Einbindung formaler Constraints in UML Spezifikationen

Theo Sattler

Karlsruhe, den 31. Januar 2000

Diplomarbeit

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Betreuer:
Prof. Dr. Wolfram Menzel
Dipl.-Inform. Thomas Baar

Ich bedanke mich bei meinem Betreuer Thomas Baar für die intensive Betreuung. Auch möchte ich mich bei der Firma **fun communications** GmbH für die freundliche Aufnahme bedanken. Besonderer Dank gebührt dabei Thomas Fuchß und Dirk Arnoldt, die mir stets hilfreich zur Seite gestanden haben. Schließlich möchte ich noch allen danken, die beim Korrekturlesen mitgeholfen haben.

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 31. Januar 2000

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bisher werden formale Methoden in erster Linie in Bereichen eingesetzt, die besondere Sicherheitsanforderungen erfüllen müssen. Dies liegt vor allem daran, daß zur nutzbringenden Anwendung formaler Methoden ein mathematisch-logisches Wissen gefordert wird, das nur selten vorausgesetzt werden kann. Um dennoch formale Methoden anzuwenden ist es daher notwendig, aufwendige Fortbildung zu betreiben oder teure externe Experten zu beauftragen.

Dieser Kostenfaktor führt dazu, daß formale Methoden nur in Bereichen angewendet werden, in denen die Sicherheitsanforderungen es verbieten, andere weniger sichere Methoden zu benutzen.

1.1 Problemstellung

Um formale Methoden nutzbringend anwenden zu können, ist es notwendig, Anforderungen, die an ein Projekt gestellt werden, formal zu fassen. Diese Formalisierung ist jedoch ein fehlerträchtiger Prozeß. Das Ziel der Diplomarbeit ist es, die Formalisierung von Anforderungen so zu unterstützen, daß diese ohne umfangreiche Einarbeitung angewandt werden können.

In diesem Zusammenhang wird anhand eines von der Firma **fun communications** GmbH entwickelten eMail-Clients (*keyMail/S*) erörtert, welche Forderungen an Software gestellt werden. Dies soll sicherstellen, daß im weiteren praxisrelevante Eigenschaften betrachtet werden.

Für die identifizierten Forderungen wird diskutiert, inwieweit sich diese in allgemeiner Weise formal ausdrücken lassen. Dabei wird auch untersucht, wie man die Transformation der allgemeinen Formulierungen in konkrete Problemstellungen automatisieren bzw. vereinfachen kann. Die Forderungen werden anschließend für das Projekt transformiert und der Übergang von der allgemeinen Formulierung auf die konkrete Situation veranschaulicht.

1.2 Umgebung

Die **UML** (Unified Modeling Language) hat sich inzwischen allgemein als Beschreibungssprache in objekt-orientierter Analyse und Design durchgesetzt und wird von den meisten CASE-Tools unterstützt. Mit einem breiten Spektrum von Diagrammen bietet sie Unterstützung für den gesamten Entwurfsprozeß. In [SMHP+99] wird die **UML** ausführlich beschrieben, wobei Kapitel 1 einen guten Überblick über die **UML** bietet. Für die weitere Diskussion ist im wesentlichen das Klassendiagramm von Bedeutung. Die Semantik der **UML** ist weitgehend semiformal gefaßt. Für Teile der **UML** wurde jedoch auch schon eine formale Definition der Semantik erstellt (siehe z.B. [EC97] und [LB98a]).

Als Sprache für die Formalisierung von Forderungen wurde die **OCL** (Object Constraint Language) gewählt. Die **OCL** ist eng mit der **UML** verbunden, so wurden zum Beispiel Teile der Definition des Meta-Modells der **UML** in **OCL** formuliert. Die **OCL** benutzt eine an OO-Programmierspachen angelehnte Notation, die sie für Programmierer gut lesbar macht. Mit Hilfe der **OCL** lassen sich Forderungen an das Design formal fassen. Eine ausführliche Beschreibung der **OCL** findet sich in [SMHP+99], Kapitel 7.

Zur Beschreibung von Problemlösungsideen haben Entwurfsmuster weite Verbreitung gefunden. Innerhalb eines Entwurfsmusters wird jeweils ein einzelnes Problem, welches häufig in Designs auftaucht, angegangen. Für dieses Problem wird dann eine bewährte Lösung in allgemeiner Form gegeben, sowie Hilfen um diese Lösung an die konkrete Situation anzupassen. Für verschiedene Problemfelder werden Entwurfsmuster in Mustersprachen zusammengefaßt. Die im weiteren benutzten Entwurfsmuster wurden den in [GHJV95], [BMR+96]und [MRB98] definierten Mustersprachen entnommen.

1.3 Lösungsansatz

Bei der Suche nach Forderungen, die an die Software gestellt werden, ergab sich, daß zwar etliche Anforderungen auftreten und auch implizit gefordert werden, diese aber eher umgangssprachlich formuliert sind. Der Aufwand, solche Forderungen formal zu fassen, wird als zu hoch erachtet.

Bei der Betrachtung der Forderungen die an *keyMail/S* gestellt werden, ergab sich, daß viele dieser Forderungen in einem direkten Zusammenhang mit Entwurfsmustern gestellt werden können. Ein weiterer Teil ist verhältnismäßig einfach, so daß dafür keine Entwurfsmuster existieren. Ein Grenzefall stellt zum Beispiel das Singleton Muster dar (siehe Abschnitt 5.1), dessen zentrale Forderung in einem einzelnen **OCL**-Ausdruck formuliert werden kann.

Die im weiteren betrachtete Lösung versucht, Forderungen, die im Zusammenhang mit Entwurfsmustern stehen, in allgemeiner Form formal zu fassen und mit dem entsprechenden Muster zu verknüpfen. Zusätzlich sollen dem Entwickler Hilfestellungen gegeben werden, um diese Forderungen bei der Verwendung des Entwurfsmusters an die speziellen Bedürfnisse anzupassen. Allgemeine Forderungen, für die noch kein Entwurfsmuster existiert, werden durch neue Muster beschrieben.

In Kapitel 3 wird eine detaillierte Beschreibung des Ansatzes gegeben. Zuvor wird in Kapitel 2 das Projekt *keyMail/S* vorgestellt. Dabei wird auf das Design eingegangen und aufgezeigt welche Forderungen bei *keyMail/S* aufgetreten sind. Kapitel 4 beschreibt Probleme und Einschränkungen, die im Zusammenhang mit **OCL** stehen. In Kapitel 5 bis 7 werden dann die identifizierten Forderungen ausführlich diskutiert.

Kapitel 2

Projekt *keyMail/S*

Die Grundlage für die Suche nach praxisrelevanten Forderungen stellt *keyMail/S* dar, ein eMail-Client der Firma **fun communications** GmbH. Analyse und Design des Projektes sind weitgehend in **UML** erstellt. Dabei wurde ausgiebig Gebrauch von Entwurfsmustern gemacht.

Im Folgenden wird das Projekt kurz beschrieben und anschließend ein Überblick über die an das Projekt gestellten Forderungen gegeben.

2.1 Überblick

keyMail/S soll zum einen eMails senden, empfangen und verwalten können. Desweiteren soll *keyMail/S* auch eine Kontaktverwaltung beinhalten und für mehrere Benutzer konfigurierbar sein. Zusätzlich soll der eMail Client Unterstützung für Verschlüsselung und Signatur von eMails bieten. Dazu stellt *keyMail/S* eine vollständige Zertifikatverwaltung zur Verfügung.

Somit besteht *keyMail/S* im wesentlichen aus fünf Teilen: Benutzerverwaltung, Accountverwaltung, Kontaktverwaltung, Nachrichtenverwaltung und Zertifikatverwaltung.

Die Benutzerverwaltung kontrolliert den Zugang zu *keyMail/S*. Ein Benutzer der *keyMail/S* benutzen will, muß sich über die Benutzerverwaltung anmelden. Jeder Benutzer kann mehrere Accounts besitzen. Jeder Account repräsentiert einen entsprechenden Mail-Server. Aufgaben in diesem Bereich werden von der Accountverwaltung übernommen. Die Kontaktverwaltung stellt jedem Benutzer ein Adreßbuch zur Verfügung, in dem der Benutzer seine Kontakte hierarchisch ordnen und zu Gruppen zusammenfassen kann. Die Nachrichtenverwaltung stellt dem Benutzer Ordner zur Verfügung, mit denen dieser Nachrichten archivieren kann.

Da alle oben aufgezeigten Teile des Systems ihre Daten in Listen bzw. Graphen verwalten, werden diese in der weiteren Diskussion eine große Rolle spielen. In diesem Zusammenhang wird besonders auf das Composite Muster (siehe Abschnitt 5.2) eingegangen. Desweiteren müssen die Daten in den einzelnen Teilen über die Laufzeit des Systems hinaus erhalten bleiben. Daher ist es notwendig, die Daten in eine linearisierte Form zu bringen,

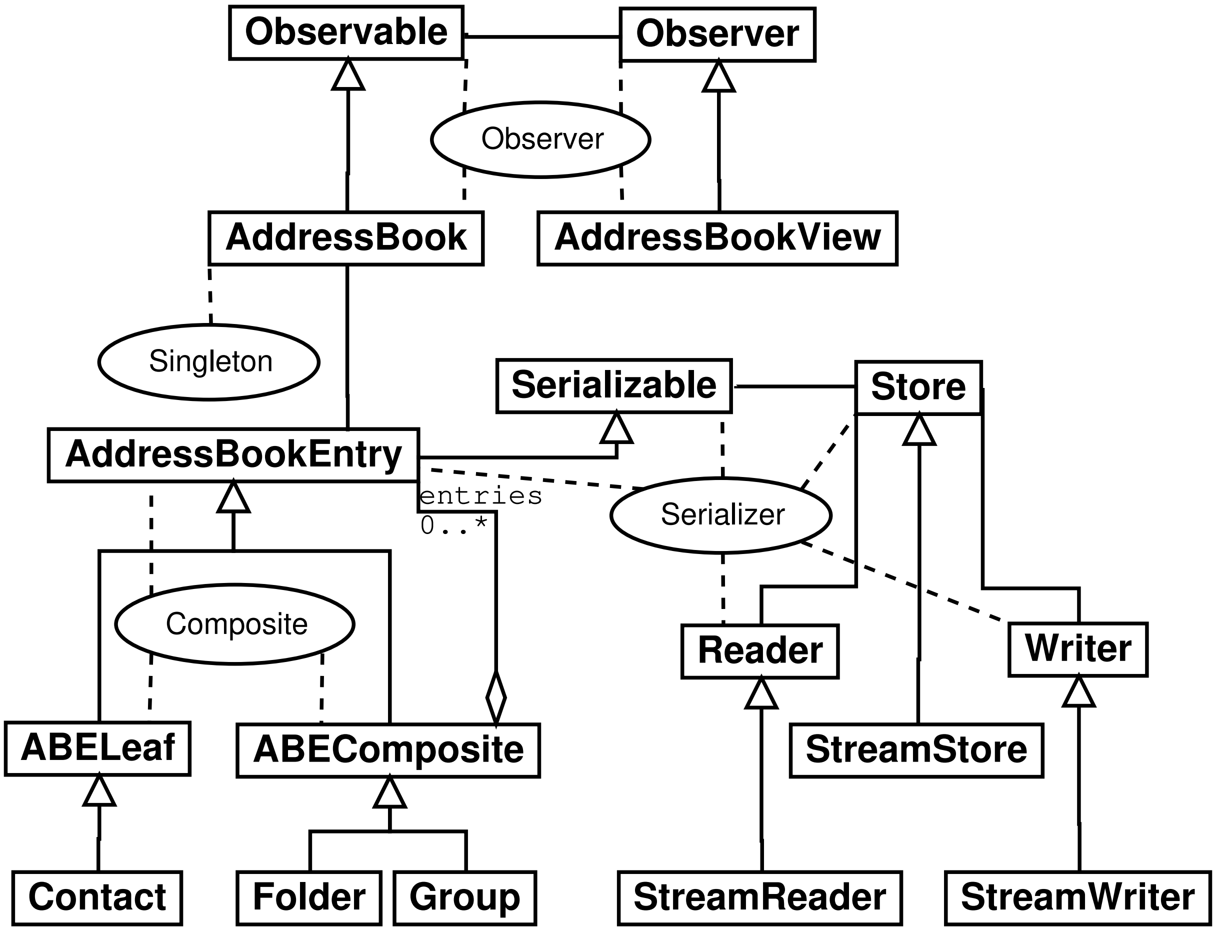

Abbildung 2.1: Ausschnitt aus dem Klassendiagramm für die Kontaktverwaltung

die z.B. in einer Datei abgelegt werden kann. Dazu bietet sich das Serializer Muster an, das in Abschnitt 6.1 behandelt wird.

Zum Zeitpunkt der Identifizierung der Forderungen war die Kontaktverwaltung bereits am weitesten entwickelt, so daß sich im weiteren die meisten Forderungen auf die Kontaktverwaltung beziehen. Bei Betrachtung der anderen Teile des Projektes traten jedoch vergleichbare Forderungen auf. Im nächsten Abschnitt wird die Kontaktverwaltung näher beschrieben.

2.2 Kontaktverwaltung

Zur Veranschaulichung der formalisierten Constraints wird auf die Kontaktverwaltung Bezug genommen. Abbildung 2.1 zeigt den für die weiteren Betrachtungen relevanten Teil des Klassendiagramms der Kontaktverwaltung. Die Identifikation passender Entwurfsmuster, sowie die Abbildung der Musterklassen auf die Klassen der Kontaktverwaltung, ist bereits erfolgt.

Die Klasse **AddressBookEntry** und ihre Unterklassen speichern die Daten der Kontaktverwaltung in einer Baumstruktur. **Contact** repräsentiert die Blätter des Baumes und repräsentieren einen einzelnen Kontakt. Die Klassen **Folder** und **Group** bilden die Knoten des Baumes und dienen der Organisation der Kontakte.

AddressBook stellt die Schnittstelle der Kontaktverwaltung zu den anderen Teilen des Systems zur Verfügung. Über den Observer-Mechanismus bietet es anderen Objekten die Möglichkeit, sich von Zustandsänderungen im **AddressBook** unterrichten zu lassen. **AddressBookView** stellt den Inhalt des Adreßbuches in einem Fenster dar. **AddressBookView** benutzt den Observer-Mechanismus, um sicherzustellen, daß stets der aktuelle Inhalt dargestellt wird.

Der Inhalt des Adreßbuches muß über die Laufzeit des eMail-Clients hinaus erhalten bleiben, zu diesem Zweck erbt **AddressBookEntry** von der Klasse **Serializable**. Diese stellt, zusammen mit den Klassen **Store**, **Reader** und **Writer** einen Mechanismus zur Linearisierung des Adreßbuches zur Verfügung. Die Klassen **StreamStore**, **StreamReader** und **StreamWriter** implementieren diesen Mechanismus.

2.3 Forderungen

Tabelle 2.1 gibt einen Überblick über die Forderungen, die im Zusammenhang mit *keyMail/S* aufgetaucht sind. Dabei bezeichnet die Spalte „Struktur“ die Datenstruktur auf der die Forderung arbeitet. „Muster“ bezieht sich auf das Entwurfsmuster mit dem die Forderung assoziiert ist, unter „Forderung“ wird eine kurze Beschreibung der Forderung gegeben. Die Spalte „Klassifikation“ gibt eine Einteilung der Forderungen in „statische“, „dynamische“ und „primitive“ Forderungen wieder. Dynamische Forderungen beziehen sich auf zeitliche Zusammenhänge von Objekten. Statische Forderungen beziehen sich auf Zusammenhänge zwischen Objekten, die ohne zeitliche Bezüge geprüft werden können. Als primitiv werden solche Forderungen klassifiziert, die keinem spezifischen Muster zugeordnet sind und mit einfachen Datenstrukturen in Zusammenhang stehen.

Die meisten Forderungen beziehen sich auf statische Aspekte des Systems. Diese eignen sich auch besonders gut, um in **OCL** formuliert zu werden. Dynamische Forderungen sind deutlich schwerer zu formulieren, da **OCL**, abgesehen vom `@pre` Operator, keine Unterstützung von zeitlichen Bezügen bietet. Dies kann teilweise mit der Definition von Hilfsmethoden umgangen werden, deren zeitliches Verhalten in Sequenzendiagrammen spezifiziert wird.

Kapitel 5 geht detailiert auf statischen Forderungen, Kapitel 6 auf dynamische Forderungen ein. Schließlich beschreibt Kapitel 7 einfache Forderungen, die nicht direkt mit einem Muster verbunden sind.

Für jedes betrachtete Muster wird eine kurze Beschreibung der Motivation und des dahinterstehenden Lösungsansatzes gegeben. Danach werden die mit dem Muster in Zusammenhang stehenden Forderungen umgangssprachlich beschrieben, anschließende faßt

Struktur	Muster	Forderung	Seite	Klassifikation
Objekt	Singleton	max. eine Instanz existiert	23	statisch
Graph	Composite	Mehrfachkanten	27	statisch
Graph	Composite	Einschränkung von Kanten	28	statisch
Graph	Composite	Zyklen	28	statisch
Graph	Composite	Erreichbarkeit	28	statisch
Graph	Composite	Kopieren einer Komponente	28	statisch
Graph	Serializer	*Read*, *Write* invers	40	dynamisch
Relation	Observer	Zustandswechsel wird propagiert	48	dynamisch
Objekt	–	Gleichheitsrelation auf Objekten	55	primitiv
Objekt	–	Objekt ist Duplikat	55	primitiv
Objekt	–	Ordnungsrelation auf Objekten	56	primitiv
Liste	–	ist Teilmenge	57	primitiv
Liste	–	ist Element	57	primitiv
Liste	–	ist Menge	57	primitiv
Liste	–	notwendige Elemente	57	primitiv
Liste	–	enthält neues Element	57	primitiv
Liste	–	Sequence ist sortiert	58	primitiv
Liste	–	Element einfügen	58	primitiv
Liste	–	Element entfernen	59	primitiv
Liste	–	Element einschränken	61	primitiv

Tabelle 2.1: Verzeichnis der Forderungen

eine Tabelle die identifizierten Forderungen nochmals zusammen.

Jede Forderung wird in **OCL** formalisiert. Die in diesem Schritt gegebene Formalisierung ist möglichst allgemein, um die Anwendbarkeit des Musters nicht einzuschränken. Für diesen Zweck wird eine erweiterte Syntax für **OCL** verwendet, die in Kapitel 3 beschrieben wird.

Zu jedem Muster folgt ein Abschnitt, indem das Muster auf das konkrete Projekt angewandt wird. Es werden jeweils nur diejenigen Forderungen benutzt, die in der konkreten Situation auch benötigt werden.

Da primitive Forderungen nicht mit einem spezifischen Muster assoziiert sind, werden diese in abgeänderter Form besprochen. Für primitive Forderungen wird eine kurze Beschreibung der Forderung, gefolgt von der dazugehörenden Formalisierung, gegeben.

Zur besseren Identifikation gelten im weiteren folgende Konventionen: Klassennamen sind **fett**, Methodennamen, Attribute und Assoziationen sind *kursiv* gedruckt. Dabei unterscheiden sich Methoden von Attributen und Assoziationen durch die angehängten Klammern. Zwischen Attributen und Assoziationen wird keine Unterscheidung getroffen, da sie für die Formalisierung unerheblich ist.

Kapitel 3

Formale Muster

Entwurfsmuster sind inzwischen eine verbreitete Methode, Lösungsansätze für softwaretechnische Probleme zu kommunizieren. Wie sich bei der Analyse realer Projekte gezeigt hat, kann damit ein großer Teil des Designs abgedeckt und ein substantieller Gewinn in der Softwarequalität erreicht werden (siehe z.B. [BCM+96]). Für die weitere Diskussion wird davon ausgegangen, daß die Identifikation des Entwurfsmuster, sowie die Anpassung des Modells, bereits erfolgt ist. [LB98b] bietet einen formalen Ansatz für entsprechende Transformationen.

In Zusammenhang mit den jeweiligen Mustern ergeben sich aber auch Forderungen, die der von dem Muster erfaßte Teil des Designs erfüllen sollte. Es ist naheliegend diese Anforderungen direkt mit dem jeweiligen Entwurfsmuster zu verknüpfen. Da diese Erweiterung nur einmal zu erfolgen hat, verteilen sich die Kosten für etwaige Experten, die zur formalen Spezifikation der Forderungen benötigt werden. Ein Entwickler kann dann durch Wahl eines so angereicherten Musters von formalen Methoden Gebrauch machen, ohne daß er die Forderungen erneut formalisieren muß.

Im allgemeinen werden die mit einem Entwurfsmuster verknüpften Forderungen nicht eindeutig sein. Bestehen die möglichen Forderungen aus einer Menge eindeutiger Forderungen, kann der Benutzer des Musters diese einfach aus dieser Menge auswählen. Werden für die genaue Formulierung der Forderung noch weitere Angaben benötigt, die beim Entwurf des Musters noch nicht bekannt sind, so versieht man das Muster entsprechend mit Parametern (siehe z.B. Einschränkung von Kanten im Composite Muster, Abschnitt 5.2.3.2).

3.1 OCL-Schablonen

OCL-Constraints, die innerhalb eines Musters formuliert werden, stellen eigentlich nur Schablonen für konkrete Constraints dar. Zuerst müssen für die konkrete Anwendung des Constraints natürlich die Namen der Klassen, Methoden, Attribute und Assoziationen angepaßt werden. Innerhalb eines Constraint können auch Teilausdrücke vorkommen, die abhängig von der Entwurfsentscheidung zum Einsatz kommen sollen. Schließlich kann es noch Teilausdrücke geben, die in der konkreten Anwendung mehrfach eingesetzt werden sollen, jeweils angepaßt für eine Menge von Objekten.

Bei der Formalisierung von Forderungen treten auch Aussagen über Modellelemente (Klassen, Methoden, Attribute, ...) auf, die vor der Instanziierung noch nicht identifiziert werden können. Um über diese Modellelemente Aussagen treffen zu können, müssen diese Modellelemente mittels Variablen repräsentiert werden. Dies ist in **OCL** nicht möglich.

Im Folgenden soll nun eine Erweiterung der **OCL** Syntax definiert werden, welche die obigen Funktionalitäten bietet. Eine Schablone wird mit **ocltemplate** eingeleitet und erhält einen Namen über den die Schablone referenziert werden kann. Um eine Schablone für ein konkretes Projekt anzupassen, wird eine Instanziierungsmethode definiert. Diese Methode erhält eine Abbildung, die das Modell der Schablone auf das Modell des Projektes abbildet. Diese Abbildung wird für das gesamte Muster definiert und gilt dann für alle darin enthaltenen Templates. Zusätzlich werden eventuelle weitere Variablen als Parameter übergeben, die Entwurfsentscheidungen widerspiegeln.

Da mit der Instanziierung das konkrete Modell verfügbar ist, kann die Belegung von Variablen für Modellelemente aufgelöst werden. Es ist somit hinreichend, Variablen für Modellelemente nur innerhalb von Schablonen zuzulassen.

Es werden zwei zusätzliche Statements eingeführt: **foreach** entspricht dabei einem **forAll**, **case** einem **if**. Die neuen Statements werden jedoch während der Instanziierung der Schablone ausgewertet und dienen der Anpassung des **OCL** Ausdrucks an die Bedürfnisse der konkreten Anwendung.

Mithilfe von *instanciate* werden Schablonen instanziiert. Dabei wird eine **Map** übergeben, welche die Abbildung der in der Schablone definierten Templatevariablen auf die Modellelemente der konkreten Anwendung enthält.

3.2 Beispiel

Als Beispiel soll eine Forderung an die Attribute einer Klasse dienen. Dabei soll der erlaubte Typ der Attribute der Klasse eingeschränkt werden. Abhängig von der Geschmacksrichtung (`flavor = 'positive'` bzw. `flavor = 'negative'`) soll für alle Attribute gelten, daß sie zu einer Ober-, bzw. Unterklasse einer gegebenen Klasse gehören. Dabei wird in einer Schleife der Typ aller Attribute geprüft und mit der Klasse verglichen.

Sowohl die Schleife, wie auch die Entscheidung, ob „positiv“ oder „negativ“ geprüft werden soll, kann zum Zeitpunkt der Anwendung der Forderung auf das konkrete Projekt aufgelöst werden.

Constraint 3.2.1 (Beispiel für Constraint-Templates)

```
ocltemplate Beispiel {flavor,RClass} ocl:
   context ConstraintClass inv:
       let attribs = Serializable.attributes in
     attribs.foreach (att |
       case (flavor = 'positive')
         then att.oclIsKindOf(RClass) and
       case (flavor = 'negative')
         then not att.oclIsKindOf(RClass) and
     ) true
```

Nun wird dieser Constraint für die Klasse **ClassA** instanziiert. Die Attribute der Klasse **ClassA** seien *aAttrib1* und *aAttrib2*. Diese sollen in positiver Weise auf **ClassB** eingeschränkt werden. Somit evaluiert

```
Beispiel.instanciate(Map{(flavor : 'positive'),(RClass : ClassB)})
```

zu:

```
context AClass inv:
  aAttrib1.oclIsKindOf(ClassB) and
  aAttrib2.oclIsKindOf(ClassB) and
  true
```

3.3 Assoziative Felder (Map)

Um die Semantik einer **OCL** Schablone zu definieren, wird eine Abbildung der Parameter auf Bezeichner im konkreten Entwurf benötigt. Dazu wird der von **Sequence** abgeleitete Typ **Map** definiert. Naheliegend wäre die Repräsentation einer **Map** als Liste von Listen. Dies ist in **OCL** jedoch nicht möglich, da Listen von Listen automatisch in einfache Listen umgewandelt werden.

Die hier aufgezeigte Vorgehensweise ist analog zu einem in [MC99] vorgeschlagenen Vorgehen zur Darstellung des kartesischen Produkts zweier Mengen.

Um eine einfache Repräsentation für Schlüssel-Wert Paare zu verwirklichen, werden Elemente mit ungeraden Indizes als Schlüssel für den jeweils nachfolgenden Wert definiert. Dies impliziert, daß die Länge der **Sequence** ein Vielfaches von Zwei ist.

```
context Map(S,T)
  inv:
    -- -- Länge der Sequence ist gerade
    self->size.mod(2) = 0;
  inv:
    -- -- Schlüssel sind vom Typ S
    -- -- Elemente sind vom Typ T
    Sequence{1..self->size}->forAll(index : Integer |
      index.mod(2) = 0 implies self->at(index).oclIsKindOf(T) and
      index.mod(2) = 1 implies self->at(index).oclIsKindOf(S) )
```

Erlaubt man, daß ein Schlüssel mehrfach vorkommt, kann man eine Abbildung auf Listen von Elementen simulieren. Als Wert zu einem Schlüssel wird dann die Liste aller Elemente, die unter diesem Schlüssel eingetragen sind, zurückgegeben.

```
context Map::atKey(s : S) : Sequence(T)
  -- -- Suche alle Elemente die über Schlüssel s indiziert werden
  let keys = Sequence{1..self->size}->select(key | self.at(key) = s) in
  post: result = keys->collect(key | self.at(key+1))
```

Kapitel 4

Probleme im Zusammenhang mit OCL

Im Folgenden sollen Einschränkungen, Probleme und Fehler betrachtet werden, die im Zusammenhang mit **OCL** bei der Formulierung von Forderungen für den **fun** eMail-Client auftreten sind.

4.1 Ausdrucksfähigkeit von OCL

Die Ausdrucksfähigkeit von **OCL** ist in mancher Hinsicht eingeschränkt. [MC99] zeigt zum Beispiel Einschränkungen in der Navigierbarkeit und Berechenbarkeit auf. Bei der Formalisierung der Forderungen an *keyMail/S* fiel jedoch hauptsächlich die Einschränkung bei der Formulierung zeitlicher Bezüge negativ auf. Zur Spezifizierung zeitlicher Abläufe mußte daher auf Sequenzendiagramme zurückgegriffen werden.

Für **OCL** ist desweiteren gefordert, daß nur Methoden benutzt werden, die den Zustand des Systems nicht verändern (Seiteneffektfreiheit). Dies führte zu Erweiterungen von **OCL**, wie zum Beispiel dem *new()*-Konstrukt.

4.2 Das *new()*-Konstrukt

Im Laufe des Entwurfs des **fun** eMail-Clients wurden Forderungen in einer von **OCL** abgeleiteten Notation formuliert. Diese Notation wurde im Zusammenhang mit dem Catalysis Prozeß [DW98] definiert.

Eine der offenkundigsten Änderungen in der Catalysis Notation ist die Einführung eines *new()*-Konstruktes. Dieses erlaubt das Anlegen neuer Instanzen in Constraints. Damit wird jedoch die Seiteneffektfreiheit von **OCL** beeinträchtigt. Das *new()*-Konstrukt wird üblicherweise in Nachbedingungen benötigt, um auszudrücken, daß eine neue Instanz angelegt und einem Attribut zugewiesen wurde.

```
aAttribute = new(aClass)
```

Was man aber eigentlich in einer Nachbedingung ausdrücken will ist, daß eine neue Instanz im Verlauf der Abarbeitung einer Methode erzeugt wurde. Das heißt, in der Nachbedingung wird keine Instanz erzeugt, sondern es wird nur gefordert, daß die Instanz vor Aufruf der Methode noch nicht existierte. Dies kann aber auch ohne das *new()*-Konstrukt ausgedrückt werden. Der folgende Ausdruck in einer Nachbedingung liefert alle im Laufe der Ausführung der Methode erzeugten und nicht wieder gelöschten Instanzen einer Klasse **aClass**.

```
aClass.allInstances->excludesAll(aClass.allInstances@pre)
```

Somit besagt der folgende Ausdruck, daß einem Attribut eine neu erzeugte Instanz zugewiesen wurde.

```
let newInstances =
  aClass.allInstances->excludesAll(aClass.allInstances@pre) in
newInstances->includes(aAttribute)
```

4.3 Eindeutige Elemente in Mengen

Mit *select()* kann man aus einer **Collection** Elemente auswählen, dazu wird ein **OCL**-Ausdruck übergeben. Das Resultat ist eine Collection, die alle Elemente enthält, auf denen der Ausdruck zu **true** evaluiert.

Ist nun davon auszugehen, daß genau für ein Element der Ausdruck zu **true** evaluiert, so besteht die Gefahr, daß als Ergebnis nicht eine Collection, die dieses Element enthält, erwartet wird, sondern das Element selbst.

```
-- -- nicht eindeutiges select
-- -- Ergebnis klar!
Set{1 .. 10}.select( i | 3 < i < 6) = Set{4, 5}
-- -- eindeutiges select
Set{1 .. 10}.select( i | i = 4) = Set{4}
-- -- häufiger Fehler!
Set{1 .. 10}.select( i | i = 4) = 4
```

Auf dieses eindeutige Element sollen meist weitere Methoden zur Anwendung kommen. In einem solchen Fall muß also vorausgesetzt werden, daß mit *select()* genau ein Element selektiert wird. Dieses Element kann dann aus dem Ergebnis extrahiert werden, um anschließend weitere Methoden darauf anzuwenden.

```
-- -- select() ausführen
let selected = aCollection->select(expr) in
-- -- Es wurde genau ein Element selektiert
selected->size = 1
-- -- extrahiere das Element
let elem = selected->asSequence->at(1) in
-- -- nun kann mit diesem Element gearbeitet werden
elem.aMethod
```

Die Eindeutigkeit und Existenz des Elements ist oft schon durch andere Constraints (z.B. Invarianten) sichergestellt. Ist dies der Fall, reduziert sich der benötigte Ausdruck auf:

```
aCollection->select(expr)->asSequence->at(1).aMethod
```

4.4 *forAll* mit mehreren Iteratoren

Bei dem *forAll* Operator auf Collections ist auch mehr als eine Iteratorvariable erlaubt. Dies entspricht einer Schachtelung von *forAll* Operatoren. Somit sind folgende **OCL**-Ausdrücke äquivalent.

```
aCollection->forAll(x, y | expression)

aCollection->forAll(x |
  aCollection->forAll(y | expression))
```

Unglücklicherweise wird jedoch oft angenommen, daß bei einem *forAll* mit mehreren Iteratoren, diese nur unterschiedliche Werte annehmen. Dies würde jedoch erst mit folgendem Ausdruck garantiert:

```
aCollection->forAll(x, y | (x <> y) implies expression)
```

Sogar in der Definition von **OCL** wird eine solche Formulierung verwendet, um zu fordern, daß die Ergebnisse der Anwendung eines Ausdrucks auf alle Elemente einer Collection paarweise verschieden sind (siehe „Predefined OCL Types in“ [SMHP+99], Kapitel 7) :

```
aCollection->isUnique(expr : OclExpression) : Boolean post:
  result = aCollection->collect(expr)->forAll(e1, e2 | e1 <> e2)
```

In dieser Form wird allerdings stets **false** zurückgegeben. Hier kann jedoch die obige Lösung nicht direkt verwendet werden, denn dann würde der Ausdruck in trivialer Weise wahr. Das Problem besteht in der Unterscheidung zwischen dem Vorkommen eines Objektes in einer **Collection** und dem Objekt selbst. Für *isUnique()* soll eigentlich gelten, daß es kein Objekt mit mehreren Vorkommen in der *Collection* gibt. Eine solche Aussage ist in direkter Weise nur für **Sequence** möglich, da in diesem Fall die einzelnen Vorkommen über ihre Stellung in der **Sequence** identifiziert werden können.

Um die beabsichtigte Aussage für *isUnique()* zu formulieren, kann folgender Ausdruck verwandt werden. Er sammelt die Vorkommen der enthaltenen Objekte, um anschließend eine Aussage über deren Anzahl zu treffen.

```
aCollection->isUnique(expr : OclExpression) : Boolean post:
  let res = aCollection->collect(expr) in
    result = res->forAll(e | res->count(e) = 1)
```

Kapitel 5

Statische Forderungen

5.1 Singleton

Das Singleton Muster legt fest, daß es zu einer Klasse nur eine einzige Instanz gibt. Es ist besonders einfach und soll daher als erstes Muster betrachtet werden.

5.1.1 Beschreibung

Um sicherzustellen, daß nur eine Instanz einer Klasse erzeugt werden kann, muß direkt auf die Instanziierungsmethode Einfluß genommen werden. Instanziierungsmethoden werden in **UML** üblicherweise mit *create()* bezeichnet, sind Klassenmethoden und geben ein Objekt der instanziierten Klasse zurück.

Um die (einzige) Instanz der Klasse zu verwalten, wird ein Klassenattribut eingeführt das auf eine eventuell bereits erzeugte Instanz verweist. Versucht man nun eine Instanz zu erzeugen, wird zuerst geprüft, ob bereits eine Instanz existiert. Ist dies der Fall, so wird die bereits in dem Klassenattribut vermerkte Instanz zurückgegeben, andernfalls wird eine Instanz erzeugt und entsprechend in dem Klassenattribut vermerkt. Abbildung 5.1 zeigt das zugehörende Klassendiagramm.

Alternativ könnten auch alle Methoden und Attribute der Klasse als Klassenmethoden bzw. -attribute definiert werden. Zusätzlich verbietet man noch die Instanziierung generell. Damit ist die Klasse gleichzeitig auch Repräsentant der einzigen Instanz.

Diese Alternative ist jedoch weniger flexibel, zum Beispiel erlaubt es das Arbeiten mit echten Instanzen, das Muster dahingehend anzupassen, daß eine begrenzte Anzahl von Instanzen erlaubt ist. Außerdem besteht beim Arbeiten mit echten Instanzen noch immer die Möglichkeit, die Instanz zu erzeugen und zu löschen. Eine ausführliche Beschreibung des Musters findet sich in [GHJV95].

5.1.2 Anforderungen

Die zentrale Forderung des Singleton Musters ist, daß zur Laufzeit maximal eine Instanz der mit dem Singleton Muster verbundenen Klasse existiert. Eine Abwandlung des Singleton

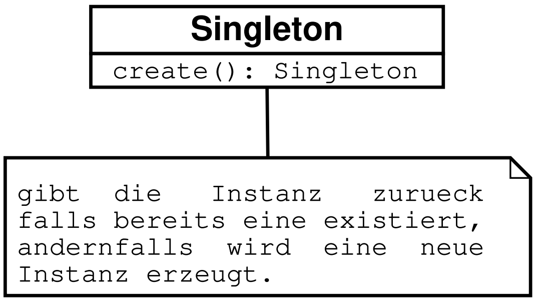

Abbildung 5.1: Struktur des Singleton Musters

Musters beschränkt die Maximalzahl der erzeugten Instanzen.

Forderung	Constraint
max. eine Instanz existiert	5.1.1
max. MAX Instanzen existieren	5.1.3

5.1.3 Formalisierung

Einschränkung auf maximal eine Instanz:

CONSTRAINT 5.1.1 (MAX. EINE INSTANZ EXISTIERT)

```
ocltemplate Singleton {} ocl:
  context Class inv:
    -- -- Zu jedem Zeitpunkt existiert maximal eine Instanz
    Class.allInstances->size <= 1
```

Der folgende Ausdruck beschreibt zusätzlich, wie die Instanziierungsmethode arbeitet.

CONSTRAINT 5.1.2 (CREATE METHODE FÜR SINGLETON)

```
ocltemplate Singleton_create {} ocl:
  context Class::create() post:
    -- -- falls bereits eine Instanz existierte,
    -- -- bleibt diese erhalten
    Class.allInstances@pre->forAll(e |
      Class.allInstances->includes(e)) and
    -- -- nach einem create existiert genau eine Instanz
    Class.allInstances->size = 1 and
    -- -- diese Instanz wird zurückgeliefert
    result = allInstances->asSequence->first
```

Soll das Muster dahingehend variiert werden, daß eine begrenzte Anzahl von Instanzen erlaubt ist, so lassen sich die obigen Ausdrücke in einfacher Weise anpassen.

Constraint 5.1.3 (Maximal MAX Instanzen existieren)

```
ocltemplate Singleton_max {MAX} ocl:
  context Class inv:
    -- -- Zu jedem Zeitpunkt existieren maximal MAX Instanz
    Class.allInstances->size <= MAX
```

Constraint 5.1.4 (Create Methode für Singleton_max)

```
ocltemplate Singleton_max_create {MAX} ocl:
  context Class::create() post:
    -- -- die bereits existierenden eine Instanz bleibt erhalten
    Class.allInstances@pre->forAll(e |
      Class.allInstances->includes(e)) and
    if Class.allInstances@pre->size < MAX then
      -- -- existieren weniger als MAX Instanzen,
      -- -- so wird durch create eine neue Instanz erzeugt
      Class.allInstances->size = Class.allInstances@pre->size + 1
      and
      -- -- wurde eine neue Instanz erzeugt,
      -- -- wird diese zurückgeliefert
      result =
        (allInstances - allInstances@pre)->asSequence->first
    endif
    -- -- sind bereits MAX Instanzen erzeugt, ist nicht eindeutig,
    -- -- welche Instanz zurückgegeben werden soll. Dies muß für
    -- -- die konkrete Anwendung entschieden werden.
```

5.1.4 Anwendung

Im **fun**-eMail-Client findet das Singleton Muster beim Adreßbuch Anwendung. Das Adreßbuch verwaltet die Kontakte des Benutzers. Um die Gewährleistung der Konsistenz der Kontakte zu vereinfachen, darf es nur ein Adreßbuch-Objekt geben. Im folgenden Constraint wird hierzu das Singleton Muster für das Adreßbuch instanziiert.

Die folgende Tabelle gibt die Abbildung des abstrakten Modells in das Projektmodell wieder:

abstraktes Modell	→	konkretes Modell
Singleton		**AddressBook**
Singleton.create()		*AddressBook.new()*

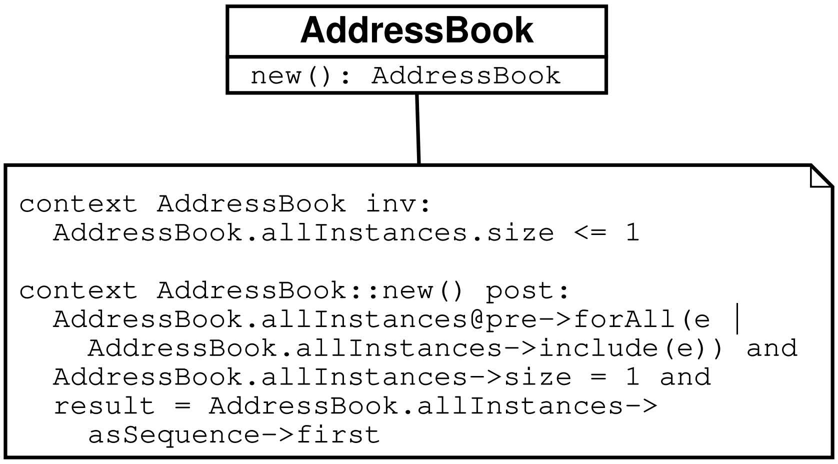

Abbildung 5.2: Anwendung des Singleton Musters auf *keyMail/S*

Mit dieser Abbildung ergeben sich folgende Constraints für **AddressBook**. Abbildung 5.2 zeigt das resultierende Klassendiagramm.

Constraint 5.1.5 (Adressbuch ist Instanz von Singleton)

```
context AddressBook inv:
  AddressBook.allInstances.size <= 1
```

Constraint 5.1.6 (*create* Methode für Adressbuch)

```
context AddressBook::new() post:
  -- -- falls bereits eine Instanz existierte,
  -- -- bleibt diese erhalten
  AddressBook.allInstances@pre->forAll(e |
    AddressBook.allInstances->includes(e)) and
  -- -- nach einem create existiert genau eine Instanz
  AddressBook.allInstances->size = 1 and
  -- -- diese Instanz wird zurückgeliefert
  result = AddressBook.allInstances->asSequence->first
```

5.2 Composite

Das Composite-Muster dient dazu, Objekte in einer Graphen-Strukturen zu organisieren (Verzeichnisse, GUI-Systeme, ...).

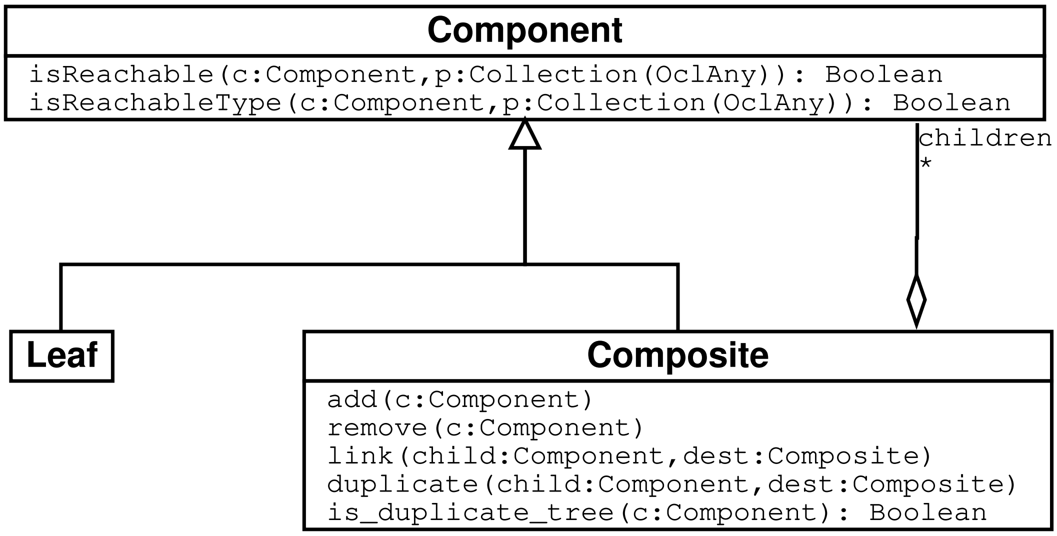

Abbildung 5.3: Struktur des Composite Musters

5.2.1 Beschreibung

Das Composite-Muster beschreibt einen gerichteten Graphen. Dazu unterscheidet es zwischen zwei Klassen von Objekten. Objekte der Klasse **Leaf** repräsentieren Endpunkte im Graphen (Blätter), Objekte der Klasse **Composite** repräsentieren innere Knoten des Graphen. Von ihnen können gerichtete Kanten auf Objekte der Klasse **Component** verweisen (*children* Assoziation). Die Klasse **Component** ist die gemeinsame Oberklasse von **Leaf** und **Composite**. Sie stellt ein gemeinsames Interface zur Verfügung und erlaubt damit Kundenklassen, ohne Rücksicht auf die konkrete Klasse, auf Objekte innerhalb der Graphenstruktur zuzugreifen. Abbildung 5.3 zeigt das entsprechende Klassendiagramm. Eine ausführliche Beschreibung des Musters findet sich in [GHJV95].

5.2.2 Anforderungen

5.2.2.1 Mehrfachkanten

Innerhalb des Composite Musters werden zwei Methoden definiert (*add()* und *remove()*). Die Funktion dieser Methoden ist intuitiv klar, aber nicht eindeutig. Es ist z.B. nicht festgelegt, was bei mehrfachem Einfügen eines Objektes passieren soll.

So kann man fordern, daß es nur eine Kante zwischen zwei Komponenten geben darf, d.h., ein nochmaliges Einfügen eines Objektes ändert den Graphen nicht mehr. Läßt man mehrfache Kanten zu, so kann man bei einem *remove()* fordern, daß alle Kanten zwischen den betroffenen Objekten entfernt werden oder nur eine.

5.2.2.2 Einschränkung der Kanten

Bei der Anwendung des Composite-Musters ist es möglich mehrere Unterklassen sowohl von **Composite**, als auch von **Leaf** zu definieren. In diesem Fall kann es notwendig sein zu fordern, daß nicht zwischen beliebigen Klassen Kanten gezogen werden können. Man braucht also eine Möglichkeit, die Kanten einzuschränken.

5.2.2.3 Erreichbarkeit

Wie bei der Frage nach Zyklen, stellt sich häufig die Frage nach der Erreichbarkeit von Objekten. Dazu wird geprüft, ob ein Pfad von einem gegebenen Objekt zu dem gesuchten Objekt existiert.

5.2.2.4 Zyklen

Eine häufig benötigte Forderung an Graphen ist die Zyklenfreiheit. Bei der Definition von Zyklen können zwei grundlegende Typen von Zyklen unterschieden werden. Objektzyklen entstehen, wenn ein Pfad von einem Objekt auf sich selbst existiert. Typzyklen enstehen bereits, wenn ein Pfad von einem Objekt zu einem Objekt gleichen Typs existiert.

5.2.2.5 Kopieren einer Komponente

Ein Element zu kopieren, hat in einem Graphen mehrere Bedeutungen. Die einfachste Form ist, daß das Element einfach zusätzlich in die *children* Liste der Zielkomponente eingefügt wird. Dadurch verweisen mehrere Elternkomponenten auf dieselbe Kindkomponente. Soll dies vermieden werden, muß eine Kopie des Elements erzeugt werden. Konsequenterweise müssen aber auch von den Kindern und Kindeskindern des Elements rekursive Kopien erzeugt werden. Um sicherzustellen, daß dieser Prozeß terminiert, darf es keinen Zyklus geben, der sowohl das zu kopierende Element, als auch seine Elternkomponente enthält.

Wird von dem Graph verlangt, daß er zyklenfrei ist, so können Probleme entstehen, falls das Kopieren durch einfaches Hinzufügen einer Kante geschieht. Um dies zu vermeiden, muß in einer entsprechenden Vorbedingung geprüft werden, ob das Anlegen einer neuen Kante ein Zyklus erzeugt.

Forderung	Constraint
Mehrfachkanten	5.2.1 – 5.2.6
Einschränkung der Kanten	5.2.7
Erreichbarkeit	5.2.8, 5.2.10
Zyklenfreiheit	5.2.12, 5.2.13
Kopieren einer Komponente	5.2.14, 5.2.15

5.2.3 Formalisierung

5.2.3.1 Mehrfachkanten

Durch einfache Nachbedingungen kann eindeutig formuliert werden, was beim Einfügen bzw. Entfernen von Kanten passieren soll. Um Mehrfachkanten auszuschließen muß jedoch sichergestellt werden, daß nur durch die *add()* Methode Kanten hinzugefügt werden.

Das Entstehen von Mehrfachkanten kann mit folgender Nachbedingung ausgeschlossen werden. Dabei muß jedoch die Möglichkeit eines Scheiterns des Einfügens berücksichtigt werden.

CONSTRAINT 5.2.1 (EINFÜGEN EINER KOMPONENTE)

```
ocltemplate Composite_Add {} ocl:
  context Composite::add(c : Component) : Boolean post:
    -- -- durch add() werden keine Mehrfachkanten erzeugt
    if (result = true ) then (self.children->count(c) = 1)
    else self.children->count(c) = self.children@pre->count(c)
    endif
```

Dieser Constraint wirkt sich jedoch nur auf das Einfügen mit *add()* aus. Sind Mehrfachkanten grundsätzlich verboten, so kann dies auch durch folgende Invariante ausgedrückt werden.

CONSTRAINT 5.2.2 (GLOBALES VERBOT VON MEHRFACHKANTEN)

```
ocltemplate Composite_Set {} ocl:
  context Composite inv:
    -- -- keine Mehrfachkanten
    self.children->forAll(c | self.children->count(c) = 1)
```

Sind Mehrfachkanten erlaubt, so garantiert die folgende Nachbedingung, daß nach dem Hinzufügen einer Kante die Anzahl der Kanten um 1 erhöht ist oder das Einfügen scheiterte.

CONSTRAINT 5.2.3 (EINFÜGEN EINER KOMPONENTE (MIT MEHRFACHKANTEN))

```
ocltemplate Composite_Add_multiple {} ocl:
  context Composite::add(c : Component) : Boolean post:
    if (result = true ) then
      (self.children->select(p | p = c)->size =
        self.children@pre->select(p | p = c)->size + 1)
    else
      (self.children->select(p | p = c)->size =
        self.children@pre->select(p | p = c)->size)
    endif
```

Eine einfache Kombination der beiden Nachbedingungen kann benutzt werden, um ein Maximum für die Anzahl der Kanten zu fordern. Dabei repräsentiert `MAX` das gesetzte Maximum.

CONSTRAINT 5.2.4 (EINFÜGEN MIT OBERGRENZE FÜR MEHRFACHKANTEN)

```
ocltemplate Composite_Add_max {MAX} ocl:
  context Composite::add(c : Component) : Booleanpost:
    -- -- mit Mehrfachkanten
    -- -- und maximaler Kantenzahl
    let pre_count : Integer = self.children@pre->select(
        p | p = c)->size in
    if (result = true ) then
      (self.children->select(p | p = c)->size =
        (pre_count + 1).min(MAX))
    else
      (self.children->select(p | p = c)->size =
        pre_count
    endif
```

Das Entfernen von Kanten ist eigentlich nur bei Mehrfachkanten mehrdeutig. Will man in diesem Fall fordern, daß nach einem *remove()* alle Kanten gelöscht sind, kann folgende Nachbedingung benutzt werden.

CONSTRAINT 5.2.5 (ENTFERNEN EINER KOMPONENTE (ALLE KANTEN))

```
ocltemplate Composite_Remove {} ocl:
  context Composite::remove(c : Component) post:
    -- -- Entfernen aller Kanten
    self.children->select(p | p = c)->size = 0
```

Soll jeweils nur eine Kante gelöscht werden, so kann man folgende Nachbedingung benutzen.

CONSTRAINT 5.2.6 (ENTFERNEN EINER KOMPONENTE (EINZELNE KANTE))

```
ocltemplate Composite_Remove_single {} ocl:
  context Composite::remove(c : Component) post:
    -- -- Entfernen einer Kante
    let pre_count : Integer = self.children@pre->select(
        p | p = c)->size in
      self.children->select(p | p = c)->size =
        (pre_count - 1).max(0)
```

Die obigen Constraints wurden nur für **Composite** formuliert, da auch nur in **Composite** die *children*-Assoziation existiert. Um ein gemeinsames Interface für Kundenklassen zur Verfügung zu stellen, können die Methoden jedoch bereits in **Component** definiert werden. Die Behandlung eines entsprechenden Aufrufs außerhalb einer **Composite** Klasse muß somit entsprechend abgefangen werden.

5.2.3.2 Einschränkung der Kanten

Die Einschränkung der Kanten (im Fall des eMail-Clients z.B. daß Gruppen keine Folder enthalten dürfen) kann sowohl positiv, als auch negativ formuliert werden. Im ersten Fall wird eine Menge von Klassen angegeben, zu denen Kanten gezogen werden dürfen. Im zweiten Fall wird die Menge der Klassen angegeben, zu denen keine Kanten gezogen werden dürfen.

Um dies auszudrücken, kann die Einschränkung von Elementen in Collections, wie sie im `Collection_restriction` Template definiert ist (siehe Abschnitt 7.2.9), verwendet werden.

Für das folgende Beispiel gelte: **CompositeA** ist abgeleitet von **Composite**; **ComponentA**ist abgeleitet von **Component**.

Wird gefordert, daß Kanten zwischen Objekten vom Typ **CompositeA** und solchen vom Typ **ComponentA** ausgeschlossen sind so benutzt man folgenden Ausdruck:

```
CONSTRAINT 5.2.7 (BEISPIEL: EINSCHRÄNKEN VON KANTEN)
       Collection_restriction.instanciate( Map{
         Class : CompositeA, coll : children,
         expression : 'c | c.oclIsKindOf(ComponentA)',
         flavor : 'negative' }
       )
```

Will man fordern, daß von Objekten vom Typ **CompositeA** nur Kanten zu Objekten vom Typ **ComponentA** existieren dürfen, so muß `flavor` entsprechend auf `positive` abgebildet werden.

5.2.3.3 Erreichbarkeit

Die Frage der Erreichbarkeit eines Objektes von einem Startobjekt aus läßt sich mit Hilfe einer Tiefensuche beantworten. Um dabei nicht in einem etwaigen Zyklus zu geraten, muß der zurückgelegte Pfad während der Suche mitgeführt werden.

Für die Suche nach einem spezifischen Objekt bietet sich somit der folgende **OCL**-Ausdruck an:

Constraint 5.2.8 (Erreichbarkeit(Composite))

```
ocltemplate Composite_isReachable {}
  context Composite::isReachable
      (obj: Component, path: Collection(Component)) : Boolean post:
    -- -- prüft, ob obj erreichbar ist
    result =
      if (path->includes(self))
      then false
      else
        children->exists(c | c = obj) or
        children->exists(c |
          c.isReachable(obj,path->union(Set{self})))
      endif
```

Constraint 5.2.9 (Erreichbarkeit (Leaf))

```
ocltemplate Leaf_isReachable {}
  Leaf::isReachable
      (obj: Component, path: Collection(Component)) : Boolean post:
    -- -- wird benötigt, um den Test auf Erreichbarkeit
    -- -- in den Blättern ordentlich Terminieren zu lassen
    result = false
```

Will man prüfen, ob ein Objekt einer spezifischen Klasse erreichbar ist, muß der Ausdruck wie folgt angepaßt werden.

Constraint 5.2.10 (Erreichbarkeit (Composite, für Klassen))

```
ocltemplate Composite_isReachable_type {}
  context Composite::isReachableType
      (type: OclType, path: Collection(Component)) : Boolean post:
    -- -- prüft, ob ein Objekt vom Typ type erreichbar ist
    result =
      if (path->includes(self))
      then false
      else
        children->exists(c | c.oclIsKindOf(type)) or
        children->exists(c |
          c.isReachableType(type,path->union(Set{self})))
      endif
```

Constraint 5.2.11 (Erreichbarkeit (Leaf, für Klassen))

```
ocltemplate Leaf_isReachable_type {}
  context Leaf::isReachableType
     (type: OclType, path: Collection(Component)) : Boolean post:
    -- -- wird benötigt, um den Test auf Erreichbarkeit
    -- -- in den Blättern ordentlich Terminieren zu lassen
    result = false
```

5.2.3.4 Zyklen

Die Forderung nach Zyklenfreiheit läßt sich mit Hilfe der bereits diskutierten Erreichbarkeit einfach formalisieren. Dafür muß nur gefordert werden, daß für kein Objekt innerhalb des Graphen ein Pfad zu sich selbst existiert.

Constraint 5.2.12 (Zyklenfreiheit (auf Objektebene))

```
ocltemplate Composite_cycle_free {}
  context Composite inv:
    -- -- enthält keine Zyklen (auf Objektebene)
    not self.isReachable(self, Collection {})
```

Um Zyklenfreiheit auf Typebene zu fordern, darf entsprechend kein Pfad zu einem Objekt gleichen Typs existieren.

Constraint 5.2.13 (Zyklenfreiheit (auf Typebene))

```
ocltemplate Composite_type_cycle_free {}
  context Composite inv:
    -- -- enthält keine Zyklen (auf Objektebene)
    not self.isReachableType(self.oclType, Collection {})
```

Die Beantwortung der Frage, was ein „Objekt gleichen Typs“ ist, wird dabei auf die Definition der Erreichbarkeit übertragen.

5.2.3.5 Kopieren einer Komponente

Ist ein einfaches Hinzufügen einer Kante hinreichend, so kann folgendes Template benutzt werden:

Constraint 5.2.14 (Komponente kopieren)

ocltemplate `Composite_simple_copy` **ocl:**
```
context Composite::link (child : Component, dest : Composite)
  pre:  self.children->includes(child)
  post: dest.children->includes(child)
```

Ist das einfache Einfügen einer zusätzlichen Kante nicht hinreichend, so muß ein Duplikat des zu kopierenden Kindes erstellt werden. Dies ist vor allem dann der Fall, wenn der Graph ein Baum ist. In diesem Fall könnte ein einfaches Hinzufügen einer Kante die Baumeigenschaft zerstören. Allerdings ist im Falle eines Baumes das Duplizieren eines Teilbaumes verhältnismäßig einfach, da keine Zyklen auftreten.

Constraint 5.2.15 (Duplizieren einer Komponente)

ocltemplate `Composite_duplicate_tree` {} **ocl:**
```
context Composite::duplicate (child : Component, dest : Composite)
  pre:  self.children->includes(child)
  post: dest.children->exists(c | c.isDuplicateTree(child))
```

Im obigen Constraint wurde die Methode *isDuplicateTree()* verwendet. Diese Methode vergleicht zwei Bäume. Der folgende Constraint formalisiert die Forderung, die an einen solchen Vergleich gestellt werden muß. Dieser Constraint benutzt wiederum *isDuplicate()*. Somit muß Constraint 7.1.3 (siehe Seite 56) für **Component** instanziiert werden.

Constraint 5.2.16 (Gleichheit auf Komponenten)

ocltemplate `Component_is_duplicate_tree` {} **ocl:**
```
context Composite::isDuplicateTree (comp : Component) : Boolean
post:
  result = self.isDuplicate(comp) and
    (self.oclIsKindOf(Composite) and comp.oclIsKindOf(Composite))
      implies (
        self.children->forAll(x | comp.children->exists(y |
          x.isDuplicateTree(y)))
        and
        comp.children->forAll(x | self.children->exists(y |
          x.isDuplicateTree(y)))
      )
```

5.2.4 Anwendung

Um das Muster auf eine konkrete Situation anzuwenden, muß als erstes eine Abbildung der konkreten Klassen auf die drei Klassen des Musters (**Component**, **Composite** und **Leaf**)

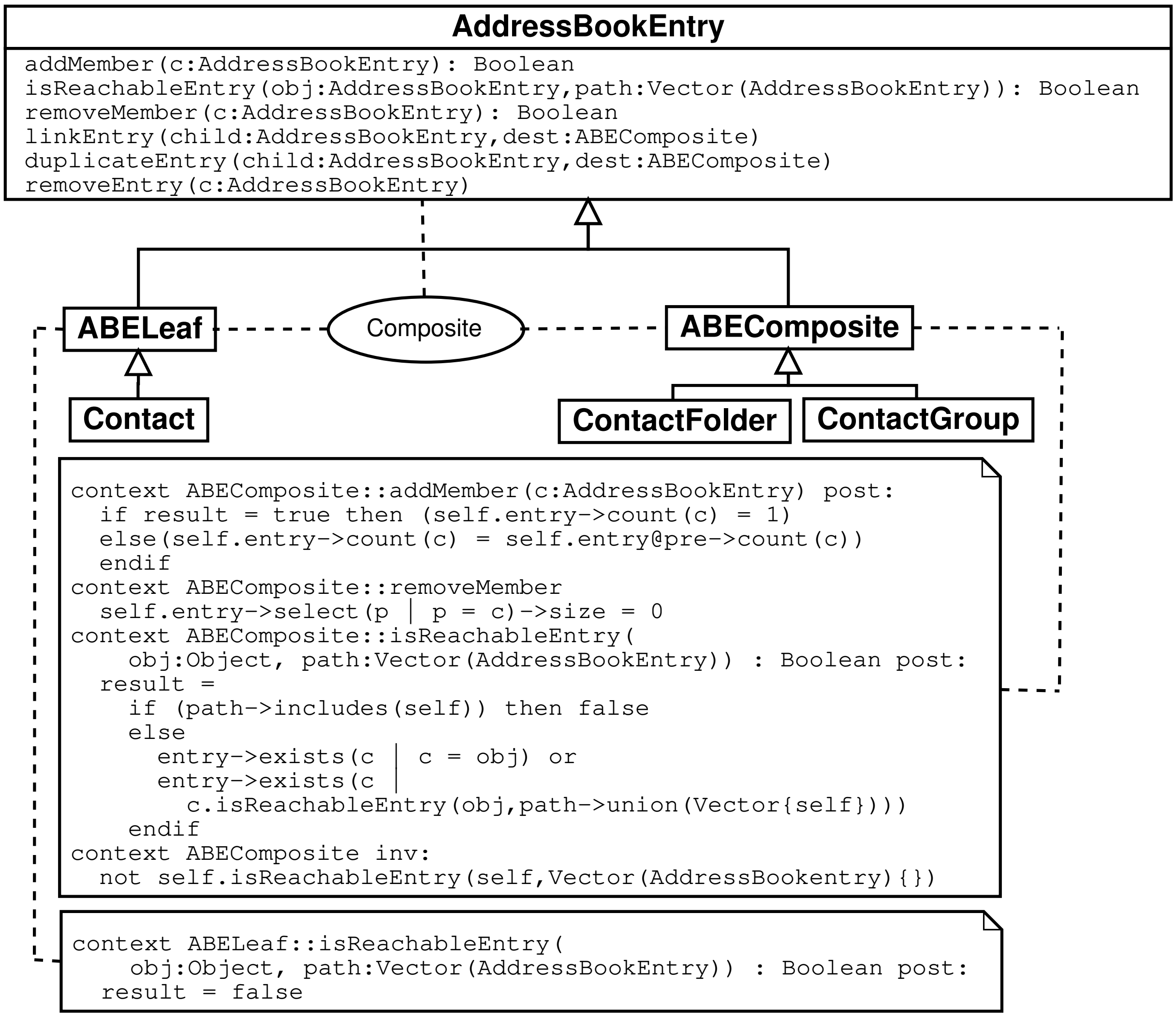

Abbildung 5.4: Anwendung des Composite Musters auf *keyMail/S*

erfolgen. Falls mehrere Klassen auf **Composite** bzw. **Leaf** abgebildet werden, ist zudem zu entscheiden, ob diese jeweils direkt als Kinder von **Component** definiert werden, oder ob sie von einer entsprechenden Oberklasse erben sollen.

Führt man eine Oberklasse für **Component**-Klassen ein, so können die entsprechenden Constraints mit dieser Oberklasse verknüpft werden. Andernfalls müssen die Constraints entsprechend angepaßt und mit allen **Component**-Klassen verknüpft werden.

Anschließend werden die *add()*- und *remove()*-Methoden identifiziert. Wurde für die *children*-Assoziation ein anderer Rollenname vergeben, muß dieser in den Constraints entsprechend angepaßt werden. Dies könnte bei entsprechender Tool-Unterstützung auch automatisch geschehen.

Im folgenden werden nun die obigen Constraints konkretisiert, um die Forderungen die an das Adreßbuch gestellt werden zu formulieren (siehe Abbildung 5.4). Die folgende Tabelle gibt die Abbildung des abstrakten Modells in das Projektmodell wieder:

Muster	→	eMail-Client
Component		**AddressBookEntry**
Component.isReachable(*obj : Component,* *path : Collection(OclAny))*		*isReachableEntry(* *obj : AddressBookEntry,* *path : Vector(AddressBookEntry))*
Component.isReachableType(*type : OclType,* *path : Collection(OclAny))*		—
Composite		**ABEComposite**
Composite.add(*c : Component) : Boolean*		*ABEComposite.addMember(* *c : AddressBookEntry) : Boolean*
Composite.link(*child : Component,* *dest : Composite,*		*ABEComposite.linkEntry(* *child : AddressBookEntry,* *dest : ABEComposite)*
Composite.duplicate(*child : Component,* *dest : Composite,*		*ABEComposite.duplicateEntry(* *child : AddressBookEntry,* *dest : ABEComposite)*
Composite.remove(c : Component)		*ABEComposite.removeMember(* *c : AddressBookEntry)*
Leaf		**ABELeaf**

5.2.4.1 Mehrfachkanten

In einem Folder bzw. einer Gruppe sollen Kontakte nur einfach vorkommen. Daher werden die *Add()*- und *Remove()*-Methoden entsprechend definiert. Aus Constraint 5.2.1 und 5.2.5 ergeben sich somit folgende Constraints:

CONSTRAINT 5.2.17 (*add()* FÜR **AddressBook**)

```
context AddressBookEntry::addMember(c : Component) post:
   -- -- keine Mehrfachkanten
   if (result = true ) then (self.entry->count(c) = 1)
   else self.entry->count(c) = self.entry@pre->count(c)
   endif
```

Constraint 5.2.18 (*remove()* für **AddressBook**)

```
context AddressBookEntry::removeMember(c : Component) post:
   -- -- Entfernen der Kante
   self.Entry->select(p | p = c)->size = 0
```

5.2.4.2 Zyklen

Für das Adreßbuch wird gefordert, daß es keine Objektzyklen enthält. Um dies zu formulieren werden die Constraints 5.2.8, 5.2.9 und 5.2.12 instanziiert. Daraus ergeben sich folgende Constraints für *keyMail/S*:

Constraint 5.2.19 (*isReachable()* für **ABEComposite**)

```
context ABEComposite::isReachableEntry
     (obj: AddressBookEntry, path : Vector(AddressBookEntry))
     : Boolean post:
   -- -- prüft, ob obj erreichbar ist
   result =
     if (path->includes(self))
     then false
     else Entry->exists(c | c = obj) or
          Entry->exists(c |
             c.isReachableEntry(obj,path->union(Set{self})))
     endif
```

Constraint 5.2.20 (*isReachable()* für **ABELeaf**)

```
context ABELeaf::isReachableEntry
     (obj: AddressBookEntry, path : Vector(AddressBookEntry))
     : Boolean post:
   -- -- wird benötigt, um den Test auf Erreichbarkeit
   -- -- in den Blättern ordentlich Terminieren zu lassen
   result = false
```

Constraint 5.2.21 (Zyklenfreiheit für das Adressbuch)

```
context ABEComposite inv:
   -- -- enthält keine Zyklen (auf Objektebene)
   not  self.isReachableEntry(self, Vector(AddressBookEntry) {})
```

5.2.4.3 Einschränkung der Kanten

Es wird gefordert, daß Gruppen kein Folder enthalten dürfen, ansonsten sind alle Kanten erlaubt. Somit ist eine negative Formulierung der Einschränkung angebracht.

Constraint 5.2.22 (**Group** enthält keine **Folder**)

```
context Group inv:
  let conform = entry->collect(c | c.oclIsKindOf(Folder))->asSet in
    conform = Set{false }
```

Kapitel 6

Dynamische Forderungen

6.1 Serializer

Das Serializer Muster beschreibt ein Verfahren um Objekt-Strukturen zu linearisieren, um sie z.B. in eine Datei zu schreiben und anschließend wieder aus der linearisierten Version rekonstruieren zu können.

6.1.1 Beschreibung

Ein zentraler Aspekt des Musters ist die Entkoppelung der zu linearisierenden Objekte von den Gegebenheiten des externen Speichers. Die zu linearisierenden Objekte sollen kein Wissen über die Art des Speichers benötigen, aber auch der Speicher soll kein Wissen über die Struktur der zu linearisierenden Objekte benötigen. Dies ermöglicht einen Wechsel des Speichertyps, ebenso wie eine Änderung der zu linearisierenden Struktur in einfacher Weise.

Um dies zu erreichen, werden **Reader** und **Writer** Klassen definiert, die das Wissen wie primitive Typen sowie Objektreferenzen linearisiert werden beherbergen. Dabei gelten alle Klassen, die nicht von **Serializable** abgeleitet sind, als primitiv.

Eine **Serializable** Klasse stellt die Methoden *readFrom()* und *writeTo()* zur Verfügung. Sie erhalten ein **Reader** bzw. **Writer** Objekt, welches das Ziel der Linearisierung repräsentiert.

Zu serialisierende Klassen erben von **Serializable** und überschreiben obige Methoden. In diesen Methoden ist das Wissen über die Struktur der Klasse repräsentiert. In ihnen werden alle persistenten Attribute des Objektes an den **Writer** geschickt, bzw. vom **Reader** erfragt.

Der **Reader** stellt für jeden primitiven Typ T eine Methode *readT()* zur Verfügung, die das nächste Objekt vom Typ T zurück gibt. Zusätzlich wird die Methode *readObject()* definiert, mit deren Hilfe Objekte vom Typ **Serializable** gelesen werden. Dazu wird ein neues Objekt erzeugt und anschließend bei diesem die *readFrom()*-Methode aufgerufen, um das Objekt korrekt zu initialisieren.

Der **Writer** stellt die inversen Methoden des **Reader** zur Verfügung. Für jeden primitiver Typ T gibt es eine Methode *writeT()*, welche ein Objekt vom Typ T linearisiert und

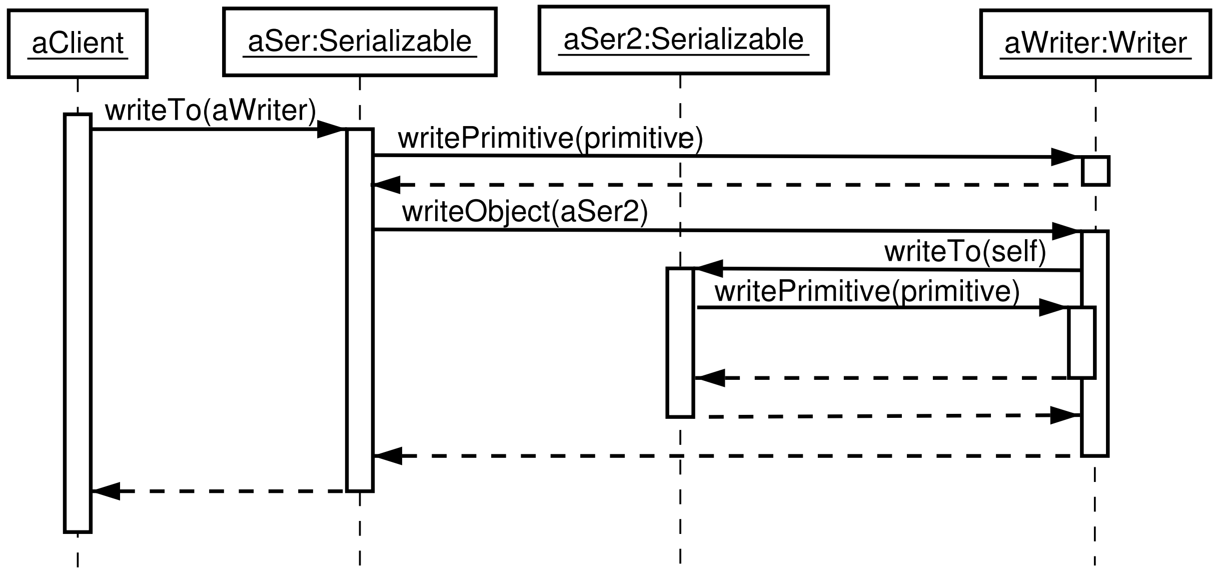

Abbildung 6.1: Das Serializer Muster: Ablauf einer Linearisierung

dem Stream anfügt. Die Methode *writeObject()* dient zur Linearisierung von **Serializable** Objekten. Sie benutzt entsprechend die *writeTo()*-Methode um das **Serializable** Objekt zu veranlassen die Linearisierung zu übernehmen.

In Abbildung 6.1 ist ein exemplarischer Ablauf für die Linearisierung einer einfachen Struktur gegeben.

Im Serializer-Muster wird ursprünglich keine explizite Verbindung zwischen **Reader** und **Writer** hergestellt. Diese Verbindung wird jedoch benötigt, um festzustellen welcher **Writer** zu einem konkreten **Reader** gehört. Daher wird eine **Store** Klasse eingeführt. Diese enthält je eine Referenz auf einen **Reader** und einen **Writer** und stellt damit die benötigte Verbindung her. Abbildung 6.2 zeigt ein Klassendiagramm für die grundlegende Struktur des Serializer Musters. Eine ausführliche Beschreibung des Musters findet sich in [MRB98].

6.1.2 Anforderungen

Die zentrale Forderung an einen Serializer ist, daß sich die Lese- und Schreiboperation invers zueinander verhalten. Wird eine Objekt-Struktur linearisiert und aus dieser linearisierten Version wieder eine Objekt-Struktur gewonnen, so soll diese mit dem Original übereinstimmen.

Um diese Forderung zu formulieren brauchen wir eine Methode die feststellt, ob zwei Objekt-Strukturen „übereinstimmen“. Da bei der Restaurierung der Objekt-Struktur neue Objekte erzeugt werden, ist die Prüfung der Identität von Objekten nicht adäquat, sondern der Inhalt der Objekte muß verglichen werden. Entsprechend stimmen Objektreferenzen dann überein, wenn die referenzierten Objekte übereinstimmen.

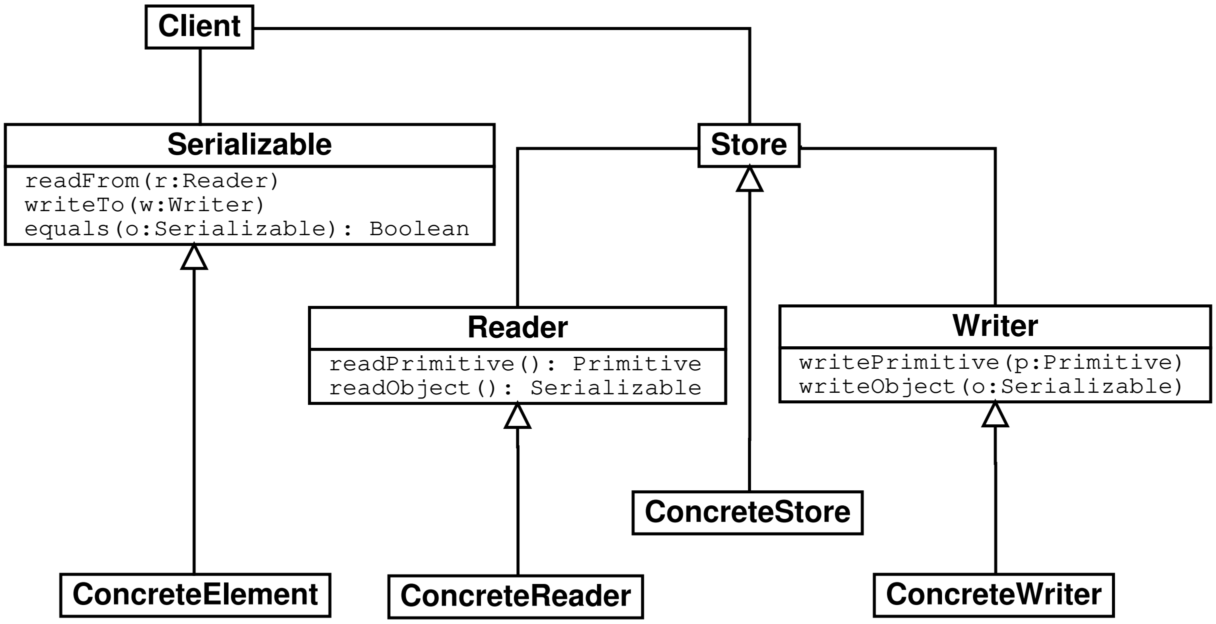

Abbildung 6.2: Das Serializer Muster

In diesem Zusammenhang ist auch die Unterscheidung zwischen transienten und persistenten Attributen wichtig. Da nur persistente Attribute bei der Linearisierung berücksichtigt werden, können auch nur solche in den Vergleich der Objekt-Strukturen eingehen. Damit hängt die Definition der „Gleichheit“ unmittelbar mit dem Umfang der Linearisierung zusammen. Persistente Attribute können weiter danach unterschieden werden, ob sie von **Serializable** abgeleitet sind. Persistente Attribute, die nicht von **Serializable** abgeleitet sind, werden vom Serializer gesondert behandelt, da diese nicht in der Lage sind ihre Linearisierung selbst zu steuern. Im weiteren Verlauf werden solche Typen als primitiv bezeichnet.

Die Entscheidung, was primitive Typen sind, ist abhängig von dem konkreten Projekt. Für Typen, die in der Zielsprache bereits als primitive Typen implementiert sind, ist die Zuordnung klar, doch bereits Strings sind üblicherweise als Klassen implementiert und werden somit über Objektreferenzen referenziert.

Entsprechend dem Muster müssen **Reader** und **Writer** primitive Typen ohne weitere Hilfe linearisieren können und somit deren Struktur kennen. Nicht-primitive Typen müssen im Gegensatz dazu das **Serializable** Interface implementieren, um linearisierbar zu werden.

Forderung	Constraint
read(), *write()* invers	6.1.3

6.1.3 Formalisierung

6.1.3.1 Charakteristik

Für die Prüfung der Gleichheit muß jedem Attribut und jeder Assoziation der zu serialisierenden Klassen eine eindeutige Charakteristik zugeordnet werden.

Constraint 6.1.1 (Charakteristik für Attribute und Assoziationen)

```
ocltemplate Serializable_characteristic {} ocl:
  context Serializable::characteristic(att : String) : String post:
    let attributes : Set{String} =
        Serializable.attributes->union(
        Serializable.associationEnds) in
      if (attributes->includes(att)) then
        Set{'primitive', 'persistent', 'transient'}
          ->includes(result)
      else result = ''
      endif
```

6.1.3.2 Gleichheit

Abhängig von der Charakteristik wird der Wert der einzelnen Attribute und Assoziationen direkt oder rekursiv verglichen.

Im Folgenden wird eine Schablone für einen entsprechenden Constraint gegeben. Dabei muß für jedes persistente Attribut und jede persistente Assoziation an entsprechender Stelle eine Zeile eingefügt werden.

Constraint 6.1.2 (Gleichheit)

```
ocltemplate Serializer_equals {} ocl:
  context Serializable::equals(obj : OclAny) post:
    -- -- Gleichheit mit dem übergebenen Objekt
    result =
      self.oclType = obj.oclType and
      let attributs : Set{String} = Serializable.attributes->union(
        Serializable->associationEnds)) in
      attributes.foreach ( att |
        case (Serializable.characteristic(att) = persistent)
          then self.att.equals(obj.att) endcase
        case (Serializable.characteristic(att) = primitive)
          then self.att = obj.att endcase
      )
```

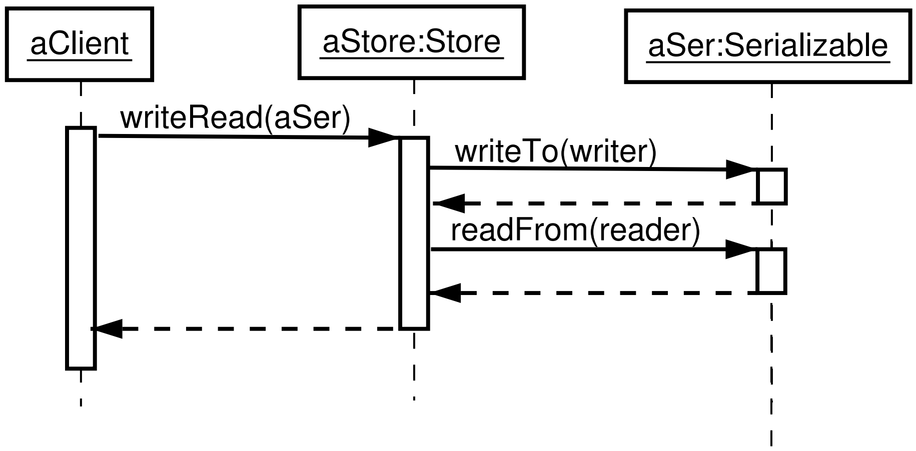

Abbildung 6.3: Hilfsmethode *writeRead*

6.1.3.3 WriteRead Methode

Um die Forderung, daß *writeTo()* und *readFrom()* invers zueinander sind zu formalisieren, wird eine Hilfsmethode definiert, die ein **Serializable** Objekt mittels *writeTo()* linearisiert um anschließend das Ergebnis mit *readFrom()* wieder in eine Objektstruktur umzuwandeln. Von dieser neuen Objektstruktur kann nun gefordert werden, daß sie mit dem Original übereinstimmt.

Der Ablauf der Hilfsmethode wird in einem Sequenzendiagramm festgelegt (siehe Abbildung 6.3). Mit dem folgenden Constraint wird anschließend die Gleichheit der beiden Objektstrukturen gefordert.

Constraint 6.1.3 (Write und Read sind invers)

```
ocltemplate Serializer_Write_Read_inverse {} ocl:
  context Store::writeRead(s : Serializable) post:
    -- -- readFrom() und writeTo() sind invers
    s@pre.equals(s)
```

6.1.3.4 Verfeinerung

Die obige Forderung kann auf einfachere Forderungen heruntergebrochen werden. Entsprechend dem Wissen, das die beteiligten Klassen beherbergen, wird von **Reader** und **Writer** gefordert, daß das Lesen und Schreiben von Typen invers zueinander ist. Von **Serializable** wird gefordert, daß die Struktur korrekt wiederhergestellt wird.

Wie oben für das Schreiben und Lesen von **Serializable** Objekten, werden nun auch für primitive Typen Hilfsmethoden eingeführt, die einen primitiven Typ schreiben und anschließend wieder einlesen. Abbildung 6.4 zeigt eine Schablone für die entsprechenden Sequenzendiagramme.

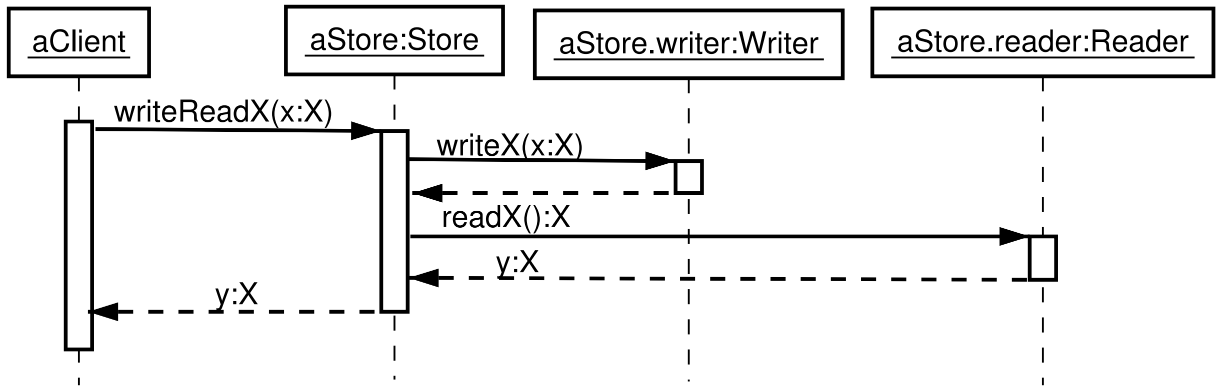

Abbildung 6.4: Hilfsmethode *writeReadX*

Mit dem folgenden Constraint wird anschließend die Gleichheit der beiden Objekte gefordert.

Constraint 6.1.4

ocltemplate `Serializer_Read_Write_attribute_inverse` {} **ocl:**
  context `Store::writeReadX(s : X) : X` **post:**
```
-- -- readX() und writeX() sind invers
result.equals(s@pre)
```

Per Induktion kann gezeigt werden, daß beliebige Blöcke von Daten linearisiert und anschließend wieder restauriert werden können. Dazu muß allerdings sichergestellt werden, daß die Attribute in der gleichen Reihenfolge gelesen werden, wie sie geschrieben wurden.

Prinzipiell kann der Ablauf von Methoden in Sequenzendiagrammen dargestellt werden. Ein Problem in diesem Zusammenhang ist, daß Sequenzendiagramme, wie alle Interaktionsdiagramme, nur exemplarische Abläufe darstellen. Ein solches Diagramm erhebt keinen Anspruch auf Allgemeinheit. Nimmt man jedoch eine solche Allgemeinheit an, so könnte auf der Grundlage von Sequenzendiagrammen für *readFrom()* und *writeTo()* die Forderung nach gleicher Abfolge der Lese- bzw. Schreibeoperationen formuliert werden.

6.1.4 Anwendung

Abgesehen von der Abbildung der Klassen und Methoden, ist beim Serializer Muster die Festlegung der Charakteristik (persistent, transient, primitive) der Attribute und Assoziationen aller von **Serializer** abgeleiteten Klassen von besonderer Bedeutung.

Die folgende Tabelle gibt die Abbildung des abstrakten Modells in das Projektmodell wieder:

Muster	→	eMail-Client
Serializable		**Serializable**
Serializable.writeTo(r : Writer)		*AddressBookEntry.writeTo(r : Writer)*
Serializable.readFrom(r : Reader)		*AddressBookEntry.readFrom(r : Reader)*
Serializable.equals(obj : OclAny)		*AddressBookEntry.equals(obj : Object)*
Serializable.characteristic(att : String)		*AddressBookEntry.characteristic(att : String)*
ConcreteElement		**ABEComosite**
ConcreteElement		**ABELeaf**
ConcreteElement		**AddressBookEntry**
Store		**Store**
Store.writeRead(c: Serializable)		*Store.testWriteReadInverse(c: Serializable)*
ConcreteStore		**ConcreteStore**
Writer		**Writer**
Writer.writePrimitive(p: Primitive)		*Writer.writeProperty(p: Property)*
Writer.writeObject(p: Serializable)		*Writer.writeObject(p: Serializable)*
ConcreteWriter		**ConcreteWriter**
Reader		**Reader**
Reader.readPrimitive(): Primitive		*Reader.readProperty(): Property*
Reader.readObject(): Serializable		*Reader.readObject(): Serializable*
ConcreteReader		**ConcreteReader**

Für die Festlegung der Charakteristik müssen zusätzliche Constraints erstellt werden. Im Falle des **ABEComposite** wird die *entry* Assoziation als persistent charakterisiert. Das **ABELeaf** speichert einen Kontakt. Die Daten eines Kontakts werden in einem **Property**-Objekt gespeichert, dieses wird in diesem Zusammenhang als primitiv charakterisiert. Eventuelle weitere Attribute und Assoziationen werden nicht weiter betrachtet. Somit ergeben sich die folgenden beiden Constraints.

Constraint 6.1.5 (Charakteristik für **ABEComposite**)

```
context ABEComposite::characteristic(att : String) post:
    if (att = 'entry') then
      result = 'persistent'
    else
      result = ''
    endif
```

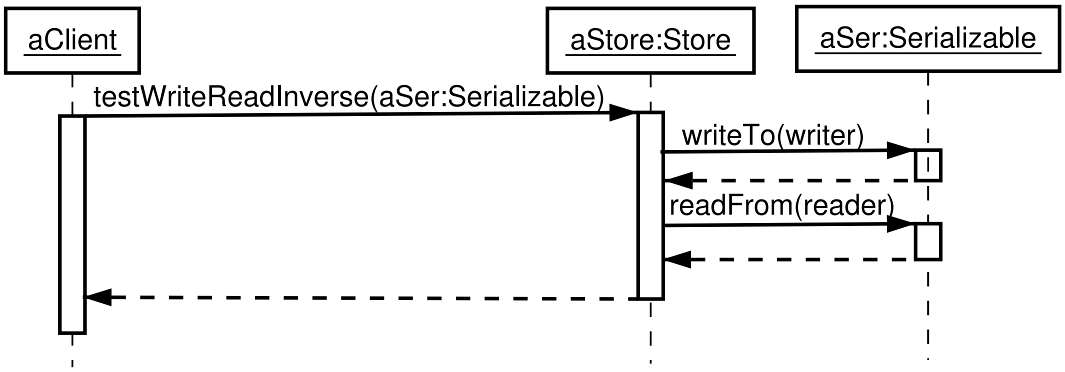

Abbildung 6.5: Hilfsmethode *testWriteReadInverse()*

Constraint 6.1.6 (Charakteristik für **ABELeaf**)

context `ABELeaf::characteristic(att : String)` **post:**

```
if (att = 'property') then
  result = 'primitive'
else
  result = ''
endif
```

Mit obiger Definition für die Charakteristik kann nun Constraint 6.1.2 für das Adreßbuch instanziiert werden. Es ergeben sich die folgenden beiden Constraints.

Constraint 6.1.7 (Gleichheit für **ABEComposite**)

context `ABEComposite::equals(obj : Object)` **post:**

```
-- -- Gleichheit mit dem übergebenen Objekt
result =
  self.oclType = obj.oclType and
  self.entry.equals(obj.entry)
```

Constraint 6.1.8 (Gleichheit für **ABELeaf**)

context `ABELeaf::equals(obj : Object)` **post:**

```
-- -- Gleichheit mit dem übergebenen Objekt
result =
  self.oclType = obj.oclType and
  self.property = obj.property
```

Abbildung 6.5 zeigt ein angepaßtes Sequenzendiagramm für die *writeRead()* Methode. Mit Hilfe dieser Methode kann nun durch Instanziierung von Constraint 6.1.3 gefordert

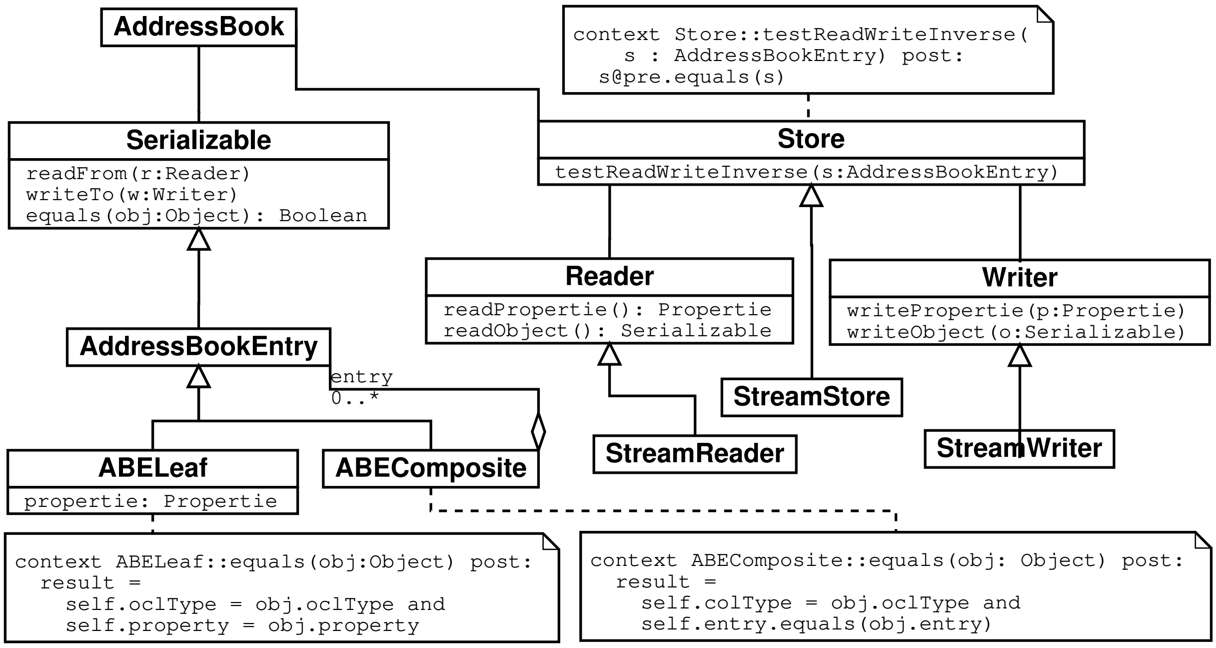

Abbildung 6.6: Anwendung des Serializer Musters auf *keyMail/S*

werden, daß *readFrom()* und *writeTo()* für das Adreßbuch invers sind.

Constraint 6.1.9 (Write und Read sind invers)

context `Store::testReadWriteInverse(s : AddressBookEntry)` **post:**
`-- -- readFrom() und writeTo() sind invers`
`s@pre.equals(s)`

Abbildung 6.6 zeigt das resultierende Klassendiagramm nach Anwendung des Serializer-Musters auf die Kontaktverwaltung.

6.2 Observer

Das Observer Muster ist für Situationen vorgesehen, in denen Objekte über Zustandsänderungen anderer Objekte informiert werden müssen. Solche Situationen treten auf, wenn z.B. der Inhalt eines Datenobjekts in einem Fenster dargestellt werden soll. Ändert sich das Datenobjekt, so soll sich diese Änderung auch im entsprechenden Fenster widerspiegeln.

6.2.1 Beschreibung

Das Observer Muster erfüllt mehrere Anforderungen, die auftreten können, wenn Objekte vom Zustand anderer Objekte abhängen. So erlaubt es, beliebig viele Objekte über eine

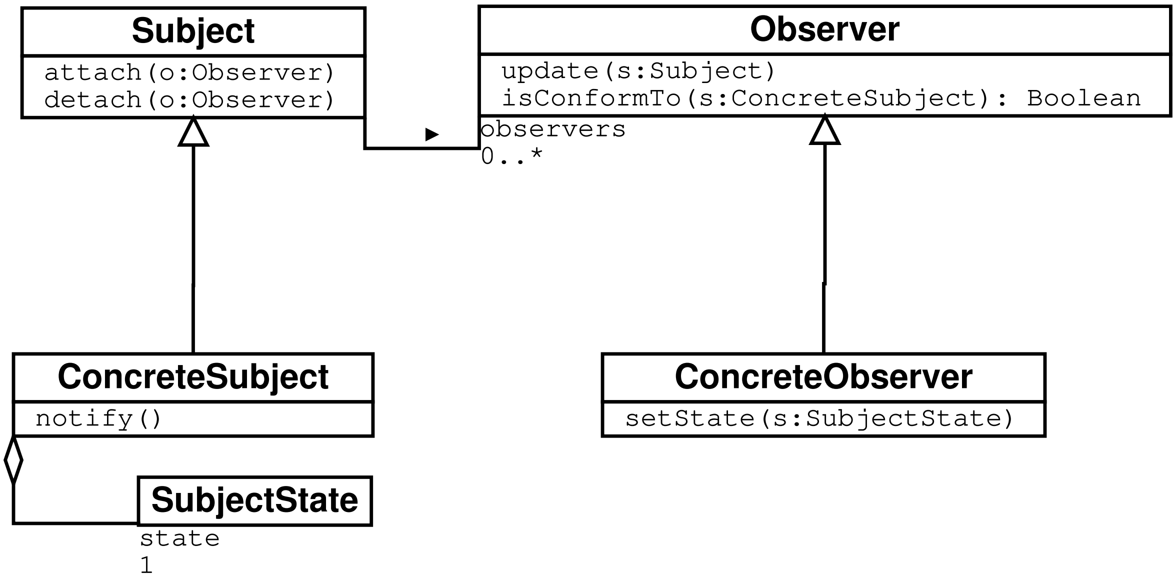

Abbildung 6.7: Struktur des Observer Musters

Zustandsänderung zu informieren. Zu informierende Objekte können jederzeit hinzugefügt oder entfernt werden.

Als grundlegende Klassen des Observer Musters werden **Subject** und **Observer** definiert. **Subject** besitzt eine Assoziation zur Klasse **Observer**. Diese Assoziation repräsentiert die Menge der Objekte, die von einem Zustandswechsel betroffen sind. Über die Methoden *attach()* und *detach()* ist es möglich Observer-Objekte hinzuzufügen bzw. zu entfernen. Die Methode *notify()* wird im Falle eines Zustandswechsels aufgerufen und dient dazu, diesen an alle assoziierten Observer zu propagieren. Dazu wird bei jedem Observer die Methode *update()* aufgerufen.

Die Assoziation *state* stellt den Zustand des Subjekts in geeigneter Form zur Verfügung. Da **ConcreteSubject** und **ConcreteObserver** dabei unterschiedliche Klassen sind und somit unterschiedliche Zustandsmengen haben, kann dabei nicht direkt die Identität der Zustände gefordert werden. Vielmehr muß eine Relation der beiden Zustandsmengen hergestellt werden, die über die Konformität zweier Zustände Auskunft gibt. Dazu wird im **Observer** die Methode *isConformTo()* definiert.

Ein Klassendiagramm für diese Struktur ist in Abbildung 6.7 gegeben. Das Sequenzendiagramm in Abbildung 6.8 zeigt den Abarbeitung eines *notify()*. Eine ausführliche Beschreibung des Musters findet sich in [GHJV95].

6.2.2 Anforderungen

Die zentrale Forderung im Observer Muster ist, daß nach einem *notify()* alle assoziierten **Observer** aktualisiert sind, d. h. das ihr Zustand konform mit dem Zustand des **Subject**s ist.

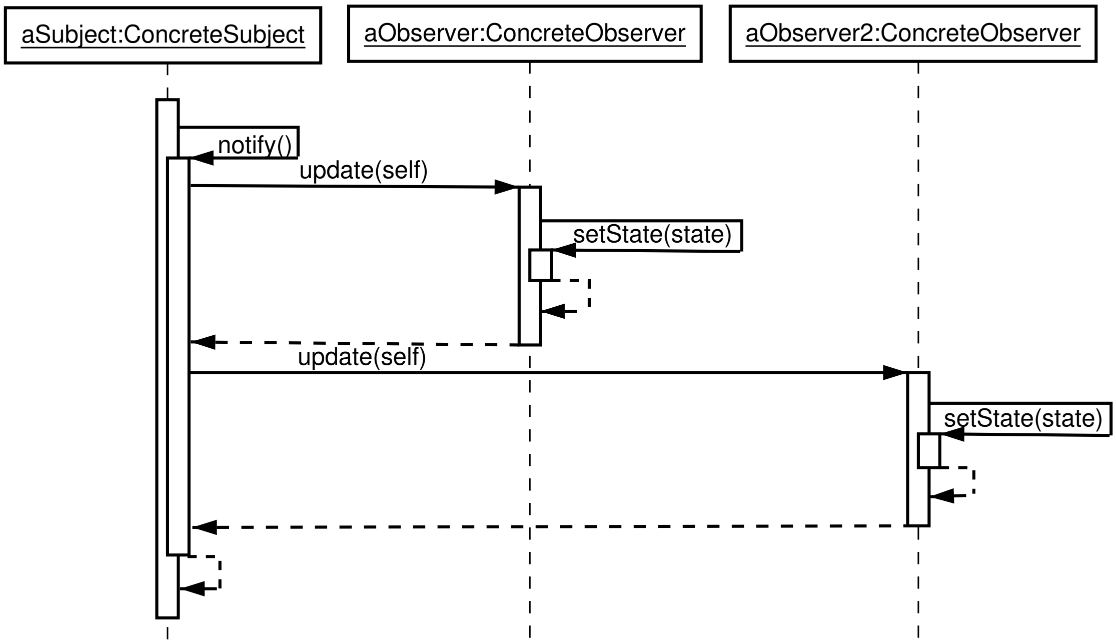

Abbildung 6.8: Muster Observer: Abarbeitung eines *notify()*

Einfachere Forderungen können mit den restlichen Methoden verknüpft werden. So muß *update()* den Zustand des **Observer**-Objektes in einen konformen Zustand bringen. Da nur für eine konkrete Anwendung festgelegt werden kann, wann zwei Zustände konform sind, lassen sich an die Methode *isConformTo()* zu diesem Zeitpunkt keine Forderungen stellen.

Die Methoden *attach()* und *detach()* stellen *add()* bzw. *remove()* Operationen für die Menge der assoziierten **Observer** dar. Da es, bei mehrfachem Einfügen eines **Observer**-Objektes hinreichend ist, wenn dieser nur einmal in der Liste erscheint, wird die *add()* Methode in der entsprechenden Ausprägung als „Einfügen in eine Menge (Set)" zur Anwendung gebracht (siehe Abschnitt 7.2.7).

Forderung	Constraint
Update aller Observer	6.2.1

6.2.3 Formalisierung

Um sicherzustellen, daß alle Observer von der Änderung Notiz genommen haben, muß für jeden gelten, daß er sich in einen Zustand befindet in dem *isConformTo()* gilt. Um dies zu fordern, muß zusätzlich gelten, daß sich der Zustand des **ConcreteSubject** während des *notify()* nicht ändert. Forderungen an die *isConformTo()* hängen von dem konkreten Projekt ab.

Constraint 6.2.1 (Update aller Observer)

```
ocltemplate UpdateAll {} ocl:
  context ConcreteSubject::notify() post:
    -- -- nach einem notify() müssen alle assoziierten
    -- -- Observer in einem konformen Zustand sein
      observers->forAll(o | o.isConformTo(self.state))
    and
    -- -- Der Zustand wird beibehalten
      self.state@pre = self.state
```

Die folgenden Forderungen beschreiben die Arbeitsweise der restlichen Methoden.

Constraint 6.2.2 (Update eines einzelnen Observers)

```
ocltemplate Update {} ocl:
  context Observer::update(sub : ConcreteSubject) post:
    -- -- nach einem Update ist der Observer Zustand
    -- -- konform mit dem Zustand des Subjects
    self.isConformTo(sub.state)
```

Constraint 6.2.3

```
ocltemplate Attach {} ocl:
    -- -- Instanz von Collection Add
    Collection_addElement.instanciate(
      Map{
        aClass : Subject, aType : Observer,
        coll : observers, addElem : attach,
        flavor : set
      }
    )
```

Constraint 6.2.4

```
ocltemplate Detach {} ocl:
    -- -- Instanz von Collection Remove
    Collection_removeElement.instanciate(
      Map{
        aClass : Subject, aType : Observer,
        coll : observers, removeElem : detach,
        flavor : set
      }
    )
```

6.2.4 Anwendung

Im **fun** eMail-Client findet das Observer Muster z.B. bei der Interaktion des Benutzers mit dem Adreßbuch Anwendung. Dabei bildet **AddressBook** das konkrete Subjekt, **AddressBookView** repräsentiert den konkreten Observer. Es präsentiert dem Benutzer eine Ansicht des Inhalts des Adreßbuches. Damit der Benutzer stets den korrekten Inhalt präsentiert bekommt, muß der **AddressBookView** über Zustandsänderungen im **AddressBook** informiert werden.

Die folgende Tabelle gibt die Abbildung des abstrakten Modells in das Projektmodell wieder:

Muster	→	eMail-Client
Subject		**Observable**
Subject.observers : Observer		*Observable.observers : Observer*
Subject.attach(o : Observer)		*Observable.attach(o : Observer)*
Subject.detach(o : Observer)		*Observable.detach(o : Observer)*
Observer		**Observer**
Observer.isConformTo(s : SubjectState) : Boolean		*Observer.conforms(s : AddressBook) : Boolean*
Observer.update(s : ConcreteSubject)		*Observer.update(s : AddressBook)*
ConcreteSubject		**AddressBook**
ConcreteSubject.state : SubjectState		*AddressBook.self : AddressBook*
ConcreteSubject.notify()		*AddressBook.notify()*
SubjectState		**AddressBook**
ConcreteObserver		**AddressBookView**
ConcreteObserver.setState(s : SubjectState)		*AddressBookView.update(s : AddressBook)*

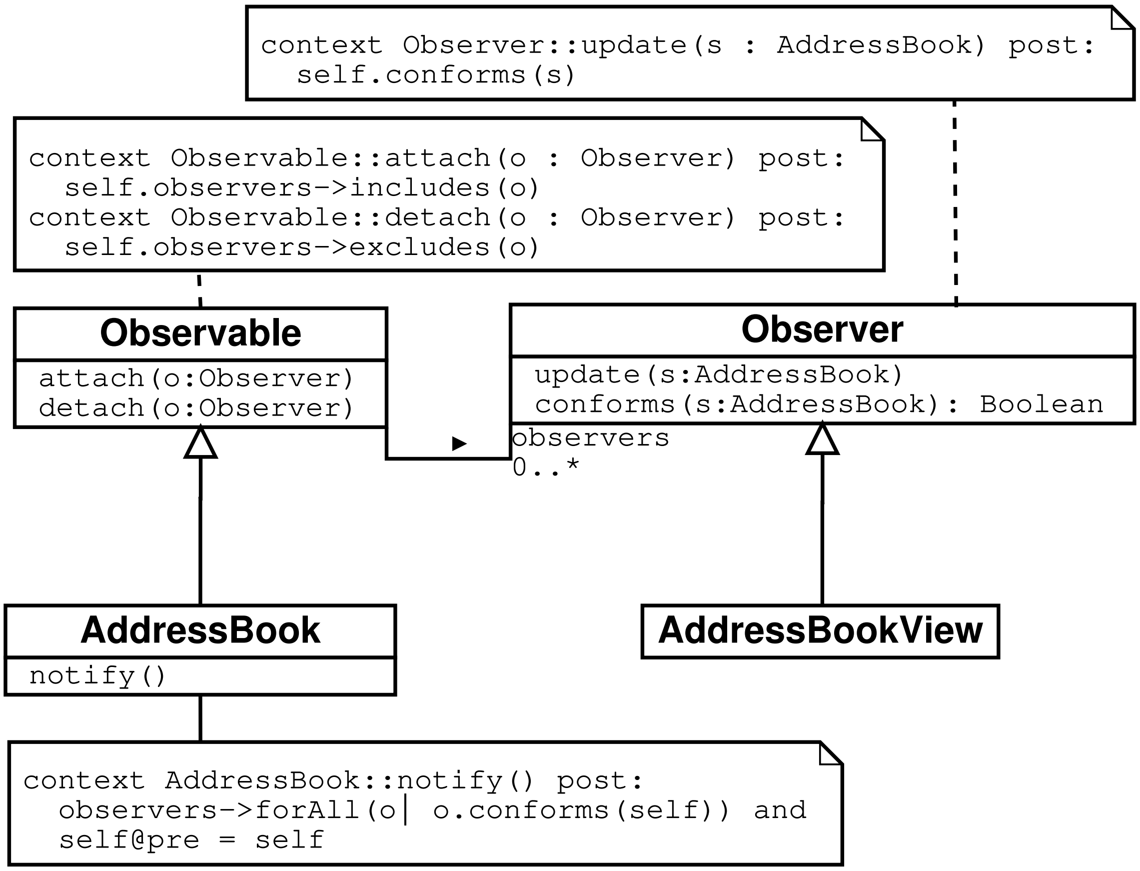

Abbildung 6.9: Anwendung des Observer Musters auf *keyMail/S*

Mit dieser Abbildung lassen sich die Constraints für das konkrete Projekt erzeugen. Abbildung 6.9 zeigt das resultierende Klassendiagramm.

CONSTRAINT 6.2.5

```
context AddressBook::notify() post:
  -- -- nach einem notify() müssen alle assoziierten
  -- -- Observer in einem konformen Zustand sein
  observers->forAll(o | o.conforms(self))
  and
  -- -- Der Zustand wird beibehalten
  self@pre = self
```

Constraint 6.2.6

```
context Observer::update(sub : AddressBook) post:
  -- -- nach einem Update ist der Observer Zustand
  -- -- konform mit dem Zustand des Subjects
  self.conforms(sub)
```

Constraint 6.2.7

```
context Observable::attach(o : Observer) post:
  self.observers->includes(o)
```

Constraint 6.2.8

```
context Observable::detach(o : Observer) post:
  self.observers->excludes(o)
```

Kapitel 7

Einfache Forderungen

7.1 Objekte

7.1.1 Gleichheit von Objekten

In **OCL** ist Gleichheit nur in der Form von Identität präsent. Soll jedoch z.B. ein Objekt dupliziert werden, so muß die Gleichheit im Sinne von inhaltsgleich definiert werden. Hier ist es jedoch nicht immer sinnvoll, Gleichheit über die Gleichheit, oder gar Identität von Attributen und Assoziationen zu definieren. Es muß also eine Gleichheitsrelation abstrakt eingeführt werden. Das folgende Template formuliert die Forderungen, die eine solche Relation erfüllen muß.

Constraint 7.1.1 (Gleichheitsrelation)

```
ocltemplate Object_equals {} ocl:
  context Object inv:
    let inst = Object.allInstances
    inv: -- -- Reflexivität
    inst->forAll(i | i.equals(i))
    inv: -- -- Symmetrie
    inst->forAll(i1, i2 |
      i1.equals(i2) implies i2.equals(i1))
    inv: -- -- Transitivität
    inst->forAll(i1, i2, i3 |
      i1.equals(i2) and i2.equals(i3) implies i1.equals(i3))
```

7.1.2 Duplizieren von Objekten

Mit Hilfe einer Gleichheitsrelation, wie sie oben definiert wurde, läßt sich zum Beispiel eine Methode zum Duplizieren eines Objektes definieren.

Constraint 7.1.2 (Erzeugen eines Duplikates)

```
ocltemplate Object_duplicate {} ocl:
  context Object::duplicate() : Object post:
    (not (result = self)) and (result.equals(self))
```

Constraint 7.1.3 (Objekte gleich aber nicht identisch)

```
ocltemplate Object_isDuplicate {} ocl:
  context Object::isDuplicate(obj: Object) : Boolean post:
    result = (not (obj = self)) and obj.equals(self)
```

7.1.3 Ordnung von Objekten

Ordnungsrelation auf Objekten; diese wird z.B. für die Forderung benötigt, daß eine Liste sortiert ist.

Constraint 7.1.4 (Ordnungsrelation)

```
ocltemplate Class_ordered {} ocl:
  context Object
    let elem = Object.allInstances in
      inv: -- -- Reflexivität
      elem->forAll(e | e.greaterEqual(e))
      inv: -- -- Antisymmetrie
      elem->forAll(e1, e2 |
        (e1.greaterEqual(e2) and e2.greaterEqual(e1))
        implies e1.equals(e2))
      inv: -- -- Transitivität
      elem->forAll(e1, e2, e3 |
        (e1.greaterEqual(e2) and e2.greaterEqual(e3))
        implies e1.greaterEqual(e3))
      inv: -- -- Totalität
      elem->forAll(e1, e2 |
        e1.greaterEqual(e2) or e2.greaterEqual(e1))
```

7.2 Collections

Im Design von *keyMail/S* beziehen sich viele Forderungen auf Collections (Assoziationen mit einer Multiplizität > 1), diese treten sowohl in Invarianten, als auch in Vor- und Nachbedingungen auf. Auch in Entwurfsmustern beziehen sich viele Forderungen auf Collections, die somit eine der grundlegenden Strukturen darstellen.

Im Folgenden sollen die häufigsten dieser Forderungen an Collections aufgelistet werden. Da **OCL** Collections bereits umfangreich unterstützt, existieren für einige der Forderungen bereits explizite Konstrukte, welche die Forderung formulieren.

7.2.1 Teil-Collection Beziehung

Eine Collection ist eine Teil-Collection einer anderen.

```
superCollection->includesAll(subCollection)
```

7.2.2 Element Beziehung

Ein Objekt ist Element einer Collection

```
collection->includes(object)
```

7.2.3 Die Collection ist eine Menge

Jedes Element kommt nur einmal vor.

```
collection->forAll(e | collection->count(e) = 1)
```

In **OCL** bereits wurde zwar bereits die Methode *isUnique()* definiert, diese ist jedoch in der **OCL** Spezifikation fehlerhaft formalisiert (siehe Diskussion des *forAll* mit mehreren Iteratoren, Abschnitt 4.4).

7.2.4 Konstante Elemente

Es gibt konstante Elemente, die in der Collection enthalten sind. Diese Forderung eignet sich als Invariante, um sicherzustellen, daß gewisse Elemente nicht gelöscht werden können.

```
collection->includesAll(Collection {aConst1, aConst2, ... })
```

7.2.5 Neues Element

Eine Menge enthält ein neues Element. Diese Forderung ist nur in Nachbedingungen sinnvoll. Um diese Forderung auszudrücken, wird in der Nachbedingung von der Collection die Collection zum Zeitpunkt des Aufrufs der Methode abgezogen. Ist diese Differenz nicht leer, so wurde ein neues Element hinzugefügt. Dabei wird nicht ausgeschlossen, daß eventuell Elemente gelöscht wurden.

```
collection->excludesAll(collection@pre)->size = 1
```

Häufig ist nicht nur von Bedeutung, daß eine Collection ein neues Element enthält, sondern daß dieses Element auch eine neue Instanz ist. Die Forderung nach einer neuen Instanz wurde bereits bei der Diskussion des *new()*-Operators (siehe Abschnitt 4.2) formuliert. Auf Collections bezogen, ergibt sich somit folgender **OCL** Ausdruck:

```
let newInstances =
  aClass.allInstances->excludes(aClass.allInstances@pre) in
newInstances->exists(e | collection->includes(e))
```

Dabei sei *collection* vom Typ **Collection(T)**.

7.2.6 Sequence ist geordnet

Oft ist es für Sequenzen wünschenswert, daß diese geordnet ist. Dazu wird eine Ordnungsrelation auf dem Elementtyp benötigt. Es ist dann hinreichend zu fordern, daß jeweils benachbarte Elemente geordnet sind.

In der **OCL** Spezifikation [SMHP+99], Kapitel 7 wird die Methode *sortedBy()* eingeführt. Die Semantik ist jedoch nur umgangssprachlich gegeben.

Ist auf dem Elementtyp eine Ordnung definiert (siehe Abschnitt 7.1.3), läßt sich in einfacher Weise fordern, daß eine Sequence entsprechend dieser Ordnung sortiert ist.

Constraint 7.2.1 (Sequence ist sortiert)

```
ocltemplate Sequence_sorted ocl:
  context Sequence(Class) inv:
    Sequence{1..self->size-1}->forAll(index : Integer |
      self->at(index).greaterEqual(self->at(index+1)))
```

Durch die Formulierung als Invariante wirkt sich die Forderung nach Ordnung direkt auf das Einfügen von Elementen aus und garantiert, daß die Ordnung nicht zerstört werden kann.

7.2.7 Element einfügen

Beim Einfügen eines Elements in eine Collection muß zuerst entschieden werden, ob es sich bei der Collection um eine Menge (d.h. ein Element ist maximal ein mal enthalten) handeln soll.

Als Nachbedingung des Einfügens wird dann gefordert, daß das Element in der Collection enthalten ist. Ist die Collection keine Menge, so wird gefordert, daß die Anzahl der Vorkommen des Elements in der Collection um eins erhöht wurde.

Im Folgenden sei **aClass** die Klasse, welche die Collection enthält, **aType** sei der Elementtyp der Collection und *coll : Collection(aType)* sei das Attribut bzw. die Assoziation, welche die Collection repräsentiert.

Constraint 7.2.2 (Einfügen eines Elements)

```
ocltemplate Collection_addElement {flavor}
  context aClass::addElem (elem : aType) post:
    case (flavor = 'set') then
      post: -- -- falls Menge
        self.coll->includes(elem)
    else
      post: -- -- falls nicht Menge
        self.coll->count(elem) = self.coll@pre->count(elem) + 1
    endcase
```

7.2.8 Element entfernen

Ist die Collection keine Menge, so muß entschieden werden, ob beim Entfernen alle oder nur ein Vorkommen des Elements in der Collection entfernt werden soll.

Im Folgenden sei **aClass** die Klasse, welche die Collection enthält, **aType** sei der Elementtyp der Collection und *coll : Collection(aType)* sei das Attribut bzw. die Assoziation, welche die Collection repräsentiert.

Constraint 7.2.3 (Entfernen eines Elements)

```
ocltemplate Collection_removeElement {flavor} ocl:
  context aClass::removeElem (elem : aType) post:
    case (flavor = 'set') then
      -- -- falls Menge
      -- -- bzw. alle Vorkommen entfernt werden sollen
        self.coll->excludes(elem)
    else
      -- -- falls nur ein Vorkommen entfernt werden soll
        self.coll@pre->includes(elem) implies
          (self.coll->count(elem) = self.coll@pre->count(elem) - 1)
    endcase
```

Ist die Collection als Sequence ausgeprägt, so stellt sich zusätzlich die Frage, welches Vorkommen gelöscht werden soll. Die naheliegendsten Vorkommen sind in diesem Fall das erste bzw. das letzte Vorkommen. Zur Formalisierung dieser Forderung wird zuerst eine weitere Funktion auf Sequenzen definiert. Diese Funktion liefert die Sequenz der Indizes aller Vorkommen eines Elementes. Dazu wird eine Sequenz aller Indizes erzeugt und anschließend daraus die Indizes selektiert, an deren Position das gesuchte Element erscheint.

Constraint 7.2.4 (Vorkommen eines Elementes)

```
ocltemplate Collection_indices {} ocl:
  context Sequence(T)::indices (e : T) : Sequence(Integer)
    result = Sequence{1..self->size}->select(i | self.at(i) = e)
    -- -- Die Indizes an denen das Objekt e
    -- -- in der Sequenz erscheint.
```

Jetzt kann gefordert werden, daß ein spezifischer dieser Indizes entfernt wird. Soll das erste Vorkommen entfernt werden, so muß `position` auf `first` gesetzt werden. Um das letzte Vorkommen zu entfernen, wird `position` auf `last` gesetzt.

Constraint 7.2.5

```
ocltemplate Collection_removeElem_Sequence {position} ocl:
  context aClass::removeElem (elem : aType)
    post: -- -- Ein Vorkommen von elem wird entfernt.
      let id = self.coll@pre->indices in
      id->size > 0 implies
        self.coll->size = self.coll@pre->size -1
      and
        self.coll->subSequence(1,id->position - 1) =
          self.coll@pre->subSequence(1, id->position - 1)
      and
        self.coll->subSequence(id->position, self.coll->size) =
          self.coll@pre->subSequence(
            id->position + 1, self.coll->size + 1)
```

Alternativ zum bloßen Entfernen des Elements aus der Collection kann auch das explizite Löschen des Elements gefordert werden. In diesem Fall müssen selbstverständlich alle Vorkommen des Elements in der Collection entfernt werden.

Constraint 7.2.6 (Löschen eines Elements)

```
ocltemplate Collection_deleteElem {} ocl:
  context aClass::deleteElem (elem : aType)
    post:
      -- -- Das Objekt ist nicht mehr in der Collection enthalten
      self.coll->excludes(elem) and
      -- -- Das Objekt existiert nicht mehr
      aType.allInstances->excludes(elem)
```

7.2.9 Element einschränken

Manche Collections dürfen nur Elemente enthalten, die eine bestimmte Bedingung erfüllen. Eine solche Einschränkung kann sowohl positiv als auch negativ formuliert werden. Im ersten Fall wird ein **OCL** Ausdruck gegeben, der für alle Elemente erfüllt sein muß, im zweiten Fall ein Ausdruck, der für kein Element erfüllt sein darf.

Im Composite Muster wird die Einschränkung von Elementen benutzt um Kanten zwischen einzelnen Component-Klassen auszuschließen (siehe 5.2.3.2).

Constraint 7.2.7 (Elemente einschränken)

```
ocltemplate Collection_restiction {expression, flavor} ocl:
  context Class inv:
    let conform = coll->collect(expression)->asSet in
      case (flavor = 'positive') then
        conform = Set{true }
      else
        conform = Set{false }
      endcase
```

Kapitel 8

Zusammenfassung

Wie sich gezeigt hat, können viele Forderungen direkt in **OCL** ausgedrückt werden. Um jedoch die Forderung in einer allgemeinen Form (unabhängig von der konkreten Situation in einem Projekt) zu formulieren, muß **OCL** erweitert werden. Dabei wird ein abstraktes Modell gegeben, welches die Situation beschreibt in der eine Forderung auftritt. Bei der Übertragung der Forderung auf die konkrete Situation wird das abstrakte Modell auf das Modell der konkreten Situation abgebildet. Zusätzlich werden Konstrukte eingeführt, die eine Einflußnahme auf die Formulierung zum Zeitpunkt der Übersetzung erlauben (**case**, **foreach**).

Die gefundenen Forderungen können zu einem großen Teil mit Entwurfsmustern verknüpft werden. Dabei wird das abstrakte Modell für die Forderungen durch das Entwurfsmuster gegeben. Die Verknüpfung der Forderungen mit Entwurfsmustern bietet zusätzlich die Möglichkeit verwandte Forderungen zusammenzufassen.

Wie sich gezeigt hat, sind jedoch nicht alle Entwurfsmuster dazu geeignet sich in obiger Weise formalisieren zu lassen. So zeigte sich zum Beispiel für das Proxy Muster, daß die einzelnen Forderungen, die im Zusammenhang mit diesem Muster auftreten, zu unterschiedlich sind, um dafür eine allgemeine Formulierung zu finden. So dient das Proxy Muster zum Beispiel dem verzögerten Einlesen von Daten, der Repräsentation von nichtlokalen Daten oder auch der Zugriffskontrolle auf Daten. Die sich daraus ergebenden Forderungen lassen sich nicht sinnvoll miteinander verbinden. Um das Proxy Muster trotzdem zu formalisieren müßte es in Teilmuster für die einzelnen Ausprägungen unterteilt werden.

In diesem Zusammenhang ist auch zu erwähnen, daß die Formalisierung die Flexibilität in gewissem Maße einschränkt. Mit Entwurfsmustern sind oft alternative Lösungsansätze oder Variationen der Modellierung gegeben. Durch die Abhängigkeit der Formulierung von dem dem Muster zugrundeliegenden Modell, können Variationen die dieses Modell betreffen nur schwer unterstützt werden.

Statische Forderungen eignen sich besonders gut zur Formalisierung in **OCL**, dynamische sind schwerer zu formalisieren. Innerhalb von **OCL** kann zur Formalisierung solcher Forderungen nur auf das *@pre* Konstrukt zurückgegriffen werden, welches innerhalb von Nachbedingungen die Bezugnahme auf den Zeitpunkt des Aufrufs einer Methode erlaubt. Um zeitliche Abläufe zu beschreiben, können jedoch **UML**-Diagramme benutzt werden

(Sequenzen- bzw Kollaborationsdiagramme).

Um ein formalisiertes Entwurfsmuster auf ein konkretes Projekt anzuwenden, müssen im wesentlichen eine Abbildung des abstrakten Modells in das konkrete Modell festgelegt und eventuell mit weiteren Parametern Designentscheidungen reflektiert werden. Durch das formale Muster sind die zur Formalisierung erforderlichen Constraints gegeben. Die Übersetzung der Constraints für das konkrete Projekt kann dann, mit Hilfe der festgelegten Abbildung automatisch erfolgen.

Kapitel 9

Ausblick

Um die hier beschriebene Vorgehensweise nutzbringend anzuwenden, müßte eine Einbindung in ein passendes CASE-Tool erfolgen. Das CASE-Tool stellt dann das Modell des realen Projektes zur Verfügung. Zusätzlich müßte eine Formalisierung für eine hinreichende Anzahl von Entwurfsmustern erfolgen. Dies würde dem Anwender erlauben, ein passendes formales Muster für ein Projekt zu wählen und das Modell des Musters auf das konkrete Modell abzubilden. Damit könnten in einfacher Weise sinnvolle und formal korrekte Constraints in das Projekt eingefügt werden.

Im weiteren wäre auch von Bedeutung, den Benutzer bei der Verfeinerung und Transformation des Designs so zu unterstützen, daß die Korrektheit von Constraints erhalten bleibt. Eine Diskussion über korrekte Transformationen für **UML**-Diagramme findet sich zum Beispiel in [LB98b].

Ein Ziel der Verwendung formaler Methoden ist die Verifikation von Eigenschaften eines Systems. Durch die Verwendung der hier propagierten formalen Muster wird in erster Linie die Formulierung von Forderungen in einer Form, die diese der Verifikation zugänglich macht, erleichtert. Da jedoch zum Zeitpunkt des Entwurfs eines formalen Musters sowohl die abstrakte Form der Forderung, als auch Charakteristika der Implementierung bekannt sind, könnten mit dem Muster auch Strategien für einen Beweis der Forderung im konkreten Projekt verknüpft werden.

Literaturverzeichnis

[BCM$^+$96] Kent Beck, Ron Crocker, Gerard Meszaros, James Coplin, Lutz Dominick, Frances Paulish, and John Vlisssides. Industrial experience with design patterns. Technical report, 1996.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns - pattern oriented software architecture*. Wiley, 1996.

[DW98] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[EC97] Andy Evans and Tony Clark. Foundations of the Unified Modeling Language. In *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997*, 1997.

[GHJV95] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[LB98a] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998.

[LB98b] Kevin Lano and Juan Bicarregui. UML refinement and abstraction transformations. Technical report, 1998.

[MC99] Luis Mandel and María Victoria Cengarle. On the expressive power of the object constraint language OCL. Technical report, Forschungsinstitut für angewandte Software-Technologie (FAST e.V.), 1999.

[MRB98] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design*. Addison-Wesley, 1998.

[SMHP$^+$99] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *Unified Modeling Language (version 1.3)*. Rational Software Corporation, June 1999.