

“Separation of Concern” mit Rollen, Subjekten und Aspekten

Ralph Depke, Katharina Mehner
Fachbereich Mathematik und Informatik
Universität Paderborn
Warburger Str. 100, D-33098 Paderborn
{depke, mehner}@upb.de

Zusammenfassung

Die Aufteilung von Modellen hinsichtlich Anforderungen (separation of concern) ist ein wichtiger aktueller Trend in der Softwaretechnik. Modelle sollen modular strukturiert und entsprechend erweiterbar sein. So werden Probleme, die aus scattering und tangling entstehen, vermieden. Die Integration von Teilmodellen ist einerseits additiv, so daß Elemente auf Ebene von Attributen und Methoden von Teilmodellen unverändert übernommen werden können. Andererseits ist invasive Integration nötig, um Methodenrumpfe zu verändern. Rollen, Subjekte und Aspekte sind Konzepte zur getrennten Beschreibung und Integration von Teilen von Struktur- und Verhaltensmodellen. In diesem Papier untersuchen wir, inwieweit additive und invasive Integration von Teilmodellen in diesen Ansätzen unterstützt werden. Wir stellen selbst ein Konzept zur aspektorientierten Modellierung vor, das beide Integrationsweisen erlaubt.

1. Einleitung

Objektorientierte Programmierung und objektorientierte Modellierung haben sich zum Ziel gesetzt, die modulare Strukturierung eines Systems mittels Klassen und Objekten zu unterstützen und somit die Verständlichkeit und die Wartbarkeit zu erhöhen. Mittlerweile ist es jedoch allgemein anerkannt, daß OO hinter diesen Zielen zurück geblieben ist. In der Praxis sind objektorientierte Entwurfsmodelle groß und monolithisch, weil die vorhandenen Sprachmittel in objektorientierten Modellierungssprachen wie Interfaces, Klassen und Pakete den Entwickler dazu zwingen, Entwurfsentscheidungen zu treffen, die nachteilige Konsequenzen haben [1]. Wesentlich ist die Diskrepanz zwischen Anforderungen und Entwurf, die dadurch entsteht, daß der Entwurf strukturell oftmals stark dem daraus resultierenden Code ähnelt. Im einzelnen wird eine Anforderung oft durch auf mehrere Klassen verteilte Funktionen erfüllt (scattering), und eine Klasse realisiert andererseits Funktionen zu mehreren Anforderungen (tangling) [1].

Diese grundsätzliche Beobachtung hat mehrere Konsequenzen. Die Umsetzung der Anforderungen ist

im Entwurf schwierig zu verfolgen (traceability) und dadurch schlecht nachvollziehbar (comprehensibility). Als Konsequenz sind Entwürfe selten wiederverwendbar und schwer zu warten. Ein noch größeres Manko stellt in größeren Projekten die Schwierigkeit dar, mit mehreren Entwicklern parallel am Entwurf zu arbeiten, da die Verantwortung nicht entsprechend den Anforderungen für isolierte Entwurfsteile vergeben werden kann.

Als Konsequenz dieser Beobachtungen wurden in den letzten Jahren einige Anstrengungen unternommen, sowohl den Entwurf als auch den Code hinsichtlich verschiedener Anforderungen zu trennen (separation of concerns [7]). Drei wichtige Ansätze sind das Rollenkonzept, subjektorientierte Programmierung und Modellierung und die aspektorientierte Programmierung.

In der objektorientierten Modellierung werden *Rollen* dazu verwendet, Struktur und Verhalten von Objekten nach Aufgaben getrennt zu modellieren. Die Zuordnung der Rollen zu Objekten ist dynamisch veränderbar. Die Sichtbarkeit auf ein Objekt über eine Rolle ist auf Attribute und Methoden der Rolle und des Objektes selbst beschränkt. Objekte können mehrere Rollen zur gleichen Zeit spielen, und diese können vom gleichen Rollentyp sein. Rollen bieten sich als ein Mittel zur stärkeren Strukturierung von Modellen gemäß den Anforderungen an.

Der subjektorientierte Ansatz zielt darauf, das funktionale Verhalten sowohl im Entwurf (subject oriented design, SOD) als auch im Code (subject oriented programming, SOP) entsprechend den Anforderungen in verschiedene unabhängige *Subjekte* zu trennen, die anhand von Kompositionsbeziehungen zu einem Gesamtsystem integriert werden.

Die *Aspektorientierte* Programmierung (AOP) zielt darauf ab, sogenannte nichtfunktionale Anforderungen wie Synchronisation, Verteilung, Fehlerbehandlung, Logging, Persistenz etc. vom Basiscode einer Klasse getrennt in einer neuen Art von Modul zu spezifizieren und anschliessend automatisch zu integrieren. Damit bietet sich dieses Konzept ebenfalls an, Modelle stärker entsprechend den Anforderungen zu spezifizieren.

Man kann diese drei Ansätze danach unterscheiden, für welche Arten von Anforderungen sie die Trennung

unterstützen. Dabei ergeben sich zwei Arten Anforderungen, die wir kurz beschreiben.

Eine Art von Anforderungen läßt sich nicht auf die objektorientierte Abstraktion, also Daten und Methoden abbilden. Hier handelt es sich um sogenannte Filter, die existierende Methoden *verfeinern*. Z.B. die Anforderung, alle Methodenaufrufe in eine Logdatei einzutragen, erfordert es, alle Methoden um das Schreiben der Logdatei zu erweitern. Diese Anforderungen nennen wir *invasiv*. Zum anderen ist zu beobachten, daß Daten und Methoden eines Objektes ganz unterschiedlichen Anforderungen dienen und sich danach orthogonal gruppieren lassen. Trotzdem handelt es sich um ein Objekt und nicht um getrennte Objekte, was nicht angemessen modelliert bzw. programmiert werden kann. Diese Anforderungen nennen wir *additiv*. Eine ähnliche Unterscheidung findet sich bei [10]. Sprachliche Mittel zur Unterstützung der Trennung dieser Arten von Anforderungen werden benötigt.

Die oben beschriebenen Ansätze SOP/D, Rollen und AOP bieten genau für die gerade beschriebenen Probleme Lösungsansätze. Zum einen wird Verhalten von Objekten in Basisverhalten und verfeinertes Verhalten getrennt. Das verfeinerte Verhalten wird in eigenen Modulen spezifiziert. Eine Spezialform von Verfeinerung des Verhaltens einer Methode sind Filter, z.B. Einschränkungen für Methodenaufrufe. In AOP werden Filter über sogenannte Advice Weaves unterstützt [8]. In SOP/D werden diese Verfeinerungen über spezielle Regeln für die Komposition von Methoden erreicht [6]. Lediglich bei Rollen wird die Verfeinerung von Methoden nicht unterstützt. Zum anderen wird verfolgt, Objekte in sogenannte Subjekte aufzuteilen, die intern objektorientiert organisiert sind, also jeweils aus Daten und Methoden bestehen, und unabhängig voneinander sind. Diese Möglichkeit gibt es auch in AOP, weil man dort auch Daten und Methoden in die Aspekte auslagern kann.

Weiterhin kann man die Ansätze danach unterscheiden, inwieweit sie sich mit Programmierung und Modellierung beschäftigen. Einige Ansätze haben sich hauptsächlich mit der Programmierung auseinandergesetzt, wie z.B. AspectJ und SOP. Noch fehlen entsprechende Ansätze für die Modellierung basierend auf AOP. Aus SOP ist der zugehörige Modellierungsansatz SOD hervorgegangen. Dieser Ansatz weist viele Ähnlichkeiten mit Rollenkonzepten auf.

Die Frage nach einer allgemeinen *aspektorientierten Modellierung* ist noch offen, und insbesondere nach der Modellierung und Integration von invasiven Anforderungen. Dazu soll in einem Vergleich von SOD, Rollen und AOP herausgefunden werden, wie additive und invasive Anforderungen unterstützt werden. Es soll untersucht werden, ob fehlende invasive Anforderungen simuliert werden können und ob das eine befriedigende Lösung darstellt, oder ob Erweiterungen gefunden werden müssen.

Im Abschnitt 2 erklären wir, wie Rollen in der objektorientierten Modellierung verwendet werden. In Abschnitt 3 gehen wir auf die subjektorientierte Mo-

dellierung ein und vergleichen dieses Konzept mit Rollen. Aspektorientierte Programmierung stellen wir in Abschnitt 4 vor und zeigen Gemeinsamkeiten und Unterschiede zu Rollen und dem subjektorientierten Ansatz auf. Wir schlagen ein Konzept zur aspektorientierten Modellierung in Abschnitt 5 vor, das mit einem bereits eingeführten Rollenkonzept verträglich ist. Ein Resümee in Abschnitt 6 schließt diesen Artikel ab.

2. Rollen in der objektorientierten Modellierung

Nach einer allgemeinen Definition des Rollenbegriffs gehen wir auf Eigenschaften von Rollen ein. Kristensen et al. definieren [9]: “a role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects.” Ein Rollenkonzept soll folgende Eigenschaften von Rollen unterstützen [9, 4]:

- *Sichtbarkeit*: Die Sichtbarkeit und der Zugriff auf ein Objekt ist auf Attribute und Methoden der Rolle oder solchen des Objektes selbst beschränkt.
- *Abhängigkeit der Lebensdauer*: Die Lebensdauer einer Rolle hängt von der des Objektes ab, zu dem sie gehört.
- *Schwache Identität*: Rollen haben eine von einem Objekt abhängige Identität und bilden mit diesem eine Einheit. Daraus folgt, daß eine Rolle nicht gleichzeitig zu verschiedenen Objekten gehören kann, sondern zu genau einem.
- *Dynamizität*: Während der Lebenszeit eines Objekt kann es Rollen annehmen und ablegen.
- *Multiplizität*: Mehrere Instanzen eines Rollentyps dürfen gleichzeitig zu einem Objekt existieren.
- *Abstraktheit*: Rollen können zueinander in Aggregations- und Generalisierungsbeziehung stehen.

Um Rollen im Rahmen der Modellierung zu nutzen, haben wir überprüft, inwieweit UML [11] diese Kriterien für ein Rollenkonzept unterstützt [2]. UML-Rollenamen schränken das Verhalten lediglich über ein Interface ein, das von dem zugehörigen Objekt implementiert wird. Derartige Rollen haben keine Attribute und keinen Zustand, wodurch Dynamizität und Multiplizität verhindert werden. Diese Argumente sprechen auch gegen den Ansatz von Steimann [12], Rollen in UML mit Interfaces gleichzusetzen. Eine andere Möglichkeit für Rollen in UML besteht darin, sie durch Vererbung oder Aggregation zu simulieren (siehe auch Kristensen et al. [9]). Bei der Verwendung von Mehrfachvererbung erbt ein Objekt von allen seinen Rollenobjekten. Die Sichtbarkeit von Attributen und Methoden ist damit nicht eingeschränkt, und Rollen können nicht dynamisch angenommen oder abgelegt

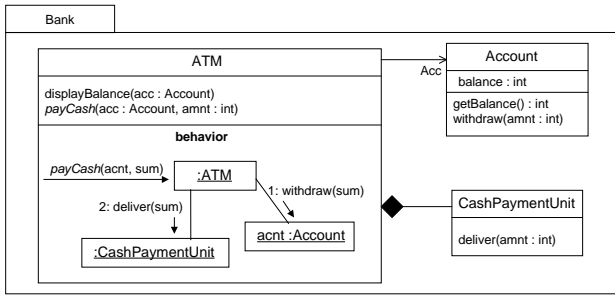


Abbildung 1. Initiales Klassendiagramm einer Bankanwendung

werden. Der nächste Fall ist die Aggregationsbeziehung. Eine Rolle soll von dem Objekt abhängen, zu dem sie gehört. Diese Eigenschaft besitzt gerade die Kompositionsbeziehung, ein Spezialfall der Aggregation. Die Sichtbarkeit von Attributen und Methoden des Basisobjektes innerhalb einer Rolle kann nur indirekt erreicht werden, da Eigenschaften des Basisobjektes nur mittels Navigation erreichbar sind.

Insgesamt gibt es kein vollständiges Rollenkonzept in UML. Die Kompositionsbeziehung erfüllt dabei noch die meisten Kriterien für Rollen. Deshalb wurde in [3, 2] die neue Beziehung *role-of* eingeführt. Die Menge von Operationen, Attributen und Assoziationen einer Klasse in einer Rolle werden einer Rollenklasse zugeordnet, die der Basisklasse über die *role-of* Beziehung zugeordnet ist. Operationen und Attribute einer Rollenklasse sind disjunkt zu denen der Basisklasse. Die Existenz einer Instanz der Rollenklasse hängt von derjenigen des Basisobjektes ab. Strukturell ähnelt die *role-of* Beziehung der Komposition in UML. Das Verhalten der Rollenklasse ändert sich, da auf Merkmale der Basisklasse genauso wie auf Merkmale der Rollenklasse zugegriffen werden kann. Zugriffe auf Attribute und Aufrufe von Methoden der Basisklasse sind auf der Rollenklasse möglich und werden an die Basisklasse *delegiert*. In Klassen- und Objektdiagrammen wird die *role-of* Beziehung durch einen Pfeil mit gefüllter Dreiecksspitze von der Rolle zur Basis angezeigt, siehe Abbildung 3.

Wir setzen Rollen im folgenden dazu ein, separierte Modellbestandteile zu integrieren. Die Sichtbarkeitsregel, die Disjunktheit der Bestandteile der Rolle zu denen des Basisobjektes und die Abhängigkeit einer Rolle von ihrem Basisobjekt sind dabei nutzbare Eigenschaften.

Die Problematik additiver und invasiver Verhaltensergänzung soll für Rollen und in den folgenden Kapiteln für SOD und AOP am Beispiel eines vereinfachten Modells von Geldautomaten (automatic teller machine, ATM) erläutert werden. Abbildung 1 zeigt innerhalb des Paketes (package) *Bank* ein UML-Klassendiagramm. Der Geldautomat wird durch die Klasse *ATM*, die für verschiedene Automatenfunktionen Methoden enthält, modelliert. Daneben ist die Klasse *CashPaymentUnit* für die Steuerung der Bargeldauszahlungseinheit des Geldautomaten zuständig.

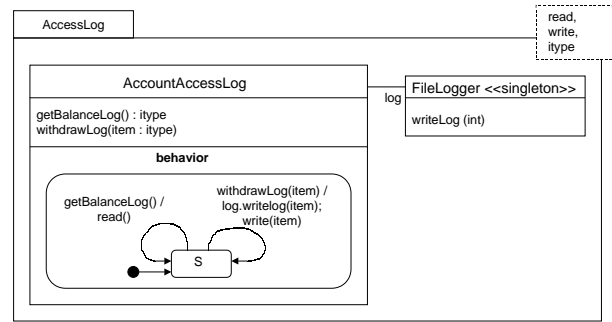


Abbildung 2. Ein Template-Paket für den Aspekt “Logging von Schreibzugriffen”

Die Konten der Kunden sind in der Klasse *Account* modelliert.

Die Klasse *ATM* bietet eine Methode *payCash* an, die für die Auszahlung eines Geldbetrages durch den Geldautomaten aufgerufen wird. Das Verhalten der Methode ist in dem zusätzlichen Bereich (compartment) *behavior* der Klasse *ATM* durch ein Kollaborationsdiagramm spezifiziert. Diese Erweiterung von Klassen ist in UML nicht vorhanden, aber im Rahmen der Erweiterungsmöglichkeiten von UML zulässig. Wie das Kollaborationsdiagramm in Abbildung 1 zeigt, wird die Methode *payCash* abgearbeitet, indem zuerst durch Aufruf der Methode *withdraw* auf einem Objekt der Klasse *Account* Geld vom Konto abgeboben wird. Danach wird die Auszahlung des Betrages durch Aufruf der Methode *deliver* auf einem Objekt der Klasse *CashPayment* initiiert. Diese Methode bewirkt den Auswurf der passenden Bargeldmenge in den Ausgabeschacht des Geldautomaten.

Generische Teilmodelle lassen sich durch UML-Templates [11] für Pakete (packages) beschreiben. Die Pakete enthalten Parameter für Klassen, Methoden, Typen, etc. Die Parameter bilden Verknüpfungsstellen, an denen die Teilmodelle mit den existierenden Modellen verknüpft werden. Gemäß UML werden Template-Parameter in einem gestrichelt umrandeten Rechteck, das auf die obere rechte Ecke des Paketes plaziert ist, angegeben. Klassen der Template-Pakete werden als Rollen mittels der *role-of* Beziehung in das existierende Modell eingefügt, und Template-Parameter werden mit Elementen des existierenden Modells belegt. Dieses Konzept ist bisher im Rahmen der agentenorientierten Modellierung verwendet worden [3]. Die Verwendung in unserem Kontext verdeutlichen wir am Beispiel.

In unserem Beispiel sorgen Methoden der Klasse *AccountAccessLog* im Template-Paket *AccessLog* (Abbildung 2) dafür, daß der Aufruf der Methode *withdrawLog* protokolliert wird. Die Klasse *FileLogger* ermöglicht das Schreiben von Protokolldaten in eine Datei (Logging). Im Zustandsdiagramm (statechart) zur Klasse *AccountAccessLog* ist festgelegt, daß ein Aufruf der Methode *withdrawLog* zu Aufrufen der Methoden *writeLog* und *write* führt. Jeder Schreibvorgang wird also protokolliert, während der Aufruf von *getBa-*

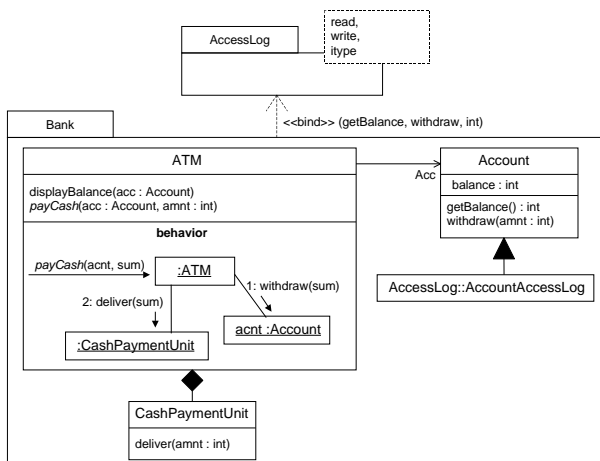


Abbildung 3. Additive Verhaltenserweiterung für das Bankbeispiel

lanceLog nur die Leseoperation read anstößt. Die Methoden write und read sind Parameter, die bei Verwendung des Paketes AccessLog mit Methoden zum Schreiben und Lesen des Datums (item) substituiert werden müssen. Weiterhin ist der Typ des Datums (itype) festzulegen.

Die eben beschriebene Protokollfunktion kann additiv dem Ausgangsmodell in Abbildung 1 hinzugefügt werden. Dazu ist in Abbildung 3 das Template-Paket AccessLog an das Paket Bank durch Belegung der Parameter mit dort vorhandenen Modellelementen angebunden. Die Bindung geschieht durch die spezielle Abhängigkeitsbeziehung <<bind>>, wobei hinter diesem Schlüsselwort eine Liste mit aktuellen Parametern folgt, die für die formalen Parameter in der gegebenen Reihenfolge eingesetzt werden. Die Klasse AccessLog::AccountAccessLog ist durch eine *role-of* Beziehung an die Klasse Account als Rolle angefügt. In Abbildung 4 ist das expandierte Klassendiagramm zu sehen, das dem Diagramm in Abbildung 3 genau entspricht und aus diesem von einem "Modellprozessor" erzeugt werden kann. Die aktuellen Parameter sind **fett** gekennzeichnet.

Die Semantik der *role-of* Beziehung führt dazu, daß Attribute und Verhalten tatsächlich additiv der vorhandenen Klasse Account hinzugefügt werden. Auf der Klasse AccountAccessLog sind nämlich alle Merkmale (features), d.h. Attribute, Assoziationen und Methoden der Klasse Account unmittelbar in der Rolle verwendbar. Zugriffe in einem Rollenobjekt der Klasse AccountAccessLog werden an das zugehörige Basisobjekt der Klasse Account delegiert. Damit wird im Beispiel die Logging-Funktion einem Objekt der Klasse Account additiv hinzugefügt. Damit die neuen Methoden verwendet werden, müssen andere Objekte allerdings auf die Rollenobjekte zugreifen. Vorhandene Zugriffe auf ein Objekt der Klasse Account werden nicht verändert. Dazu wären die aufrufenden Methoden (invasiv) zu verändern.

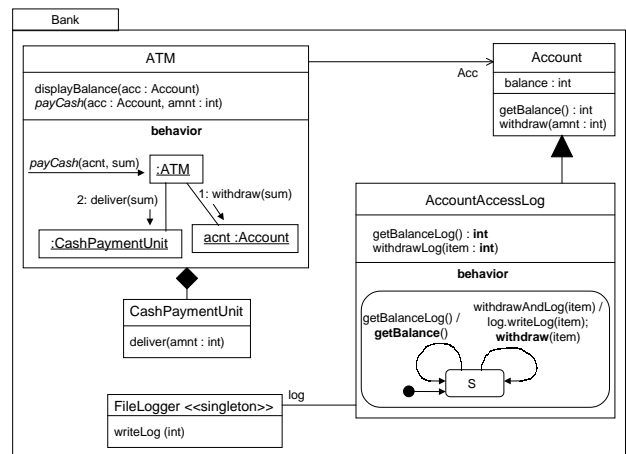


Abbildung 4. Expandiertes Klassendiagramm zu Abbildung 3

An diesem Beispiel ist gezeigt worden, wie Teilmodelle mit Hilfe von Rollen *additiv* zusammengefügt werden können. In den folgenden beiden Abschnitten gehen wir darauf ein, welche Formen invasiver Modellintegration existieren. Dazu betrachten wir zuerst die subjektorientierte Entwicklung und danach das Konzept der Aspektorientierung.

3. Subjektorientierte Entwicklung

Subjektorientierte Programmierung [6] und Modellierung [1] sind weitere Vorschläge zur Trennung von Code und Entwurf hinsichtlich bestimmter Anforderungen (separation of concerns). Das zugrundeliegende Problem ist die im objektorientierten Vorgehen inhärente Notwendigkeit, die funktionalen Anforderungen an ein Softwaresystem durch ein Modell bzw. Code aus Paketen, Interfaces und Klassen mit zugehörigem Verhalten zu erfüllen. In Folge werden, wie in der Einleitung erwähnt, einzelne Funktionen auf verschiedene Klassen (scattering) verteilt und Attribute und Methoden verschiedener Funktionen in einer Klasse gebündelt (tangling). Dadurch wird in Bezug auf die Anforderungen das Ziel der minimalen *Kopplung* zwischen Systemteilen und der maximalen *Kohäsion* in den Teilen nicht erreicht. Der Übergang vom objektorientierten Modell zur Implementierung wird allerdings tendenziell erleichtert.

Subjektorientierte Entwicklung beginnt mit der Gliederung des Entwurfsmodells in *Subjekte*, d.h. in funktionale Einheiten, die streng mit funktionalen Anforderungen an das System korrespondieren und auch überlappen können. Der zugrundeliegende Entwicklungsprozeß ist, wie in [1] dargestellt, dahingehend vereinfacht, daß das Entwurfsmodell direkt aus den Anforderungen gewonnen wird. Auf eine Analyse wird verzichtet. Einzelne Subjekte werden im Entwurf durch Pakete modelliert, die als strukturelles Modell ein Klassendiagramm enthalten. Dabei sind alle Klassen, Assoziationen, Attribute und Methoden angege-

ben, die für die betrachtete funktionale Einheit notwendig sind.

Kompositionsregeln bestimmen, wie zwei Subjekte zu einem gemeinsamen Modell zusammengefügt werden. Zwischen den verschiedenen Paketen zu Subjekten des Entwurfs werden Kompositionsbeziehungen definiert, an denen auch die Regeln annotiert werden, nach denen die Pakete paarweise integriert werden. Die einfachste Methode besteht darin, Elemente mit übereinstimmenden Namen zu integrieren (merge integration). Aus zwei Klassen gleichen Namens zu unterschiedlichen Subjekten wird eine einzige Klasse, in der alle Attribute und Methoden beider Klassen aufgenommen werden. Gleichnamige Attribute und Methoden werden identifiziert. Dieses Vorgehen führt zu einer additiven Verknüpfung von Subjekten, da die integrierten Klassen mindestens das Verhalten der Ausgangsklassen aufweisen. Methoden selbst werden nicht verändert, so daß sich deren Verhalten nicht verändert. Alternativ können Methoden eines Subjekts auch durch Methoden eines anderen überschrieben werden (override integration). Dadurch wird es möglich, das Verhalten von Klassen auf der Ebene der Methodenschnittstelle zu verändern. Als flexible dynamische Methode bietet der subjektorientierte Ansatz an, Kriterien zu spezifizieren, anhand derer zur Laufzeit entschieden wird, welche der zu integrierenden Methoden einer Klasse aufgerufen werden (select integration). Die Kriterien können von Attributwertbelegungen abhängen, wodurch die laufzeitabhängige Entscheidung möglich wird.

Die zuletzt genannten beiden Alternativen zur Verhaltensmodifikation in komponierten Modellen sind invasiver Art. Allerdings findet die Modifikation auf der Ebene der Methodenschnittstelle statt. Das Verhalten *in* den Methoden wird nicht verändert. Das unterscheidet den subjektorientierten Ansatz vom aspektorientierten, bei dem auch das Verhalten der Methoden selbst verändert wird.

Ähnlich wie Rollen werden Subjekte im wesentlichen als Bausteine verwendet, um ein Modell oder ein Programm additiv zu verknüpfen. Rollen bieten im Unterschied zu Subjekten keine Möglichkeit, das Verhalten vorhandener Teilmodelle invasiv zu verändern. In unserem Beispiel in Abbildung 3 werden Merkmale der Klasse `Account` nicht verändert. Unser bisheriges Separationskonzept kann also subjektorientierte Modellierung nicht integrieren.

Im folgenden Abschnitt gehen wir auf die aspektorientierte Programmierung ein und zeigen, wie invasive Verhaltensänderung unterhalb der Methodenschnittstelle möglich ist. Derartige Verhaltensänderungen erlauben subjektorientierte Programmierung und Modellierung nicht.

4. Aspektorientierte Programmierung

Aspekte sind Anforderungen an ein System, die einer oder mehreren Klassen eines Systems zukommen und deren Struktur und Verhalten auf systemati-

sche Weise beeinflussen, wie z.B. die nichtfunktionalen Anforderungen Synchronisation, Tracing, Verteilung, u.a. [8]. Dabei lassen sich klare Schnittstellen finden, an denen die Klassen und die Aspekte interagieren. Dies ermöglicht es, Aspekte getrennt zu spezifizieren. Diese Ideen wurden bereits in neue Programmiersprachen umgesetzt.

Wichtigster Wegbereiter der Aspektorientierten Programmierung ist die Sprache AspectJ von Xerox [13]. Sie unterstützt die Kapselung von additiven und invasiven Anforderungen in einer neuen Art Modul namens *Aspect*. Die hierin verwendete Programmiersprache basiert auf einer Untermenge von Java, ergänzt um Konstrukte zur Interaktion von Aspekten mit Java-Klassen. Um ausführbaren Code zu bekommen, werden Aspekte von einem Precompiler in die Java-Klassen "eingewebt".

Die Stellen, an denen Aspekte mit dem normalen Javacode interagieren können, nennt man *Join Points*. Dazu zählen Methodenaufrufe, angenommene Methodenaufrufe, Methodenausführung, Konstruktoren, Ausnahmebehandlung und noch ein paar andere. Im Aspekt muß festgelegt werden, für welche Join Points zusätzliche Anweisungen spezifiziert werden. Dies geschieht jeweils für eine Menge von Join Points, die als *Pointcuts* bezeichnet wird. Für einen Pointcut wird in einem sogenannten *advice*-Block mittels Javacode spezifiziert, was passiert, wenn ein Join Point des Pointcuts bei der Programmausführung erreicht wird. Soll zusätzlicher Javacode vor dem Join Point ausgeführt werden, muss er als *before-advice* definiert werden, für nachträgliche Ausführung als *after-advice*. Dieser Mechanismus erstellt Aspekte für invasive Anforderungen.

Für die additive Form des Aspekts werden mittels eines *introduction*-Blocks existierende Javaklassen um Felder, Methoden, Konstruktoren und Interfaces erweitert.

Für unser Beispiel untersuchen wir, wie man die Modellierung aus Abbildung 3 in AspectJ umsetzen kann. Die Logging-Funktionalität zu der Klasse `Account` ist zu ergänzen, aber analog der Rolle als Einheit gekapselt. Die Ausgangsklasse soll nicht verändert werden. Die Klienten sollen über die Rolle auf das Objekt zugreifen, was auf den Zugriff über ein neues Interface abgebildet wird:

```
interface AccountAccessLog {
    public int getBalanceLog();
    public withdrawLog(amnt: int);
}
```

Das Interface `AccountAccessLog` wird der Klasse `Account` in einem Aspekt zugewiesen. Der Aspekt benutzt analog der Rolle die Klasse `FileLogger` und spezifiziert die neuen Methoden der Rolle. Während `getBalanceLog` nur auf `getBalance` umgelenkt wird, schreibt `withdrawLog` das Log bevor sie `withdraw` aufruft. Zur Laufzeit gibt es für jedes Konto (`Account`) eine Aspektinstanz. Mit diesem Aspekt wird die Implementierung

des durch die Rolle zusätzlich gegebenen Verhaltens erreicht, wie es in Abbildung 4 dargestellt ist.

```

aspect AccountAccessLog {
  introduction Account {
    implements AccountAccessLog;

    static myLog =
      FileLogger('myAccount.log');

    void withdrawLog(int i){
      myLog.writeLog(i);
      withdraw(i);
    }

    int getBalanceLog(){
      return getBalance();
    }
  }
}

```

Weiter sieht das Beispiel vor, daß die Klasse ATM, wann immer sie die Methoden von Account aufruft, diese durch die Methoden von AccountAccessLog ersetzt. Wir können einen Klienten von Account aber nicht durch Aspekte automatisch verändern. Aspekte können nur den Code der aufgerufenen Klasse erweitern, nicht aber den der Aufrufer. Möglich wäre nur, für alle Klienten von Account die Aufrufe umzulenken.

Es fehlt also bei AspectJ die Unterstützung des Konzepts, daß Aspekte in Abhängigkeit von Aufrufer und Aufgerufenem wirksam werden. In der Sprache Sally [5] wird genau dieses unterstützt. Pointcuts sind hier Interaktionen, d.h. Kombinationen aus Aufrufer und Aufrufendem, und werden als Paare von Join Points gebildet.

Wir diskutieren kurz das Verhältnis von Aspektorientierter Programmierung (AOP) und Rollenkonzepten. Wenn Objekte als Komponenten aufgefaßt werden, können Rollen zur Repräsentation von Eigenschaften verschiedener Objekttypen eingesetzt werden. Allerdings findet man bei Rollen nicht das Verändern von Verhalten analog dem invasiven Einweben von Aspektcode, sondern nur das Hinzufügen von weiteren Eigenschaften analog dem additiven Einweben von Aspektcode, wie oben am Beispiel der Auslagerung der Logging-Funktion in einen Aspekt gezeigt.

AOP erlaubt im Unterschied zur subjektorientierten Entwicklung auch invasive Verhaltensänderungen unterhalb der Methodenschnittstelle. Wie Eingriffe in das Verhalten nicht nur auf der Seite des Aufgerufenen sondern auch auf der Seite des Aufrufers modelliert werden können, zeigen wir im folgenden.

5. Aspektorientierte Modellierung

Wir haben gerade gesehen, daß AOP nicht in der Lage ist, Klienten an den neuen Aspekt anzupassen, was auf Modellierungsebene bedeutet, das Objekt

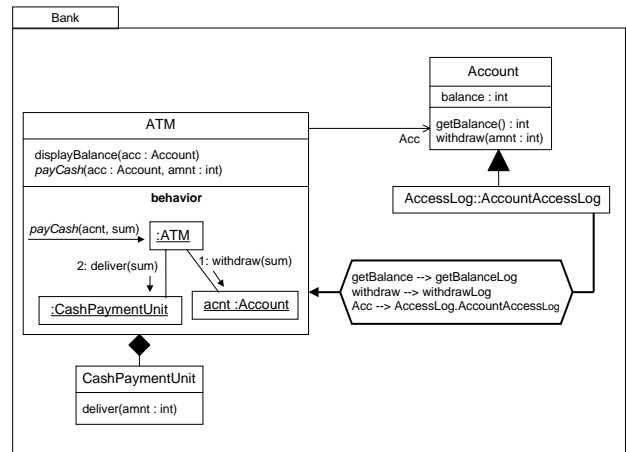


Abbildung 5. Invasive Verhaltensänderung durch Einweben eines Aspektes

über die neue Rolle anzusprechen. Auch im Rollenkonzept wird dies noch nicht unterstützt. Daher soll hier eine Modellierung vorgestellt werden, die es erlaubt, die Nutzung der Rolle als eine eigene Anforderung modular zu spezifizieren. Unser Vorschlag zur aspektorientierten Modellierung, der speziell auch die Möglichkeit zur subjektorientierten Modellierung einschließt.

Invasives Verhalten erfordert, die Verhaltensspezifikation von Methoden selbst zu ändern, damit bei Aufrufen verändertes Verhalten wirksam wird. Zu diesem Zweck spezifizieren wir eine Modelltransformation durch eine *aspect-of* Beziehung. Diese Beziehung wird durch ein Sechseck notiert (siehe 5), in dem die struktur- und verhaltensändernde Transformation definiert ist. Das Sechseck wird mit einem Pfeil mit der Klasse verbunden, deren Attribute, Methoden und Assoziationen durch den Webvorgang betroffen sind. Es wird mit einer einfachen Kante mit der Klasse verbunden, in der der invasive Aspekt spezifiziert ist. In Abbildung 5 ist der invasive Aspekt für die Klassen ATM und AccountAccessLog spezifiziert. Die Transformationsvorschrift im Sechseck beschreibt die Umlenkung bestehender Aufrufe auf die neuen Methoden.

Das Weben (weaving) von Aspekt und Modell liefert ein Zielmodell, das dadurch entsteht, daß die spezifizierten Transformationen angewendet werden. In Abbildung 6 ist das Klassendiagramm zu sehen, das aus der Transformation entsprechend Abbildung 5 entsteht. Im Kollaborationsdiagramm zur Methode payCash ist die Methode withdraw durch die Methode withdrawLog ersetzt worden. Die Assoziation auf die Klasse Account ist in eine zur Klasse AccountAccessLog geändert worden. Jetzt werden alle Auszahlungen des Geldautomaten protokolliert.

Die angegebene Spezifikation von Aspekten baut auf dem in Kapitel 2 beschriebenen Rollenkonzept auf. Trotz der Umlenkung von Assoziationen (hier Acc) sind Attribute, Assoziationen und Methoden der ursprünglichen Zielklasse (hier Account) weiterhin ver-

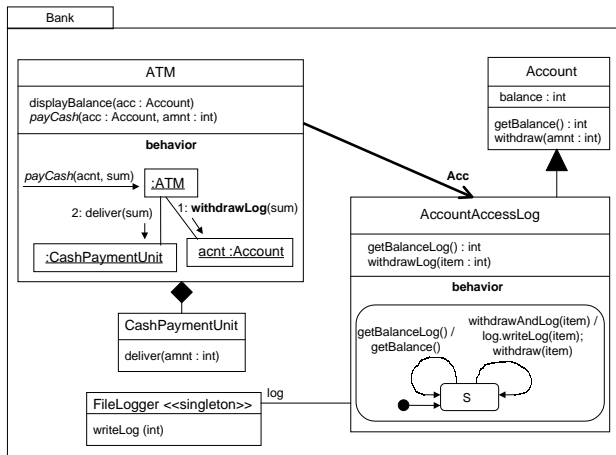


Abbildung 6. Finales expandiertes Klassendiagramm

wendbar, da entsprechende Zugriffe entlang der *role-of* Beziehung delegiert werden.

Die in der subjektorientierten Entwicklung möglichen Formen der invasiven Verhaltensintegration können durch spezielle Transformationen bewirkt werden. Beispielsweise kann das Überschreiben von Methoden (override integration) dadurch erreicht werden, daß das *komplette* Verhalten einer Methode durch dasjenige einer Methode einer anderen Klasse ersetzt wird.

6. Resümee

Wir haben gezeigt, daß Rollen nur die additive Integration von Teilmodellen zu einem Gesamtmodell unterstützen, während Subjekte eine einfache, auf der Ebene der Methodenschnittstelle wirkende, invasive Verhaltensänderung erlauben. Aspektorientierung verändert auch Methodenrümpfe, allerdings werden nur die aufgerufenen Methoden modifiziert. Wir haben ein Konzept zur aspektorientierten Modellierung vorgestellt, das auch auf der Seite der aufrufenden Methoden eine Modifikation des Verhaltens erlaubt. Unser Konzept wird durch eine UML-basierte aspektorientierte Modellierungssprache unterstützt. Um diese neuen Konzepte in der Modellierung einzusetzen bedarf es der Entwicklung passender Entwurfsmethoden.

Bisher wurde nur die Verwendung von Template-Paketen mit einzelnen bzw. von einander völlig unabhängigen Rollen und mit Hilfsklassen wie dem *FileLogger* untersucht. Dies soll in Zukunft auf die Verwendung von Rollensystemen in Template-Paketen erweitert werden. Hier ist die Frage, wie spezifiziert werden kann, daß die Verwendung einer Rolle die einer anderen nachzieht, voraussetzt oder ausschließt, und wie dies für Modelle überprüft werden kann.

Unser Vorschlag zur aspektorientierten Modellierung hat Gemeinsamkeiten mit Verfahren zur Integration von Komponenten. Beispielsweise dienen in [10] *pluggable composite adapters* der dynamischen Inte-

gration von Komponenten. Eine Schicht aus Adaptern modifiziert, ähnlich wie unsere Modelltransformation, das Verhalten der zu integrierenden Komponenten. Gemeinsamkeiten und Unterschiede derartiger Verfahren zu unserem Vorschlag sind noch zu untersuchen.

Unsere Diskussion der Modellierungsansätze bezieht sich bisher rein auf die syntaktische Ebene. Offen ist noch die Frage nach der *Konsistenz* bei der Integration von Verhalten, was in Ansätzen wie SOD und Rollen bisher nur sehr spärlich behandelt wird.

Literatur

- [1] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proc. OOPSLA '99*. ACM Press, 1999.
- [2] R. Depke, G. Engels, and J. M. Küster. On the integration of roles in the UML. Technical Report No. 214, University of Paderborn, Dep. of Comp. Sci., Aug. 2000. <http://www.upb.de/cs/ag-engels/Papers/2000/DepkeTR214.pdf>.
- [3] R. Depke, R. Heckel, and J. M. Küster. Roles in agent-oriented modeling. *Int. Journal on Software Engineering and Knowledge Engineering*. To appear.
- [4] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.
- [5] S. Hanenberg. Sally language issues. 2001. <http://www.cs.uni-essen.de/dawis/research/aop/sally/>.
- [6] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In A. Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28, pages 411–428. ACM Press, 1993.
- [7] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Feb. 1995.
- [8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maleda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on object-oriented programming (ECOOP)*, Springer LNCS 1241, Berlin, 1997.
- [9] B. B. Kristensen and K. Østerbye. Roles: Conceptual Abstraction Theory and Practical Language Issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.
- [10] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.
- [11] Object Management Group. UML specification version 1.3, June 1999. <http://www.omg.org>.
- [12] F. Steimann. A radical revision of UML's role concept. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML'2000*, number 1939 in Lecture Notes in Computer Science. Springer-Verlag, 2000.
- [13] Xerox PARC. AspectJ/COOL language specification. 2001. <http://www.aspectj.org>.