Visualisierung einer allgemeinen Baumstruktur

Bei der graphischen Visualisierung einer Baumstruktur müssen zwei Dinge berücksichtigt werden: der Baum als Wurzel aller Zweige und die einzelnen Knoten darin. Jede graphische Repräsentation eines Knoten besteht aus den darzustellenden Daten plus einer Menge graphischer Attribute, die das Aussehen dieser Daten bestimmen. Mögliche Attribute sind z.B. Farbe, Schrift, ein zugehöriges Bild, etc. welche alle zu einem Stil zusammengefasst werden. Eine abstrakte Modellierung dieser Beschreibung, kann man in Abbildung 4.6 sehen.

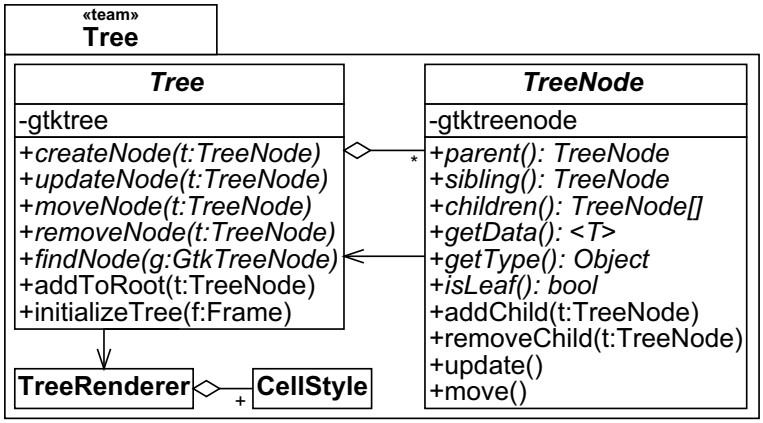

Abbildung 4.6: Ein abstraktes Baummodell

Es wird ein abstraktes Team `Tree` definiert. In diesem Team gibt es vier Klassen. Die Klasse `Tree` modelliert ein Anzeigeelement, welches eine komplette Baumstruktur darstellen kann. Da ein allgemeines Baummodell modelliert wird, ist das Team wie auch die Klasse `Tree` abstrakt. Sie definiert nur zwei Methoden, die für alle spezifischen Bäume Gültigkeit besitzen: `addToRoot` und `initializeTree`. Mit der ersten Methode können Blätter in die oberste Ebene des Baumes eingefügt werden. Die zweite Methode ist dem Fehlen von benutzerspezifischen Konstruktoren geschuldet. Eine Rolle wird immer vom Laufzeitsystem erzeugt. Die Rollenklasse darf deshalb nur einen parameterlosen Konstruktor implementieren. Um in dem Objekt einen definierten Anfangszustand zu setzen, muss eine dedizierte Methode aufgerufen werden, die Parameter erlaubt. Der Aufruf dieser Methode liegt in der Verantwortlichkeit des Anwenders des Teams – also dem Programmierer. Es werden weitere fünf Methoden abstrakt deklariert. All diese Methoden haben in einer speziellen Baumform eine eigene Ausprägung und können deshalb nicht verallgemeinert werden. Die Methoden decken dabei den Lebenszyklus eines einzelnen Knoten ab: Erzeugung (`createNode`), Synchronisation (`updateNode`), Umordnung innerhalb des Baumes (`moveNode`) und Löschen (`removeNode`) eines Knotens.

Die abstrakte Klasse `TreeNode` repräsentiert einen Knoten ein einem Baum. Der

Baum wird durch drei Methoden aufgespannt: `parent`, `sibling`, `children`. Jeder Knoten in einem Baum kann selbst eine Menge von Unterknoten referenzieren, die über die Methode `children` zugreifbar sind. Die Position eines einzelnen Knoten im Baum wird dabei durch die Referenz auf den Eltern- und den Nachbarknoten eindeutig gekennzeichnet. Diese drei Methoden werden hier abstrakt deklariert. Es wird davon ausgegangen, dass die zu repräsentierende Baumstruktur im Modell vorhanden ist. Ist dies nicht der Fall, kann diese Funktionalität in einem erbenden Team allgemein implementiert werden. Dieses erbende Team stellt dann eine Standardbaumstruktur zur Verfügung. In einem Konnektor muss in jedem Fall definiert werden, ob der darzustellende Knoten in einem Baum überhaupt Unterknoten haben kann. Zu diesem Zweck existiert die abstrakte Methode `isLeaf`. Der darzustellende Inhalt eines Knoten wird über die Methode `getData` erfragt. Der Typ des Rückgabewertes des darzustellenden Datums kann an dieser Stelle nicht definiert werden. Er hängt von der jeweiligen speziellen Baumform ab, die diesen Typparameter binden muss. Es existieren vier Methoden in der Klasse `TreeNode`, die für die Synchronisation der graphischen Repräsentation mit dem Modell zuständig sind. Sie stehen stellvertretend für die einzelnen Aktionen, die auf einem graphischen Knoten möglich sind:

- es wird ein Unterknoten hinzugefügt (`addChild`)
- es wird ein Unterknoten gelöscht (`removeChild`)
- der darzustellende Inhalt des Knoten hat sich geändert (`update`)
- der Knoten wurde an eine andere Stelle im Baum bewegt (`move`)

Diese Methoden müssen im Konnektor an all jene Methoden des Modells gebunden werden, die diese einzelnen Aktionen ausführen. Es müssen natürlich nur all jene Aktionen gebunden werden, die im Modell auch wirklich vorkommen. Unterstützt eine Anwendung bestimmte Aktionen nicht, werden die betroffenen Bindungen einfach weggelassen.

Um das Aussehen der einzelnen Knoten im Baum zu kontrollieren gibt es die Klasse `TreeRenderer` und `CellStyle`. Die Klasse `TreeRenderer` ist für das Zeichnen der Knoten im Baum verantwortlich. Dabei stehen ihr mehrere Stile zur Verfügung. Ein Stil `CellStyle` kapselt die graphischen Attribute, wie ein Knoten dargestellt werden soll. Beide Klassen müssen für die Möglichkeiten der spezifischen Baumformen verfeinert werden. Es ist möglich, für verschiedene Arten von Knotentypen, verschiedene Stile zu registrieren. Die Klasse `TreeNode` hat zu diesem Zweck die abstrakte Methode `getType` deklariert. Das Objekt, welches diese Methode liefert, wird beim `TreeRenderer` benutzt, um einen bestimmten Stil zu erfragen, der für diesen Typ hinterlegt wurde. In der Ansicht einer Verzeichnisstruktur würde es beispielsweise einen Stil für eine Datei und einen Stil für einen Ordner geben. Ein eindeutiges Typobjekt könnte hier das jeweilige Klassenobjekt sein. Beim zeichnen der

einzelnen Knoten wird über das jeweilige Typobjekt ermittelt, welcher Stil zum anzeigen verwendet werden soll. Die Varianten, die bei der Darstellung angewendet werden, obliegen den Möglichkeiten der jeweiligen Baumstruktur.

Um diese Schnittstelle als Team zur Verfügung zu stellen, wurden die existierenden Klassen der GTK-Bibliothek adaptiert [Gamma u. a. 1996, ObjectAdapter]. Das bedeutet, dass die Klasse `Tree` und die Klasse `TreeNode` ein äquivalentes Objekt der GTK-Bibliothek kapseln und benutzen. Um die gewünschte Schnittstelle bereitzustellen wäre der Weg der Vererbung der günstigere gewesen. Die GTK-Bibliothek erlaubt (in der benutzten Version 1.2) leider das explizite instantiieren von Knotenobjekten nicht, so dass dieser Umweg nötig war. Dieser Nachteil wird deutlich, wenn es um die Abarbeitung von Ereignissen geht. Der Ereignismanager liefert zu jedem Ereignis das zugehörige Objekt, welches das Ereignis auslöste. Wird beispielsweise ein Knoten im Baum selektiert, wird das zugehörige Knotenelement in die Ereignisverwaltung gereicht. Dieses Objekt ist nun das adaptierte Objekt einer zugehörigen Instanz der Klasse `TreeNode`. Es existiert eine eins zu eins Relation zwischen dem adaptierten Objekt und dem Adapter. Um diese Relation zur Verfügung zu stellen, implementiert die Klasse `Tree` die Methode `findNode`. Diese Methode ordnet einem adaptierten Knotenelement den Adapter zu. In der Ereignisverwaltung des Baumes, muss also immer diese Methode zu Hilfe genommen werden, um die eigentliche Quelle des Ereignisses in Erfahrung zu bringen. Diese Einschränkung beruht auf den spezifischen Eigenschaften der GTK-Bibliothek (in der aktuellen Version von GTK (GTK-2.0) wurde dieser Mangel behoben), und ist nicht von allgemeiner Natur.

4.2.3.2 Verfeinerung der allgemeinen Baumstruktur

Die modellierte allgemeine Baumstruktur des abstrakten Teams `Tree` ist für sich alleine genommen nicht nutzbar. Die abstrakten Methoden der Klasse `TreeNode` können nicht weiter verfeinert werden. Sie beinhalten die Logik, die von der jeweiligen Domäne bestimmt wird. Die Klasse `Tree` wurde sehr allgemein modelliert, damit spezielle graphische Baumformen von dem abstrakten Team erben und die abstrakten Methoden implementieren. Für die Projektmanagementsoftware wurden zwei Typen von graphischen Bäumen implementiert und benutzt.

- `SimpleTree`: Dieses Team realisiert die einfachste Repräsentation einer graphischen Baumstruktur: der einzelne Knoten besteht aus einer einfachen Zeichenkette, die dargestellt wird. Der Typparameter in `TreeNode` wird also an die Klasse `String` gebunden. Es werden keine graphischen Attribute unterstützt, so dass auch eine Erweiterung der Klasse `CellStyle` und `TreeRenderer` entfällt. Alle abstrakten Methoden der Klasse `Tree` wurden implementiert. Das gesamte Team bleibt aber wegen der abstrakte Rolle `TreeNode` weiterhin abstrakt.

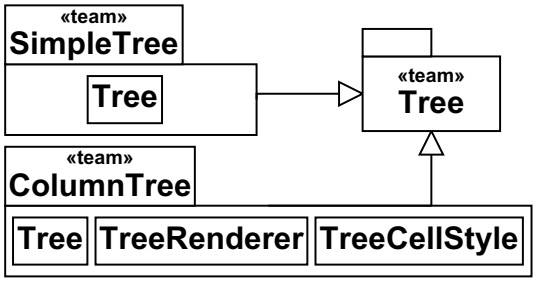

Abbildung 4.7: Verfeinerung der allgemeinen Baumstruktur

- `ColumnTree`: Eine Baumform, in der jeder Knoten eine Zeile in einer Tabelle darstellt, wird von diesem Team realisiert. Man kennt diese Form der Baumstruktur z.B. aus der Ablage seines EMail-Programms (siehe Abbildung 4.8). Das darzustellende Datum besteht aus einem Tupel aus Zeichenketten. Die

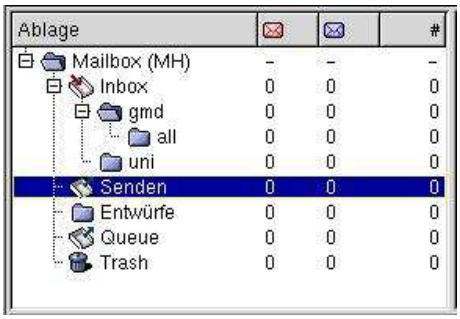

Abbildung 4.8: Spaltenbaumsicht in einer Emailablage.

  Größe des Tupels kann an dieser Stelle wieder nicht definiert werden und muss auf den bindenden Konnektor verschoben werden. Zu jedem Eintrag im Baum kann ein kleines Bild definiert werden. Es ist außerdem möglich die Farbe und den Typ der Schrift einer Zeile festzulegen. All diese Attribute werden in einer neuen Klasse manifestiert, die die Klasse `CellStyle` verfeinert. Auch muss die Klasse `TreeRenderer` angepasst werden, damit die einzelnen Attribute richtig dargestellt werden. Auch hier werden alle Methoden der Klasse `Tree` implementiert, wo hingegen die Klasse `TreeNode` weiterhin abstrakt bleibt.

Die drei so entstandenen Teams `Tree`, `SimpleTree` und `ColumnTree` wurden in Hinsicht auf rapide Anwendungsentwicklung und Wiederverwendbarkeit entworfen. Sie haben keine Beziehung zu einer speziellen Domäne. Sie definieren einen expliziten Vertrag (*'expected Interface'*), den der Benutzer der Teams erfüllen muss. Es existieren dabei prinzipiell immer zwei Wege, um diesen Vertrag zu erfüllen: (1) man verfeinert das Team, passt es an die speziellen Anforderungen einer speziel-

len Domäne an und implementiert die geforderte Schnittstelle oder (2) man bindet die geforderte Funktionalität an die Funktionalität der Domäne. Die beiden speziellen Baumformen `SimpleTree` und `ColumnTree` wurden so modelliert, dass nur noch die eigentliche Baumstruktur und der anzuzeigende Inhalt von der Domäne definiert werden muss. Hinzu kommt natürlich die Synchronisation des Modells mit der Repräsentation. Im einfachsten Fall muss nur noch ein Konnektor definiert werden, der die geforderte Schnittstelle bindet und die Synchronisation sicherstellt.

Anwendung der Baumsicht im Projektplan Um das Beispiel zu komplettieren, soll an dieser Stelle auch die Anwendung der erstellten Teams gezeigt werden. Im Falle von PromOTe wurde vor allem die Spaltenbaumsicht an mehreren Stellen verwendet. Der Projektleiter hat beispielsweise in der Projektplanung (siehe Anhang, Abbildung A.5) die Möglichkeit Meilensteine und Aufgaben zu definieren und diese einzelnen Mitarbeitern im Team zuzuweisen. Die einzelnen Aufgaben können dabei geschachtelt werden. Hiermit modelliert man die verschiedenen Granularitäten von Aufgaben. Eine „große“ Aufgabe kann in viele kleine zergliedert werden, was in jeder Ebene von Aufgaben möglich ist. Um eine Aufgabe zu erfüllen, müssen auch alle Unteraufgaben erfüllt sein. Das Resultat solch einer Definition ist ein Baum von Aufgaben. Dieser Baum von Aufgaben (und Meilensteinen die hier im Beispiel vernachlässigt werden, da alle Aussagen für Aufgaben ganz analog für Meilensteine gelten) soll in der Projektplanungssicht angezeigt werden. Der Projektleiter kann neue Aufgaben und Unteraufgaben definieren, existierende Aufgaben löschen, bzw. einzelne Aufgaben im Baum verschieben. Zu jeder Aufgabe in diesem Baum soll der Name (eine Art Kurzbeschreibung), der geplante Aufwand und der geplante Fertigstellungstermin angezeigt werden. Es sind außerdem Icons definiert, die den derzeitigen Zustand der Aufgabe ausdrücken sollen. Eine Aufgabe kann folgende Zustände haben:

- In Planung.
- Offen.
- In Bearbeitung.
- Abgeschlossen.

Die Planungsansicht für den Projektleiter soll einen schnellen Überblick über den Stand der Planung geben. Interessiert eine bestimmte Aufgabe genauer, wird die jeweilige Aufgabe angewählt. Ein neues Fenster öffnet sich, indem alle genauen Daten zu dieser Aufgabe festgehalten sind und geändert werden können.

Um diese Funktionalität zu erfüllen, wird das Team `ColumnTree` an die Domänenklassen gebunden, wie dies in Abbildung 4.9 zu sehen ist. Zu beachten ist, dass hier nur die Bindung der Klasse `TreeNode` an die Klasse `Task` explizit gezeigt wird. Die Bindung für die Meilensteine und das Projekt selber werden der Übersicht halber

nur verkürzt dargestellt. Um alle Aufgaben und Meilensteine eines Projektes sichtbar zu machen, spielt die Klasse `Project` die Rolle `Tree`. Bei dieser Bindung müssen keine abstrakten Methoden gebunden werden. Es ist nur eine Synchronisation zu spezifizieren. Das Projekt hält eine Liste von Meilensteinen, die wiederum Aufgaben und Meilensteine referenzieren können. Diese Liste des Projektes muss mit dem Baum synchronisiert werden. Wenn immer ein Meilenstein auf der obersten Ebene hinzugefügt oder gelöscht wird, muss der Baum neu gezeichnet werden. Die Rolle `TreeNode` wird an die Klassen `MileStone` und `Task` gebunden. Sie stellen die sichtbaren Elemente im Baum dar. Da die Bindung dieser Klassen etwas komplexer daher kommt, sollen die einzelnen Punkte Schritt für Schritt erläutert werden. Dies geschieht nur am Beispiel der Aufgaben, da die Bindung für Meilensteine sehr ähnlich ist.

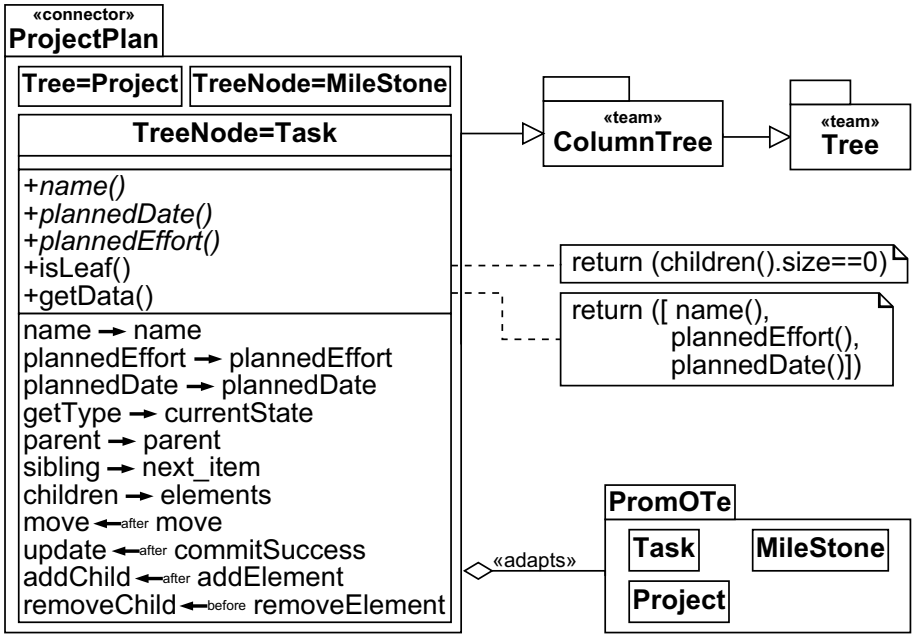

Abbildung 4.9: Der Projektplan als Spaltenbaum.

Der darzustellende Inhalt einer jeden Aufgabe besteht aus dem Namen der Aufgabe (einer Kurzbeschreibung), dem geplanten Aufwand in Tagen oder Stunden und dem geplanten Fertigstellungstermin. Diese Funktionalität wird mit `getData` erreicht. Es wäre möglich, diese Methode in der Klasse `Task` zu definieren und im Konnektor zu binden. Dies hätte aber zweierlei Nachteile: (1) dies bedeutet GUI-Funktionalität im Modell und (2) in einer anderen Baumsicht kann diese Methode u.U. nicht benutzt werden, wenn andere Informationen dargestellt werden sollen. Hier müsste also für jede View eine eigene Methode im Modell definiert werden. Eine einfache Lösung dieses Problems besteht in der Möglichkeit, die Methode im Konnektor selber zu definieren. Die Rolle `TreeNode` wird verfeinert und die ab-

strakte Methode `getData` implementiert. Diese spezifische Eigenschaft ist somit nur für diese konkrete Bindung vorhanden. Die View bleibt somit abstrakt und wiederverwendbar, das Modell benötigt kein Wissen über GUI-spezifische Eigenschaften.

Die Methode `getData` wird im Konnektor derart definiert, dass die gewünschten Einträge als Liste zurückgegeben werden. Da die Klasse `TreeNode` keine der anzuzeigenden Daten kennt, diese sind ja in der Klasse `Task` spezifiziert, werden diese Methoden wiederum abstrakt definiert und an `Task` gebunden (`name`, `plannedEffort` und `plannedDate`). Die Implementierung von bindungsspezifischen Funktionalitäten im Konnektor stellt eine sehr mächtige Technik dar und findet oft Anwendung.

Die zweite Methode die hier implementiert wird, ist die Methode `isLeaf`. In dieser Anwendung gibt es keine Einschränkungen für einzelne Knoten, ob dieser wiederum Unterknoten haben kann (wie dies z.B. bei einer Verzeichnisstruktur bei Datei und Ordner ganz klar definiert werden kann). Hier kann jede Aufgabe wiederum Unteraufgaben haben. Ob die Aufgabe ein Blatt in dem Baum ist, ergibt sich lediglich aus der Eigenschaft, ob die Aufgabe Unteraufgaben hat oder nicht. Die Methode `isLeaf` gibt also einen Wert zurück, der abhängig von der Anzahl der Unterknoten ist.

Eine ähnliche Technik wird auch für die Methode `getType` benutzt. Diese Methode wird vom `TreeRenderer` benutzt, um einen registrierten Stil auf den Knoten anzuwenden. Dieser Stil legt das Aussehen eines Knotens fest. Eine Aufgabe soll je nach Bearbeitungszustand verschieden dargestellt werden. So bekommt jede fertig gestellte Aufgabe beispielsweise ein Häkchen, jede Aufgabe in Bearbeitung eine grüne Ampelleuchte, jede Aufgabe in Planung wird grau dargestellt. Diese Stile müssen erstellt und beim `TreeRenderer` unter Angabe eines Typobjektes registriert werden. Als Typobjekt wird hier der Status einer Aufgabe genutzt. Dieser Status ist ein Aufzählungstyp (*'enumeration type'*) und kann die erwähnten Stati annehmen (in Planung, Offen, In Bearbeitung, Abgeschlossen). Die Methode `getType` wird aus diesem Grund an die Methode `currentState` gebunden. Zum zeichnen einer Aufgabe wird also ein Stil genutzt der für den derzeitigen Status derselben hinterlegt wurde.

Auf funktionaler Ebene sind nun alle Methoden implementiert, die spezifisch für die Bindung der Klasse `TreeNode` an die Klasse `Task` sind. Alle weiteren Funktionalitäten sind im Modell selbst vorhanden und müssen vom Konnektor gebunden werden, damit die erwartete Schnittstelle von `TreeNode` erfüllt wird. Diese Funktionalität betrifft die Baumstruktur selber: die Methoden `parent`, `sibling` und `children` müssen gebunden werden. Diese Funktionalität dieser Methoden ist in der Klasse `Task` vorhanden, wenn auch unter anderem Namen. Die Klasse `TreeNode` wird mit dieser Bindung effektiv und somit das gesamte Team. Die Synchronisation von `TreeNode` und `Task` muss aber trotzdem noch definiert werden.

Zu diesem Zweck hat die Klasse `TreeNode` vier Trigger-Methoden implementiert, die vom Konnektor gebunden werden:

- `update` ←**after** `commitSuccess`:
  Jedes Domänenobjekt in PromOTe kann nur über einen transaktionalen Mechanismus, in einen zweistufigen Commit-Verfahren verändert werden. Dieses Verfahren ermöglicht transaktionale Zustandsübergänge, die nur valide Zustände zulässt. Am Ende solch einer Transaktion wird immer die Methode `commitSuccess` aufgerufen. Sie signalisiert, dass sich der Zustand des Objektes verändert hat. Eine Bindung an diese Methode stellt sicher, dass der dargestellte Inhalt mit dem zugrunde liegenden Modell immer übereinstimmt.

- `move` ←**after** `move`:
  Mit der Methode `move(beneath, nextTo)` der Klasse `Task` kann eine existierende Aufgabe, unter Angabe des neuen Eltern- und Geschwisterknotens, im Baum bewegt werden. Die Projektplansicht stellt die Funktionalität zur Verfügung, dies auch graphisch zu tun. Jede Bewegung einer Aufgabe, auch die graphisch unterstützte, verändert trotzdem immer nur das Modell. Die Bindung an `move` stellt sicher, dass die neue Position der Aufgabe auch graphisch repräsentiert wird.

- `addChild` ←**after** `addElement`:
  `removeChild` ←**before** `removeElement`:
  Diese beiden Bindungen synchronisieren jeden einzelnen graphischen Knoten mit dem Knoten des Modells. Immer wenn ein Knoten als Unterknoten hinzukommt oder gelöscht wird, muss der jeweilige graphische Knoten neu gezeichnet, bzw gelöscht werden.

Mit dieser Bindung können die Aufgaben in einem Projekt graphisch dargestellt werden. Eine äquivalente Bindung erfährt die Klasse `MileStone`, so dass auch alle Meilensteine in der Projektplansicht korrekt dargestellt werden. Im Konnektor muss nun noch die Controllerfunktionalität implementiert werden, damit diese View ihren Zweck erfüllt. Das Team `ProjectPlan` kann nun instantiiert und aktiviert werden. Die Aktivierung ordnet jeder Aufgabe und jedem Meilenstein eines Projektes ein Objekt der Klasse `TreeNode` zu, die in einem Objekt der Klasse `Tree` dargestellt werden, die dem Projekt zugeordnet wurde. Die Domänenklassen wie auch das Team `Tree` und `ColumnTree` wurden dabei nicht verändert.

4.2.3.3 Komposition von Repräsentationen

Am Beispiel einer abstrakten Baumrepräsentation wurde gezeigt, wie graphische Standardelemente an Klassen der Domäne gebunden werden können. Für die Projektmanagementsoftware PromOTe wurden die dort benutzten Standardkomponenten als Rollen implementiert. Es bleibt nun zu zeigen, dass dieses Konzept der

Rollenbindung nicht nur für einzelne Standardelemente genutzt werden kann, sondern dass damit genauso auch komplexe Schnittstellen implementiert werden können.

Eine typische graphische Benutzeroberfläche besteht in der Regel aus einem Fenster, einem Statusbalken, einer Werkzeugleiste, einem Menü und natürlich der graphischen Darstellung einzelner Entitäten der Domäne. Solche komplexen Anzeigeelemente werden typischerweise durch Aggregation bestehender Standardelemente zusammengesetzt. Durch Synthese von einzelnen Standardelementen entsteht auf diese Weise ein neues komplexes Anzeigeelement. In dieser Schnittstelle werden u.U. mehrere Entitäten der Domäne graphisch repräsentiert. Eine Bindung solcher komplexen Elemente an eine Menge von Entitäten der Domäne ist dabei aber genau so einfach wie für einfache Standardelemente. Das erwartete Interface ist hier in der Regel nur umfangreicher. Der Konnektor muss in diesem Fall eine Menge einzelner Bindungen zusammenfassen — jede Bindung entspricht einer Repräsentation einer Entität der Domäne.

Da sich die Bindung einfacher Anzeigeelemente von komplexen Anzeigeelementen nur in ihrem Umfang unterscheidet, besteht auch die Möglichkeit GUI-Builder zu benutzen. Diese Werkzeuge erlauben das interaktive erstellen einer graphischen Benutzeroberfläche. Jeder, der einmal eine etwas komplexere Oberfläche manuell erstellt hat, weiß um den Vorteil eines solchen Hilfsmittels. Die Ausgabe solcher GUI-Builder ist in der Regel generierter Quellcode oder eine formale Beschreibung, die mit Hilfe des zugehörigen Toolkits geladen werden kann. Beide Formen sind in der Regel nicht für eine manuelle Nachbereitung geeignet. Diese negative Eigenschaft kann man mit Hilfe von Object Teams geschickt umgehen, da die Bindung solcher Sichtelemente sowieso extern vorgenommen wird. So muss in der View immer nur die erwartete Schnittstelle deklariert werden, welche dann vom Konnektor gebunden werden kann. Für PromOTe wurde von der Kombination von GUI-Builder und Object Teams intensiv Gebrauch gemacht.

In Abbildung 4.10 ist am Beispiel der Projektansicht das generelle Vorgehen für die GUI-Programmierung in PromOTe zu sehen. Es wurde ein Team `View` erstellt, in dem alle möglichen Sichten der einzelnen Entitäten der Domäne implementiert sind. Solch eine Sicht besteht als Wurzelelement immer aus einem `Frame`, der in verschiedene Umgebungen gebettet werden kann. Dabei gibt es grundsätzlich die Unterscheidung zwischen einfacher Sicht und Bearbeitungssicht. In einer einfachen Sicht werden die jeweiligen Entitäten nur repräsentiert. Die Möglichkeit, die einzelnen Daten der Sicht zu bearbeiten, wurde in einer Klasse implementiert, die von der jeweiligen Anzeigesicht erbt. Die hier vorkommenden graphischen Elemente erlauben eine Bearbeitung. Für eine Repräsentation als einfache Zeichenkette oder als Listenelement oder als Baumelement gibt es allgemeine wiederverwendbare Rollen, die im Konnektor für die Bindung an die jeweilige Entität angepasst werden

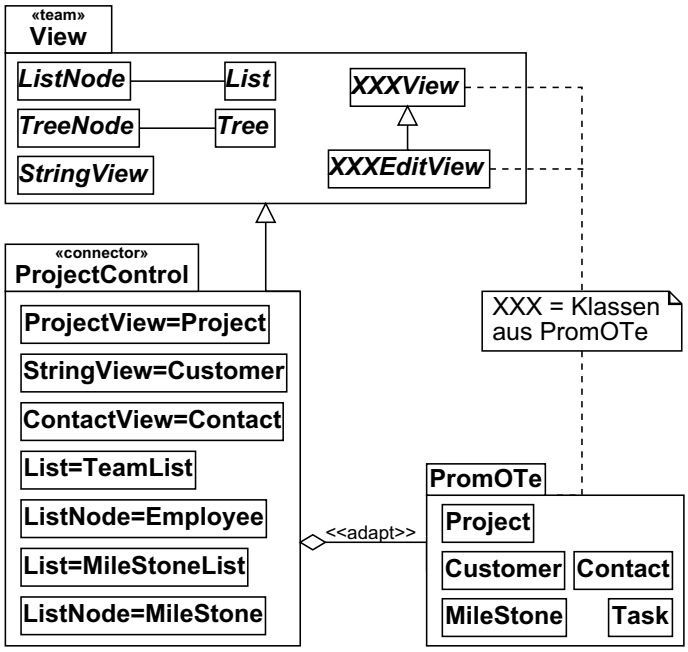

Abbildung 4.10: Bindung komplexer Anzeigeelemente

müssen. Mit diesem Satz von Anzeigeelementen können nun einzelne graphische Schnittstellen jeweils neu zusammengesetzt werden.

Abbildung 4.10 zeigt solch eine Zusammenstellung einzelner Sichten für die Projektansicht. Eine Projektansicht[3] besteht aus vier Reitern (*'tabbed pane'*), die Informationen zum Projekt offerieren.

- Projektdaten: Hier werden die organisatorischen Daten des Projektes wie Name, Beschreibung, Anfang und geplantes Ende, derzeitiger Status etc. dargestellt. Die Darstellung ist in der Klasse `ProjectView` implementiert und wird an das aktuelle Projekt gebunden.

- Team: Dieser Reiter zeigt eine Liste aller Angestellten, die in diesem Projekt arbeiten. Hier wurde die abstrakte Listenimplementierung an die Mitarbeitersicht angepasst. Die Klasse `TeamList` hält alle Mitarbeiter des Projektes. Diese Klasse bekommt die Rolle `List`. Die Klasse `Employee` spielt in dieser Konstellation die Rolle `ListNode`. Diese Rolle wird so angepasst, dass von jedem Mitarbeiter der Name, der Vorname und der Loginname sichtbar ist.

[3] Um sich ein Bild von der realisierten Projektansicht zu machen, wurden Bildschirmfotos im Anhang A.2 abgebildet

- Projektplan: Ein Projekt besteht unter anderem aus einer Liste von Meilensteinen. Im Gegensatz zur Projektplansicht, wo unter jedem dieser Meilensteine beliebig verschachtelte Aufgaben sichtbar sind, wird hier nur eine flache Liste gezeigt. Es werden zu jedem Meilenstein Informationen über den Status, den geplanten und den vereinbarten Fertigstellungstermin, den geplanten Aufwand und den jeweiligen Fehl-, bzw. Überstunden gezeigt. Diese Liste gibt also einen schnellen Überblick über den derzeitigen Stand des Projektes. Wie bei der Listensicht des Teams wird hier die Rolle `List` an die Klasse `MileStoneList` und die Rolle `ListNode` an die Klasse `MileStone` gebunden, um dies zu ermöglichen.

- Kundenkontakt: Der Reiter Kundenkontakt gibt Information über den jeweiligen Kunden und zeigt den zugeordneten Kundenkontakt. Der Kunde wird als einfache Zeichenkette (*'label'*) dargestellt. Die abstrakte Rolle `StringView` wird an die Klasse `Customer` gebunden, um diese Sicht zu realisieren. Der Kundenkontakt soll detailliert angezeigt werden. Deshalb wird die Rolle `ContacView` an die Klasse `Contact` gebunden, welche die Informationen über den Kundenkontakt explizit darstellt.

Der Konnektor `ProjectControl` fasst diese einzelnen Bindungen zusammen. Damit das Projekt richtig dargestellt wird, ist eine Initialisierung des zugehörigen Fensters mit den einzelnen Sichten nötig. Die Klasse `ProjectControl` verwaltet ein Fenster, in der er es eine leere Reiterstruktur gibt. Die einzelnen Frames der einzelnen Views werden in diese Reiterstruktur eingebettet. Für die korrekte Anzeige des Projektes sind die Bindungen zuständig. Im Konnektor wird nur noch Controllerfunktionalität implementiert. Im Beispiel der Projektansicht besteht diese Funktionalität im wesentlichen nur aus der Fensterverwaltung. Dies liegt an der eingeschränkten Funktionalität einer einfachen Sicht - sie dient nur der Anzeige, nicht dem Bearbeiten. In der Bearbeitungssicht eines Projektes können die einzelnen Daten über die Sichtelemente geändert werden. Außerdem kommen noch verschiedene Knöpfe hinzu, die die Bearbeitung vereinfachen. Die dahinter stehende Controllerfunktionalität ist im Konnektor `EditProjectControl` implementiert.

Am Beispiel der Projektansicht wurde gezeigt, wie einzelne Entitäten der Domäne repräsentiert werden und wie diese Repräsentationen zu einer graphischen Benutzerschnittstelle aggregiert werden können. Die Anordnung der einzelnen Views, also der Rollen des Teams `View`, ist dabei spezifisch für die Projektansicht. Jedes Fenster in PromOTe bindet eine Untermenge dieser Rollen — je nachdem welche Entitäten der Domäne in diesem Fenster angezeigt werden sollen. Die einzelnen Views sind dabei untereinander synchronisiert. D.h. wo immer sich der Zustand eines Modells ändert, ändern sich alle Views dieses Modell automatisch, als eine Folge der gewobenen Callin-Bindungen für die Synchronisation. Als Programmierer einer graphischen Benutzeroberfläche muss hier nicht mehr auf die Synchronisation aller einzelnen dargestellten Entitäten der Domäne geachtet werden. Diese kann

bei einer komplexen Sicht mit vielen Entitäten schnell unübersichtlich werden und führt außerdem zu einem erhöhten Aufwand, da immer das ganze Fenster, also alle dargestellten Views aktualisiert werden, obwohl sich u.U. nur ein kleiner Teil geändert hat. Die Granularität dieser Synchronisation wird auf ein überschaubares Maß reduziert: Die Synchronisation einer einzelnen Entität mit ihrer Repräsentation.

4.2.3.4 Erweiterte Techniken für MVC

Das Laufzeitsystem von Object Teams stellt erweiterbare Funktionalitäten zur Verfügung, die gerade für die Programmierung einer graphischen Benutzeroberfläche von großem Nutzen sind. Diese einzelnen Funktionalitäten sind über die Basisklasse `Team` verfügbar so dass keine neuen Sprachkonstrukte nötig sind.

Schwaches Lifting Das grundlegende Konzept von Object Teams beruht auf der Bindung von Rollenklassen an Klassen der Domäne. Wird die View als Rolle begriffen und an das Modell durch einen Konnektor gebunden, meint man in der Regel die Sicht für genau eine Instanz einer Domänenklasse und nicht für alle Instanzen der Klasse. Wird beispielsweise die Projektansicht aktiviert, sollen nicht alle Projekte des Systems visualisiert werden, sondern genau ein ausgewähltes Projekt. Es gibt offensichtlich einen Konflikt zwischen Klasse und Instanz. Damit die Sicht für eine Instanz realisiert werden kann, muss die gesamte Klasse verändert werden, damit eine Synchronisation möglich ist. Im klassischen MVC-Paradigma existiert genau das gleiche Dilemma. Soll eine Domänenklasse durch eine View visualisiert werden, muss eine explizite Beobachterinfrastruktur in der Klasse implementiert werden. Die Anmeldung als Beobachter funktioniert hingegen auf Instanzebene. Es wird also eine Funktionalität implementiert, die nur von einer Untermenge aller Instanzen der Klasse genutzt wird. Um dieses Konzept für Object Teams zur Verfügung zu stellen, wird ein Mechanismus bereit gestellt, der eine explizite Kontrolle über die Rollenerzeugung erlaubt. Ein Rollenverhalten wird für eine Instanz einer Domänenklasse nur dann wirksam, wenn ihre Rolle existiert. Für die Rollenerzeugung ist das Team verantwortlich. Wird eine Methode mit Callin-Bindung aufgerufen, so wird das Basisobjekt zu ihrer Rolle geliftet. Eine neue Rolle wird erzeugt, wenn diese noch nicht existiert. Es existiert nun die Möglichkeit in Object Teams, diesen Lifting-Mechanismus abzuschwächen. Eine Basis wird nur zu ihrer Rolle geliftet, wenn das Rollenobjekt schon existiert (siehe 3.2.3.3). Existiert kein Rollenobjekt, kann die Instanz nicht von der Rollenfunktionalität profitieren. Dieser Mechanismus wird für die Erzeugung von Views genutzt. Ein Konnektor liftet ein ausgewähltes Basisobjekt zu seiner Rolle. D.h. es wird eine View für ein bestimmtes Modell erzeugt. Nachdem die View erzeugt wurde, wird der Lifting-Mechanismus des Konnektors abgeschwächt, so dass nur Objekte mit einer existenten View synchro-

nisiert werden. Instanzen der gleichen Klasse ohne eine graphische Repräsentation sind davon nicht betroffen.

Zeitweilige Deaktivierung Bei der Definition der Synchronisation von Modell und View scheint es zwei konkurrierende Designentscheidungen zu geben. Auf der einen Seite soll jede kleinste Änderung sofort eine Änderung der View hervorrufen. Auf der anderen Seite sollen viele, zeitlich nahe Änderungen in gebündelter Form übernommen werden, damit z.B. die Oberfläche nicht hakt bzw. flickert. In der aspektorientierten Welt hat dieses Phänomen den Namen *'Jumping Aspects'* (siehe 3.2.4). Die Granularität der Observerstruktur korreliert offensichtlich mit der Granularität von *'Join-Points'*. Object Teams bietet hier die Möglichkeit der kurzzeitigen Deaktivierung eines Teams. In einer Rollenmethode können die Methoden `suspend` und `resume` benutzt werden, um die Deaktivierung und Aktivierung zu kontrollieren. Auf diese Weise ist es möglich, über verschiedene Eintrittspunkte in die Rolle ein unterschiedliches Rollenverhalten zu erzwingen. Um z.B. neue Mitglieder in einem Team sichtbar zu machen, könnte es zwei Methoden in der Listenrolle geben: `addNode`, welche immer dann aufgerufen wird, wenn ein einzelner Mitarbeiter ins Team aufgenommen wird und `addNodeList` um eine ganze Gruppe von Mitarbeitern anzuzeigen, die neu im Team sind. Um zu verhindern, dass der Trigger für jeden einzelnen Mitarbeiter ausgelöst wird, könnte die Methode `addNodeList`, das Team kurzzeitig deaktivieren, für jeden neuen Mitarbeiter `addNode` aufrufen, dann die Liste neu zeichnen und das Team wieder aktivieren. Der Programmierer des Teams hat somit die volle Kontrolle darüber, wann solch eine zusammengesetzte Aktion den Triggermechanismus atomarer Aktionen außer Kraft setzen soll.

4.2.4 Zusammenfassung

Das MVC-Paradigma steht für eine saubere Trennung einzelner Zuständigkeiten — einer klaren Architektur. Es wurde gezeigt, dass solch eine Trennung eine Entkoppelung der einzelnen kollaborierenden Entitäten voraussetzt, die in streng getypten objektorientierten Sprachen nicht erzielt werden kann (die dynamischen Eigenschaften von Ruby wurden hier bewusst nicht verwendet). Das Object Teams Paradigma führt eine neue Art der Bindung von Klassen ein, die die geforderte Entkoppelung ermöglicht. Die existierenden Kommunikationskanäle werden durch Callin- bzw. Callout-Bindungen ersetzt. Die Entkoppelung schließt die Definition aller Zuständigkeiten an genau der Stelle ein, wo die Anforderungen entspringen. So wird die Synchronisation der Repräsentation mit dem Modell in der View selbst definiert und nicht im Modell.

Die Implementierung einer View als Rolle eines Teams ermöglicht eine abstrakte Definition derselben, die auf keinem spezifischen Typ von Modell arbeitet. Jede View definiert eine erwartete Schnittstelle, die vom Modell erfüllt werden muss.

Die so definierten Views sind in einem höherem Maße wiederverwendbar, als dies mit klassischen objektorientierten Mitteln möglich wäre. Es besteht die Möglichkeit der Anpassung von abstrakten Rollen im Konnektor, so dass die View abstrakt und das Modell unabhängig von der View definiert werden kann. Durch Rollenbindung kann Funktionalität der Rolle an die Basis gebunden werden. Dies ist in Sprachen mit einfacher Vererbung schwierig, da hier Kontrakte in der Regel als Schnittstelle definiert werden, welche keine Implementierung zulassen.

Die Modellierung von Views als Rolle einer Domänenklasse beinhaltet eine aussagekräftige Metapher und gibt dem Programmierer einen Leitfaden für ein gutes Design in die Hand. Für die Realisierung der verschiedenen Abstraktionsebenen, im Domänenmodell sowie in der Modellierung der Views, und der automatischen Kontext- und Identitätswahl durch die Technik des Lifting und Lowering, ist eine Technik vorhanden, die den Programmierer unterstützt und die Komplexität des Gesamtsystems reduziert. Die verschiedenen Standarddienste eines Teams haben sich in der Praxis als sehr vielseitig und nützlich herausgestellt.

4.3 Persistenz von Objekten

In einem System wo Informationen verwaltet werden sollen, müssen die anfallenden Daten gespeichert werden. Sollen die Daten unabhängig von der Laufzeit eines Programms verfügbar sein, benötigt man eine permanente Speicherung, z.B. als Datei auf der Festplatte oder als Einträge in einer Datenbank. Die Abbildung der Daten im Hauptspeicher auf die Daten im Permanentspeicher und umgekehrt nennt man Persistenz. Wird eine Domäne objektorientiert modelliert, werden die dort vorkommenden Entitäten typischerweise als Klassen gekapselt. Instanzen dieser Klassen repräsentieren somit eine konkrete Entität in der Domäne, mit all ihrer Funktionalität und all ihren Daten — das Grundprinzip der Objektorientierung. Eine permanente Speicherung von Daten sollte in einer objektorientierten Umgebung also Speicherung von Objekten bedeuten.

Das Konzept „Objekt“ ist ein Laufzeitkonzept. Objekte existieren nur während der Ausführung eines Programmes. Sollen Objekte permanent gespeichert werden, muß eine Abbildung des Objektes auf den Permanentspeicher erfolgen. Man nennt diesen Prozess *Passivierung*. Es wird eine weitere Abbildung benötigt, um die gespeicherten Daten wieder in das Objekt zu verwandeln. Dieser Prozess heißt *Aktivierung*. Je nach Programmiersprache, Anwendung und technischen Rahmenbedingungen gibt es eine ganze Reihe von möglichen Abbildungen und Speichermedien. Moderne Persistenzschichten trennen aus diesem Grund beide Funktionalitäten und erlauben die Wahl verschiedener Abbildungen und verschiedener Speichermedien (wobei nicht jede Kombinationen möglich ist, bzw. Sinn macht).

Ein Persistenzmechanismus ist eine typische Systemfunktionalität und sollte als Dienstleistung der Laufzeitumgebung aufgefasst werden. Komponentensysteme wie z.B. Enterprise Java Beans [Sun Microsystems 2002a] oder das Corba Component Model offerieren solch eine Dienstleistung völlig transparent (*'container managed persistence'* [Stearns 2001]). Eine einfache Deklaration im Deployment Deskriptor reicht hierfür aus. Solch ein Komponentencontainer generiert die nötige Funktionalität um die jeweilige Klasse herum, so dass der Entwickler davon befreit ist. Er nutzt dabei in der Regel einen speziellen Persistenzschichten. Diese Persistenzschichten beruhen auf Techniken wie Reflexion, bzw. Generierung von Code. Der Programmierer hat weitreichende Möglichkeiten in diesen Prozess einzugreifen.

Anstelle solcher hoch spezifischen und äußerst komplexen Techniken soll ein vergleichsweise sehr leichtgewichtiger Persistenzmechanismus vorgestellt werden, der eine ebenbürtige Funktionalität bereitstellt. Er beruht auf der Idee, dass auch die Funktionalität der Persistenz allgemein in einem Team definiert werden kann. Diese Herangehensweise verzichtet auf Deskriptoren, die in ihrer Komplexität schon als Spracherweiterung aufgefasst werden können. Die Ausdrucksmittel von Object Teams reichen für die Definition der einstellbaren Parameter völlig aus.

Der hier vorgestellte Persistenzmechanismus wird in PromOTe verwendet, um die dort existierenden Domänenobjekte persistent zu machen. Als Speichermedium wurde eine relationale Datenbank gewählt. Dies ist vor allem dem Fakt geschuldet, dass es für Ruby bis dato keine technisch hochwertigeren Möglichkeiten, wie z.B. eine objektorientierte Datenbank, gibt. Der Vorteil dieser Entscheidung ist die hohe Akzeptanz und Verfügbarkeit verschiedener relationaler Datenbanken. Um die Anforderungen und Probleme bei der Abbildung von Objekten in Tabellen einer relationalen Datenbank besser zu verstehen, werden in Kapitel 4.3.1 Grundbegriffe und Techniken erläutert, um dann in Kapitel 4.3.2 das generelle Vorgehen zu beschreiben. Kapitel 4.3.3 führt das Team `Persistence` ein und zeigt ganz allgemein, wie man dieses Team an Klassen der Domäne bindet. Zum Schluss wird gezeigt, wie dieses Team in PromOTe verwendet wird.

4.3.1 Relationale Datenbanken als Persistenzmedium

Relationale Datenbanken[Oheim 1999, Relationale Datenbanken] beruhen auf der mathematischen Mengentheorie. Eine Datenbank besteht aus einer Reihe von Tabellen, wobei jede Tabelle eine Menge von Daten enthält. Über diese Tabellen, genauer gesagt über die Mengen der in den Tabellen enthaltenen Daten, können Relationen erstellt werden, was zu einer neuen Menge von Daten führt. Man spricht aus diesem Grund von *relationalen Datenbanken*. Die Anordnung von Daten in Tabellen und die vielen Möglichkeiten der Verknüpfungen und Selektionen erlauben eine sehr flexible Dateninterpretation. Langjährige Entwicklungen auf dem Gebiet der Datenbankoptimierung haben zudem dafür gesorgt, dass relationale Datenbanken extrem leistungsfähig und effizient sind.

Um Daten in einer Tabelle eindeutig zu referenzieren, werden eindeutige Attribute, so genannte Schlüsselattribute gefordert. Mit Hilfe eines Schlüsselattributes kann somit ein kompletter Datensatz referenziert und gefunden werden. Die Angabe einer ISBN-Nummer in einer Buchhandlung reicht z.B. als eindeutige Referenz für einen bestimmten Buchtitel völlig aus. Von dem zugrunde liegenden Datenmodell muss folgendes gefordert werden: sie enthält keine redundanten Daten, keine Seiteneffekte, keine Mehrdeutigkeiten und keine Inkonsistenzen. Sind diese Punkte erfüllt, spricht man von einem normalisierten Datenmodell [Oheim 1999, relationale Normalform].

Werden Daten in einer Datenbank verändert, laufen diese Veränderungen immer in einer *Transaktion* [Oheim 1999, Transaktionskonzepte, Integritäten, Protokolle]. Eine Transaktionen fasst einzelne Änderungen von Daten in der Datenbank zu einer einzigen Aktion zusammen. Eine Transaktion wird nur dann effektiv, wenn jede einzelne Teilaktion erfolgreich ist. So ist es mit Hilfe von Transaktionen möglich, nur valide Zustandsübergänge von Daten zu erlauben. Der Vorgang des Geldüberweisens beispielsweise, besteht aus der Aktion: (1) Betrag x von Konto A abbuchen und

(2) Betrag x auf Konto B buchen. Eine Überweisung ist nur dann erfolgreich, wenn beide Aktionen durchgeführt werden können. Beide Änderungen hängen voneinander ab und haben nur gemeinsam Gültigkeit. Man spricht im Zusammenhang von Transaktionen immer von *ACID*-Kriterien, die bei einer Transaktion sicher gestellt werden muss. Der Name ergibt sich aus den Anfangsbuchstaben der vier Regeln:

- *'Atomicity'*: Eine Transaktion ist immer atomar. Das bedeutet sie wird als eine Aktion aufgefasst, deren einzelne Teile entweder komplett oder gar nicht ausgeführt werden.
- *'Consistency'*: Eine Datenbank wird mit Hilfe von Transaktionen immer von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Während der Transaktion muß diese Regel nicht erfüllt sein.
- *'Isolation'*: Transaktionen können immer als isoliert betrachtet werden. Das bedeutet, es müssen keine weiteren Transaktionen berücksichtigt werden, die u.U. zur gleichen Zeit laufen. und die gleichen Daten verändern.
- *'Durabity'*: Als eine Folge der Konsistenzbedingung besagt diese Regel, dass nur konsistente Zustände wirklich gespeichert werden.

Die genannten Techniken machen relationale Datenbanken universell einsetzbar und sehr robust. Dies sind wohl die Gründe, warum sich diese Datenbanken so stark durchgesetzt haben. Die Abbildung von Objekten auf Tabellen birgt aber eine Menge Probleme, die unter dem Begriff *Impedance Missmatch* [Oheim 1999, Kopplungen von Programmiersprachen und Datenbanken] bekannt geworden sind: (1) der Zugriff auf Daten in einer Datenbank ist mengenorientiert und nicht satzorientiert wie in einer Programmiersprache und (2) erfolgt die Referenzierung in einer Datenbank deskriptiv, die einer Laufzeitumgebung aber direkt. Eine Abbildung von Objekten auf Daten in einer Tabelle muss diese Probleme lösen.

4.3.2 Persistenz von Objekten in einem RDBMS

Eine Klasse in einer objektorientierten Sprache kapselt Eigenschaften und Daten, die spezifisch sind für eine bestimmte Entität. Objekte sind Instanzen einer Klasse, also eine einmalige, spezifische Ausformung der in der Klasse definierten Eigenschaften. Um das Laufzeitkonzept „Objekt“ in eine Datenbank abzubilden, müssen alle Eigenschaften des jeweiligen Objektes gespeichert werden. Unterschlägt man bei einer Klasse die Eigenschaften, so ist der schablonenhafte Charakter einer Klasse auf eine Tabelle übertragbar. Beide sind eindeutig referenzierbar und definieren einen bestimmten Satz von Daten, welches jedes Element hat. In diesem Sinne können Instanzen einer Klasse als Zeilen in einer Tabelle angesehen werden - wieder nur die Daten und nicht die Eigenschaften betrachtet. Jedes Objekt ist einmalig in der

Laufzeitumgebung, es hat eine eindeutige Identität. Diese Identität muss berücksichtigt werden. Da Objekte nun nicht mehr an die Laufzeitumgebung gebunden sind, wird eine allgemeinere Identität benötigt. Dieses Problem wird in den meisten objektorientierten Datenbanken durch ein '*Surrogate*' [Oheim 1999, Objektorientierte Datenbanken] umgangen. Jedes Objekt bekommt zusätzlich ein weiteres Attribut, welches das Objekt eindeutig, über Laufzeitgrenzen hinweg, referenziert.

Wenn eine Klasse als Tabelle und ein Objekt als Eintrag in der Tabelle abgebildet wird, muß weiterhin die Normalform der Tabelle beachtet werden. Das Problem erschließt sich in Abbildung 4.11. Das Objekt `a1` hält eine Referenz auf `a2`. Wenn

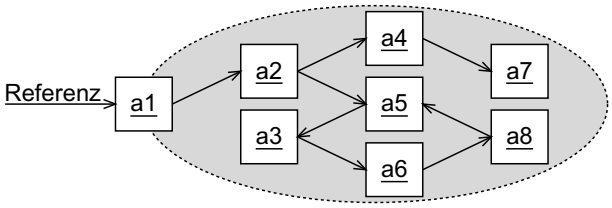

Abbildung 4.11: Mögliche Laufzeitstruktur eines Objektes

`a1` gespeichert werden soll, muß auch `a2` gespeichert werden. Um die Normalform nicht zu verletzen, muss jedes Objekt für sich gespeichert werden. Es könnte ja sein, dass noch eine andere Referenz auf `a2` zeigt, so dass das gleiche Objekt zweimal an zwei unterschiedlichen Orten gespeichert werden würde, was erstens redundant ist und zweitens die Konsistenzregel bricht. Da `a2` wiederum andere Objekte referenziert, müssen auch diese gespeichert werden - als Folge der Speicherung von `a1`. Betrachtet man den Graph der erreichbaren Objekte von `a1` aus gesehen, wobei die Kanten Referenzen der einzelnen Objekte sind, so ergibt sich die Menge der Objekte die einzeln gespeichert werden müssen, wenn `a1` persistent gemacht werden soll. Man bezeichnet diese Menge von erreichbaren Objekten als *transitive Hülle* von `a1`. Bei einer Speicherung eines Objektes müssen alle Objekte der transitiven Hülle gespeichert werden. Um die Normalform nicht zu verletzen, muß jedes dieser Objekte in die für die erzeugende Klasse vorgesehene Tabelle gespeichert werden.

Bei der Speicherung eines einzelnen Objektes kann es folgende Attribute geben, die auf die Datenbankstrukturen abgebildet werden müssen:

- Basistyp: Jede Programmiersprache und auch jede Datenbank definiert einen Satz von Basistypen, die in der Regel eins zu eins abgebildet werden können. Zu diesen Typen gehören in der Regel: Zeichenketten, Ganzzahlen, Fließkommazahlen, Datumseinträge etc.

- Referenzen auf andere Objekte: Die Referenzierung von Objekten kann nur emuliert werden und ist kein adäquater Ersatz zum Laufzeitkonstrukt. Objekte werden als Einträge einer Tabelle abgebildet. Diese Einträge sind eindeutig

referenzierbar. Die Eindeutigkeit wird durch ein *'Surrogate'* erreicht - eine vom System eindeutig vergebene Identität. Diese Identität kann als Fremdschlüssel in die Tabelle eingetragen werden, von wo aus die Referenz ausgeht. Objektreferenzen werden also als Datenbankreferenzen abgebildet. Die Art dieser Relation wird damit nicht abgebildet. Befindet sich das referenzierte Objekt z.B. in einer Kompositionsrelation, muß es aus der Datenbank gelöscht werden, wenn das referenzierende Objekt gelöscht wird. Die Referenz müsste zusätzlich attribuiert werden (z.B. exklusiv/teilbar, abhängig/unabhängig).

- Sammelattribut (Collection,Array): Sammelattribute müssen separat behandelt werden. Hier wird eine eigene Tabelle benötigt, die die einzelnen Elemente der Kollektion aufnimmt. Der Schlüssel für die Menge der referenzierten Objekte, bzw. Basistypen ist die Identität des Sammelattributs.

Ein Beispiel solch einer Abbildung von Objekten in einer Datenbank, kann man in Abbildung 4.12 sehen. Ein Objekt vom Typ `Person` soll gespeichert werden. Die

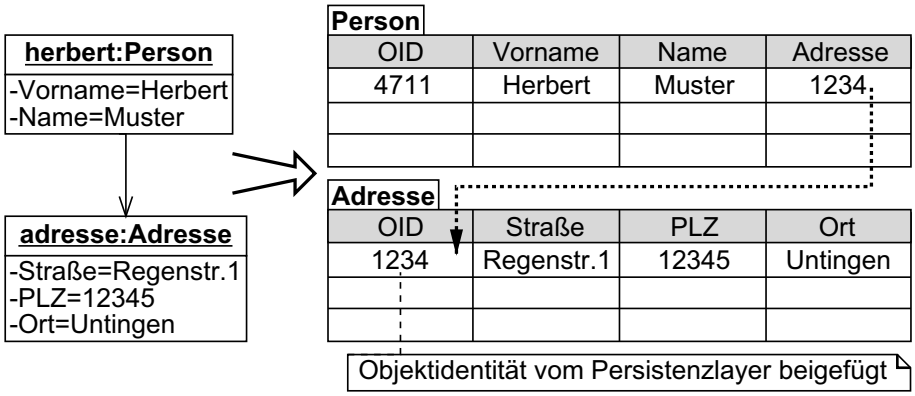

Abbildung 4.12: Abbildung von Objektzuständen in Tabellen einer Datenbank

Klasse `Person` kapselt einen Vornamen, einen Nachnamen und eine Adresse. Eine Adresse wird in einer eigenen Klasse modelliert und kapselt die Daten: Straße, Postleitzahl und Ort. Bei der Passivierung einer Instanz der Klasse `Person` müssen zwei Objekte passiviert werden. Das Objekt selber und das zugehörige Adressobjekt. Außer der Referenz auf die Adresse sind alle anderen Daten Basistypen und können direkt in der Datenbank abgebildet werden. Jedes persistente Objekt muß vom System eindeutig referenzierbar sein. Der Persistenzmechanismus vergibt für diese Objekte aus diesem Grund eine eindeutige Nummer, die als Identität im Sinne des Persistenzmechanismus angesehen werden kann und über die aktuelle Laufzeit hinweg Gültigkeit besitzt. Diese Identität ist ein vom System vergebenes, eindeutiges Schlüsselattribut (Primärschlüssel) und kann in anderen Tabellen zur Referenzierung genutzt werden. Bei der Speicherung einer Person wird die Referenz auf das Adressobjekt nur als Fremdschlüssel gespeichert, wobei die Adresse selber in

einer zweiten Tabelle gespeichert wird. Diese Abbildung ist reversibel, das heißt der Zustand des passivierten Personenobjektes kann wiederhergestellt und das Objekt aktiviert werden.

4.3.3 Ein Team für Persistenz

Jede Klasse, welches Persistenz unterstützen soll, muss eine Funktionalität bereitstellen, um den aktuellen Zustand eines Objektes zu extrahieren (für die Passivierung), bzw. einen externen Zustand zu setzen (für die Aktivierung). Es ist weiterhin ein eindeutiger Identifikationsmechanimsus zu einer Instanz dieser Klasse nötig. Um den Persistenzmechanismus nicht an die spezifischen Eigenschaften der persistenten Klassen anpassen zu müssen, soll eine Schnittstelle definiert werden, die die nötige Interaktion definiert. Ein Persistenzmechanismus arbeitet dann nur auf einer Schnittstelle, die von der jeweiligen Klasse implementiert werden muss. Da eine Domänenklasse kein Wissen über einen Persistenzmechanismus haben soll, wird dieser Vertrag in einer Rolle definiert. Diese Rolle kann nun an die jeweiligen Domänenklassen gebunden werden, wobei alle nötigen funktionalen Erweiterungen für den Persistenzmechanismus in der Rolle implementiert werden können. Die Domänenklasse bleibt somit unberührt und weiß nichts über Persistenz.

4.3.3.1 Abstrakte Teamvereinbarung

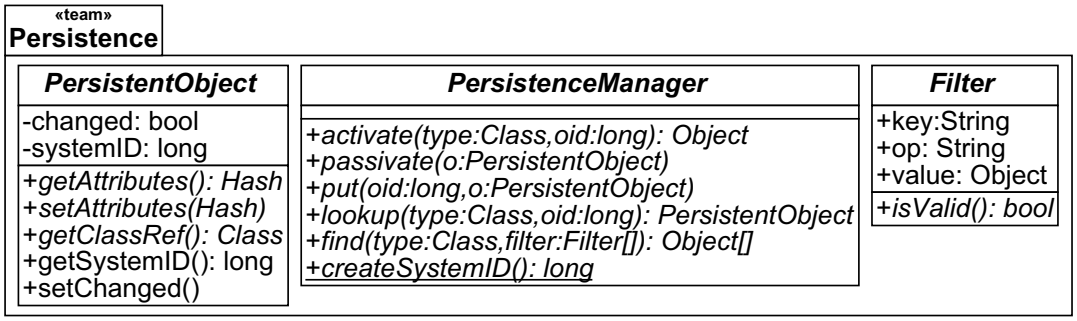

Abbildung 4.13: Ein Team für Persistenz

Abbildung 4.13 zeigt das abstrakte Team `Persistence`. Hier sind zwei Rollen definiert: `PersistentObject` und `PersistenceManager`. In der ersten Rolle ist die Funktionalität für den Zugriff auf den aktuellen Zustand der Basis mit den organisatorischen Daten definiert, die zweite Rolle deklariert die Funktionalität einen Zustand zu speichern, bzw. zu laden. Alle Methoden der Klasse `PersistenceManager` sind abstrakt deklariert. Es wurde an dieser Stelle noch

nicht definiert, welche Art der Abbildung und welche Art des Speichermediums genutzt werden soll.

Für welche Abbildung und für welches Speichermedium man sich auch immer entscheidet, es müssen folgende Funktionalitäten implementiert werden: Die Methode `createSystemID` erzeugt eine für den Persistenzmechanismus eindeutige Zahl, welche der Identifikation von Objekten dient. Jedes persistente Objekt bekommt solch eine eindeutige Identifikation, um darüber referenzieren zu können. Um eine Instanz vom Typ `PersistentObject` zu passivieren, existiert die Methode `passivate`. Das persistente Objekt wird nur dann passiviert, wenn sich der Zustand im Gegensatz zum passivierten Zustand auch wirklich verändert hat. Aus diesem Grund gibt es in der Klasse `PersistentObject` ein Indikatorattribut `changed`. Ist dieses gesetzt, ist der aktuelle Zustand nicht mehr synchron zum gespeicherten Zustand und muss aktualisiert werden. Mit der Methode `activate` kann ein passiviertes Objekt wieder aktiviert werden. Um das zu aktivierende Objekt eindeutig zu spezifizieren, ist die erzeugende Klasse und die Objektidentität nötig. Anhand dieser Informationen kann das passivierte Objekt gefunden und aktiviert werden.

Ist die Objektidentität nicht bekannt, gibt es die Möglichkeit Instanzen einer bestimmten Klasse nach ihrem Inhalt zu finden. Die Methode `find` deklariert solch eine Funktionalität. So kann über alle passivierten Instanzen einer bestimmten Klasse gesucht werden. Um das Suchergebnis einzuschränken, kann man dieser Methode eine Menge von Filtern übergeben, die das Suchergebnis einschränken. Ein Filter besteht dabei aus einem Attribut, einer Operation und einem Wert (z.B. Name=Muster). Treffen die Aussagen aller definierten Filtern für eine Instanz der gesuchten Klasse zu, ist diese Instanz Teil der Ergebnismenge. Die Methode `find` gibt immer eine Liste von Objekten zurück, auch wenn die Filter ein Objekt eindeutig markieren. Die Klasse Filter ist auch hier nur abstrakt deklariert, da verschiedene Persistenzmanager unterschiedliche Suchmöglichkeiten zulassen. Diese Klasse muss dort realisiert werden, wo auch die Klasse `PersistenceManager` realisiert wird.

Die von einem Persistenzmanager aktivierten Objekte werden nicht automatisch in einen Cache geschrieben. Diese funktionale Erweiterung wird auf die jeweilige Basis verlagert. Es ist also prinzipiell möglich, das selbe Objekt mehrmals zu aktivieren, was in Hinsicht auf Performanz und Identität keine gute Idee ist. Die Klasse `PersistenceManager` deklariert aus diesem Grund zwei abstrakte Methoden, um mit einem Cache zusammen zu arbeiten. Die Methode `activate` soll ein persistentes Objekt nur dann wirklich aktivieren, wenn die Methode `lookup` kein Objekt für die jeweilige Klasse und Objektidentität liefert, das Objekt nicht gecached vorliegt. Nachdem ein passiviertes Objekt aktiviert wurde, wird die Methode `put` mit diesem Objekt und dessen Identität aufgerufen — ein neuer Eintrag für den Cache. Je

nach Art der Anwendung kann mit Hilfe dieser beiden Methoden ein Cache realisiert werden, es besteht aber kein Zwang dies zu tun.

Die Rolle `PersistentObject` kapselt eine dem Basisobjekt zugehörige eindeutige Objektidentität (`systemID`), auf die mit der Methode `getSystemID` zugegriffen werden kann. Diese Identität erweitert den persistenten Zustand des Objektes. Sie kapselt weiterhin ein transientes Attribut (`changed`), welches Auskunft über die Synchronisation des passivierten Zustandes mit dem aktiven Zustand gibt. Um dieses Attribut aktuell zu halten, muß die Methode `setChanged` immer dann aufgerufen werden, wenn sich der Zustand der gebundenen Basis verändert hat. Sie muß also im Konnektor an alle Methoden der Basisklasse per Callin gebunden werden, die den Zustand verändern. Die Rolle definiert zwei abstrakte Methoden, die im Konnektor verfeinert werden müssen. `getClassRef` liefert eine Referenz auf das `Class`-Objekt der Basisklasse. Die Kombination aus Klasse und Objektidentität ist eine hinreichende Information, um ein persistentes Objekt zu referenzieren.

Um Zugriff auf den internen Zustand eines Objektes zu bekommen, muß die Methode `getAttributes` implementiert werden. Diese Methode liefert ein `Hash` [4]-Objekt, wobei die Schlüssel die Namen der internen Attribute und die Werte Referenzen auf diese Attribute sind (z.B. Vorname→Herbert, Name→Muster etc.). Dies stellt einen allgemeinen Mechanismus dar, generisch auf einen Objektzustand zuzugreifen. Der Hash stellt den internen Zustand eines Objektes dar. Genau solch ein Hash wird auch zur Aktivierung benutzt. Der Persistenzmanager speichert die Werte zu den einzelnen Attributen und kann diese wieder laden. Es wird ein neues Objekt erzeugt und der geladene Zustand gesetzt. Diese Möglichkeit wird mit der abstrakten Methode `setAttributes` definiert, die den geladenen Zustand in Form eines Hash übergeben bekommt. Wird ein Objekt persistent, wird der Typ, die Identität und der Zustand gespeichert. Aus diesen Daten lässt sich das passivierte Objekt rekonstruieren.

4.3.3.2 Persistenz für relationale Datenbanken

Um Persistenz in eine relationale Datenbank zu ermöglichen, muss die abstrakte Rolle `PersistenceManager` implementiert werden (siehe Abbildung 4.14). Die Methode `passivate` benutzt die Methoden `insertObject` und `updateObject` um eine Instanz vom Typ `PersistentObject` zu passivieren. Wird das Objekt zum ersten Mal passiviert, muß es in die Datenbank eingetragen werden (`insertObject`). Bei jeder weiteren Passivierung müssen die gespeicherten Daten nur aktualisiert werden (`updateObject`). Bei einer Passivierung werden drei Arten von Attributen unterstützt:

[4] Ein Hash realisiert eine assoziative Abbildung. Unter einem gegebenen Schlüsselobjekt kann ein Objekt hinterlegt werden. Mit diesem Schlüsselobjekt kann das Objekt erfragt werden.

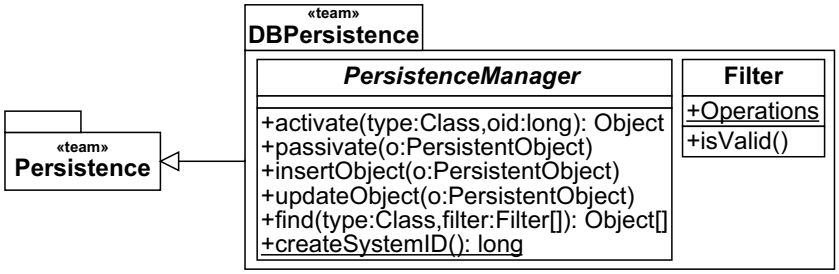

Abbildung 4.14: Persistenz in eine relationale Datenbank

- Basistypen: ganze Zahlen, Fließkommazahlen, Zeichenketten etc. können direkt auf Datentypen in der Datenbank abgebildet werden.

- Sammelattribut: alle Kollektionen, Listen und Mengen werden auf weitere Tabellen abgebildet, welche die Mengen repräsentieren. Jede Menge wird in einer Tabelle gespeichert, die durch Tupel in der Art von (Objektidentität, Wert in der Menge) repräsentiert werden. Die Menge ergibt sich also aus allen Werten, zu einer bestimmten Objektidentität. Trifft der Persistenzmanager bei der Aktivierung auf solch ein Sammelattribut, werden alle Werte der Menge zu dem Objekt geladen und als Kollektion dem Objekt verfügbar gemacht.

- Objektreferenzen: Um Persistenz für Objekte zu ermöglichen, existiert die Rolle `PersistentObject`, welche an die instantiierende Klasse des Objektes gebunden sein muss. Hier ist der Vertrag definiert, wie Objektreferenzen passiviert werden können. Die Objektidentität wird als eindeutiger Schlüssel (Primärschlüssel, bzw. Fremdschlüssel) in der Datenbank benutzt. Wird ein Objekt *o* von einem zu speichernden Objekt *r* referenziert, so müssen beide Objekte gespeichert werden. Die Referenz von *r* auf *o* wird als Datenbankreferenz realisiert, wobei die Objektidentität von *o* als Referenz in *r* eingetragen wird. Da bei der Passivierung eines Objektes alle Objekte der transitiven Hülle passiviert werden müssen, müssen auch alle diese Objekte in das Raster der drei unterstützten Attributtypen fallen: Basistyp oder Sammelattribut oder Objekte vom Typ `PersistentObject`. Ist dies nicht der Fall, kann das Objekt nicht gespeichert werden. Der Persistenzmanager speichert die einzelnen Attribute in eine Tabelle, deren Namen sich aus der instantiierenden Klasse ergibt. Da die transitive Hülle einen Graph von Objekten darstellt (es kann zirkuläre Referenzen geben), wird das Attribut `changed` der einzelnen `PersistentObject`-Objekte nach der Speicherung zurückgesetzt. Jede erneute Speicherung des Objektes hätte in diesem Fall keine weitere Auswirkung. Ist das Objekt passiviert, sind alle Objekte der transitiven Hülle passiviert.

Das Finden von Daten gestaltet sich mit Hilfe einer relationalen Datenbank sehr einfach. Die Klasse `Filter` wurde für die in einer relationalen Datenbank möglichen Operationen für die einzelnen Datentypen angepasst. Die Methode `find` erstellt eine Anfrage an die Datenbank, in der die Ausdrücke aller übergebenen Filter eingearbeitet werden. Die Filterung wird also von der Datenbank übernommen und muß nicht implementiert werden. Das Ergebnis solch einer Suche sind alle Objektidentitäten, deren Typ der übergebenen Klasse entspricht, und deren Zustand den übergebenen Filtern genügt. Die Menge der gefundenen Objektidentitäten kann nun vom Persistenzmanager aktiviert und als Resultat zurückgegeben werden.

4.3.3.3 Einfaches Anwendungsbeispiel für Persistenz

Um den Prozess der Persistenz zu demonstrieren, soll die Bindung der abstrakten Rolle `PersistentObjekt` an die Klasse `Person` gezeigt werden. Die Klasse Person kapselt die Daten Vorname und Name. Um diese Daten zu lesen und zu schreiben, existieren die zugehörigen get- und set-Methoden. Abbildung 4.15 zeigt eine Bindung der Rolle `PersistentObjekt` an `Person`. Die abstrakten Methoden der

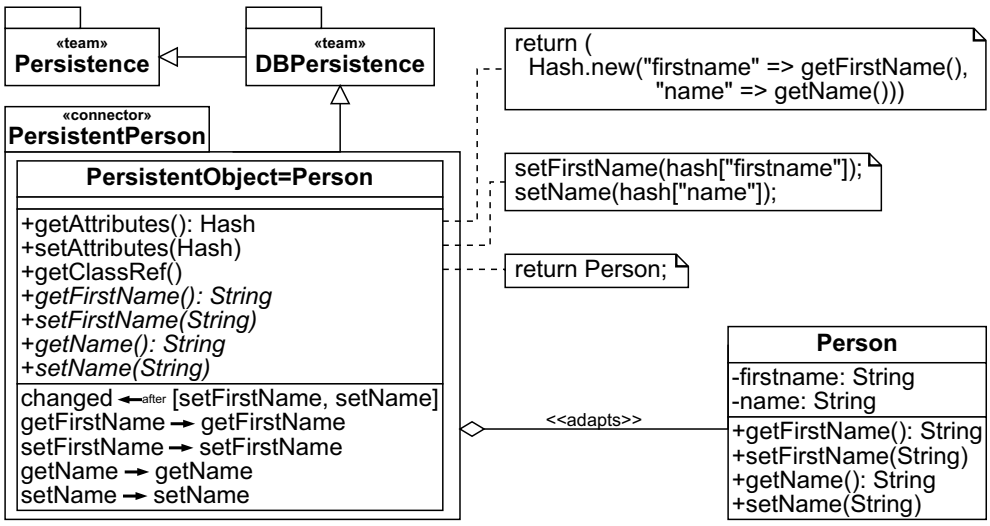

Abbildung 4.15: Beispiel eines persistenten Objektes

Rolle werden im Konnektor implementiert. Die Methode `getAttributes` erzeugt einen Hash, der gefüllt wird mit dem Zustand des Objektes. Der Zustand der Klasse Person besteht aus dem Vornamen und dem Namen der Person. Um diesen Zustand bei der Aktivierung wieder zu setzen, wird die Methode `setAttributes` implementiert. Sie bekommt als Parameter den gespeicherten Zustand und setzt die übergebenen Attribute. Die Methode `getClassRef` gibt eine Referenz auf die

Basisklasse `Person` zurück. Um den Zustand einer Person korrekt lesen und schreiben zu können, werden die jeweiligen get- und set-Methoden von Person genutzt. Sie müssen aus diesen Gründen abstrakt deklariert und an die Basis gebunden werden. Um den persistenten Zustand mit dem aktiven Zustand als verändert zu markieren, muß die Methode `setChanged` immer dann aufgerufen werden, wenn sich der Zustand des Objektes verändert. Eine Änderung des Zustandes wird in `Person` nur über die set-Methoden erreicht. Sie werden aus diesem Grund an die Methode `setChanged` Callin-gebunden. Eine Instanz vom Typ Person kann nun vom Persistenzmanager passiviert und auch wieder aktiviert werden.

Eine Bindung der Rolle `PersistenceManager` wurde in diesem Beispiel der Übersichtlichkeit halber ausgelassen. Erst das Binden der Methoden `activate` und `passivate` und der Aufruf der gebundenen Methode mit einer Instanz von Person führt zur Passivierung, bzw. Aktivierung dieses Objektes. Dies wird bei der Persistenz in PromOTe gezeigt.

Wie man in diesem Beispiel erkennen kann, scheinen die zu implementierenden Methoden der Rolle `PersistentObjekt` von sehr generischer Natur. Setzt man eine Konvention voraus, wie sie z.B. Java-Beans definiert (eine Klasse implementiert einen Standardkonstruktor und für jedes Attribut zugehörige get- und set-Methoden), würde die Angabe der zu speichernden Attribute völlig ausreichen. Der Name der zu speichernden Attribute lässt auf die entsprechenden Namen der get- und set-Methoden schließen. Verfügt die benutzte Programmiersprache dann auch noch über reflektive Mechanismen, kann selbst die Angabe der zu speichernden Attribute entfallen, da diese Liste durch Introspektion erstellt werden kann. Auch ein generativer Mechanismus ist sehr gut vorstellbar. Anhand der Attribute einer Klasse wird die zugehörige Rolle `PersistentObjekt` von einem Generator erzeugt. Diese Möglichkeit wurde für die Objekte in PromOTe genutzt.

4.3.3.4 Transaktionale Persistenz

Relationale Datenbanken stellen Mechanismen bereit, um Transaktionen zu unterstützen. Wie in Kapitel 4.3.1 erläutert wurde, können mit Hilfe von Transaktionen einzelne atomare Veränderungen zu einer „Gesamtänderung“ zusammengefasst werden. Die Änderung findet auf diese Weise entweder komplett oder gar nicht statt. Das Team `DBPersistence` fasst jede Passivierung eines Objektes als eine eigene Transaktion auf (so genannte implizite Transaktion, bzw. *'autocommit'*). Um die Zustandsänderungen verschiedener Objekte voneinander abhängig zu machen, werden explizite Transaktionen benötigt. Mit Hilfe solcher Transaktionen können die Abhängigkeiten der Entitäten einer Domäne ausgedrückt werden. Eine persistente Zustandsänderung findet nur dann statt, wenn alle an der Transaktion beteiligten Zustandsänderungen vollzogen sind. Das Verhalten der Datenbank kann auf

die Benutzung expliziter Transaktionen umgestellt werden. Eine so eingestellte Datenbank vollzieht die eingeleiteten Zustandsänderungen erst, wenn diese Veränderungen explizit bestätigt werden (`commit`). Es ist auch möglich, diese Änderungen zurückzunehmen (`rollback`), womit der letzte gültige Zustand wieder aktiv wird.

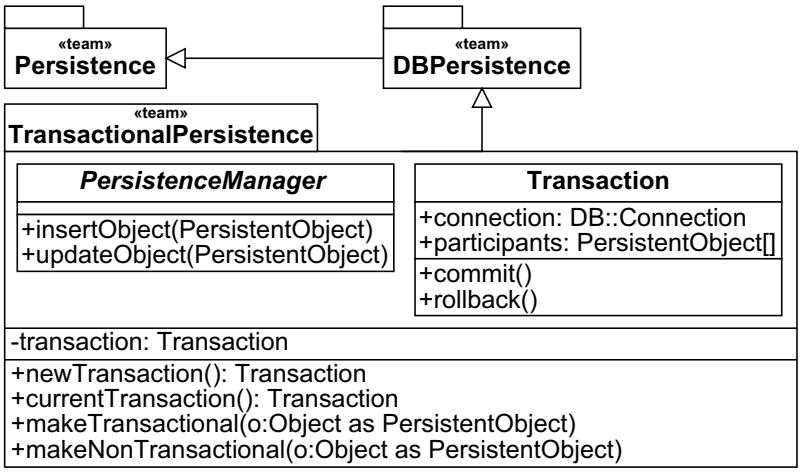

Abbildung 4.16: Auch transaktionale Persistenz ist möglich

Abbildung 4.16[5] zeigt das Team `TransactionalPersistence`, die das Team `DBPersistence` verfeinert. Eine neue Klasse `Transaction` wird hier eingeführt und die Klasse `PersistenceManager` verfeinert. Alle Teilnehmer einer Transaktion müssen vom Typ `PersistentObject` sein und werden in dem Attribut `participants` gehalten. Die Klasse `Transaction` kapselt außerdem eine Verbindung zu der Datenbank (`connection`). Alle Zustandsänderungen der involvierten Teilnehmer der Transaktion werden über diese Verbindung realisiert. Die Transaktion kann mit `commit` abgeschlossen werden. Alle Zustandsänderungen seit der Erzeugung der Transaktion bis zum `commit` werden wirksam. Soll eine Transaktion abgebrochen werden, wird dies mit der Methode `rollback` signalisiert. Die Methoden `commit` und `rollback` leiten die gewünschte Funktionalität an die Datenbankverbindung weiter. Die Datenbank kümmert sich um den gewünschten Zustandstransfer. Nachdem eine Transaktion abgeschlossen ist, kann das zugehörige Transaktionsobjekt nicht mehr benutzt werden. Für eine weitere Transaktion muß ein neues Transaktionsobjekt erzeugt werden.

Damit dieses Verhalten möglich wird, muss die Klasse `PersistenceManager` angepasst werden. Die Methoden, die eine Zustandsänderung von persistenten Objekten vollziehen, müssen in das transaktionale Vorgehen involviert werden. Das Vorgehen ist sehr einfach: die bei einer Passivierung ausführenden Methoden

[5] Hier wird von der Paketnotation mit Attributen und Methoden Gebrauch gemacht. Ein Team wird in UFA als Paket dargestellt, aber als Klasse implementiert. Die Attribute und Methoden des Teams werden in der Paketnotation wie in einer Klasse in einzelnen *'Compartments'* visualisiert.

`insertObject` und `updateObject` kontrollieren, ob das zu passivierende Objekt an einer Transaktion teilnimmt. Ist dies der Fall, wird der Zustandsübergang über die gekapselte Verbindung der Transaktion abgewickelt. Die Zustandsänderung wird somit nicht sofort wirksam, sondern erst nach erfolgreichem Abschluss der Transaktion. Ist das zu passivierende Objekt nicht Teil einer Transaktion, wird die Veränderung über eine „normale“ Datenbankverbindung abgewickelt, welche implizite Transaktionen unterstützt. Wird also keine explizite Transaktion angefordert und mit Teilnehmern gefüllt, ändert sich am Verhalten, wie in `DBPersistence` implementiert, nichts.

Transaktionen bilden Abhängigkeiten von Entitäten der Domäne ab. Sie müssen aus diesem Grund als Teil der Domäne angesehen werden und können nicht als Rollenverhalten modelliert werden. Die Klasse Transaktion ist aus diesem Grund eine normale Klasse und keine zu bindende Rolle des Teams. Um Zugriff auf das Verhalten einer Transaktion zu erhalten, sind die nötigen Methoden als Teammerkmale implementiert. Die Teammethode `newTransaction` erzeugt eine neue Transaktion und gibt diese zurück. Um die Abhängigkeiten einzelner Entitäten der Domäne auszudrücken, müssen sie als transaktional markiert werden. Dies geschieht mit der Methode `makeTransactional`. Als Parameter dieser Methode wird das abhängige Objekt übergeben, dessen instantiierende Klasse die Rolle `PersistentObjekt` gebunden haben muss. Dieses Objekt wird zu seiner Rolle geliftet (`as`-Operator) und in die Teilnehmerliste der laufenden Transaktion eingetragen. Um diese Abhängigkeit aufzuheben, muss ein Aufruf an die Methode `makeNonTransactional` erfolgen. Nachdem alle nötigen Zustandsübergänge vollzogen sind, kann die Transaktion abgeschlossen werden.

4.3.4 Persistenz in PromOTe

Die Funktionalität der Persistenz ist im Projektmanagementsystem PromOTe in einen transaktionalen Mechanismus auf Objektebene eingebettet. Bevor die Bindung der Domänenklassen gezeigt wird, soll kurz auf diesen transaktionalen Mechanismus eingegangen werden. Die Anforderungsbeschreibung der Firma ITSO fordert eine Kontrolle über jede Zustandsänderung einer jeden Entität der Domäne. Das bedeutet, wann immer ein Objekt verändert wird, soll vorher sicher gestellt werden, dass der resultierende Zustand korrekt ist. Da solch eine Kontrolle für atomare Änderungen schwer durchsetzbar ist und weil manche interne Zustandsänderungen die Validität eines Objektes zeitweise brechen dürfen, wurde ein transaktionaler Mechanismus auf Objektebene entworfen, der in Abbildung 4.17 zu sehen ist.

Die abstrakte Klasse `Commitable` ist die Basisklasse aller Domänenklassen in PromOTe. Ihre Funktionalität ist also dort überall verfügbar. Sie definiert die abstrakte

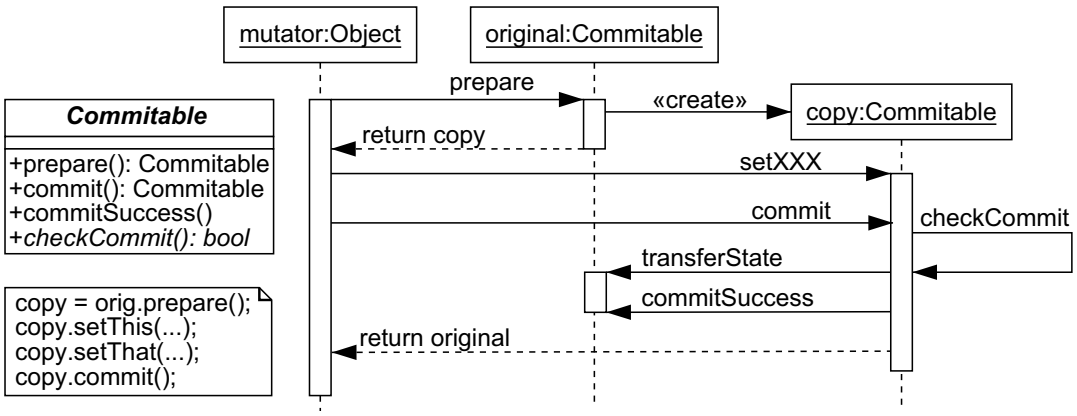

Abbildung 4.17: Zusammenfassen von Zustandsänderungen auf Objektebene

Methode `checkCommit`, die in den einzelnen Domänenklassen implementiert werden muss. Diese Methode prüft den internen Zustand der einzelnen Instanz auf Validität. Da nur die konkrete Domänenklasse das Wissen trägt, was ein valider Zustand im Sinne der Domänenklasse bedeutet, muss die Kontrollfunktionalität auch in der Klasse selbst definiert werden. Diese Prüfung ist erst möglich, wenn der Zustand schon gesetzt ist — ergibt die Prüfung, dass der Zustand nicht valide ist, kann nicht einfach zum ursprünglichen Zustand zurückgekehrt werden. Aus diesem Grund werden Zustandsänderungen immer auf einer Kopie des Originals ausgeführt.

Wie im Sequenzdiagramm in Abbildung 4.17 zu sehen, beginnt jede Zustandsänderung mit einem Aufruf der Methode `prepare`. Diese Methode erzeugt eine Kopie des Originalobjektes, indem zusätzlich ein Verweis auf das Originalobjekt gesetzt wird. Die Kopie kennt ihr Original. Die Instanz `mutator` spielt in diesem Diagramm keine Rolle, sie stellt nur eine zustandsverändernde Entität dar. Auf der Kopie können nun alle zustandsändernden Methoden aufgerufen werden (was im Diagramm nur beispielhaft durch `setXXX` angedeutet wurde). Alle gemachten Zustandsänderungen werden erst dann wirksam, wenn auf der Kopie `commit` aufgerufen wird. Die Methode `commit` ruft als erstes die Methode `checkCommit` auf. Es wird also auf der Kopie geprüft, ob der aktive Zustand gültig ist. Nur wenn dieser gültig ist, wird dieser Zustand in das Original übertragen. Nach einer erfolgreichen Zustandsänderung wird die Methode `commitSuccess` im Original aufgerufen und die originale Instanz zurückgegeben. Die Methode `commitSuccess` stellt also einen eindeutigen Trigger dar. Immer wenn diese Methode aufgerufen wird, hat sich der Zustand eines Objektes transaktional geändert. Genau diesen Mechanismus macht sich auch die Persistenzkomponente zu Nutze. Eine Entität wird immer dann passiviert, wenn sich der Zustand einer Instanz transaktional verändert hat.

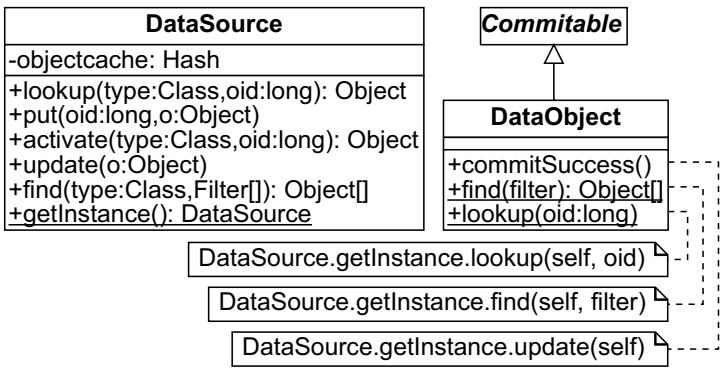

Abbildung 4.18: Ein Cache für Objekte

Als Datenquelle von PromOTe wurde ein Objektrepository implementiert. In einer verteilten Umgebung kann es mehrere gleichzeitig laufende Klienten geben, die auf der gleichen Menge von Objekten arbeiten. Das Repository soll die Konsistenz der Daten zwischen den einzelnen Klienten sicher stellen. Dies bedeutet, dass die Veränderung einzelner Objekte dem Repository bekannt gemacht werden und das diese Veränderung dann an alle benutzenden Klienten propagiert wird. Die Klasse `DataSource` stellt eine sehr einfache Implementierung solch einer Funktionalität zur Verfügung. Die Verteilung und Konsistenzprüfung wurde in der derzeitigen Implementierung nur vorgesehen, aber nicht implementiert. Die Klasse `DataSource` agiert in PromOTe z.Zt. also nur als einfacher Objektcache, der die einzelnen Entitäten der Domänenklassen aufnimmt. In diesem Objektcache können Objekte mit einem Schlüssel hinterlegt und über diesen wiedergefunden werden. Es ist außerdem möglich Objekte nach ihrem Inhalt zu suchen, wofür es die Methode `find` gibt. Die Klasse `DataSource` hat nur genau eine Instanz und implementiert das Singleton-Muster [Gamma u. a. 1996, Singleton]. Um die Zusammenarbeit der Domänenklassen mit dem Objektrepository zu erleichtern, wurde die Klasse `DataObjekt` definiert, die von `Commitable` erbt und die die Elternklasse aller Domänenklassen darstellt. Die Methode `commitSuccess` wird verfeinert, so dass bei einem Zustandsübergang die Methode `update` in `DataSource` aufgerufen wird. Diese Zustandsänderung kann nun vom Repository propagiert werden. Es werden weiterhin Methoden implementiert, die den Umgang mit den Klassen bequemer machen sollen. In `DataObject` ist unter anderem die statische Methode `find` definiert, die eine relativ natürliche Schreibweise im Quelltext erlaubt: `Person.find("name","=","Muster")` — finde alle Personen, die den Namen Muster tragen.

Die Methoden der Klasse `DataSource` ähneln doch sehr stark den Methoden des Persistenzmanagers. In der ungebundenen Form stellt diese Klasse einen transienten Objektcache dar. Er kann um die Funktionalität der Persistenz erweitert werden, indem die Rolle `PersistenceManager` an diese Klasse gebunden wird. Da diese

Rolle in seiner allgemeinen Form keine Aussage über Art der Abbildung und des Speichermediums macht, kann dieser Objektcache auf ganz verschiedene Weise persistent gemacht werden. Für PromOTe wurde die Persistenz in eine relationale Datenbank gewählt, das Team `DBPersistence`. Um die Domänenklassen in PromOTe persistent zu machen, ergibt sich somit eine Struktur, die man in Abbildung 4.19 sehen kann.

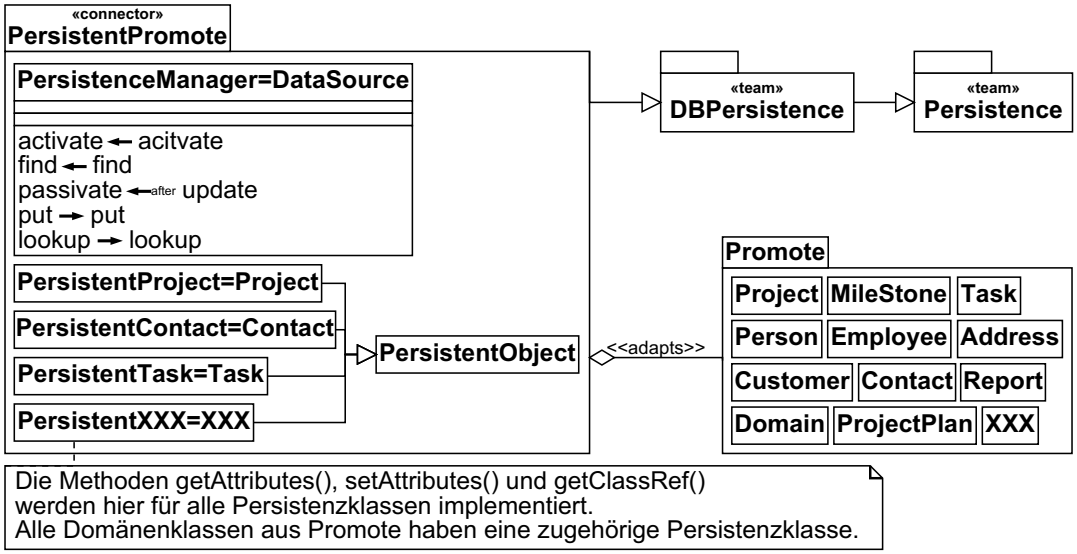

Abbildung 4.19: Persistente Domänenklassen in PromOTe

Die Rolle `PersistentObject` wird an alle Klassen der Domäne PromOTe gebunden. Da die abstrakten Methoden dieser Klasse für alle Domänenklassen separat implementiert werden müssen, reicht eine einfache Implementierung nicht aus. Aus diesem Grund werden im Konnektor neue Rollenklassen definiert, die von `PersistentObject` erben. Für jede Klasse der Domäne existiert eine spezielle Rolle für die Persistenz. In dieser speziellen Rolle werden die Methoden `getAttributes`, `setAttributes` und `getClassRef` implementiert, die für das Zusammenspiel mit dem Persistenzmanager nötig sind. Hier muß also für jede Klasse der Domäne der jeweilige Zustand ausgelesen und gesetzt werden können. Außerdem müssen alle zustandsändernden Methoden der Domänenklassen an die Methode `setChanged` der Rolle gebunden werden, so dass der persistente Zustand mit dem aktiven Zustand verglichen werden kann.

Die Funktionalität der Objektspeicherung ist in der Rolle `PersistenceManager` definiert. Diese Rolle wird an die Klasse `DataSource` gebunden. Die Rolle ersetzt die Methoden `activate` und `find`. Diese Funktionalität von `DataSource` wird also durch die gleichnamige Funktionalität der Rolle ersetzt, denn dort hat sie eine spezielle Bedeutung. Das Objektrepository soll bei einer Zustandsänderung

(`update`), diese Änderung an alle Klienten propagieren. Dieser Trigger wird ausgenutzt, um an dieser Stelle auch das jeweils veränderte Objekt zu speichern. Nach einem Zustandsübergang eines Objektes, wird dieses Objekt also passiviert. Damit der Cache mit dem Persistenzmanager zusammen arbeitet, werden die abstrakten Methoden `put` und `lookup` der Rolle gebunden. Der Persistenzmanager aktiviert auf diese Weise nur dann ein Objekt, wenn dieses im Cache noch nicht vorhanden ist, es also noch nicht aktiviert wurde. Ein aktiviertes Objekt wird im Cache unter seiner Objektidentität hinterlegt und kann auf diese Weise wiedergefunden werden.

Die im Konnektor `PersistentPromote` definierte Funktionalität existiert genau einmal im System. Jedes Objekt soll nur an genau einer Stelle passiviert, bzw. aktiviert werden. Der Konnektor wird aus diesem Grund statisch deklariert (siehe Kapitel 2.1.3.5). Die jeweilige Bindung existiert somit immer und nur genau einmal.

4.3.5 Java Data Objects

Um die Realisierung von Persistenz mit Hilfe von Object Teams mit aktuellen Techniken vergleichen zu können, soll hier eine Java-Technologie vorgestellt werden: *Java Data Objects*[Sun Microsystems 2002b; Branca und Oser 2002]. Aus der Fülle der Möglichkeiten soll gerade diese gewählt werden, da sie erstens einem sehr jungen Standard zu Grunde liegt (die Spezifikation in der Version 1.0 wurde am 25.03.2002 verabschiedet) und da sie zweitens eine im Vergleich zu anderen Persistenzmechanismen sehr interessante Idee verfolgt.

Java Data Objects, kurz JDO, definiert einen Satz von Schnittstellen, die einen Vertrag zur persistenten Speicherung von Objekten definieren. Wie bei den Spezifikationen von Sun üblich, benötigt man eine Komponente, die diese Schnittstellen implementiert. Im Rahmen dieses Vergleichs wurde die kostenfreie Implementierung *LiDO Community Edition* [Libelis 2002] genutzt. Die für den Programmierer wichtigsten Schnittstellen sind in Abbildung 4.20 ausschnittsweise[6] dargestellt.

Die Implementierung der Klasse `PersistenceManager` realisiert die eigentliche Funktionalität der Persistenz. Der Persistenzmanager kann für verschiedene Speichermedien und Abbildungen konfiguriert werden. Standardmäßig werden objektorientierte und relationale Datenbanken, sowie Persistenz in das Dateisystem unterstützt. Der Persistenzmanager unterstützt explizite Transaktionen und steht als Mediator zur Verfügung, um Anfragen an die jeweilige Datenquelle zu ermöglichen. Es wurde eine neue Anfragesprache *JDOQL* entwickelt, die sich an die Standardanfragesprache OQL (*'Object Query Language'*) anlehnt.

[6] Viele Methoden existieren jeweils für die einzelnen Java-Typen. Diese wurden hier verkürzt mit `XXX` dargestellt. Es sind also eigentlich immer 10 Methoden, wobei `XXX` durch die jeweiligen Java-Datentypen ersetzt werden muß. Alle Methoden aus `PersistentCapable` heißen eigentlich jdo<MethodenName>. Die Schnittstellen definieren noch eine ganze Reihe von Methoden, die der Übersichtlichkeit halber nicht dargestellt wurden. Dies sollte mit `...()` angezeigt werden

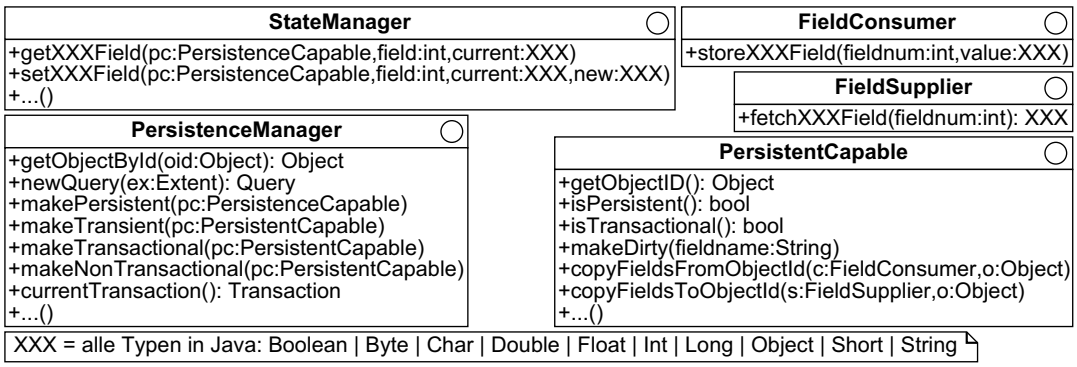

Abbildung 4.20: Die wichtigsten Schnittstellen in JDO

Jede Klasse die eine Objektspeicherung unterstützen soll, muß die Schnittstelle `PersistentCapable` implementieren. In dieser Schnittstelle ist die Interaktion mit dem Persistenzmanager definiert. Auch JDO stellt die Funktionalität der Persistenz als Systemfunktionalität dar, die in einer Domänenmodellierung keinen Platz hat. Ein Programmierer soll sich um Persistenz nicht kümmern. Die Deklaration und Implementierung der Schnittstelle `PersistentCapable` für die einzelnen Klassen übernimmt ein Werkzeug mit dem Namen *Enhancer*. Dieses Werkzeug nutzt die Technik der Bytecodetransformation. Eine übersetzte Klasse wird um die Schnittstelle `PersistentCapable` erweitert und die darin definierte Funktionalität für die spezielle Klasse implementiert. Der Programmierer muß lediglich einen Persistenzdeskriptor schreiben: eine XML-Datei, die die zu speichernde Struktur beschreibt.

JDO benötigt eine eindeutige Objektidentität für jedes persistente Objekt. Implementiert die jeweilige Klasse selbst solch ein Attribut, kann dieses als Objektidentität genutzt werden (so genannte *'application identity'*). Ist dies nicht der Fall, wird vom System eine Identität erzeugt (*'datatstore identity'*). Der Zugriff auf diese Identität wird durch die Methode `getObjectID` realisiert. Um einen allgemeinen Zugriff auf die einzelnen Attribute eines Objektes zu erhalten, ohne die jeweilige Struktur und die nötigen Zugriffsmethoden zu kennen, werden die Methoden `copyFieldsFromObjectId` und `copyFieldsToObjectId` implementiert. Die Attribute einer Klasse werden durchnummeriert, der Zugriff erfolgt über einen sequentiellen Index. Die Implementierung der Schnittstelle `FieldConsumer` kann eine Liste von getypten Attributen speichern. Das Auslesen einer Liste von getypten Attributen ist mit der Implementierung von `FieldSupplier` möglich. Typ meint in JDO alle in Java-existierenden Datentypen: `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, `Object` und `Short` und nicht der Typ einer Klasse, bzw. deren Vererbungshierarchie. Der Persistenzmanager kann mit Hilfe einer Implementierung von `StateManager` den Zustand eines Objektes lesen bzw. schreiben, ohne die implementierte Funktionalität der Klasse zu benutzen. Um eine Veränderung

am Zustand des Objektes zu registrieren, existiert die Methode `makeDirty`, die als Parameter den Namen des Attributes erhält. Damit diese Methode aufgerufen wird, werden alle Zugriffe auf Attribute in der Klasse durch einen Methodenaufruf ersetzt. In der aufgerufenen Methode kann nun zu `makeDirty` verzweigt werden. Da auf diesem Wege eine Veränderung jedes einzelnen Attributes registriert wird, müssen bei einer Passivierung wirklich nur die geänderten Attribute passiviert werden und nicht das gesamte Objekt samt transitiver Hülle. Dieses Vorgehen hat einen weiteren Vorteil, den man als transparente Aktivierung bezeichnet. Da der Zugriff auf ein Attribut einer Klasse auf eine Stelle reduziert wurde, muß dieses Attribut bei einer Aktivierung erst dann geladen werden, wenn darauf auch wirklich zugegriffen wird. Das bedeutet, das bei einer Aktivierung nicht notwendigerweise die gesamte transitive Hülle aktiviert wird, sondern nur ein Teil. Erst der Zugriff auf ein Attribut sorgt für dessen Aktivierung.

Für einen Test wurde die Klasse `Person` mit zugehöriger Adresse implementiert und ein einfacher Persistenzdeskriptor geschrieben, der die Klassen und deren Attribute auflistet. Der Enhancer erweiterte die Klassen um die notwendige Schnittstelle und deren Implementierung und richtete die Datenbank für die Persistenz ein. Die Klasse `Address` hatte vor der Erweiterung 3 Attribute und 8 Methoden, nach der Erweiterung 13 Attribute (8 statisch und 2 transient, so dass der Objektstrom bei einer Serialisierung nicht belastet wird) und 44 Methoden (9 statische). Sowohl Passivierung als auch Aktivierung verhielten sich wie erwartet.

4.3.6 Zusammenfassung

Persistenz als Systemfunktionalität zu begreifen birgt große Vorteile. In der UML-Modellierung reicht die Markierung eines Domänenobjektes mit einem einfachen Stereotyp aus und in der Implementierung soll diese Funktionalität nicht in der Klasse selbst implementiert werden. Es ergeben sich somit folgende Probleme:

- Der in einem Objekt gekapselte Zustand soll nur über Methoden der Klasse zugreifbar und veränderbar sein — ein Grundprinzip der Objektorientierung.
- Die Identität eines Objektes soll über Laufzeitgrenzen hinweg Gültigkeit besitzen.
- Man möchte den Prozess der Passivierung und Aktivierung kontrollieren und beeinflussen können.

Es wurden zwei Möglichkeiten solch einer Realisierung vorgestellt, beide mit ihren ganz eigenen Stärken und Schwächen.

JDO kommt mit einem sehr mächtigen Funktionsumfang: es existiert keine Voraussetzung an eine Klasse, die Persistenz unterstützen soll. Es existieren eine ganze Reihe von unterstützten Speichermedien und Abbildungen. Es werden explizite Transaktionen und transparente Aktivierungen unterstützt. All die Mächtigkeit von JDO wird aus einem neuen Werkzeug, dem Enhancer, gezogen, welches Funktionalität in die jeweilige Klasse einbringt. Der Umgang mit den persistenten Eigenschaften einer Klasse wird dagegen explizit vollzogen: In einem Persistenzdeskriptor müssen die zu speichernden Attribute definiert werden. Der Build-Prozess wird um einen weiteren Schritt angereichert. Die Benutzung des Persistenzframeworks in der Anwendung erfolgt explizit. Außerdem soll hier festgehalten werden, dass der Enhancer für genau die eine Funktionalität der Persistenz entwickelt wurde und keine allgemeinere Lösungsstrategie verfolgt. So sollte doch einigermaßen kritisch gefragt werden, ob es bald für jedes spezielle Problem einen speziellen Enhancer geben wird.

Die vorgestellte Realisierung von Persistenz für PromOTe ist im Gegensatz zu JDO sehr viel leichtgewichtiger, weniger Komplex und nutzt das allgemeine Paradigma Object Teams. Die Realisierung der Persistenz als Team ermöglicht eine schrittweise Verfeinerung und Anpassung der einzelnen Funktionalitäten. Dabei werden die persistenten Klassen auf der einen Seite und die Abbildung in ein spezielles Speichermedium auf der anderen Seite grundsätzlich unterschieden. Erweiterungen der einzelnen Funktionalitäten sind sehr einfach möglich, wie dies mit den expliziten Transaktionen in Kapitel 4.3.3.4 gezeigt werden sollte. Grundlage der Kopplung einer Persistenzrolle an eine Domänenklasse ist ein explizites Zusammenspiel. Nur wenn die Basisklasse einen Zugriff des internen Zustandes über Methoden erlaubt, ist eine Bindung überhaupt möglich. Die Kapselung von administrativen Daten und Funktionalitäten für die Persistenz ist in einer eigenen Rolle manifestiert, die in der Basisklasse fehl am Platze ist. So ist z.B. in PromOTe diese Rollenbindung nur in der Laufzeitumgebung des Objektrepository's aktiv. Die Klienten, die auf diesen Objekten arbeiten, sehen aber immer nur transiente Instanzen von Domänenklassen ohne persistente Rolle.

4.4 Zugriffskontrolle auf Basis von Privilegien

In fast jedem Informationssystem gibt es Normen, wer auf bestimmte Informationen zugreifen und wer diese verändern darf. Es müssen Regeln definiert werden, die vom System geprüft werden können. Unter Zugriffskontrolle versteht man ganz allgemein den Prozess der Bewilligung bzw. Verweigerung von Zugriffen auf Ressourcen, bzw. Daten in einem System. Werden beispielsweise sensitive Firmendaten verfügbar gemacht, sollte man den Zugriff auf diese Daten sehr genau einschränken können, so dass damit kein Missbrauch möglich ist. Die Funktionalität der Zugriffskontrolle kann als vermittelnde Schicht zwischen der Ressource und dem jeweiligen Nutzer begriffen werden. Es muss sichergestellt werden, dass diese Ressource immer nur über diese Vermittlerschicht angefordert werden kann, so dass auf diesem Wege eine Kontrolle möglich ist. Die Definition einer Vermittlerschicht legt nahe, dass diese Kontrollfunktionalität nicht Teil der angeforderten Daten bzw. Funktionalitäten einer speziellen Domäne ist.

Nach einer Einführung über allgemeine Modelle und Mechanismen der Zugriffskontrolle, soll die rollenbasierte Zugriffskontrolle näher erläutert werden. Die dort dargestellten Mechanismen sollen dann in einem Team modelliert und besprochen werden. Nach einem einfachen Anwendungsbeispiel, soll diese Lösung für die Anwendung in PromOTe gezeigt werden. Da gerade die Frage der Zugriffskontrolle sehr spezifische Möglichkeiten erfordert, sollen weitere mögliche Adaptionen aufgezeigt werden.

4.4.1 Zugriffskontrolle: Modelle und Mechanismen

Samarati und Vimercati [2000] definieren drei Abstraktionsebenen, die bei einer Zugriffskontrolle unterschieden werden müssen:

'*Security Policy*': definiert die generellen Prinzipien, die bei einer Zugriffskontrolle beachtet werden sollen. Hier werden Regelungen auf ganz abstraktem Niveau aufgestellt, was und wie reguliert werden soll.

'*Security Model*': definiert die formale Repräsentation und Arbeitsweise der in der ersten Ebene definierten Prinzipien. Eine Überprüfung auf Vollständigkeit und Sicherheit der erstellten Mechanismen ist auf dieser Ebene möglich.

'*Security Mechanism*': Die Realisierung eines bestimmten Modells in funktionale Kontrollmechanismen bestimmt diese Ebene. Sie entspricht der Implementierung eines definierten Modells.

Diese Herangehensweise unterscheidet sich zu der bekannten Modellierung und Implementierung in der formalen Überprüfung des Modells auf Sicherheit. Auf der Ebene des Modells müssen folgende Punkte sichergestellt werden: (1) das Modell

muß den Zugriff auf die zu schützenden Ressourcen sicher stellen (*'tamper proof'*). Es muß (2) geprüft werden, ob alle möglichen Zugriffsvarianten abgedeckt wurden (*'non bypassable'*). Es darf also keine möglichen Umwege geben, sich einen Zugriff zu verschaffen. Und es muß (3) sicher gestellt werden, dass das Sicherheitssystem selbst geschützt ist, so dass hier keine Veränderung möglich ist (*'security kernel'*). Sind diese drei Punkte auf der Ebene des Modells formal sichergestellt, können die erstellten Regeln implementiert werden.

Samarati und Vimercati [2000] teilen die bestehenden Zugriffskontrollprinzipien in 3 Kategorien ein, die im folgenden kurz dargestellt werden sollen.

Discretionay Access Control

Dieses Modell der Zugriffskontrolle beruht auf der Basis einer Identität. Jeder Benutzer eines Systems ist mit einer Identität gekennzeichnet. Für jede vorkommende Identität werden explizite Zugriffsrechte auf bestimmte Ressourcen vergeben. Diese Zugriffsrechte werden auch *Privilegien* genannt. Es muss eine administrative Instanz geben, welche die Vergabe von Privilegien regelt. Jede Identität des Systems kann die ihm zur Verfügung stehenden Privilegien an eine andere Identität übertragen — daher auch der Name (*'discretionary'*=Handlungsfreiheit). Prinzipiell gibt es zwei Möglichkeiten, den Zugriff zu beschränken:

Closed Policy: Ist wohl die bekannteste Form: Die Regelung der Zugriffe erfolgt hinsichtlich einer Freigabe bestimmter Ressourcen. Bei einem Zugriff auf die Ressource wird kontrolliert, ob die Identität das erforderliche Privileg für diese Handlung besitzt. Ist dieses nicht vorhanden, wird der Zugriff verweigert.

Open Policy: Dieses Modell verhält sich genau anders herum. Einer Identität können negative Privilegien erteilt werden. Das bedeutet, dass bei einem Zugriff geprüft wird, ob solch ein negatives Privileg vorliegt. Ist dies nicht der Fall, wird der Zugriff gewährt.

Die beiden Modelle unterscheiden sich also in der Art der Freigabe auf Ressourcen, wenn keine Zugriffsbeschränkungen definiert sind. Aus diesem Grund findet man die zweite Variante nur in solchen Szenarien, wo Sicherheit keine wichtige Rolle spielt.

Mandatory Access Control

Diese Art der Zugriffskontrolle basiert auf Restriktionen, die im Auftrag einer zentralen Autorität festgelegt werden. Dieses Modell klassifiziert alle Entitäten nach Subjekt und Objekt. Objekte sind passive Entitäten, die Daten kapseln. Subjekte sind aktive Entitäten, die auf Objekte zugreifen. Unter Subjekt wird nicht unbedingt ein Benutzer, also ein Mensch, verstanden, sondern vielmehr ein Programm oder ein

Prozess. Dieses Modell definiert Sicherheitsebenen plus zugehörigen *Dominanzrelationen*. Solch Sicherheitsebenen könnten beispielsweise: Streng Geheim > Geheim > Vertraulich > Unklassifiziert sein. Alle Objekte werden einer Sicherheitsebene zugeordnet. Hat ein Subjekt Zugriff auf Objekte die als Geheim eingestuft sind, darf er auch auf alle vertraulichen und unklassifizierten Objekte zugreifen, nicht aber auf Objekte der Kategorie Streng Geheim.

Role Based Access Control

Der Zugriff auf Ressourcen in einem System wird hier durch die jeweiligen Zuständigkeiten einer Person in diesem System geregelt. Diese Möglichkeit ist vor allem für Geschäftslösungen interessant, da hier die Sicherheitsprinzipien mit der Organisationsstruktur der Firma korrelieren. Es können also Zugriffsbeschränkungen auf Basis von Rollen einer Organisationsstruktur (z.B. Geschäftsführer, Mitarbeiter, Sekretär) definiert werden. Die zentrale Idee hinter einer rollenbasierten Zugriffskontrolle, ist die Möglichkeit Privilegien für eine Gruppe von Identitäten zu definieren. Man hat weiterhin die Flexibilität von expliziten Restriktionen, die aber automatisch für eine Gruppe von Identitäten wirksam sind. Mit der Definition von Rollen können verschiedene Ziele verfolgt werden, beispielsweise:

- Aktivitätsbereiche: Werden Privilegien für einzelne Aktivitäten definiert, so können Rollen dafür genutzt werden, bestimmte Arbeitsbereiche voneinander zu trennen. Die einzelnen Mitarbeiter können nur organisatorisch begrenzte Aktivitäten ausführen.

- hierarchische Rollen: Rollen können in einer hierarchischen Ordnung definiert werden, die sich in einer `ist ein` Beziehung befinden. So könnte die Rolle Mitarbeiter und die Rolle Sekretär definiert werden, wobei der Sekretär ein Mitarbeiter ist. Er hat somit alle Rechte, die auch ein Mitarbeiter hat, plus den spezifischen Rechten eines Sekretärs. Eine Firmenhierarchie könnte so abgebildet werden, wobei die Anzahl der Privilegien nur steigen kann, je höher die Rolle in der Hierarchie definiert ist.

- Trennung von Pflichten: Rollen können so definiert werden, dass kein Benutzer ausreichend Rechte hat, ein System zu missbrauchen. So könnte es beispielsweise in einem Kassensystem die Rolle Kassierer und Storno geben. Ein Kassierer darf kein Storno ausführen, derjenige der storniert darf nicht kassieren.

Eine Rolle steht stellvertretend für eine Menge von Privilegien. Ressourcen müssen also weiterhin gegen ein bestimmtes Privileg geschützt werden und nicht gegen verschiedene Rollen. Es ist eine Administration notwendig, die (1) Rollen als Liste von Privilegien definiert, (2) den Benutzern Rollen zuweist und (3) den Objekten Restriktionen auferlegt.

4.4.1.1 Anwendung im Umfeld einer Projektmanagementsoftware

Eine Software zur organisatorischen Unterstützung von Projektabläufen findet ihre Anwendung typischerweise im Umfeld eines Unternehmens. Die von der Firma ITSO definierten Anforderungen [Glöckner und Storl 2001] zielen auf genau diese Zielgruppe ab. Die in einem Unternehmen bestehende Organisationsstruktur spiegelt sich in den einzelnen spezifizierten Rollen wieder. Es wurden folgende Rollen definiert: Geschäftsführer, Projektleiter und Mitarbeiter. In einer Erweiterung wurde eine Abrechnungskomponente vorgesehen, die sich mit einer Rolle Sekretär verbindet. Die Anforderungen von ITSO an eine Projektmanagementsoftware wurden als einzelne Aktivitäten spezifiziert, zu denen jeweils Akteure definiert wurden, wie in Abbildung 4.1 zu sehen ist. Diese Aktivitäten lassen sich auf keine hierarchische Struktur zurückführen. Das bedeutet, die Organisationsstruktur korreliert nicht mit den möglichen Aufgaben im Unternehmen, sondern bilden Aktivitätsbereiche.

Die Anforderungen und Definitionen von ITSO für eine Projektmanagementsoftware sind in Beziehung auf einen Sicherheitsmechanismus ein Beispiel par excellence für die rollenbasierte Zugriffskontrolle. Jeder definierten Aktivität kann ein Privileg zugeordnet werden. Die einzelnen Privilegien können aus dem Anwendungsfalldiagramm 4.1 abgelesen werden und den einzelnen Rollen als Privileg zugeordnet werden. Die einzelnen Rollen ergeben sich somit als eine Sammlung der einzelnen Privilegien (siehe Abbildung 4.4.1.1). Nachdem gezeigt wurde, welche Art der Zu-

Privileg	Geschäftsführung	Projektleiter	Mitarbeiter
Mitarbeiterverwaltung	⊙		
Kundenverwaltung	⊙		
Bereichsverwaltung	⊙		
Kontaktverwaltung	⊙	⊙	
Zeitaufschreibung		⊙	⊙
Reportverwaltung		⊙	⊙
Meilensteinverwaltung		⊙	⊙
Aufgabenverwaltung		⊙	⊙
Projektplanung		⊙	⊙

Abbildung 4.21: Verteilung der Privilegien auf Rollen

griffskontrolle für eine Projektmanagementsoftware sinnvoll ist, nämlich die rollenbasierte Zugriffskontrolle (Ebene 1: *'Security Policy'*), welche Rollen in dem von ITSO spezifizierten System vorkommen und welche Privilegien existieren, soll nun gezeigt werden, wie die Privilegien der Rollen überhaupt ausgewertet werden, wie also das Sicherheitsmodell zu den allgemeinen Prinzipien aussieht (Ebene 2: *'Security Model'*).

4.4.2 Sicherheit mit vereinten Kräften

Im Kontext einer objektorientierten Umgebung, wo Daten nur in Form von Objekten existieren, bedeutet Zugriffskontrolle den Schutz von Objekten. Der Zugriff auf Methoden von Klassen muß hier kontrolliert werden. Damit solch eine Kontrolle möglich ist, muß die Identität des zugreifenden Subjektes bekannt sein. Auch hier ist unter Subjekt nicht notwendigerweise ein Benutzer zu verstehen, sondern ein Programm, Prozess etc. Einem Subjekt sind bestimmte Privilegien zugeordnet, die vom System erfragt werden können. Anhand dieser Privilegien kann entschieden werden, ob ein Zugriff erlaubt ist, oder nicht.

Die zu erbringende Funktionalität der Zugriffskontrolle soll unabhängig von den zu schützenden Klassen entwickelt werden können. Andernfalls müsste in allen sensitiven Methoden einer Domänenklasse solch eine Kontrolle erfolgen. Dieses Vorgehen wäre nicht nur schlechte Modellierung, sie erfordert zusätzlich auch, dass jeder Aufrufer der Methode die nötigen Informationen zur Zugriffskontrolle kennt. So werden unter Umständen Variablen durch eine ganze Ablaufstruktur geschliffen, nur weil an einer Stelle im Quellcode eine Zugriffskontrolle erfolgt (es besteht natürlich auch die unschöne Variante von globalen Variablen).

Um diese Probleme zu umgehen, könnte man sich ein Vermittlerobjekt [Gamma u. a. 1996, Proxy] vorstellen, welche jeden Methodenaufruf an das ummantelte Objekt weiterleitet, vorher aber prüft, ob der Zugriff erlaubt ist oder nicht. Diese Möglichkeit wird in der Regel von verteilten Systemen genutzt. Hier existiert sowieso schon eine Vermittlerschicht für die Kommunikation mit zugehörigem Vermittlerobjekt, welches auch für die Zugriffskontrolle genutzt werden kann. Die Identität wird von der jeweiligen benutzten Middleware bereitgestellt, so dass dieser Weg sehr praktikabel und in der Realität anzutreffen ist (z.B. CORBA).

Existiert in der Zielanwendung keine Middleware mit Zugriffskontrolle, oder fehlt die Muße für jede existierende Domänenklasse eine Vermittlerklasse mit Kontrollfunktionalität zu programmieren, bietet die aspektorientierte Programmierung eine interessante Herangehensweise. Der Aspekt der Zugriffskontrolle kann separat definiert und an alle Stellen der Domäne gebunden werden, wo eine Zugriffskontrolle sichergestellt werden soll. Voraussetzung ist die Existenz einer Identität, welche über Privilegien verfügt, die kontrolliert werden können. In Abbildung 4.22 sieht man solch eine Modellierung als Rollen eines Teams.

Das Team `AccessControl` definiert drei Klassen `Subject`, `Permission` und `GuardedObject`. Die abstrakte Klasse `Permission` stellt ein Privileg dar. In einer Teamverfeinerung können eine Reihe von Privilegien definiert werden, welche in einer speziellen Domäne vorkommen. Jedes Privileg muß die Methode `implies` implementieren, die eine *Dominanzrelation* verwirklicht. Das bedeutet, dass ein Privileg gegen ein anderes Privileg geprüft werden kann. Das Ergebnis solch einer Dominanzrelation ist immer ein boolscher Wert. Ein Privileg schließt ein gegebenes

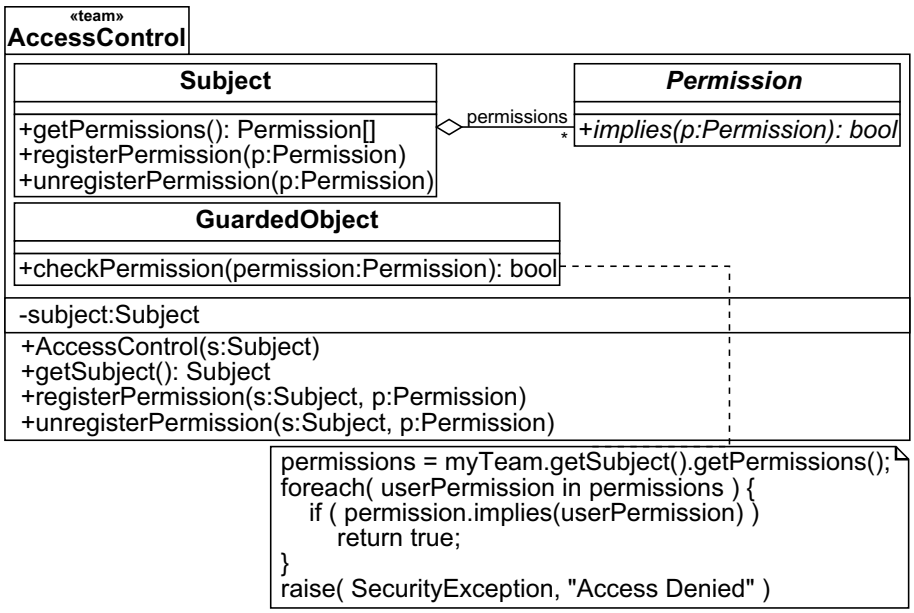

Abbildung 4.22: Zugriffskontrolle auf Grundlage von Privilegien

Privileg entweder ein, in diesem Fall genügt das gegebene Privileg dem vergleichenden, oder nicht. Jede zu schützende Methode muß mit solch einem Privileg versehen werden, gegen das geprüft werden soll. Bei einem Zugriff werden alle Privilegien eines Subjektes gegen dieses Privileg geprüft. Gibt es mindestens ein Privileg, welches das geforderte Privileg einschließt, ist der Zugriff gewährt - andererseits nicht. Durch die Einführung der Klasse Privileg mit der abstrakten Methode `implies` ist ein Standardmechanismus gegeben, wie man Restriktionen und Rechte in einem System definiert. Objekte können durch Instanzen von `Permission` geschützt werden. Sie müssen in diesem Sinne als Restriktion aufgefasst werden, da die dort definierte Dominanzrelation durch Privilegien des Benutzers erfüllt sein muss. In der Implementierung von `implies` kann ganz genau definiert werden, was für ein Privileg von dem Benutzer erwartet wird, damit eine Zugriffsberechtigung erteilt werden kann. Die Instanzen von `Permission` die einem Benutzer zugeordnet sind, können als Privilegien verstanden werden, da sie einen Zugriff unter Umständen ermöglichen.

Die Rolle `Subject` modelliert den jeweiligen Benutzer, bei dem ein Zugriff geprüft werden soll. Jedes Subjekt hat hier eine Menge von Privilegien, auf die mit `getPermissions` zugegriffen werden kann. Existiert in der jeweiligen Domäne die Abstraktion eines Subjektes, so kann die jeweilige Klasse an die Rolle `Subject` gebunden werden. In der Domäne muss also die dort definierte Identität eines Subjektes nicht um die Verwaltung von Privilegien erweitert werden. Bei einer Prüfung wird diese Identität zu dem assoziierten Subjekt geliftet, an dem die Privilegien er-

fragt werden können. Existiert eine solche Identität in der Domäne nicht, muß die Rolle `Subject` nicht gebunden werden, sondern kann als normale Klasse benutzt werden. Hiermit erreicht man einen hohen Grad an Flexibilität, um verschiedene Anwendungen zu unterstützen.

Die effektive Rolle `GuardedObject` kann an alle Klassen der Domäne gebunden werden, die eine Zugriffskontrolle sicher stellen sollen. Diese Rolle implementiert genau eine Methode `checkPermission`, die als Parameter das Privileg übergeben bekommt, gegen das geprüft werden soll. Dieses Privileg wird gegen alle Privilegien des zugreifenden Subjektes geprüft. Existiert mindestens ein Privileg, das das geforderte Privileg einschließt, kehrt die Methode zurück. Existiert solch ein Privileg nicht, muss der Zugriff verweigert werden. Da diese Möglichkeit in der Domänenklasse nicht vorgesehen werden konnte, wird hier eine Ausnahme ausgelöst, die nicht deklariert werden muss (*'unchecked exception'*). Dies ist ein üblicher Weg, um sich vor Missbrauch zu schützen (z.B. SecurityManager in Java).

Die Methode `checkPermission` wird an alle Methoden der Domänenklasse gebunden, wo eine Zugriffskontrolle erfolgen soll. Da die Kontrolle immer vor dem Aufruf der Methode geschehen soll, wird der `before` Qualifizierer genutzt. Außerdem muß die Signatur angepasst werden. Die Methode `checkPermission` erwartet ein Objekt vom Typ `Permission`, die in der Methode der Domänenklasse eigentlich nicht vorkommen sollte. In der Signaturanpassung muss ein für diese Methode spezifisches Privileg definiert werden, welches bei Aufruf der Methode erzeugt und an die Methode `checkPermission` übergeben wird.

Das Team `AccessControl` kapselt die Identität, unter der die Anwendung ausgeführt wird. Eine Instanz dieses Teams wird unter Angabe des Subjektes erzeugt. Dies könnte beispielsweise passieren, nachdem sich ein Benutzer in dem System identifiziert hat. In einer Teamverfeinerung müssen alle Domänenklassen spezifiziert werden, die zugriffsbeschränkt werden sollen. Dies geschieht durch eine spielt Beziehung zur Rolle `GuardedObject`. Durch Callin-Bindung der Methode `checkPermission` wird der Zugriffskontrollmechanismus in die einzelnen Methoden gewoben. Um solch eine zugriffsbeschränkte Methode ausführen zu können, muss das Subjekt über ein spezielles Privileg verfügen, welches die definierte Restriktion erfüllt. Das Team `AccessControl` implementiert auch administrative Methoden, Privilegien bei einem Subjekt zu registrieren, bzw. zu entfernen. Diese Methoden könnten selbst durch ein Administrationsprivileg geschützt werden, so dass hier kein Missbrauch betrieben werden kann.

Um eine Zugriffskontrolle auf Basis von Rollen zu unterstützen, kann das Team `AccessControl` um diesen Mechanismus erweitert werden (Hinweis: es sind die Rollen im Sinne von rollenbasierter Zugriffskontrolle gemeint, nicht die Rollen im Sinne von Object Teams), wie dies in Abbildung 4.23 zu sehen ist. Eine Rolle kann als eine Aggregation von Privilegien verstanden werden. Eine Rolle wird aus diesem

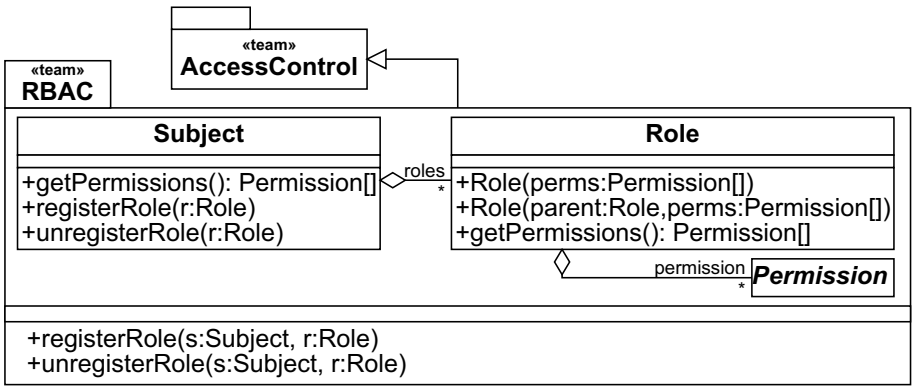

Abbildung 4.23: Rollenbasierte Zugriffskontrolle

Grund mit einer Menge von Privilegien erzeugt. Da Rollen in einer Anwendung eine relativ statische Natur haben, könnten sie beispielsweise als statische Variablen in einer Teamverfeinerung definiert werden. Die Rollen sind auf diese Weise unabhängig von jeder Domäne — Art und zugehörige Privilegien lassen sich jeweils neu definieren. Um den Mechanismus von hierarchischen Rollen zu unterstützen, kann eine Rolle eine Elternrolle haben, von der alle Privilegien „geerbt“ werden. Die Methode `getPermissions` liefert alle Privilegien der Rolle und aller Elternrollen. Damit die Privilegien der Rollen eines Subjektes geprüft werden können, muss die Methode `getPermissions` in der Klasse `Subject` angepasst werden. Um Rollen eines Subjektes zu verwalten, werden die Methoden `registerRole` und `unregisterRole` in der Klasse `Subject` und im Team für ein gegebenes Subjekt implementiert.

4.4.2.1 Ein gesicherter Banktresor

Ein einfaches Beispiel für eine Zugriffskontrolle sieht man in Abbildung 4.24. Um dies Beispiel möglichst einfach zu halten, wurde auf die Einführung von Rollen verzichtet. Aus diesem Grund wird das Team `AccessControl` direkt verfeinert und nicht das Team `RBAC`.

In einer Bank steht ein Tresor (`Safe`), der von Mitarbeitern (`Clerk`) der Bank geöffnet werden darf, um Geld zu entnehmen (`get`) oder Geld aufzubewahren (`put`). Damit nicht jeder Mitarbeiter der Bank an den Tresor kommt, wird der Tresor zugriffsbeschränkt. Da sich die Zugriffsbeschränkung auf Mitarbeiter bezieht, wird die Rolle `Subject` an die Klasse `Clerk` gebunden. Das Team `SafeBank` führt zwei neue Privilegien ein. Das Privileg `Deposit` ermächtigt einen Mitarbeiter Geld in den Tresor zu legen. Die Methode `implies` prüft nur, ob ein Privileg vom Typ `Deposit` vorhanden ist, wobei die einzuzahlende Summe keine Rolle spielt. Das

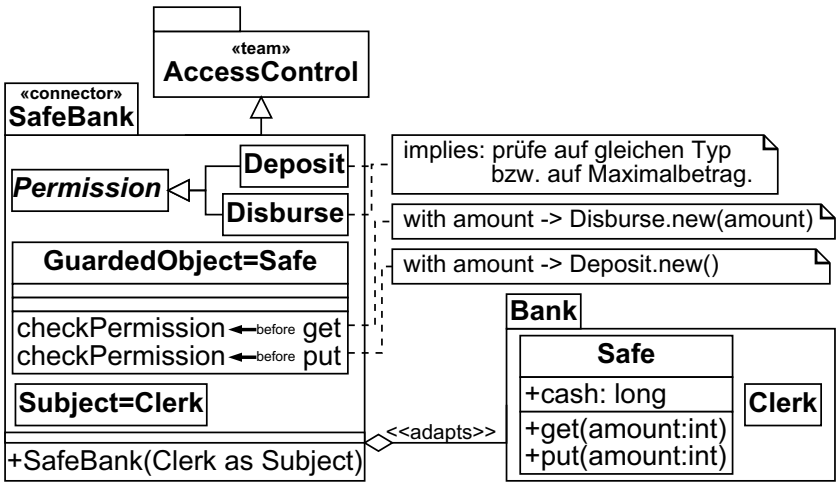

Abbildung 4.24: Ein sicherer Banktresor

Privileg `Disburse` ermächtigt einen Mitarbeiter der Bank, Geld aus dem Tresor zu entnehmen. Hier wird nicht nur das Vorhandensein des Privilegs selbst geprüft, sondern auch der angeforderte Betrag. So ist es möglich diesen Betrag im Privileg der einzelnen Mitarbeiter verschieden einzustellen. So hätte der Filialleiter z.B. unbegrenzten Zugang, ein einfacher Mitarbeiter aber nur auf einen begrenzten Betrag.

Damit diese Zugriffskontrolle nutzbar ist, muß von administrativer Seite aus jedem Mitarbeiter der Bank, der Zugriff auf den Tresor haben soll, das entsprechende Privileg zugeordnet werden. Es muss außerdem sicher gestellt werden, dass dieses Team instantiiert wird, nachdem sich der Benutzer im System identifiziert hat. Damit dies möglich ist, existiert im Team `SafeBank` ein Konstruktor, der eine Instanz der Klasse `Clerk` zu einer Instanz der Klasse `Subject` liftet, die im Team gespeichert wird. Immer wenn die Methode `get` oder `put` an der Klasse `Safe` aufgerufen wird, kann anhand der Privilegien dieses Subjektes geprüft werden, ob die Aktion erlaubt ist.

4.4.2.2 Veränderte Funktionalität in Folge von Sicherheitsmechanismen

Das Team `AccessControl` bietet die Möglichkeit, den Zugriff auf einzelne Methoden von Domänenklassen durch Privilegien zu kontrollieren. Indem die Methode `checkPermission` vor jeden Aufruf der zu schützenden Methode gebunden wird, kann diese Kontrolle sicher gestellt werden. Diese Methode realisiert dabei aber nur eine ausschließliche Kontrolle. Hat der Benutzer das nötige Privileg, darf auf die Methode zugegriffen werden; hat er es nicht, wird der Zugriff verweigert. In der Domäne können aber auch Methoden existieren, die sich durch eine Zugriffskontrolle verändern! Verändern heißt hier im allgemeinen Fall einschränken. So möchte man u.U. einen Zugriff prinzipiell erlauben, die resultierende Information aber

auf Basis von Privilegien einschränken, so dass nur Teile einer Gesamtinformation zugreifbar sind. Denkt man beispielsweise an eine Mitarbeiterliste in einer Projektmanagementsoftware, so könnte einem Mitarbeiter Zugriff auf alle Mitarbeiter der Projekte gewährt werden, in denen er mitarbeitet. Ein Sekretär dagegen sollte Zugriff auf jeden Mitarbeiter haben. Dies gilt natürlich nur für Funktionalitäten, die sich aus einer Zugriffsbeschränkung ergeben und nicht Teil der Domänenfunktionalität sind.

Um solch eine Funktionalität bereitzustellen, bietet Object Teams die Möglichkeit, Methoden von Domänenklassen in einer Rolle zu verfeinern. Dies wird durch *'Replacements'* realisiert, welche die überschriebene Basismethode weiterverwenden. Die Klasse `GuardedObject` kann im Konnektor um die entsprechenden Methoden erweitert werden, die diese Funktionalität bereitstellen. Bei einer Aktivierung werden auf diese Weise die speziellen Methoden der Rolle genutzt, die für den Kontext der Zugriffskontrolle adaptiert wurden. Solche Methoden sind spezifisch für die jeweilige Basisklasse, so dass diese Funktionalität nicht verallgemeinert werden kann.

4.4.2.3 Anwendung in PromOTe

Die Anforderungen an eine Zugriffskontrolle in PromOTe wurden von ITSO eindeutig definiert. Es soll drei Rollen geben, Geschäftsführer, Projektleiter und Projektmitarbeiter, die jeweils eine abgegrenzte Funktionalität nutzen können. Die einzelnen Funktionalitäten wurden als Aktionen spezifiziert. Für diese Aktionen wurden Privilegien definiert, die in den einzelnen Rollen zusammengefasst werden.

Die an den Aktionen beteiligten Domänenklassen müssen vorbereitet werden, damit die erstellten Privilegien geprüft werden können. Alle zu prüfenden Methoden dieser Klassen werden an die Methode `checkPermission` gebunden.

Die Nutzer von PromOTe werden als Instanzen der Klasse `Person`, bzw. ihren spezialisierenden Klassen modelliert. Jede einzelne Person muss nun um den Mechanismus erweitert werden, Privilegien in Form von Rollen verwalten zu können. Auch dies ist nur eine Anforderung der Zugriffskontrolle und wird deshalb separat im Team spezifiziert.

Eine Realisierung der Anforderungen kann man in der Modellierung in Abbildung 4.25 sehen. Die Rolle `Subject` wurde an die Klasse `Person` gebunden. Da jeder Mitarbeiter und jeder Kunde eine Spezialisierung dieser Klasse ist, reicht diese Definition zur Modellierung aus. Hat sich die Person am System angemeldet, wird eine neue Instanz von `PromteControl` erzeugt. Die Teammerkmale führen ein Lifting der übergebenen Person zu der Rolle `Subject` durch. Ist die Teaminstanz aktiviert, werden alle Callin-Bindungen in die Domänenklassen gewoben, so dass eine Zugriffskontrolle möglich wird. Die Privilegien einer Person werden

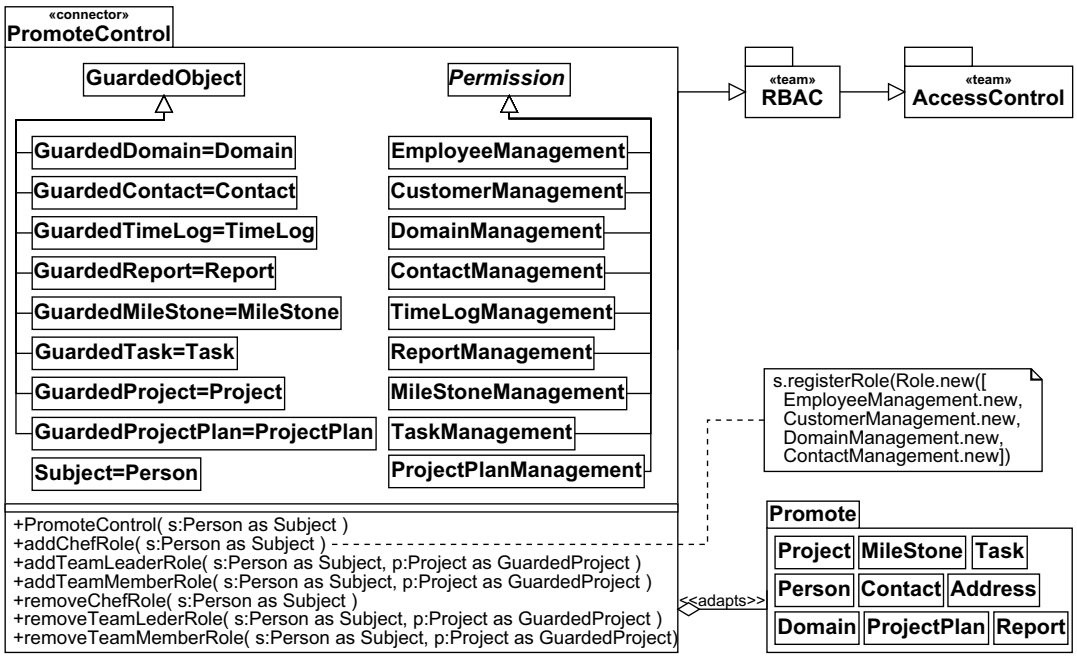

Abbildung 4.25: Zugriffskontrolle in Promote

in Rollen gekapselt. Um einer Person Privilegien zuteil werden zu lassen, können die Teammerkmale benutzt werden (`addChefRole`, `addTeamLeaderRole` und `addTeamMemberRole`). In PromOTe basieren die Rollen Projektleiter und Projektmitglied auf einem ganz speziellen Projekt. Diese Rollen können also auf Basis von Projekten mehrfach vergeben werden. Um solch spezielle Rollen zu erzeugen, muss das Projekt übergeben werden. Da die Klasse `Project` selbst einer Zugriffskontrolle unterliegt, wird hier zu der Rolle geliftet. Benötigt ein Privileg eine besondere Information aus einem Projekt, so müsste sie in diesem Beispiel in der Rolle deklariert und gebunden werden.

Art und Umfang der Zugriffskontrolle ist in den Spezialisierungen der Klasse `GuardedObject` manifestiert. Hier werden die einzelnen zu schützenden Methoden an die Methode `checkPermission` gebunden. Dies ist am Beispiel von `GuardedProject` in Abbildung 4.26 zu sehen: Die einzelnen Methoden werden mit Privilegien deklariert, die von dem jeweiligen Benutzer erfüllt werden müssen. Hier wird die Möglichkeit der Signaturanpassung ausgenutzt, um aus den Aufrufparametern ein Privileg zu generieren. In dem dargestellten Beispiel sind die Parameter oft nicht von Wichtigkeit. Dies liegt aber an der Definition der Privilegien. Hat man das Privileg `EmployeeManagement`, können unabhängig vom Projekt Mitarbeiter einem Team zugeordnet oder umgeordnet werden. Für die Privilegien `TaskManagement` und `MileStoneManagement` ist es aber sinnvoll auch das Projekt zu überprüfen, indem die Aufgabe oder der Meilenstein definiert wurden. Nur

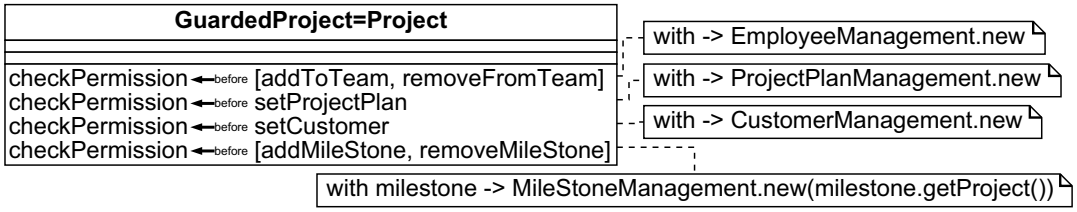

Abbildung 4.26: Zugriffskontrolle für Projekte

so kann sichergestellt werden, dass ein Projektleiter auch nur auf die Planungselemente Zugriff hat, wo er auch Projektleiter ist.

Die Bindung am Beispiel der Klasse `Project` soll für das Verständnis hier ausreichen. Die Bindung der anderen Klassen erfolgt ganz analog, in der Regel sogar weniger komplex. Eine andere Art der „Zugriffskontrolle“ ist in PromOTe durch den Einsatz einer graphischen Benutzeroberfläche gegeben. Um die einzelnen Rollen bestmöglich zu unterstützen, gibt es verschiedene Sichten auf die gleichen Entitäten. So hat ein Teammitglied eine andere Sicht auf ein Projekt, als ein Projektleiter. So entscheidet die jeweilige Stellung eines Mitarbeiters über die jeweilige graphische Schnittstelle. Die verschiedenen graphischen Schnittstellen erlauben auch verschiedene Möglichkeiten der Datenmanipulation. Indem der funktionale Umfang einer Schnittstelle festgelegt ist, sind auch nur diese Möglichkeiten verfügbar. Mit Hilfe einer graphischen Benutzeroberfläche kann man also festlegen, welche Manipulation überhaupt möglich sein sollen und muss weniger definieren, was nicht erlaubt ist. Aus diesem Grund sind die hier definierten Privilegien weniger rigide und umfangreich, als sie vielleicht sein müssten. Wenn diese Anwendung in einer verteilten Umgebung eingesetzt wird, sollte man hier sehr viel mehr tun.

4.4.2.4 Adaptierungen und Möglichkeiten

In der Modellierung des Teams `AccessControl` wird die Annahme gemacht, dass die Anwendung immer von einem einzigen Benutzer ausgeführt wird. Aus diesem Grund wird in dem Team ein konkretes Subjekt modelliert. Diese Annahme trifft auf PromOTe zu, auch wenn es ein gemeinsames Objektrepository gäbe, welches sich mehrere Benutzer teilen. Denkt man aber beispielsweise an einen Webserver, so kann es in der gleichen Laufzeitumgebung mehrere Benutzer geben. Das ist kein Problem, solange ein Zugriff einem Subjekt explizit zugeordnet werden kann. Man würde in solch einer Umgebung den Benutzer nicht im Team definieren, sondern eine Möglichkeit spezifizieren, explizit auf diesen zuzugreifen. Die Auswahl des Subjektes könnte pro Ausführungspfad (*'Thread'*) stattfinden, oder die Anwendung hat selbst ein System zur Verwaltung von Identitäten, die mit der Zugriffskontrolle kooperieren kann (so z.B. bei einem Webserver in einem Session-Objekt). Ansonsten

müsste an dem Team nichts verändert werden, um es auch in einem Mehrbenutzersystem anzuwenden. Die Rolle `Subject` kann überhaupt entfallen, wenn die jeweilige Identitätsform in einem System schon ein Rechtesystem implementiert.

4.4.3 Zusammenfassung

Mit Hilfe von Object Teams ist hier eine sehr elegante Form der Zugriffskontrolle möglich. Die einzelnen Methoden der Domänenklassen werden mit Restriktionen versehen, auf die ein Benutzer nur zugreifen darf, wenn er das nötige Privileg hat.

Der Nachweis über die Sicherheit dieser Herangehensweise konnte hier nicht geführt werden. Diese muss aber auch in der Anwendung geprüft werden und nicht anhand der Programmierschnittstelle. Für die Sicherheit des Mechanismuses selber kommt hier die Tatsache zu Hilfe, dass Rollen ihr Team nie verlassen. Um also die Sicherheit der Zugriffskontrolle selbst zu schützen, muß nur die öffentliche Schnittstelle des Teams geprüft werden. Um eine Zugriffskontrolle zu garantieren, muss sichergestellt werden, dass das Team aktiviert ist. Hier kann die statische Aktivierung ihren Dienst leisten. Ein statisches Team kann nicht instantiiert werden und ist immer aktiv. Hier muss nur das Subjekt gesetzt werden, gegen die die Zugriffskontrolle geprüft werden soll.

5 Fazit

Die Evolution von Software läuft häufig nach einem ähnlichen Schema ab. In der Phase der Modellierung wird eine klare Architektur entworfen, in der die einzelnen Entitäten und Zuständigkeiten explizit separiert werden. Diese Separation von Zuständigkeiten ist schon in der Phase der Implementierung sehr schwierig einzuhalten. Verändern sich dann Anforderungen an das System, oder es kommen neue Anforderungen hinzu, wird die einst klare Struktur mehr und mehr aufgeweicht. Der Entwickler muß die einzelnen Abhängigkeiten des Systems kennen, um dieses zu verändern. Die vielen Aufgaben und Zuständigkeiten erschweren das Verständnis, die Wartbarkeit und die Wiederverwendbarkeit.

Mit der Einführung der Strukturierungsform *Komponente* [Szyperski 2002] rückt dieses Problem in den Vordergrund. Eine Komponente fasst funktionale Zuständigkeiten zusammen und macht diese über Schnittstellen (*'provided interfaces'*) verfügbar. Eine Komponente setzt u.U. andere Schnittstellen voraus (*'required interfaces'*), die für die korrekte Funktionsweise nötig sind. Die Bindung von Komponenten über Schnittstellen erlaubt die explizite Definition von Abhängigkeiten. Eine Komponente wird als *'black box'* begriffen, so dass die interne Struktur und Funktionsweise nicht interessiert. Eine (vom Komponentenentwickler nicht vorgesehene) Anpassung der Funktionalität ist nicht möglich, eine Wiederverwendung schwierig. Szyperski [2002] argumentiert sogar, dass die Wiederverwendung nicht das Hauptziel von Komponenten sind, sondern dass die Bindung über Schnittstellen vor allem Austauschbarkeit garantieren soll.

Das Object Teams Paradigma erlaubt die Kapselung von funktionalen Aspekten als Rollen in einem Team. Eine Bindung dieser Rollen an Domänenklassen wird hier über *spielt*-Beziehungen in einem Konnektor definiert. Auch hier wird eine Schnittstelle definiert *'required interface'*, die bei einer Bindung erfüllt werden muss. Die abstrakten Methoden der Rolle müssen implementiert oder gebunden werden. Die angebotenen Schnittstellen (*'provided interfaces'*) sind die öffentlichen Schnittstellen der Rollenklassen. Teams unterstützen Vererbung. Es können Teams in einer Vererbungshierarchie definiert werden, die verschiedene Abstraktionsebenen zulassen. Teams haben somit auch eine explizite Adaptionsschnittstelle. Rollen werden implizit vererbt und können im erbenden Team, im klassisch objektorientierten Sinne, adaptiert werden. Dieses Vorgehen fördert in einem hohen Maße die Wiederverwendbarkeit.

Eine Strukturierung nach funktionalen Zuständigkeiten verbessert die Lesbarkeit und das Verständnis der zu erbringenden Funktionalität. Die einzelnen Aspekte können separat modelliert und implementiert werden und müssen nicht im Zusammenspiel in einem bestimmten Kontext betrachtet werden. Die so erstellten Aspekte können auf diese Weise auch besser gewartet und erweitert werden, da die einzelnen Abhängigkeiten in den Domänenklassen nicht beachtet werden müssen. Die Änderung eines Rollenverhaltens ist automatisch in jeder Domänenklasse verfügbar, die diese Rolle bindet. Die erbrachte Funktionalität einer bestimmten Methode einer Domänenklasse ist auf diese Weise nicht mehr auf einen Blick erkennbar. Die Funktionalität wird um das Rollenverhalten angereichert, wobei die Definition dieser Bindung extern geschieht. Die Komplexität eines Systems wird auf einzelne Aspekte heruntergebrochen, um dessen Zusammenspiel sich der Entwickler nicht kümmern muß.

Eines der größten Schwierigkeiten mit Aspekten ist die Frage: Was ist überhaupt ein Aspekt? Wie findet man Aspekte? Evaluiert man die Standardbeispiele aktueller Aspekttechniken, so könnte der Eindruck entstehen, dass Aspekte immer etwas mit Logging zu tun haben. Object Teams führt die Metapher *Rolle* ein, die in einem *Team* agiert. Diese Metapher fügt sich in das Bild der Objektorientierung: Klassen bilden Entitäten einer Domäne ab. Kann man ein stereotypes Verhalten unterschiedlicher Entitäten beobachten, so kann dieses Verhalten in einer Rolle gekapselt werden. Der Kontext solch eines Verhaltens wird durch das Team festgelegt. In dieser Diplomarbeit wurden drei Aspekte vorgestellt. Es wurde gezeigt, dass sogar die graphische Repräsentation einer Entität als Rolle implementiert werden kann.

Damit die Technik der Object Teams in den Prozess der Softwareentwicklung integriert werden kann, sind nicht nur Werkzeuge zur Implementierung nötig, sondern auch Modellierungstechniken. Mit der UML Erweiterung UFA [Herrmann 2002a] kann die statische Struktur von Teams beschrieben werden. Von dieser Möglichkeit wurde im Verlauf dieser Arbeit häufig Gebrauch gemacht. Um die Funktionsweise der einzelnen Rollen besser zeigen zu können, werden Diagramme benötigt, mit denen auch das dynamischen Verhalten gezeigt werden kann. Solche Diagramme konnten aus Mangel an Möglichkeiten nicht gezeigt werden.

5.1 PromOTe - eine Statistik

Das von ITSO definierte Domänenmodell (siehe Abbildung 4.2) bildete die experimentelle Grundlage für die Projektmanagementsoftware PromOTe. Diese Implementierung enthält ausschließlich[1] Funktionalitäten, die den Anforderungen die-

[1] Alle Domänenklassen implementieren einen transaktionalen Mechanismus auf Objektebene (siehe Abbildung 4.17). Dieser Mechanismus erwächst aus der Anforderung, dass alle Domänenklassen ihren Zustand bei einer Zustandsänderung prüfen müssen. Um diese Funktionalität zu vereinheit-

ser Domäne genügen. Alle Anforderungen an eine Projektmanagementsoftware, die auf diesem Domänenmodell arbeitet, wurden als Rollen in einem Team implementiert. Die Domänenklassen sollten für die Implementierung dieser Funktionalitäten nicht adaptiert werden. Zu diesen Anforderungen gehört:

- die Bereitstellung einer graphischen Schnittstelle, welche die Objekte der Domäne graphisch repräsentiert.
- die Speicherung von Instanzen der Domänenklassen in eine relationale Datenbank.
- eine Zugriffskontrolle des Systems auf Basis von Privilegien.

In Abbildung 12 sieht man eine kleine Statistik, in der die Anzahl der Klassen und die Anzahl der Zeilen im Quellcode (*'lines of code'*), jeweils mit der prozentualen Verteilung, zu den einzelnen Paketen aufgetragen wurden. (Die graphische Benutzeroberfläche wurde mit einem Werkzeug erstellt. Die Schnittstellenbeschreibung wurde hier nicht mit eingerechnet.) Wie man in dieser Tabelle sehen kann, stel-

Paket	**Klassen**	**Quellcodezeilen**
Graphische Benutzeroberfläche	57 (46 %)	2423 (46 %)
Persistenz	29 (24 %)	1585 (30 %)
Zugriffskontrolle	9 (8 %)	159 (4 %)
PromOTe Domäne	28 (22 %)	1052 (20 %)
Gesamt	123	5219

Abbildung 5.1: Statistik zum Quellcode von PromOTe

len die eigentlichen Domänenklassen nicht den Hauptanteil an Klassen und Zeilen im Quellcode dar. Dies liegt am Wesen einer solchen Anwendung. Eine Projektmanagementsoftware ist vor allem ein Informationssystem, mit relativ wenig (Domänen-)Funktionalität. Trotzdem soll hier festgehalten werden, dass nur 20% des Quellcodes in PromOTe wirklich Domänenfunktionalität beinhalten und 80% für einzelne Aspekte aufgewendet wurden.

Die in PromOTe identifizierten und implementierten Teams sind unabhängig von PromOTe und können wiederverwendet werden. Bei der Implementierung der Rollen wurde versucht eine Adaption bestmöglich zu unterstützen, so dass die einzelnen Funktionalitäten an eine spezielle Domäne angepasst werden können.

lichen, erben alle Klassen von der gleichen Basisklasse. Diese Funktionalität ist aber auch Domänenfunktionalität.

5.2 Erweiterungen von Teams

Alle Teams erben direkt oder indirekt vom Basisteam `Team`. In diesem Team sind Funktionalitäten implementiert, die jedem erbenden Team zur Verfügung stehen. Die Aktivierung oder Deaktivierung wird beispielsweise über Methoden geregelt, die hier definiert sind. Da über die Teamklasse das Verhalten der Rollen und des Teams gesteuert werden kann, ist hier eine Möglichkeit gegeben, die Eigenschaften von Object Teams zu erweitern.

Bei der Realisierung von graphischen Repräsentationen, als Rolle zu einem Modell, wurde beispielsweise festgestellt, dass es möglich sein sollte, den Lifting-Mechanismus an einem Team abzuschwächen (*'weak activation'*, siehe Kapitel 3.2.3.3). Solch eine funktionale Erweiterung konnte in der Teamklasse selbst implementiert werden (da diese Funktionalität von allgemeiner Nützlichkeit ist, wurde sie im Basisteam `Team` implementiert). Der Mechanismus ist auf diese Weise ohne eine Syntaxerweiterung im Team nutzbar. Sollten sich weitere Anforderungen an ein allgemeines Verhalten von Rollen und Teams ergeben, so können diese Erweiterungen in der Teamklasse implementiert werden.

5.3 Wie groß ist ein Team?

Ein Team ist ein abgeschlossenes Konstrukt, indem alle Rollen implementiert werden sollen, die für eine zu erbringende Funktionalität nötig sind. In einer Teamvererbung können neue Rollen definiert werden, die somit jedem erbenden Team zur Verfügung stehen.

Bei der Realisierung der graphischen Benutzeroberfläche für PromOTe wurden abstrakte Teams definiert, die graphischen Standardelementen entsprachen. Dies wurde am Beispiel der Visualisierung einer Baumstruktur (Kapitel 4.2.3.1) gezeigt. In einer Projektansicht werden z.B. verschiedene Domänenobjekte durch solche Standardelemente visualisiert. Damit dies möglich war, musste ein Team geschaffen werden, dass die Rollen dieser unabhängigen Teams vereinigt. Dieses Team erbt sozusagen von allen Teams, welche graphische Standardelemente visualisieren können. Da in Ruby nur einfache Vererbung unterstützt wird, mussten Rollen in diesem neuen Team definiert werden, welche die einzelnen Rollen dieser Teams beerben. Diese Möglichkeit ist nicht zufriedenstellend. Die Abgeschlossenheit eines Teams und die damit verbundene Konsistenz, sollte gewahrt bleiben. In weiterführenden Forschungen sollte daher der Frage nachgegangen werden, wie mehrere Teams, bzw. die Rollen eines Teams, in einem Team *benutzt* werden können.

5.4 Ausblick

Mit Hilfe von Object Teams können Aspekte entkoppelt werden. Solch eine Entkoppelung hilft die Komplexität eines Systems zu reduzieren, indem nur die einzelnen Aspekte und deren Bindung betrachtet werden müssen. Die Metapher von Aspekten als Rolle in einem Team geben dem Entwickler dabei Orientierungshilfe. Aber wie mit jedem neuen Paradigma, so muss auch hier erst ein Gefühl, wie mit dieser neuen Technik umgegangen werden kann, entwickelt werden.

Das diese Technik reif ist für reale Anwendungen, sollte mit PromOTe gezeigt werden. Natürlich sind noch viele Änderungen und Verbesserungen am Modell zu erwarten. So ist die Frage der Abgeschlossenheit eines Teams (Kapitel 5.3) aktuell in der Diskussion. Es könnte überlegt werden, ob Bindungen nur auf Methodenebene begrenzt bleiben, oder ob bestimmte Methoden als nicht bindungsfähig deklariert werden. Für solche Verfeinerungen bedarf es weiterer Evaluierungen und Erfahrungen. Die hier vorliegende Diplomarbeit wollte in diesem Sinne einen ersten Beitrag leisten.

A Anhang

A.1 Begriffserklärung

Access Control	Zugriffskontrolle.
Assertion	Überprüfung einer Behauptung
Browser	Anzeigegerät für Webseiten
Controller	Im Model-View-Control Paradigma übernimmt der Controller die Rolle eines Mediators zwischen Model und View.
Discretionary	Handlungsfreiheit, Vollmacht.
Effektive Klasse	Eine Klasse ist solange abstrakt, solange sie abstrakte Methoden enthält. Sind alle abstrakten Methoden definiert, wird die Klasse effektiv.
GUI Graphical User Interface	Graphische Benutzeroberfläche
Label	Textfeld in einer graphischen Benutzeroberfläche.
Logging	Kontrolldaten, die Programmfluss beschreiben
Mandatory	Obligatorisch, Zwingend.
Middleware	Kommunikationsmedium zwischen verschiedenen Laufzeitumgebungen. In der Regel wird sie zur Kommunikation über Rechnergrenzen hinaus genutzt. Bekannteste Formen: CORBA, RMI, SOAP.
Persistent	Dauerhafte Speicherung von Daten. Beständig, Fortbestehend.
Precondition	Vorbedingung
Postcondition	Nachbedingung
Reiter	Geschichtete Fenster in einer graphischen Benutzeroberfläche, denen Laschen zugeordnet werden. Anhand der Lasche kann das jeweilige zugeordnete Fenster in den Vordergrund geholt werden. Die jeweiligen Fenster heißen Reiter.
Role Based Access Control	Rollenbasierte Zugriffskontrolle.
Transient	Gegenteil von Persistent. Vergänglich, Flüchtig.
View	Repräsentation einer Modellklasse im Model-View-Controller Paradigma.

A.2 Bildschirmfotos der Projekbearbeitungssicht

Abbildung A.1: Die Projektsicht.

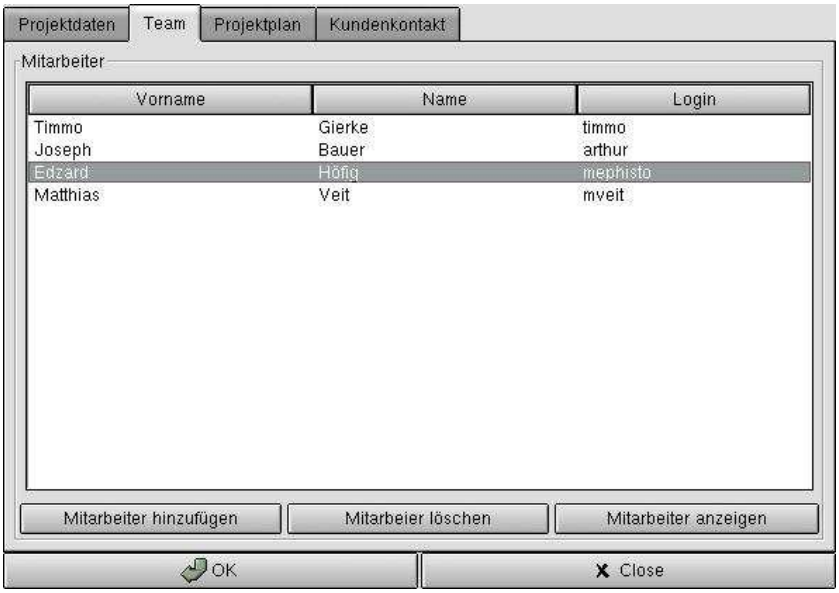

Abbildung A.2: Alle Mitarbeiter im Team.

Abbildung A.3: Alle Meilensteine des Projektes.

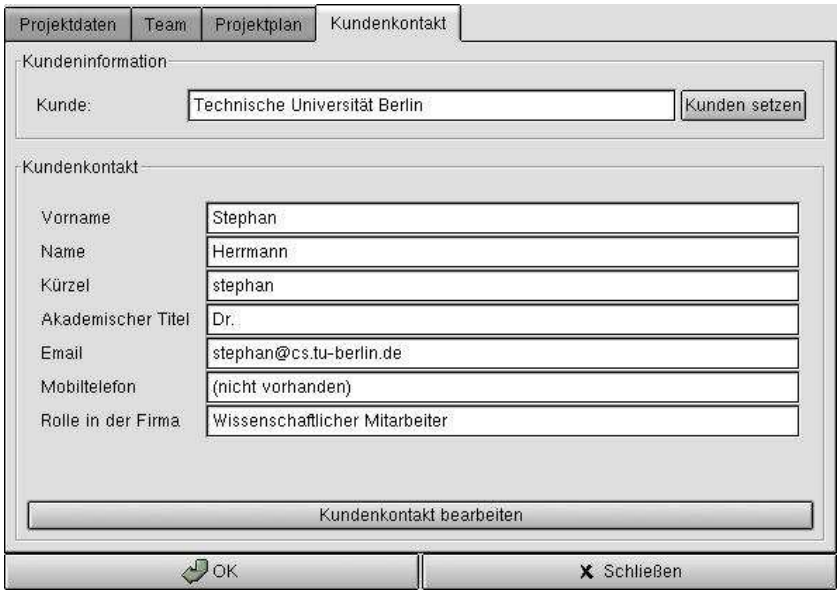

Abbildung A.4: Der Auftraggeber mit zugehörigem Kontakt.

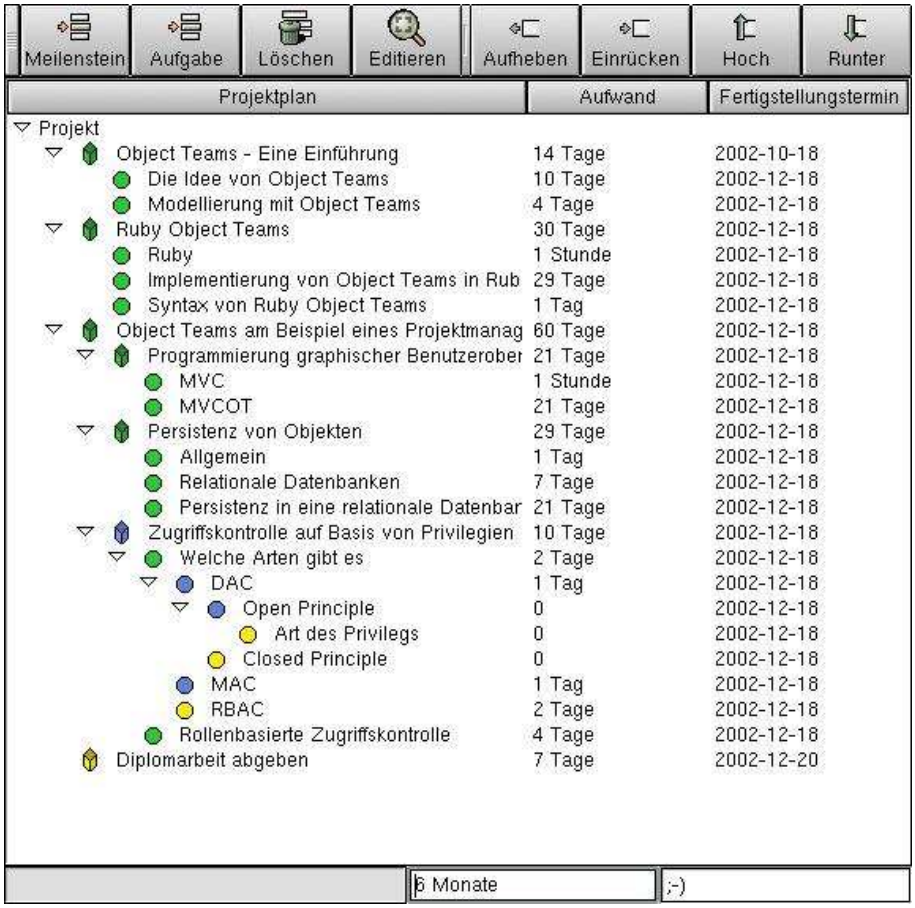

Abbildung A.5: Projektplanung - ein Baum von Aufgaben und Meilensteinen.

A.3 Danksagung

Hiermit danke ich meinen fleißigen Korrektoren Timmo, Stephan, Jule und Joseph für die inhaltlichen und sprachlichen Anregungen und Verbesserungen dieser Arbeit.

Literaturverzeichnis

[AOSD 2002] AOSD: *AOP*. WEB. 2002. – URL `http://aosd.net/`. – Zugriffsdatum: September 2002

[Binder und Hundt 2002] BINDER, Christoph ; HUNDT, Christine: *Java Object Teams*. 2002. – URL `http://www.objectteams.org/software/javaot.html`. – Zugriffsdatum: Juli 2002

[Branca und Oser 2002] BRANCA, Andrea ; OSER, Philipp: Gut abgelegt. In: *IX* (2002), Dezember, S. 106–108

[Brichau u. a. 2000] BRICHAU, Johan ; DE MEUTER, Wolfgang ; DE VOLDER, Kris: Jumping Aspects / Programming Technology Lab Vrije Universiteit Brussel. Pleinlaan 2, 1050 Brussel, Belgium, 4 April 2000. – Forschungsbericht. ECOOP 2000, Workshop: Aspects & Dimensions of Concerns workshop

[Chaplin 2001] CHAPLIN, Damon: *GLADE User Interface Builder*. WEB. 2001. – URL `http://glade.gnome.org`. – Zugriffsdatum: 20.12.2002

[Fischer 2000] FISCHER, Thorsten: *GUI-Programmierung mit GTK+*. SuSE Press, 2000. – ISBN 3-934678-42-4

[Fulton 2001] FULTON, Hal: *The Ruby Way*. Sams Publishing, Dezember 2001. – ISBN 0-672-32083-5

[Gamma u. a. 1996] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns*. Addidon Wesley Longman Verlag GmbH, 1996. – ISBN 0-201-63361-2

[Glöckner und Storl 2001] GLÖCKNER, Peter ; STORL, Christian: *Fallstudie Domänenmodell "Projektmanagement- und Informationsportal"*. ITSO. Mohrenstr. 63-64 10117 Berlin: IT Service OMIKRON GmbH (Veranst.), 20 Dezember 2001

[GTK+ Team 2001] GTK+ TEAM: *Gimp Tool Kit*. WEB. 2001. – URL `http://www.gtk.org`. – Zugriffsdatum: 20.12.2002

[Herrmann 2000] HERRMANN, Stephan: Dynamic view connectors for separating concerns in software engineering environments. In: *Workshop on multi-dimensional separation of concerns in software engineering (icse 2000)*, URL `http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/herrmann.pdf`, Juni 2000

[Herrmann 2002a] HERRMANN, Stephan: Composable designs with UFA. In: *Workshop on aspect-oriented modeling with UML (aosd-2002)*, URL `http://lglwww.epfl.ch/workshops/aosd-uml/Allsubs/Herrmann.pdf`, März 2002

[Herrmann 2002b] HERRMANN, Stephan: ObjectTeams: Improving Modularity for Crosscutting Concerns / Technical University Berlin. 2002. – Forschungsbericht

[Herrmann und Mezini 2001] HERRMANN, Stephan ; MEZINI, Mira: Combining composition styles in the evolvable language LAC. In: *Workshop on advanced separation of concerns in software engineering (icse 2001)*, URL `http://www.research.ibm.com/hyperspace/workshops/icse2001/Papers/herrmann.pdf`, Mai 2001

[Igarashi u. a. 2001] IGARASHI, Hiroshi ; CONWAY, Neil ; MUTOH, Masao: *Ruby-Gnome*. WEB. 2001. – URL `http://ruby-gnome.sourceforge.net`. – Zugriffsdatum: 20.12.2002

[Kiczales u. a. 2001] KICZALES, G. ; HILSDALE, E. ; HUGUNIN, J. ; KERSTEN, M. ; PALM, J. ; GRISWOLD, W. G.: An overview of AspectJ. In: KNUDSEN, J. L. (Hrsg.): *Proc. ecoop 2001, lncs 2072*. Berlin : Springer-Verlag, Juni 2001, S. 327–353

[Krasner und Pope 1988] KRASNER, Gelnn E. ; POPE, Stephen T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk 80. In: *Joop*, September 1988, S. 26–41

[Libelis 2002] LIBELIS: *LiDO Community Edition*. WEB. 2002. – URL `http://www.libelis.com/`. – Zugriffsdatum: 11.12.2002

[Matsumoto 2002] MATSUMOTO, Yukihiro: *Ruby Programming Language*. WEB. 2002. – URL `http://www.ruby-lang.org/en/index.html`. – Zugriffsdatum: Juli 2002

[Meyer 1997] MEYER, Bertrand: *Object -oriented software construction*. Prentice Hall PTR, 1997. – ISBN 0-13-629155-4

[MySQL AB 2002] MYSQL AB: *MySQL*. WEB. 2002. – URL `http://www.mysql.com`. – Zugriffsdatum: 20.12.2002

[Neumann 2002] NEUMANN, Michael: *Ruby/DBI*. WEB. 2002. – URL `http://ruby-dbi.sourceforge.net`. – Zugriffsdatum: 20.12.2002

[Oestereich 2001] OESTEREICH, Bernd: *Die UML-Kurzreferenz für die Praxis*. Oldenbourg Wissenschaftsverlag GmbH, 2001. – ISBN 3-486-25637-8

[Oheim 1999] OHEIM, Jürgen: *Datenbanksysteme*. Vorlesungsskript. Berlin: Technische Universität (Veranst.), 23 August 1999

[OMG 2002] OMG: *Unified Modelling Language*. 2002. – URL `http://www.omg.org/uml`. – Zugriffsdatum: Septmeber 2002

[Samarati und Vimercati 2000] SAMARATI, Pierangela ; VIMERCATI, Sabrina de Capitani d.: Access Control: Policies, Models, and Mechanisms. In: SPRINGER-VERLAG BERLIN HEIDELBERG (Hrsg.): *FOSAD* Bd. 2171. Via Bramante 65, 26013 Crema (CR), Italy : Forcadi, R. Gorrieri, R., 2000, S. 137–196. – 2001

[Stearns 2001] STEARNS, Beth: *Container-Managed Persistence Example*. WEB. Juli 2001. – URL `http://developer.java.sun.com/developer/technicalArticles/ebeans/EJB20CMP/`. – Zugriffsdatum: 20.12.2002

[Sun Microsystems 2002a] SUN MICROSYSTEMS: *Enterprise JavaBeans Specifications*. WEB. 2002. – URL `http://java.sun.com/products/ejb/docs.html`. – Zugriffsdatum: 20.12.2002

[Sun Microsystems 2002b] SUN MICROSYSTEMS: *JSR-000012 Java Data Objects (JDO) Specification*. WEB. 11 Dezember 2002. – URL `http://jcp.org/aboutJava/communityprocess/final/jsr012/index.html`. – Zugriffsdatum: 11.12.2002

[Szyperski 2002] SZYPERSKI, Clemens: *Component Software*. Addison Wesley, 30 November 2002. – ISBN: 0201745720

[Thomas und Hunt 2000] THOMAS, David ; HUNT, Andrew: *Programming Ruby*. Addidon Wesley Longman Verlag GmbH, 15 Dezember 2000. – ISBN 0201710897

[Veit 2002] VEIT, Matthias: *Ruby Object Teams*. WEB. 2002. – URL `http://robjectteam.sourceforge.net`. – Zugriffsdatum: 20.12.2002

Index