TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Modelle und Techniken für eine effiziente und lückenlose Zugriffskontrolle in Java-basierten betrieblichen Anwendungen

Dipl.-Ing. Kathrin Lehmann
Lehrstuhl für Software Engineering betrieblicher Informationssysteme
Institut für Informatik
Technische Universität München

Lehrstuhl für Software Engineering betrieblicher Informationssysteme
Institut für Informatik
Technische Universität München

Modelle und Techniken für eine effiziente und lückenlose Zugriffskontrolle in Java-basierten betrieblichen Anwendungen

Kathrin Lehmann

Vollständiger Ausdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Alfons Kemper, Ph. D.

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Florian Matthes

2. Univ.-Prof. Dr. Helmut Seidl

Die Dissertation wurde am 31.05.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.11.2007 angenommen.

Zusammenfassung

Bei der Entwicklung betrieblicher Anwendungen wird der Aspekt der Sicherheit häufig vernachlässigt. Die Implementierung von Sicherheitslösungen wendet zunächst nur potentielle Schadenseintritte ab, eine direkte Gegenüberstellung von Kosten und Nutzen ist nicht möglich. So werden Anforderungen an die Sicherheit einer Anwendung häufig durch Adhoc-Lösungen implementiert.

Im Fall der Zugriffskontrolle bedeutet so eine Adhoc-Lösung, dass der Entwickler der Anwendung Zugriffskontrollüberprüfungen in den Quellcode der Anwendung platziert, wo immer es notwendig erscheint. Diese unsystematische Vorgehensweise erschwert die Änderbarkeit, die Wartung und die Überprüfung der Zugriffskontrolle auf Vollständigkeit und Korrektheit.

In dieser Arbeit wird ein systematisches Vorgehen für die Integration von Zugriffskontrollfunktionalität in eine betriebliche Anwendung entwickelt. Eine Architektur für die Zugriffskontrolle entkoppelt die Definition der Berechtigungspolicy von der Durchsetzung derselben innerhalb der Anwendung. Anpassungen der Zugriffskontrollanforderungen, d. h. Änderungen und Konfiguration definierter Berechtigungen, können flexibel und unabhängig von der eigentlichen Funktionalität der Anwendung vorgenommen werden. Der Aufwand für die Anpassungen kann gering gehalten werden.

Für die Definition der Berechtigungspolicy wird die regelbasierte, deklarative Berechtigungsbeschreibungssprache *PathExpressions* entwickelt, welche auf die Zugriffskontrollanforderungen betrieblicher Anwendungen zugeschnitten ist. Sie kann sowohl benutzerbestimmbare als auch rollenbasierte Berechtigungen darstellen. Die Verwendung von deklarativ spezifizierten Regeln für die Spezifikation der Berechtigungen ermöglicht eine zentrale Sicht auf alle Berechtigungen innerhalb der Anwendung.

Berechtigungen beziehen sich auf die in der Anwendung verwalteten Geschäftsobjekte und können auch abhängig von den Beziehungen der Geschäftsobjekte untereinander sein. Bei Änderungen des Datenmodells, welches die Geschäftsobjekttypen und ihre Beziehungen abbildet, müssen auch die Berechtigungen aktualisiert werden. Bisher wurden deklarativ spezifizierte Berechtigungspolicies entkoppelt von der Anwendung erstellt und in einem von der Anwendung unabhängigen Dokument oder einer unabhängigen Komponente abgelegt. Die Subjekte, Objekte und Aktionen, die in den Zugriffsregeln spezifiziert werden, müssen auf Subjekte, Objekte und Aktionen innerhalb der Anwendung abgebildet werden und umgekehrt. Hierfür muss auf Objektidentifier oder ähnliche Referenztechniken zugegriffen werden. Die Objektidentifier bestehen aus Zeichenketten und unterliegen keiner Typüberprüfung. Weiterhin ist nicht gewährleistet, dass die verwendeten Objektidentifier tatsächlich in der Anwendung existieren. Die PathExpressions lösen dieses Problem. Die spezifizierten Berechtigungen werden direkt mit dem Datenmodell der Anwendung verknüpft. So ist es einfach, die Zugriffsregeln auch bei Änderung des Datenmodells konsistent mit den in der Anwendung verwalteten Geschäftsobjekttypen zu halten.

Die Durchsetzung der Zugriffsregeln erfolgt direkt im Quellcode der Anwendung. Die Zugriffskontrolldurchsetzung gehört zu den so genannten Querschnittsaspekten, d. h. der Quellcode für die Zugriffsdurchsetzung ist über die gesamte Anwendung verstreut. Ein systematischer Ansatz für die Einbettung von Zugriffskontrollabfragen fehlt bislang noch. In dieser Arbeit wird ein Framework entwickelt, welches dedizierte Eingriffspunkte für die Berechtigungsüberprüfung für jeden Dienst der betrieblichen Anwendung vorsieht. Weiterhin können auch die Antworten, welche ein Dienst einem

anfragenden Client zurückliefert, konfiguriert und gefiltert werden, so dass der Client nur die Teile der Antwortnachricht erhält, für die er leseberechtigt ist.

Wenn der Entwickler Quellcode für die Zugriffsdurchsetzung per Hand in die Anwendung integriert, so ist die Korrektheit und Vollständigkeit der Berechtigungsüberprüfungen nicht gewährleistet. In dieser Arbeit wird der Ansatz verfolgt, die Zugriffskontrollinformationen automatisch aus dem Quellcode zu inferieren. Hierfür werden Techniken der statischen Quellcodeanalyse und der abstrakten Interpretation verwendet. Für jeden Dienst der betrieblichen Anwendung wird die Datenstruktur *AccessTypes* erstellt, welche alle notwendigen Informationen enthält, um Quellcode für die Zugriffsdurchsetzung automatisch in die vom Framework bereitgestellten Eingriffspunkte zu generieren. Die AccessTypes stellen quasi eine Zusammenfassung der im Quellcode vorkommenden, zugriffsgeschützten Anweisungen dar. Die geschützten Anweisungen sind jeweils mit Zugriffsberechtigungen verknüpft. Die automatische Generierung von Quellcode für die Zugriffsdurchsetzung sorgt dahingehend für eine effiziente Zugriffsüberprüfung, als dass doppelte Überprüfungen vermieden werden. Jede Zugriffskontrollüberprüfung ist in der Regel mit einer kostspieligen Datenbankabfrage verbunden, insbesondere, wenn aufgrund der Komplexität der Anfragen eine Caching-Strategie bereits durchgeführter Anfragen schwierig zu realisieren ist. Wiederholte, unnötige Berechtigungsüberprüfungen führen zu Performanzeinbußen. Ebenso wird durch die statische Codeanalyse eine lückenlose Zugriffskontrolle gewährleistet, d. h. es gibt keinen Ausführungspfad innerhalb der betrieblichen Anwendung, welcher nicht durch die Zugriffskontrolle abgedeckt wird. Ansonsten könnte es Ausführungshistorien geben, die zum Verlust der Vertraulichkeit und / oder der Integrität der in der Anwendung verwalteten Daten führen.

Es wurde eine prototypische Umsetzung der Zugriffsentscheidungskomponente, des Werkzeugs zur Integration der Zugriffsdurchsetzung sowie der für die Zugriffskontrolle erweiterten Architektur in einem Java-Framework für betriebliche Anwendungen implementiert.

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen bedanken, die mich während meiner Promotion unterstützt haben. Mein herzlicher Dank gilt Florian Matthes für die Möglichkeit am sebis-Lehrstuhl zu arbeiten. Ihm möchte ich für die Bereitstellung des Themas, für die Betreuung und die vielen hilfreichen Kommentare und Verbesserungsvorschläge beim Entstehen dieser Arbeit danken. Helmut Seidl danke ich für das Interesse am Thema und das Übernehmen der Zweitbetreuung.

Ebenso möchte ich mich bei Peter Thiemann für die Zusammenarbeit und die wertvolle Einweisung in die abstrakte Interpretation bedanken.

Mein Dank gilt auch Vanda Lehel für die gute Zusammenarbeit am Lehrstuhl, Thomas Büchner für die anregenden Diskussionen, die für meine Arbeit sehr hilfreich waren, Michael Schermann, Martin Wimmer und Wolfgang Wörndl für die Anregungen im Rahmen unseres inoffiziellen Sicherheitskolloquiums.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Betriebliche Anwendungen verwalten große Datenbestände. Hierbei handelt es sich sowohl um Daten zu den unternehmensinternen Assets wie Personal-, Produkt- und Leistungsdaten als auch um Verwaltungsdaten zu Kunden und Lieferanten. Die Informationen, die diese Daten darstellen, bilden in der Informationsgesellschaft einen Anteil am Vermögenswert eines Unternehmens. Information ist ein Wirtschaftsgut, dessen Nutzung in einem Unternehmen unverzichtbar ist.

Betriebliche Anwendungen stellen dem Anwender Dienste für die Informationsnutzung zur Verfügung. Die Anwendungen sind zunehmend untereinander vernetzt und über das Internet verfügbar, so dass Anwender ortsungebunden Zugriff auf notwendige Informationen erhalten können, um so schnell und flexibel agieren zu können. Im Zuge der immer weiter reichenden Vernetzung und Zugriffsmöglichkeiten auf die Dienste ergibt sich ein hoher Bedarf an Informationssicherheit. Es ist ein Informations-Recht gefordert, welches Konflikte auflöst, die sich durch die Nutzung der Information ergeben [Spi85].

Betriebliche Anwendungen müssen also hohe Anforderungen an die Zugriffskontrolle erfüllen, um die Integrität und Vertraulichkeit der verwalteten Daten und damit der verwalteten Information zu gewährleisten. Die Anforderungen an die zu implementierende Zugriffskontrolle sind häufig komplex. Erwähnt sei hier das zentrale R/3-Modul der zurzeit marktführenden betrieblichen Standardsoftware SAP. Es sieht im Release 4.6C bereits 900 verschiedene Berechtigungsobjekte vor [LfD02]. Obwohl das R/3-Berechtigungskonzept eine große Flexibilität bei der Gestaltung der Berechtigungen bietet, so ist es „gerade die außergewöhnliche Komplexität des Konzepts, die seine Schwachstellen ausmacht und die in der Fachöffentlichkeit zur Diskussion über die informationstechnische Sicherheit der verbreiteten Standardsoftware geführt hat“ [JBD98].

Die Informationssicherheit, und hier insbesondere die Zugriffskontrolle, ist schon einige Jahrzehnte Gegenstand der Forschung. Es ergibt sich immer wieder neuer Forschungsbedarf, da sich die Nutzungsszenarien computergestützter Systeme und Anwendungen ständig ändern und weiter entwickeln.

1.2 Ziel und Problemstellung

Die Zugriffskontrolle für betriebliche Anwendungen ist häufig unzureichend realisiert. Unternehmen sind an einer Optimierung des Kosten-Nutzen-Verhältnisses interessiert. Die Implementierung von Maßnahmen zur Erhöhung der Sicherheit eines Systems kostet zunächst nur Aufwand, Zeit und Geld und bringt keinen direkten bzw. unmittelbaren Nutzen für das Kerngeschäft eines Unternehmens. Deshalb wird die Realisierung von Sicherheitsmaßnahmen häufig vernachlässigt. Hierbei wird aber übersehen, dass im Falle der Ausnutzung einer Sicherheitslücke die Kosten für die Beseitigung des entstandenen Schadens sehr viel höher sein können, als die Kosten einer adäquaten Sicherheitslösung

zum Zeitpunkt der Entwicklung des Systems. Diese Arbeit soll einen Beitrag zur Modellierung einer verbesserten Zugriffskontrollarchitektur und einer verbesserten Durchsetzung der Zugriffskontrolle leisten, als dies in zurzeit verwendeten betrieblichen Anwendungen der Fall ist.

Konzeptuelle Fehler, Implementierungsfehler und Sicherheitslücken treten in jeder Phase des Softwareentwicklungsprozesses auf. Je später sie entdeckt werden, desto kostspieliger ist ihre Beseitigung ([Bal00], [Som05]). Besonders kostspielig ist die Beseitigung von Fehlern nach dem Testen und der Auslieferung einer Software. Bei der Realisierung der Zugriffskontrolle einer Anwendung kommt es häufig zur Vermischung des Quellcodes, der die eigentliche Funktionalität bereitstellt, mit dem Quellcode, der der Zugriffskontrolle dient. In Abb. 1.1 ist eine betriebliche Anwendung mit einem solchen vermischten Quellcode schematisch dargestellt. Ein Client greift über die von der betrieblichen Anwendung angebotenen Schnittstellen, die jeweils einen Dienst bereitstellen, auf die betriebliche Anwendung zu. Während der Ausführung der Transaktion für einen Dienst finden an vielen Stellen im Quellcode Berechtigungsüberprüfungen statt. Die Einbettung der Zugriffskontrollüberprüfung ist unsystematisch über den Quellcode verteilt. Häufig wird der Code für die Zugriffskontrolle erst nachträglich in den Quellcode eingebaut, dort, wo es dem Entwickler gerade sinnvoll erscheint. Durch diese Unübersichtlichkeit des Quellcodes werden sowohl das Testen der reinen Funktionalität der Software als auch die Beseitigung fehlerhafter Zugriffskontrollfunktionalität erschwert. Ebenso sind die Wartbarkeit und die Änderbarkeit einer bestehenden Anwendung nur eingeschränkt oder nur mit hohem Aufwand realisierbar.

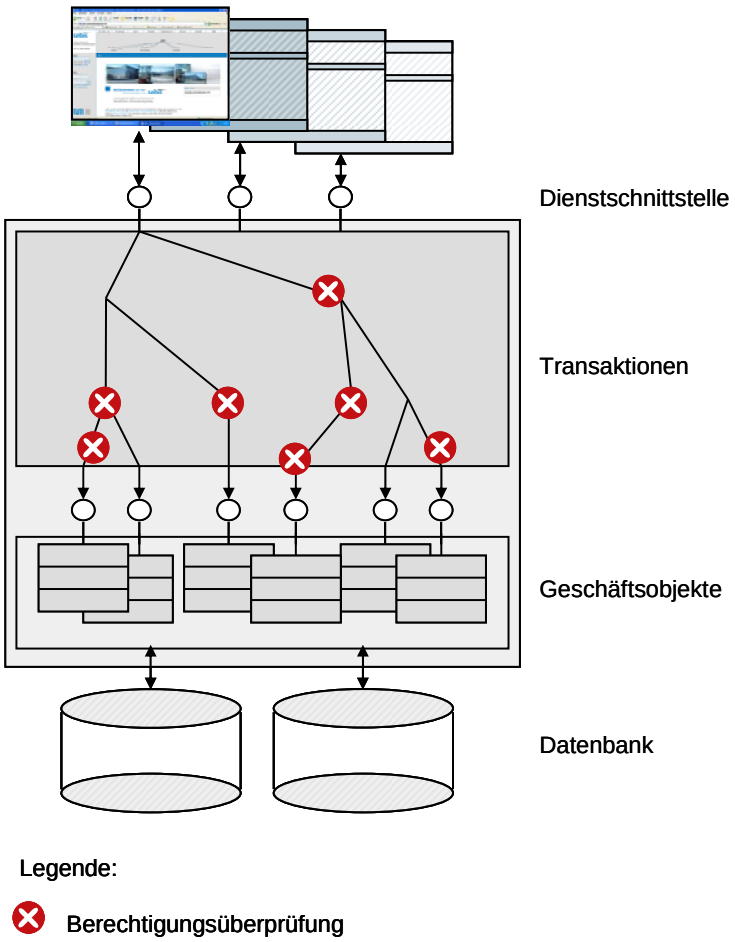


Abb. 1.1: Unsystematische Einbettung von Zugriffskontrollüberprüfungen

Für die Zugriffskontrolle kann hier eine Verbesserung erreicht werden, indem die Architektur einer betrieblichen Anwendung Elemente für die Einbettung der Zugriffskontrolle vorsieht. Die ISO-Norm 10181-3 (siehe Kapitel 2.4) stellt hierfür eine konzeptuelle Architektur bereit, die die Trennung von Zugriffsentscheidung und Zugriffsdurchsetzung vorsieht. Diese Trennung erhöht die Übersichtlichkeit

und die Konfigurierbarkeit von verwalteten und im System durchzusetzenden Berechtigungen. Das Ziel dieser Arbeit ist es, eine Architektur für betriebliche Anwendungen zu konkretisieren.

Weiterhin soll der Code zur Durchsetzung der Zugriffskontrolle unabhängig von dem Quellcode der eigentlichen Anwendung erzeugt und in die Anwendung integriert werden können. Denn betriebliche Standardanwendungen werden vor ihrer Auslieferung an verschiedene Kundenwünsche angepasst. Jeder Kunde hat hier andere Vorstellungen und Wünsche bezüglich der Zugriffskontrolle. Verschiedene Installationen derselben Software müssen für verschiedene Zugriffskontrollanforderungen konfiguriert werden können. Das Ziel ist es also, eine Architektur zu entwickeln, bei der die Spezifikation von Berechtigungen für die Zugriffskontrolle flexibel und unabhängig von den Mechanismen für die Durchsetzung der Zugriffskontrolle innerhalb der Anwendung gestaltet werden kann. Um sicherzustellen, dass die Einbettung von Quellcode für die Zugriffskontrolldurchsetzung korrekt ist, soll hier ein Verfahren entwickelt werden, mit dem diese Einbettung automatisch und ohne Mehraufwand für den Entwickler einer Anwendung durchführbar ist. Die Einbettung soll hierbei sowohl effizient als auch vollständig sein. Effizient bedeutet hier, dass doppelte Überprüfungen aus Performanzüberlegungen heraus vermieden werden sollen. Vollständig bedeutet hier, dass es keinen Ausführungspfad innerhalb der Anwendung gibt, der nicht der Zugriffskontrolle unterliegt.

Insgesamt sind die Beiträge dieser Arbeit

- ein Vorgehensmodell, welches die Konfiguration und Integration von Berechtigungen in bestehende und neu entwickelte Anwendungen erlaubt.
- eine Berechtigungsbeschreibungssprache, welche auf die Anforderungen an die Ausdrucksmächtigkeit der in einer betrieblichen Anwendung benötigten Berechtigungen abgestimmt ist und auch bei Änderung des Datenmodells der Anwendung konsistent mit den in der Anwendung verwalteten Geschäftsobjekttypen gehalten werden kann.
- eine Architektur für die Zugriffskontrolle, welche die Anforderungen an die Konfigurierbarkeit von Dienstanfrage und Dienstantwort sowie einer eventuell verwendeten Benutzeroberfläche realisiert.
- ein Algorithmus, welcher die automatische und systematische Einbettung von Quellcode für die Zugriffskontrollüberprüfung für die einzelnen Dienste einer datenzentrierten und dienstbasierten Anwendung erlaubt.
- eine prototypische Implementierung, welche die Realisierbarkeit oben genannter Konzepte, dies sind die Berechtigungsbeschreibungssprache, die Architektur und der Algorithmus für die statische Quellcodeanalyse, demonstriert.

1.3 Aufbau der Arbeit

Kapitel 2 dient der grundlegenden Begriffsklärung und der Einordnung der Zugriffskontrolle in das Umfeld der IT-Sicherheit. Hier werden sowohl die einzelnen Phasen des Sicherheitsengineering-Prozesses, die an die Phasen des allgemeinen Softwareengineering-Prozesses angelehnt sind, als auch Berechtigungsmodelle vorgestellt, die sich für betriebliche Anwendungen etabliert haben. Weiterhin wird die oben angesprochene konzeptuelle Sicherheitsarchitektur für die Zugriffskontrolle der ISO-Norm 101081-3 vorgestellt. Sie dient als Grundlage für die in dieser Arbeit entwickelte Sicherheitsarchitektur.

Kapitel 3 stellt ein allgemeines Modell für die in dieser Arbeit untersuchte Klasse von Anwendungen auf. Die Architektur dieses Modells ist datenzentriert und dienstbasiert. Das Modell deckt ein breites Spektrum an betrieblichen Anwendungen ab, denn mit diesem Modell können die typischen Charakteristika betrieblicher Anwendungen abgebildet werden. Es wird gezeigt, dass sich die Dienste, die eine betriebliche Anwendung nach außen hin anbietet, in eine Handvoll verschiedener Diensttypen klassifizieren lassen.

Kapitel 4 beschreibt die Analysephase des in Kapitel 2 erläuterten Sicherheitsengineering-Prozesses. Es werden die in dieser Arbeit verfolgten Schutzziele und ein geeignetes Bedrohungsmodell aufgestellt. Auf Basis der Analyse einer Beispielanwendung sowie anderer betrieblicher Anwendungen werden die Anforderungen an die Zugriffskontrolle einer betrieblichen Anwendung aus Sicht des Benutzers erstellt. Weiterhin wird ein Vorgehensmodell vorgestellt, welches sowohl die

Konfigurierbarkeit der für eine Anwendung spezifizierten Berechtigungen als auch die flexible Integration von Quellcode für die Berechtigungsdurchsetzung ermöglicht. Es werden bestehende Java-Bibliotheken und -Frameworks auf ihre Zugriffskontrollmechanismen hin untersucht. Die bestehenden Mechanismen sind nicht ausreichend, um die Ziele dieser Arbeit zu erreichen. Das Kapitel schließt mit einer Vorstellung der wichtigsten verwandten Arbeiten und Ansätze.

Kapitel 5 leitet die Entwurfsphase des Sicherheitsengineering-Prozesses ein. Es bildet den ersten Kernteil der Arbeit – die Spezifikation einer Berechtigungspolicy für betriebliche Anwendungen. Zunächst werden der Begriff der Berechtigung selbst sowie die einzelnen Bausteine einer Berechtigung eingehend präzisiert und klassifiziert. Weiterhin wird auf gängige Mechanismen und Policysprachen für die Formulierung von Berechtigungen eingegangen und die Schwächen bestehender regelbasierter Berechtigungsbeschreibungssprachen werden aufgezeigt. Der Hauptteil des Kapitels widmet sich der in dieser Arbeit entwickelten Berechtigungsbeschreibungssprache *PathExpressions*, die den diskutierten Schwächen gängiger regelbasierter Beschreibungssprachen begegnet. Das Kapitel enthält auch Erläuterungen zu der prototypischen Implementierung der PathExpressions.

Kapitel 6 bildet den zweiten Kernteil der Arbeit – die Durchsetzung der Zugriffskontrolle mittels einer geeigneten Architektur und eines Algorithmus, welcher die systematische Einbettung von Quellcode für die Berechtigungsdurchsetzung in eine Anwendung erlaubt. Das Kapitel gliedert sich in zwei Hauptteile. Im ersten Teil wird ein allgemeines Framework für die Zugriffskontrolle entwickelt. Es sieht mehrere Eingriffspunkte vor, in die Quellcode für die Durchsetzung von Berechtigungen eingebettet werden kann. Es wird gezeigt, dass dieses Framework die Zugriffskontrollanforderungen an die verschiedenen Diensttypen, die in Kapitel 3 herausgearbeitet wurden, realisieren kann. Im zweiten Teil wird ein Algorithmus vorgestellt, welcher auf Basis einer statischen Quellcodeanalyse und der abstrakten Interpretation die berechtigungsrelevanten Anweisungen, die während einer Diensterbringung ausgeführt werden, identifiziert und für jeden angebotenen Dienst bündelt. Die Bündelung der Berechtigungen findet auf Basis der in dieser Arbeit entwickelten Datenstruktur *AccessTypes* statt. Aus den AccessTypes kann automatisch oder semi-automatisch der Quellcode für die Berechtigungsdurchsetzung eines Dienstes in die vom Framework vorgesehenen Eingriffspunkte generiert werden. Der Autorin ist kein anderer bestehender Ansatz bekannt, mit dem eine automatische und systematische Integration von Berechtigungsüberprüfungen in den Quellcode einer Anwendung möglich ist.

Die Arbeit schließt mit einem Fazit und einem Ausblick auf verwandte Forschungsthemen wie der Informationsflusskontrolle, der Berechtigungsadministration und der Durchsetzung von Berechtigungen in verteilten Umgebungen.

2 Stand der Forschung in der Zugriffskontrolle

Diese Arbeit beschäftigt sich mit der Zugriffskontrolle. In Abschnitt 2.1 wird zunächst eine Einordnung der Zugriffskontrolle in das Umfeld der IT-Sicherheit vorgenommen. Die Entwicklung von Modellen und Techniken für eine effiziente und lückenlose Zugriffskontrolle in Java-basierten Anwendungen wird in einen Sicherheitsengineering-Prozess eingebettet. Dieser ist ein Softwareengineering-Prozess, welcher auf die speziellen Anforderungen eingeht, die für die Realisierung von Sicherheitszielen erfüllt werden müssen. Der Sicherheitsengineering-Prozess wird in Abschnitt 2.2 umrissen. Innerhalb der Entwurfsphase des Sicherheitsengineering werden für die Zugriffskontrolle geeignete Berechtigungsmodelle sowie Sicherheitsarchitekturen, die die Zugriffskontrolle abbilden, herangezogen. Die für betriebliche Anwendungen gängigen Berechtigungsmodelle werden in Abschnitt 2.3 beschrieben. In Abschnitt 2.4 werden konzeptuelle Architekturen für die Einbettung der Zugriffskontrolle vorgestellt. Das Kapitel schließt mit einer Zusammenfassung.

2.1 Einordnung der Zugriffskontrolle in das Umfeld der IT-Sicherheit

In diesem Unterkapitel wird eine Reihe von Begriffsdefinitionen vorgenommen. Beginnend mit dem Begriff der IT-Sicherheit und seinen verwandten Begriffen in Abschnitt 2.1.1 wird in Abschnitt 2.1.2 genauer auf die Zugriffskontrolle eingegangen. Mit der Zugriffskontrolle eng verwandt ist die Rechteverwaltung, die in Abschnitt 2.1.3 erläutert wird.

2.1.1 IT-Sicherheit

IT-Sicherheit beschäftigt sich mit der Sicherheit von IT-Systemen. Ein *IT-System* ist ganz allgemein eine Funktionseinheit, welche Daten verarbeiten kann [DIN88, ISO93, Die05]. Unter Verarbeitung von Daten ist hier das Aufnehmen (Erfassen), das Aufbewahren (Speichern), das Weitergeben (Übertragen), das Umformen (Transformieren) und das Nutzen (Korrelieren und Interpretieren) von Daten gemeint. Ein IT-System kann hierbei nur einem, einigen ausgewählten oder allen oben aufgeführten Zwecken dienen. Unter einer Funktionseinheit können hier jegliche Hard- und Softwareeinheiten, wie etwa Rechner, Betriebssysteme und Datenbanken sowie die Netzwerkinfrastruktur, die zur Kommunikation der verschiedenen Systeme und Systemeinheiten notwendig ist, verstanden werden. Insbesondere kann ein IT-System wiederum aus IT-Untersystemen aufgebaut sein. In einem soziotechnischen System wird auch der Mensch selbst als eine solche Funktionseinheit verstanden [Eck04, Die05]. Grundsätzlich können IT-Systeme auch aus mehreren Komponenten (oder Funktionseinheiten) aufgebaut sein. In dieser Arbeit werden Anwendungssysteme betrachtet. Ein *Anwendungssystem* ist ein IT-System, welches eine konkrete Anwendung mit einer Benutzerschnittstelle bereitstellt. Ein Anwendungssystem besteht nicht nur aus der Anwendung selbst, sondern baut auf den Schichten der Middleware, des Betriebssystems, der Hardware und ggf. der vorhandenen

Infrastruktur auf. Die Schichten sind so aufgebaut, dass eine Schicht immer Funktionen und Schnittstellen der direkt unter ihr liegenden Schicht verwendet, um ihre Dienste zu erfüllen [Rud99].

IT-Sicherheit für IT-Systeme hat viele Facetten. Gemeinhin unterscheidet man zwischen *Funktionssicherheit* bzw. technischer Sicherheit (engl.: safety) und *Informationssicherheit* (engl.: security). Ein IT-System ist funktionssicher, wenn es erwartungsgemäß funktioniert, d. h., wenn es keine unzulässigen Zustände einnimmt und unter allen Betriebsbedingungen vorhersagbare Ergebnisse liefert. Die Informationssicherheit hingegen hat zum Ziel, die von einem IT-System gespeicherten und verwalteten Informationen vor unberechtigtem Zugriff durch nicht autorisierte Personen zu schützen [Eck04]. Eine Diskussion über die Abgrenzung von technischer Sicherheit und Informationssicherheit findet sich in [GKM+03]. In neuerer Zeit ist noch der Aspekt der Privatheit (engl.: privacy) hinzugekommen [Wei02, Sac07]. Dieser beschäftigt sich mit der informationellen Selbstbestimmung, d. h., jede Person sollte festlegen dürfen, zu welchem Zweck ihre persönlichen Daten verwendet werden, wie lange sie gespeichert werden dürfen, etc. Heutzutage existieren Gesetze und Datenschutzrichtlinien, die den Gebrauch von personenbezogenen Daten regeln. Genannt seien hier das Bundesdatenschutzgesetz (BDSG), die Datenschutzgesetze der einzelnen Länder sowie bereichsspezifische Datenschutzgesetze wie etwa die Telekommunikations-Datenschutzverordnung und das Telekommunikationsgesetz. Das BDSG fungiert hierbei als Auffanggesetz, d. h., falls ein Sachverhalt nicht in einem spezielleren Landesgesetz oder einem anderen bereichsspezifischen Gesetz geregelt ist, so gelten die Bestimmungen des BDSG [The97, Buc03]. Allerdings werden diese gesetzlichen Richtlinien noch selten durch eine entsprechende Implementierung erzwungen. Die beiden W3C-Projekte P3P („Platform for Privacy Preferences“) [W3C02b] und APPEL („A P3P Preference Exchange Language”) [W3C02a] haben zum Ziel, ein standardisiertes Format für Internetseiten zur Verfügung zu stellen, mit dem Webseitenbetreiber ihre Datenschutzpraktiken formulieren und Internetnutzer diese automatisiert mit den eigenen Datenschutzpräferenzen abgleichen können.

Nach [Die04] kann die Sicherheit in zwei andere komplementäre Sichten unterteilt werden, die *Verlässlichkeit* und die *Beherrschbarkeit*. Die Verlässlichkeit ist die Sicherheit des Systems selbst; ein System mitsamt seinen Daten darf nicht unzulässig beeinträchtigt werden. Die Beherrschbarkeit ist die Sicherheit vor dem System; hiermit ist gemeint, dass die von einem System Betroffenen bei der Anwendung von IT-Systemen keine Beeinträchtigungen erfahren dürfen. Dieses Konzept wird mit *dualer Sicherheit* bezeichnet.

Jeder Bereich der IT-Sicherheit verfolgt verschiedene *Schutzziele*, bei deren Einhaltung die Sicherheit im Sinne des jeweiligen Sicherheitsbereichs gewährleistet ist. Praktisch kann die totale Sicherheit eines IT-Systems in Hinblick auf die Erreichung eines Schutzziels jedoch nicht gewährleistet werden, da der Mensch als Funktionseinheit eines IT-Systems nicht als vollständig verlässlich und vor allem als nicht kontrollierbar einzustufen ist [Die04]. Mit der Informationssicherheit eines IT-Systems, die in dieser Arbeit näher betrachtet wird, werden in der Regel fünf allgemeine Schutzziele assoziiert: *Vertraulichkeit, Integrität, Verfügbarkeit, Nicht-Abstreitbarkeit* (auch: *Zurechenbarkeit*) und *Authentizität* (auch: *Echtheit, Glaubwürdigkeit, Revisionsfähigkeit, (Rechts-)Verbindlichkeit*) [Eck04].

Innerhalb eines Unternehmens dienen vertrauliche Informationen dem Wettbewerbsvorteil [EA06]. Unbefugte können durch Kenntnis vertraulicher Daten und Informationen dem Unternehmen Schaden zufügen. Von daher ist die Vertraulichkeit von Daten und Informationen ein wichtiges Sicherheitsgrundbedürfnis. Durch die Vertraulichkeit wird der lesende Zugriff auf Daten und Informationen geschützt. Die Integrität hingegen beschäftigt sich mit dem schreibenden Zugriff. Die Daten, die innerhalb eines Unternehmens verwaltet werden, müssen den Wirklichkeitsausschnitt, den sie abbilden, korrekt darstellen. Deshalb dürfen die Daten nicht unbefugt geändert, gelöscht oder neu angelegt werden, so dass die Konsistenz der verwalteten Daten nicht mehr gegeben ist. Das Schutzziel der Verfügbarkeit besagt, dass die in einem IT-System verwalteten Daten immer für die autorisierten Personen zugreifbar sein müssen. Die Nicht-Abstreitbarkeit verlangt, dass Aktionen, die innerhalb eines IT-Systems durchgeführt wurden, im Nachhinein nicht geleugnet werden können. Die Authentizität ist die Echtheit und Glaubwürdigkeit eines Subjekts [Eck04]. Die genannten Schutzziele werden nicht exklusiv der Informationssicherheit zugeschrieben. Für die Verfügbarkeit gilt z. B., dass es sich hierbei auch um ein Ziel der Funktionssicherheit handelt. Bei [Die04] werden die drei ersten Schutzziele Vertraulichkeit, Integrität und Verfügbarkeit der Sicht der Verlässlichkeit zugeordnet, während die letzteren beiden, Nicht-Abstreitbarkeit und Authentizität, der Beherrschbarkeit zugeordnet werden.

Für jedes Schutzziel gibt es Maßnahmen, die das Erreichen dieser Schutzziele unterstützen. Auf technischer Ebene setzen sich diese Maßnahmen aus den *Sicherheitsgrundfunktionen* zusammen. Dem Unsicherheitsfaktor Mensch wird durch organisatorische Maßnahmen begegnet.

In der vorliegenden Arbeit werden die Maßnahmen der *Zugriffskontrolle* und der *Rechteverwaltung* näher betrachtet. Diese beiden Maßnahmen unterstützen die Erreichung der Schutzziele Vertraulichkeit und Integrität im Bereich der Informationssicherheit sowie das Schutzziel der Privatheit im Bereich des Datenschutzes (siehe Abb. 2.1). Es sei hier angemerkt, dass die Zugriffskontrolle und die Rechteverwaltung alleine nicht ausreichend sind, um die die genannten Schutzziele zu gewährleisten. Es kann sich nur um unterstützende Maßnahmen handeln. Insbesondere ist eine Zugriffskontrollüberprüfung nur dann möglich, wenn die mit dem IT-System interagierende Entität, dies ist eine Person oder ein anderes IT-System, eindeutig identifiziert werden kann, diese sich also zuvor authentifiziert hat. Weiterhin sind für die Übertragung von Daten weitere Sicherheitsmechanismen, wie etwa die Verschlüsselung und die Verwendung geeigneter Kommunikationsprotokolle notwendig, um Angriffen während des Datenaustausches zwischen zwei IT-Systemen entgegen zu wirken. Ebenso sind für die Sicherstellung der Integrität weitere Maßnahmen, wie etwa die Überprüfung und Erhaltung der referentiellen Integrität von in einer Datenbank gespeicherten Daten und der Einsatz von kryptographischen Prüfsummen, notwendig.

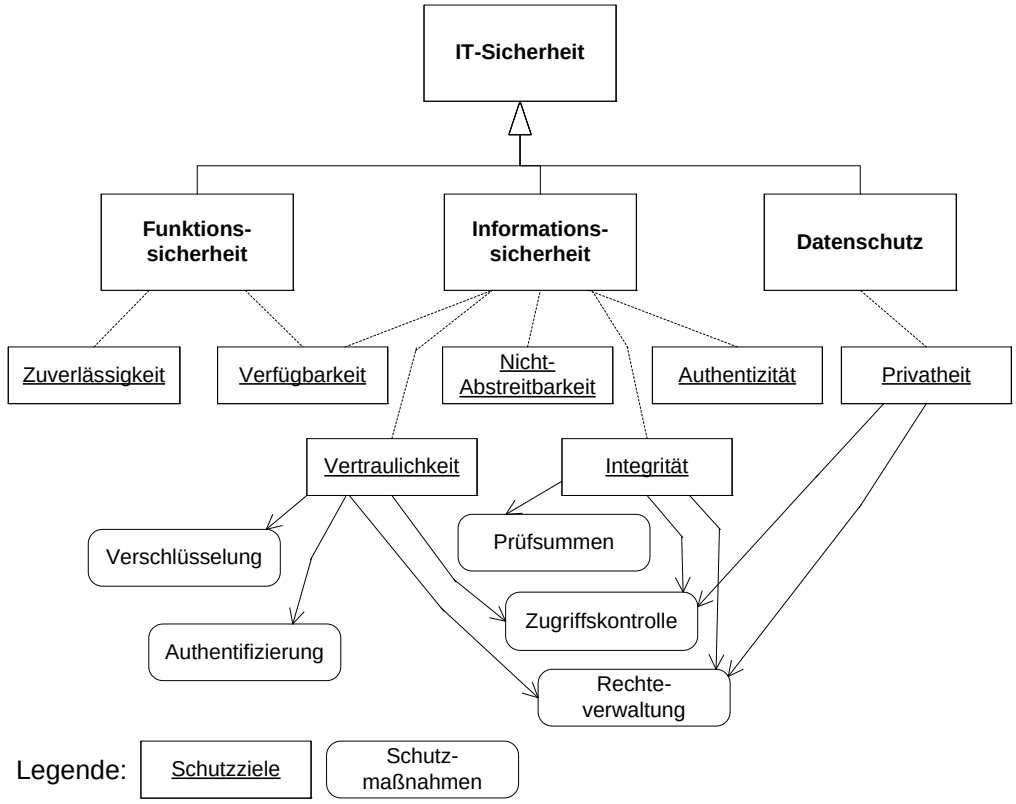


Abb. 2.1: Übersicht über Maßnahmen zur Erreichung der Schutzziele Vertraulichkeit und Integrität sowie der Privatheit

2.1.2 Zugriffskontrolle

Die *Zugriffskontrolle* (auch: *Rechteprüfung*, *Berechtigungsüberprüfung*, *Autorisierung*) kontrolliert den Zugriff von Prinzipalen auf Ressourcen [And01, S. 51]. Ein *Prinzipal* ist eine Entität, welche in einem IT-System agiert [And01]. Der Prinzipal hat eine eindeutige Identität, über die ihm Zugangs- und Zugriffsberechtigungen zugeordnet werden können. Die Zugriffskontrolle setzt also eine weitere Sicherheitsmaßnahme, die *Zugangskontrolle* (oder *Authentifizierung*) voraus, über die die Identität eines Prinzipals geprüft wird. Bei einem Prinzipal kann es sich nach [And01] um ein Subjekt, eine

Person, eine Rolle, ein Terminal, eine Smartcard, einen Prozess, der im Namen eines Benutzers läuft o. ä. handeln. Ein *Subjekt* ist dort eine physische Person, wohingegen mit dem Begriff *Person* auch eine juristische Person gemeint sein kann [And01, S. 9]. Nach [Gol02] wird der Benutzer eines Systems, also die physische Person, mit Prinzipal bezeichnet und das Subjekt agiert in einem IT-System im Namen des Prinzipals. Nach [Oak01] können einem Subjekt, welches einen Benutzer eines IT-Systems darstellt, mehrere Prinzipale zugeordnet werden, da sich ein Subjekt über verschiedene Merkmale identifizieren kann, z. B. über eine eindeutige ID, über ein Passwort oder über eine Gruppen- oder Rollenzugehörigkeit. Es lässt sich also feststellen, dass in der Literatur die Unterscheidung zwischen Subjekt und Prinzipal unscharf ist. In dieser Arbeit werden die Begriffe Subjekt und Prinzipal als Synonyme betrachtet. Eine genauere Differenzierung der Begrifflichkeiten ist im Rahmen dieser Arbeit nicht notwendig. Im Bereich der Zugriffskontrolle hat sich in der Literatur der Begriff Subjekt durchgesetzt, so dass im Folgenden nur noch dieser Begriff verwendet wird. Soweit nicht anders beschrieben, ist im Folgenden das Subjekt immer an eine physische Person gebunden.

Eine *Ressource* ist eine Entität in einem IT-System, auf welcher Operationen ausgeführt werden können. In betrieblichen Anwendungen können diese Entitäten Anwendungsobjekte wie etwa Geschäftsobjekte sein, in einer relationalen Datenbank kann es sich um Tabellen, Zeilen und Spalten handeln, in einem Betriebssystem um Dateien oder Prozesse. Ein anderer Begriff für Ressource ist *Objekt*. Insbesondere kann es in einem IT-System Objekte geben, welche auch als Subjekte fungieren können. In einem Betriebssystem z. B. gibt es ausführbare Dateien als Objekte. Zum einen können Zugriffsrechte auf diese Dateien spezifiziert werden, zum anderen können die Dateien während ihrer Ausführung als Subjekte agieren und wieder auf andere Objekte zugreifen.

Ein *Zugriff* auf eine Ressource kann entweder lesend oder schreibend erfolgen. Lesende Zugriffe ändern den Systemzustand nicht. Unter schreibenden Zugriffen werden hier alle Zugriffe verstanden, die den Systemzustand ändern. Der Systemzustand kann durch Editieren oder Löschen einer bestehenden Ressource oder durch Anlegen einer neuen Ressource verändert werden. Ein Zugriff beinhaltet also eine bestimmte Funktionalität, – hier mit *Aktion* bezeichnet –, die auf einer Ressource ausgeführt wird.

Subjekt, Objekt und Aktion bilden zusammen eine (Zugriffs-)*Berechtigung*. Neben den drei eben genannten Elementen können Berechtigungen auch *Nebenbedingungen* enthalten. Eine Nebenbedingung beschreibt eine weitere Bedingung, die auf den Systemzustand oder den Zustand von Subjekt und / oder Objekt zutreffen muss, damit ein gegebenes Subjekt diese Berechtigung innehat. Für eine Anwendung wird ein ganzer Satz an Berechtigungen definiert. Dieser bildet die *Berechtigungspolicy* der Anwendung. Auf Berechtigungen und Policies wird in Kapitel 5.1 näher eingegangen. Die Zugriffskontrolle überprüft, ob für ein bestimmtes Subjekt eine bestimmte, angeforderte Berechtigung existiert.

Weiterhin sind die Anforderungen an die Zugriffskontrolle einer betrieblichen Anwendung eng mit der Geschäftslogik der betrieblichen Anwendung verwoben, da ja überprüft werden soll, ob ein bestimmtes Subjekt, sei dies eine Person oder ein anderes System, eine bestimmte Funktionalität ausführen darf. Die Implementierung einer solchen Berechtigungsüberprüfung stellt eine *Querschnittsfunktion* dar, welche für die Anwendung in ihrer Gesamtheit gilt, da Berechtigungen prinzipiell an vielen Stellen innerhalb der Anwendung überprüft werden müssen. Von daher ist es schwierig, die Zugriffskontrolle von der Geschäftslogik zu entkoppeln. Eine Entkopplung, und damit eine klare Trennung von der Geschäftslogik, ist aber notwendig, um eine einfache Administrierbarkeit der Berechtigungen zu erreichen.

2.1.3 Rechteverwaltung

Die *Rechteverwaltung* administriert für jede zugriffsgeschützte Ressource innerhalb eines IT-Systems die Zugriffsrechte für jedes Subjekt. Zugriffsrechte sind für jede Schicht eines IT-Systems spezifisch. In einem Betriebssystem gibt es z. B. Lese- und Schreibrechte auf Dateien, in einer relationalen Datenbank gibt es z. B. das Recht zum Einfügen neuer Zeilen oder Spalten in eine Datenbanktabelle. Die Rechteverwaltung vergibt und ändert die Zugriffsrechte und legt fest, unter welchen Umständen die Rechte wahrgenommen werden dürfen. Zugriffsbeschränkungen können z. B. zu bestimmten

Tageszeiten auftreten oder davon abhängen, mit welchem Authentifizierungsmechanismus ein Subjekt seine Identität bescheinigt hat.

Eine umfassende Rechteverwaltung sollte nach dem *Vollständigkeitsprinzip* realisiert sein, d.h. grundsätzlich ist jeder Zugriff zu überprüfen [Eck04]. In der englischsprachigen Literatur wird dieses Prinzip als *complete mediation* [SS75] bezeichnet. Eine Rechteverwaltung, die nach dem Vollständigkeitsprinzip alle Subjekte und alle Objekte verwaltet, kann sehr umfangreich werden. Von daher werden Rechte häufig nach einem der beiden komplementären Prinzipien, dem *Erlaubnisprinzip* oder dem *Verbotsprinzip*, realisiert. Beim Erlaubnisprinzip sind alle Aktionen verboten, welche nicht explizit erlaubt werden [SS75, Eck04, Die05]. Beim Verbotsprinzip gilt die umgekehrte Regel [Die05]. Natürlich ist hier das konservative Design die bessere Wahl [SS75]. Ein Designfehler in der Rechteverwaltung wird beim Erlaubnisprinzip eher dazu führen, dass zu wenige Rechte für ein Objekt vergeben worden sind. Dieser Fehler kann bei Verwendung des IT-Systems schnell erkannt und beseitigt werden. Auf der anderen Seite führt ein Designfehler beim Verbotsprinzip dazu, dass mehr Zugriffsrechte auf einem Objekt existieren als gewünscht. Dieser Fehler könnte bei normaler Verwendung des IT-Systems unbemerkt bleiben.

Weiterhin gelten zwei weitere Designprinzipien für eine Rechteverwaltung. Zum einen sollte das *Prinzip der minimalen Rechte* (engl.: principle of least privilege / need-to-know principle) [SS75, And01, Eck04, Die05] realisiert werden. Dieses Prinzip besagt, dass jedem Subjekt nur so viele Rechte zugesprochen werden sollten, wie das Subjekt minimal benötigt, um seine Aufgaben innerhalb des IT-Systems zu erfüllen. Ein weiteres Prinzip ist das *Prinzip der Aufgabenteilung* (*Prinzip der Trennung von Zuständigkeiten*, engl.: separation of duty / separation of concerns) [SS75, And01, Die05]. Dieses ist ähnlich dem Prinzip der minimalen Rechte, wurde aber speziell durch das rollenbasierte Berechtigungsmodell (siehe 2.4.2) geprägt. Jede Person hat innerhalb eines Unternehmens bestimmte, spezialisierte Aufgaben zu übernehmen. Einige dieser Aufgaben sollten so verteilt sein, dass sie nicht durch dieselbe Person ausgeführt werden können. Ein Beispiel hierfür ist das Vier-Augen-Prinzip, bei dem es einen Ausführer und einen Überwacher gibt. Die beiden Rollen des Ausführers und des Überwachers sollten nicht von derselben Person eingenommen werden. Zum Beispiel könnte ein Mitarbeiter bei der Bank gleichzeitig Kunde der Bank sein. Natürlich sollte es ihm nicht erlaubt sein, schreibend auf sein eigenes Konto zuzugreifen, auch wenn er die Rechte besitzt, Geld für andere Bankkunden von einem Konto abzuheben oder auf ein Konto gutzuschreiben.

Das Ziel der Rechteverwaltung ist es, zu gewährleisten, dass die Subjekte nicht durch eine zu strenge Rechteeinschränkung an der legitimen Ausführung ihrer Arbeit gehindert werden und dass andersherum kein Schaden dadurch entsteht, dass Subjekten zu viele Rechte eingeräumt wurden, so dass sie versehentlich oder absichtlich die im System verwalteten Objekte kompromittieren. Harrison, Ruzzo und Ullman haben ein Berechtigungsmodell aufgestellt, welches auf einer Zugriffsmatrix basiert. Die Zugriffsmatrix beschreibt den Zustand eines Systems. Dieser kann durch Ausführen von Kommandos (engl. commands) verändert werden. In diesem Berechtigungsmodell werden Subjekten Zugriffsrechte auf Objekten gewährt; die Subjekte sind auch Objekte; und neue Objekte (und damit auch Subjekte) können durch das Absetzen von Kommandos erstellt werden. Die Zugriffsrechte können durch den Ersteller des Objekts geändert werden. In diesem Modell ist es nicht entscheidbar, ob es eine Änderungshistorie der Zugriffsrechte geben kann, so dass eine Konfiguration des Systems entsteht, in dem es einem unzuverlässigen Subjekt gelingt, einem anderen Subjekt ein Recht auf ein Objekt einzuräumen, welches es vorher nicht hatte und damit das intendierte Rechtesystem zu unterlaufen. Es kann also nicht entschieden werden, ob das Modell in dieser Hinsicht „sicher“ ist [HRU76]. Diese Sicherheitseigenschaft ist nur für sehr beschränkte Modelle gezeigt worden, in denen alle Kommandos aus nur einer einzigen Operation bestehen oder die Anzahl der Subjekte eingeschränkt ist [Gol02].

2.2 Der Sicherheitsengineering-Prozess

Die Anforderungen an die Sicherheit eines IT-Systems werden innerhalb des *Sicherheitsengineering-Prozesses* (engl.: security engineering process) festgelegt. Der Sicherheitsengineering-Prozess ist ein Softwareengineering-Prozess, welcher gesondert auf die Modellierung und Realisierung von Sicherheitsvorgaben eingeht. Er ist noch nicht methodisch ausgearbeitet [Eck04, And01]. Von daher werden nachfolgend einige Methoden und Maßnahmen beschrieben, die sich bisher bewährt haben.

In der Analysephase des Sicherheitsengineering werden spezielle Werkzeuge und Techniken innerhalb der Sicherheitsanforderungsanalyse (engl.: security requirements analysis) eingesetzt, um die Sicherheitsanforderungen herauszuarbeiten. Die Sicherheitsanforderungen an ein System gehören zu den *nicht-funktionalen Anforderungen* [Som05, Bal00]. Im Gegensatz zu den *funktionalen Anforderungen* lässt sich der Erfüllungsgrad nicht-funktionaler Anforderungen nur schwer oder gar nicht messen. Gerade Maßnahmen zur Erhöhung der Sicherheit eines Systems sind präventiver Art, so dass sie zunächst nur Kosten verursachen aber kein direkter Nutzen ersichtlich ist. Sicherheitsmaßnahmen wehren ja nur potentielle Schadenseintritte ab. Die Kosten, die mit der Realisierung von Sicherheitsanforderungen anfallen, lassen sich nur schwer einem Kapitalertrag (engl.: return on investment) zuordnen.

Innerhalb der Sicherheitsanforderungsanalyse wird zunächst ein *Bedrohungsmodell* erarbeitet, welches Interessen potentieller Angreifer auf das System mit einbezieht. Ein Bedrohungsmodell beschreibt mögliche Bedrohungen für ein System. Im Rechnungswesen z. B. soll versehentlichen und absichtlichen Fehlern durch die doppelte Buchhaltung entgegengewirkt werden; in der Bank und im Krankenhaus sollen Social Engineering Angriffe eingeschränkt und erschwert werden, indem nur möglichst wenigen Personen Zugang zu vertraulichen Daten gewährt wird (für eine Erläuterung von Social Engineering siehe z. B. [And01, Mit02]). Das bekannteste Werkzeug für die Bedrohungsanalyse sind die Bedrohungsbäume (engl.: attack trees) [Sch99]. Im Anschluss an das Bedrohungsmodell werden die Risiken und Schadensszenarien für konkrete, mögliche Angriffe bewertet.

In der Modellierungsphase des Sicherheitsengineering-Prozesses wird auf Basis der gefundenen Bedrohungsmöglichkeiten eine *Sicherheitsstrategie* [Eck04] entwickelt. Die Sicherheitsstrategie beschreibt formal oder informell Maßnahmen, die zur Erfüllung des gewünschten Sicherheitsstandards notwendig sind. Neben Maßnahmen zur sicheren Kommunikation mittels kryptographischer Algorithmen und Protokollen gehören hierzu auch Maßnahmen zum Schutz persistent gespeicherter Daten. Die Zugriffskontrolle sollte als Maßnahme eingesetzt werden, um die Vertraulichkeit von gespeicherten Daten zu gewährleisten. Die Rechteverwaltung dient der Organisation und Administration spezifizierter Rechte. Die Sicherheitsstrategie mündet in einem *Sicherheitsmodell* [Eck04]. Anhand des Sicherheitsmodells können die zu erfüllenden Eigenschaften überprüft werden.

Das Sicherheitsmodell enthält ebenfalls ein Modell für die Zugriffskontrolle, das *Berechtigungsmodell*. Ein Berechtigungsmodell beschreibt allgemeine, wiederkehrende Berechtigungsmuster, welche sich in der Praxis etabliert haben oder welche aufgrund ihrer Eigenschaften für den praktischen Einsatz geeignet sind. Das Berechtigungsmodell bildet also ein Rahmenwerk, in dem konkrete Berechtigungen definiert werden können. Die beiden für betriebliche Anwendungen wichtigsten Berechtigungsmodelle werden im nächsten Unterkapitel 2.3 näher beschrieben. Auf Basis des Berechtigungsmodells können die konkreten Berechtigungen für eine Anwendung definiert werden.

In der Entwurfsphase des Sicherheitsengineering wird eine Sicherheitsarchitektur entworfen. Ein Architekturvorschlag für die Zugriffskontrolle ist in der ISO/IEC-Norm 10181-3 gegeben. Dieser wird in Kapitel 2.4.1 behandelt.

In der Implementierungsphase schließlich werden die durch die Architektur festgelegten IT-Maßnahmen umgesetzt. Für die Implementierung der Zugriffskontrolle werden herkömmlich Zugriffskontrolllisten (ACLs) verwendet. Neuere Implementierungen verwenden für die Beschreibung und Durchsetzung von Berechtigungen regelbasierte Berechtigungsbeschreibungssprachen. Auf ACLs und regelbasierte Berechtigungsbeschreibungssprachen wird in Kapitel 5 näher eingegangen.

Die letzte Phase eines Softwareentwicklungsprozesses ist die Wartungsphase. In dem hier behandelten Bereich der Zugriffskontrolle und Rechteverwaltung ist es vor allem nötig, die Berechtigungen zu warten. Diese können sich im Laufe des Betriebs eines Informationssystems ändern und ein Administrator muss diese Änderungen konsolidieren und verwalten können, und zwar so, dass die Sicherheitsrichtlinien zu jeder Zeit erfüllt werden und die definierten Berechtigungen konsistent bleiben.

In der vorliegenden Arbeit werden aufbauend auf einem geeigneten Bedrohungsmodell (siehe 4.1) eine Berechtigungsbeschreibungssprache, die auf einem für betriebliche Anwendungen geeigneten Berechtigungsmodell aufsetzt, zur Definition einer Berechtigungspolicy (siehe Kapitel 5) sowie eine Sicherheitsarchitektur und ein Algorithmus, die eine Durchsetzung der in der Policy spezifizierten Berechtigungen ermöglichen (siehe Kapitel 6) erarbeitet. Diese Arbeit deckt also die Modellierungs- und die Implementierungsphase des Sicherheitsengineering-Prozesses ab. Eine Behandlung der Wartungsphase würde den Rahmen dieser Arbeit sprengen.

2.3 Berechtigungsmodelle für betriebliche Anwendungen

In der Literatur existiert eine Vielzahl unterschiedlicher Berechtigungsmodelle, die jeweils für unterschiedliche Anwendungsbereiche konzipiert wurden. Im militärischen Umfeld wurde das Bell-LaPadula-Modell [BL76] entwickelt, welches insbesondere die Vertraulichkeit von Informationen und Daten innerhalb eines Anwendungssystems sicherstellt. Die Umkehrung des Bell-LaPadula-Modells ist das Biba-Modell [Bib77], welches nicht die Vertraulichkeit sondern die Integrität der im System verwalteten Daten in den Vordergrund stellt. Das Clark-Wilson Modell [CW87] stellt ebenfalls die Integrität der Daten in den Vordergrund. Hier können die Daten nur über so genannte Transformationsprozeduren verändert werden. Das Chinese-Wall-Modell [BN89] schließlich ist ein Berechtigungsmodell für den kommerziellen Sektor, durch welches Insidergeschäfte verhindert werden sollen. Im Umfeld der Zugriffskontrollanforderungen von betrieblichen Anwendungen, die in dieser Arbeit genauer untersucht werden, sind die oben genannten Modelle nur von untergeordneter Bedeutung, da sie sich im Unternehmensumfeld nicht durchgesetzt haben. Es sind zwei Berechtigungsmodelle von besonderem Interesse, das benutzerbestimmbare Zugriffskontrollmodell und das rollenbasierte Zugriffskontrollmodell. Diese Zugriffskontrollmodelle werden im Folgenden näher erläutert.

2.3.1 Benutzerbestimmbares Zugriffskontrollmodell (DAC)

Die *benutzerbestimmbare Zugriffskontrolle* (engl.: discretionary access control, DAC) [FKC03] gehört zu den ältesten und wohl bekanntesten Zugriffskontrollmodellen. Sie basiert auf einem sehr einfachen Konzept. Jedem Objekt, welches in einem IT-System verwaltet wird, ist ein Eigentümer zugeordnet. In der Regel handelt es sich hierbei um den Ersteller des Objekts. Der Eigentümer hat vollständige Kontrolle über sein Objekt. Er hat alle Zugriffsrechte auf das Objekt. Weiterhin kann er Zugriffsrechte auf sein Objekt für andere Subjekte innerhalb des IT-Systems definieren. Die Rechteverwaltung ist dezentral organisiert. Die Zugriffsrechte beschränken sich häufig auf einige wenige Rechte, in der Regel auf das Lesen und Schreiben.

Im Betriebssystem UNIX, dem wohl bekanntesten Vertreter von DAC, sind die Objekte Dateien und Verzeichnisse, wobei Drucker und andere Endgeräte auch als Dateien behandelt werden. Entsprechend dem Eigentümerprinzip besitzt jedes Objekt einen *owner*, der zunächst alle Rechte auf das entsprechende Objekt hat. Rechte, die vergeben werden können, sind *read* (r), *write* (w) und *execute* (x). Je nachdem, ob es sich bei dem Objekt um eine Datei oder ein Verzeichnis handelt, werden diese Rechte unterschiedlich interpretiert [Tan02]. In Unix gibt es drei Berechtigungsblöcke: eigene Rechte, Rechte für genau eine Gruppe, Rechte für den Rest der Welt. Hierbei kann ein Benutzer Mitglied in beliebig vielen Gruppen sein. Mit dem *chmod*-Befehl (chmod = change mode: Änderung von Dateiattributen u. a. der Rechteattribute r, w und x) kann der Eigentümer die Rechte für jede seiner Dateien und für jede der drei Berechtigungsblöcke setzen. Die Eigentümerschaft auf ein Objekt kann, abgesehen vom Administrator (superuser), nicht übertragen werden. Somit handelt es sich bei Unix um eine strikte, benutzerbestimmbare Zugriffskontrolle (engl.: strict DAC). Manchmal ist es notwendig, Rechte für Unterprogramme zu erweitern. Hierfür wird das zusätzliche Schutzbit SETUID verwendet. Wenn ein Benutzer einen Prozess anstößt, so ist normalerweise der Benutzer der Eigentümer des Prozesses. Insbesondere hat der Prozess dann dieselben Rechte wie der Benutzer. Ist das SETUID-Bit gesetzt, so ist der Besitzer der ausführbaren Datei auch gleichzeitig der Eigentümer des angestoßenen Prozesses, so dass der Prozess mit den Rechten des Dateibesitzers ausgestattet ist. Diese Art der Rechteüberschreibung ist nützlich, wenn Benutzer Systemprogramme ausführen möchten. Bspw. gehört die Datei, die den Drucker repräsentiert, dem Administrator und nur dieser hat Lese- und Schreibrechte auf die Datei. Damit aber trotzdem alle Benutzer drucken können, kann jetzt der Prozess, der den Drucker anstößt mit den Rechten des Administrators ausgeführt werden.

In einer anderen Variante von DAC, der liberalen, benutzerbestimmbaren Zugriffskontrolle (engl.: liberal DAC) kann der Eigentümer das Recht zur Verwaltung der Rechte seiner Objekte an andere Subjekte weitergeben. Dieses Prinzip wird mit *Delegation* bezeichnet. Hierdurch kann sich eine sehr unübersichtliche Berechtigungsstruktur ergeben. Durch mehrfaches Weiterreichen von Administra-

tionsrechten bleibt die Verwaltung zwar dezentral, eine Rechterücknahme gestaltet sich aber als schwierig, da kein Überblick mehr darüber besteht, wer von wem ein Recht erhalten hat [SS94, SC00].

Die benutzerbestimmbare Zugriffskontrolle eignet sich für Anwendungen, in denen die verwalteten Daten einen starken Bezug zu einer Person haben. Ein Beispiel hierfür sind Weblogs, welche im Unternehmenskontext mit K-Logs (knowledge logs) bezeichnet werden [LM04]. Neben den „normalen“ Weblogs, die von nur einer Person publiziert werden, gibt es Teamlogs, in denen eine ganze Autorengruppe Einträge in einem Weblog veröffentlichen kann. Weblogeinträge in einem K-Log stellen z.B. persönliches Wissen und Erfahrungen dar, über dessen Verfügung der Eigentümer des Wissens gerne selbst bestimmen möchte. Die Objekte, die in einer Weblog-Community verwaltet werden, sind also die Weblogs mit den dazugehörigen Weblogeinträgen. Die Subjekte sind die Personen, die Weblogs besitzen und Weblogeinträge in das System einstellen sowie die Personen, die die einzelnen Einträge lesen. In einem System, in dem Informationseigentümer den Zugriff auf die von ihnen bereitgestellte Information, hier also die einzelnen Weblogeinträge, steuern können, haben die Benutzer mehr Vertrauen in die Anwendung, als wenn die Zugriffskontrolle zentral gesteuert würde.

Die Objekte, die mit einer benutzerbestimmbaren Zugriffskontrolle verwaltet werden können, können entweder grobgranularer oder feingranularer Natur sein. In einer Weblog-Community können z. B. ganze Weblogs nur für gewisse Lesergruppen zugänglich sein, oder aber einzelne Weblogeinträge für unterschiedliche Leser bestimmt sein. In einer Community-Software, in welcher private Adressdaten und andere personenbezogene Daten verwaltet werden, wie z.B. in Xing (http://www.xing.com, vorher: openBC) kann eine feingranulare Aufteilung der Personendaten in einzelne Adressfelder, wie Straße, Telefonnummer beruflich, Telefonnummer privat, Geburtsdatum, … erfolgen. Die einzelnen Benutzer können dann für jeden Kontakt bestimmen, welche Datumsfelder dieser lesen darf. In solch einer Community-Software spielt der Aspekt der Privatheit eine große Rolle. Nach dem Prinzip der informationellen Selbstbestimmung können die Benutzer des Systems bestimmen, wer ihre persönlichen Daten einsehen darf.

Insgesamt ist die benutzerbestimmbare Zugriffskontrolle ein sehr einfaches und nützliches Konzept für die Verwaltung von Zugriffsrechten auf persönliche Daten und Informationen, welches auch in betrieblichen Anwendungen Bedeutung hat.

2.3.2 Rollenbasiertes Zugriffskontrollmodell (RBAC)

Die *rollenbasierte Zugriffskontrolle* (engl.: role-based access control, RBAC) [FSG+01, AI04] gehört zu den neueren Zugriffskontrollmodellen und hat sich als Zugriffskontrollmodell für betriebliche Anwendungen durchgesetzt. Es existiert ein Standard vom National Institute of Standards and Technologies (NIST) [AI04]. Im rollenbasierten Zugriffskontrollmodell werden Berechtigungen zu Rollen zusammengefasst. Jede Rolle beschreibt dabei eine bestimmte Aufgabe innerhalb eines Unternehmens. Die einzelnen Benutzer, also die Mitarbeiter des Unternehmens, werden dann den Rollen zugewiesen.

Im NIST-Standard sind zu diesem sehr einfachen Modell, dem Kern (engl.: core RBAC), noch einige Erweiterungen aufgeführt. Die hierarchische, rollenbasierte Zugriffskontrolle (engl.: hierarchical RBAC) führt eine Rollenhierarchie ein. Eine Rolle erbt alle Berechtigungen ihrer untergeordneten Rollen und enthält alle Benutzer der übergeordneten Rollen. Bei der *statischen Aufgabenteilung* (engl.: static separation of duty) soll verhindert werden, dass derselben Person zwei sich gegenseitig ausschließende Rollen zugewiesen werden (siehe 2.1.3). Dasselbe Prinzip der statischen Aufgabenteilung wird auch für administrative Rollen angewandt. So gibt es innerhalb eines Systems häufig einen allmächtigen Administrator mit allen Rechten. Diese Konzentration der Rechte auf eine einzige Person bzw. auf eine einzige Rolle birgt natürlich auch Sicherheitsrisiken. Von daher gibt es z. B. im Betriebssystem Solaris drei Administratorrollen mit getrennten Aufgaben (primärer Administrator, Systemadministrator, Operator), um die Machtkonzentration zu verringern [Cha03]. Bei der *dynamischen Aufgabenteilung* (engl.: dynamic separation of duty) können zwar prinzipiell im Konflikt stehende Rollen derselben Person zugeordnet werden, allerdings wird zur Laufzeit verhindert, dass diese Person beide Rollen zur selben Zeit, bzw. zur Erfüllung derselben Aufgabe, einnehmen kann.

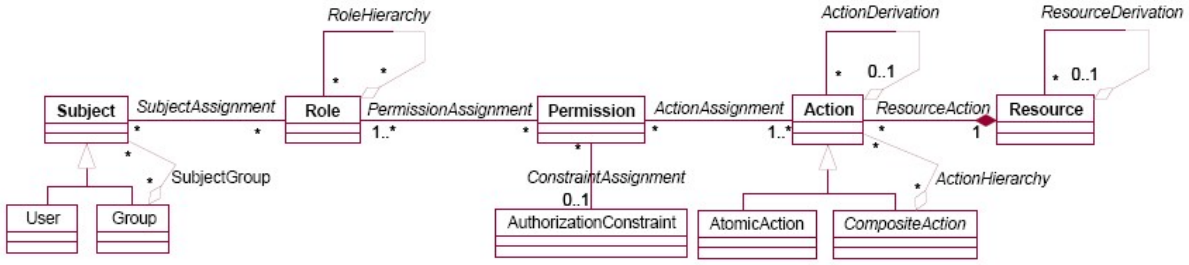


Abb. 2.2: Konzeptuelles Klassendiagramm für eine rollenbasierte Zugriffskontrolle, Quelle: [LBD02]

In Abb. 2.2 ist ein konzeptuelles Modell für die rollenbasierte Berechtigungsstruktur aufgezeigt. Die Subjekte (*Klasse: Subject*) sind Benutzer (*Klasse: User*) oder Gruppen (*Klasse: Group*), welche hierarchisch geschachtelt werden können (*Assoziationsname: SubjectGroup*). Die Subjekte werden den Rollen (*Klasse: Role*) zugeordnet, welche wiederum eine Rollenhierarchie (Assoziationsname: *RoleHierarchy*) bilden können. Eine Rolle ist eine Zusammenfassung mehrerer Rechte (Klasse: *Permission*). Berechtigungen können Nebenbedingungen (Klasse: *AuthorizationConstraint*) haben und gelten für ein oder mehrere Aktionen (Klasse: *Action*). Die Aktionen selbst sind Basisfunktionen (Klasse: *AtomicAction*) oder setzen sich aus den Basisfunktionen (Klasse: *CompositeAction*) zusammen. Das Recht zum Ausführen einer Aktion auf einem Objekt (Klasse: *Resource*) kann das Recht zum Ausführen einer Aktion auf einer anderen Ressource implizieren. Z. B. kann das Recht zum Lesen eines Verzeichnisses das Recht zum Lesen aller im Verzeichnis enthaltenen Dateien implizieren. Diese Rechteableitungen werden in diesem Modell für Aktionen (Assoziationsname: *ActionDerivation*) und Objekte (Assoziationsname: *ResourceDerivation*) dargestellt. Eine Berechtigung besteht prinzipiell aus Subjekt, Objekt und Aktion, wie unter 2.1.2 bereits kurz erläutert. Aktionen können aber objektspezifisch sein, so dass hier das Objekt nicht direkt an der Berechtigung sondern an der Aktion hängt.

Mit Hilfe von RBAC kann auch eine DAC-Strategie realisiert werden. DAC-Strategien werden implementiert, indem für jedes zugriffsgeschützte Objekt mehrere Rollen angelegt werden: eine Rolle pro geschützter Aktion, eine oder mehrere Rollen für die Verwaltung der Rechte. Im strikten DAC, wo alleine der Eigentümer des Objekts über die Rechtevergabe bestimmen kann, gibt es genau eine Rolle, die diese Verwaltungsrechte besitzt. In Systemen, wo mehrstufige Rechteweitergabe gewünscht ist, gibt es dementsprechend mehrere Rollen, welche wiederum je nach DAC-Strategie das Recht besitzen, einer weiteren Rolle die Zugriffsrechte weiterzugeben [SM98, OSM00]. Die Realisierung von benutzerbestimmbarer Zugriffskontrolle in RBAC führt also möglicherweise zu einer Explosion vorhandener Rollen. Bei der Einbettung von DAC in RBAC ergeben sich eine Vielzahl sehr kleiner und ähnlicher Rollen, so dass zu prüfen ist, ob sich die Kleinstrollen nicht zusammenfassen lassen. Ansonsten würde die Administration des Berechtigungssystems sehr unübersichtlich werden.

Obwohl RBAC sehr ausdrucksstark ist, so verwendet doch jedes System seine eigenen Rollen und Rechtestrukturen, so dass eine Interoperabilität zwischen zwei Systemen, die RBAC verwenden, im Allgemeinen nicht möglich ist.

Trotzdem ist die rollenbasierte Zugriffskontrolle der de facto Standard für die Berechtigungsstruktur einer betrieblichen Anwendung, da sich die einzelnen Rollen des Berechtigungskonzepts sehr gut auf die Rollen innerhalb eines Unternehmens abbilden lassen. Die rollenbasierte Berechtigung ist für einzelne betriebliche Anwendungen wegen der intuitiven Gestaltung der Rollen nach der Unternehmenshierarchie sehr gut anwendbar.

2.4 Sicherheitsarchitekturen für die Zugriffskontrolle

Für die Realisierung einer Zugriffskontrolle ist auf Basis der Sicherheitsstrategie, die auf ein Zugriffskontrollmodell abgebildet wird, eine konzeptuelle Sicherheitsarchitektur zur Einbettung der Zugriffskontrolle in die Anwendung zu entwickeln [Eck04]. In 2.4.1 wird eine Norm für die Zugriffskontrolle vorgestellt, die diese in zwei Komponenten aufspaltet, eine Komponente für die Berechtigungsspezifikation und eine für die Berechtigungsdurchsetzung. Diese Aufspaltung eignet sich sehr gut, um eine Zugriffskontrolle in einer betrieblichen Anwendung systematisch umzusetzen. In 2.4.2 wird die Quasar Authorization beschrieben. Dieses ist eine Komponente, mit Hilfe derer die Berechtigungsspezifikation von der Anwendung entkoppelt vorgenommen werden kann. Diese Entkopplung ist für betriebliche Anwendungen sehr nützlich. Dennoch fehlt bei dieser Lösung die systematische Durchsetzung der Zugriffskontrolle.

2.4.1 Konzeptuelle Architektur nach ISO/IEC-10181-3

Weiter oben wurde bereits festgehalten, dass die Zugriffskontrolle ein Querschnittsaspekt ist, d. h. sie durchzieht die gesamte Anwendung. Die Zugriffsüberprüfung findet an vielen verschiedenen Stellen innerhalb der Anwendung statt. Alle Berechtigungsabfragen einer Anwendung sollten auf Grundlage derselben Berechtigungsspezifikation ausgeführt werden. Wenn die Regeln für die Berechtigungsüberprüfungen fest in der Implementierung der Anwendung verankert sind, muss, falls sich die Berechtigungsspezifikation einmal ändert, die gesamte Anwendung daraufhin untersucht werden, ob Berechtigungsüberprüfungen angepasst werden müssen. Um dies zu verhindern, ist es sinnvoll, die Spezifikation der Berechtigungen von der Implementierung der Durchsetzung derselben weitestgehend zu trennen. Ein Rahmenwerk für die Zugriffskontrolle, welches eine solche Trennung vorsieht, wird in der ISO/IEC-Norm 10181-3 [ISO96b] vorgeschlagen.

Diese ISO/IEC-Norm sieht vor, die Berechtigungsüberprüfung in eine *Zugriffsentscheidungsfunktion* (access decision function, ADF) und eine *Zugriffsdurchsetzungsfunktion* (access enforcement function, AEF) aufzuspalten. Eine Zugriffsanfrage eines Subjekts auf ein Objekt wird hierbei zunächst an die AEF weitergeleitet. Das Subjekt wird in dieser Norm mit *Initiator* bezeichnet, während das Objekt *Target* genannt wird. Im Sinne des Vollständigkeitsprinzips sollte jede Zugriffsanfrage an eine AEF weitergeleitet werden. Die AEF leitet die Anfrage an die Entscheidungskomponente weiter und erwartet das Ergebnis der Zugriffsüberprüfung zurück (siehe Abb. 2.3). Die AEF wiederum „weiß“, wie mit dem Ergebnis der Zugriffsüberprüfung umzugehen ist. Bei einer Zugriffsverweigerung z. B. könnte dem anfragenden Subjekt eine vorkonfigurierte Fehlermeldung zurückgeliefert werden. Die Trennung in ADF und AEF sorgt also für die Konfigurierbarkeit des Umgangs mit Zugriffsentscheidungen, die die ADF bereitstellt. Die AEF kann an einer bestimmten Stelle im Quellcode der Anwendung platziert werden und kann daraufhin konfiguriert werden, wie mit positiver und negativer Zugriffsentscheidung umzugehen ist. Die eigentliche Spezifikation der Berechtigungen findet aber in der ADF statt. Jetzt können AEF und ADF unabhängig voneinander geändert werden. Bei der ADF handelt es sich häufig um eine zentrale Komponente, so dass die Berechtigungsspezifikation an einem zentralen Punkt definiert und geändert werden kann. Bei der AEF handelt es sich nicht um eine Komponente im engeren Sinn, sondern der jeweilige Code zur Zugriffskontrolldurchsetzung ist über die gesamte Anwendung verstreut. Da die AEF dezentral realisiert ist, sind Änderungen an der AEF aufwendig durchzuführen, da Code der gesamten Anwendung von der Änderung betroffen sein kann. In dieser Arbeit sollen ein Vorgehensmodell und ein Zugriffskontrollframework geschaffen werden, welches die Änderungen der AEF mit nur geringem Aufwand ermöglicht.

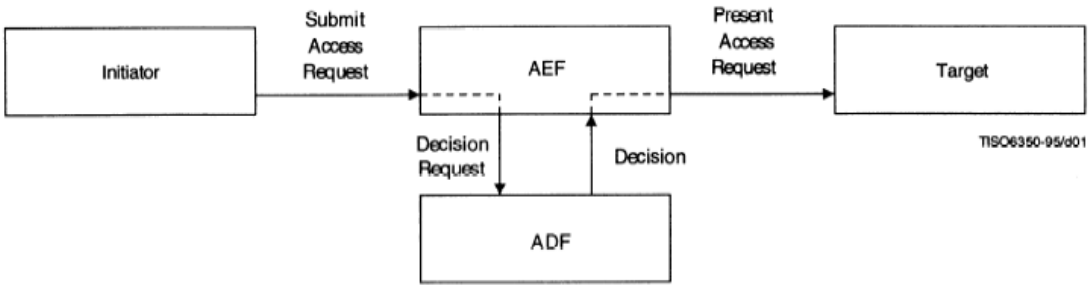


Abb. 2.3: Darstellung des Zusammenwirkens von AEF und ADF, Quelle: [ISO96]

2.4.2 Die Quasar Authorization als Beispiel für eine generische Berechtigungskomponente

Die Quasar Authorization [Nie03, Abb. 2.4] ist ein Beispiel für eine ADF-Komponente. Sie ist als Open-Source-Komponente erhältlich [OQ05]. Quasar [Qua07, Sie04] ist ein interner Architekturstandard, der von der Firma sd&m (http://www.sdm.de) entwickelt worden ist, mit dem Ziel robuste und änderbare Software zu erstellen. Quasar steht für Qualitätsarchitektur. Quasar basiert auf der Trennung von Zuständigkeiten. Insbesondere wird die Architektur der technischen Infrastruktur von der Software-Architektur getrennt. Die Software-Architektur wiederum gliedert sich in die Anwendungsarchitektur mit ihren fachlichen Elementen sowie die technische Architektur, die wieder verwendbare Komponenten und Bibliotheken beinhaltet. Die Quasar Authorization ist ein Bestandteil einer solchen generischen, in vielen betrieblichen Anwendungen wieder verwendbaren Komponente der technischen Architektur.

Die Quasar Authorization kapselt Autorisierungsaufgaben. Die Authentifizierung sowie die Verwaltung von Benutzern und Zugriffsrechten kann von externen Systemen über eine Schnittstelle angebunden werden. Das Berechtigungsmodell der Quasar Authorization ist sehr ausdrucksmächtig, beinhaltet positive und negative Rechte und vielfältige Möglichkeiten der Spezifikation von Nebenbedingungen. Da die Authorization eine separate Komponente darstellt, werden die Subjekte und Objekte in der Authorization über ID-Strings identifiziert.

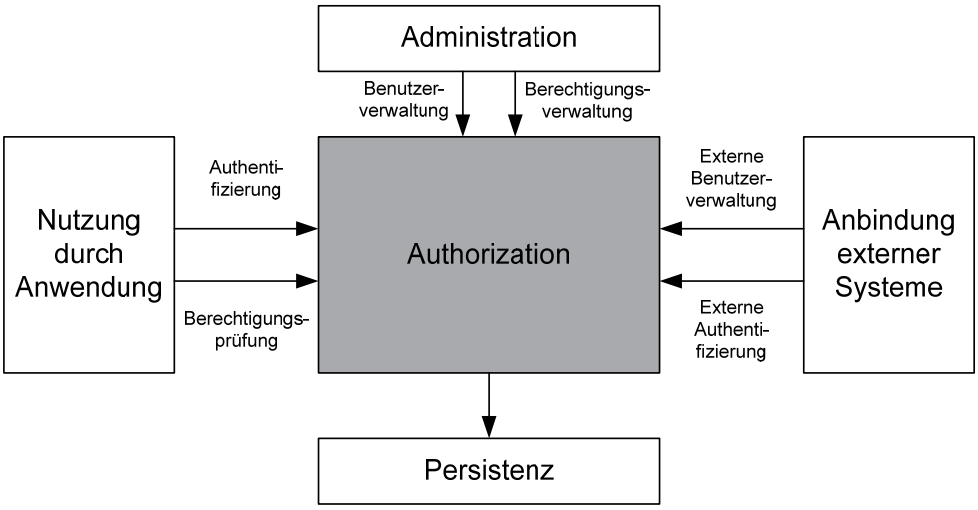


Abb. 2.4: Grobaufbau der Quasar Authorization, Quelle: [Nie03]

Die Quasar Authorization Komponente teilt sich in eine Berechtigungsverwaltung (siehe Abb. 2.5) und eine Benutzerverwaltung (siehe Abb. 2.6) auf. Die Benutzerverwaltung kann auch über externe Systeme angebunden werden.

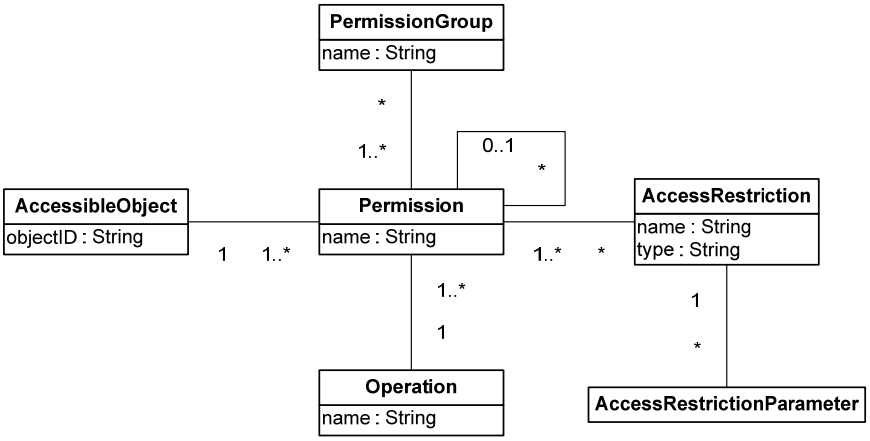


Abb. 2.5: Datenmodell für Berechtigungen, Quelle: [Nie03]

In der Quasar Authorization werden zu schützende Objekte mit *AccessibleObjects* bezeichnet. Es kann sich hierbei um jedwede Entitäten innerhalb einer Anwendung handeln, die über eine ID identifiziert werden können, also z. B. Instanzen von Klassen aber auch Funktionen und Methoden. Die *objectID* ist diese eindeutige ID. Sie wird als String bereitgestellt und macht daher eine Typüberprüfung schwierig. Aktionen werden als *Operation* bezeichnet und können lesende oder schreibende Zugriffe auf veränderbare Objekte oder die Ausführung einer Funktion sein [Nie03]. Zu jeder Berechtigung (*Permission*) können eine Vielzahl von Nebenbedingungen festgelegt werden, z. B. kann es eine Berechtigung geben, die den Zugriff auf Personalakten dahingehend einschränkt, dass nur auf Personen, deren Nachnamen mit den Buchstaben A – K beginnt, zugegriffen werden darf. Die Beschränkung A – K wäre dann so eine Nebenbedingung (*AccessRestriction*). Je nach Nebenbedingungstyp kann es verschiedene Parameter (*AccessRestrictionParameter*) geben, welche die Nebenbedingung beschreiben. Im vorigen Beispiel wären die Parameter das Intervall von A bis K. Andere Nebenbedingungstypen verlangen andere Parameter, wie z. B. „<“, „<=“, etc.

Das Schutzobjekt, die Operation und die Nebenbedingung bilden eine Berechtigung. Berechtigungen können sich gegenseitig implizieren (dieses wird in der Abbildung durch die Assoziation der Permission auf sich selbst dargestellt). So impliziert eine Schreibberechtigung auf ein Schutzobjekt, dass es auch gelesen werden darf.

Berechtigungen werden analog zum Konzept der rollenbasierten Berechtigungsverwaltung (siehe 2.3.2) in Rollen, hier mit Berechtigungsgruppen (*PermissionGroup*) bezeichnet, gruppiert.

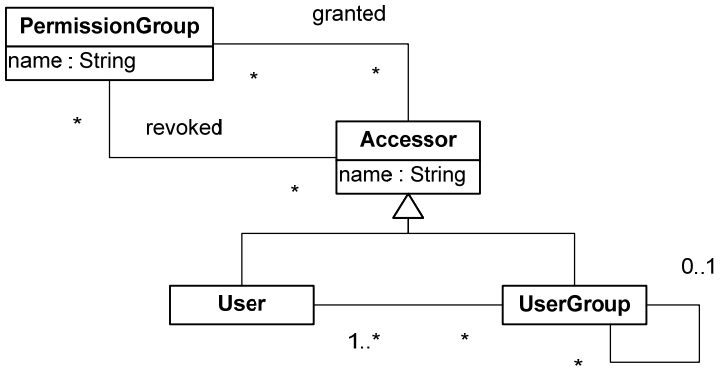


Abb. 2.6: Datenmodell für Benutzer, Quelle: [Nie03]

Das Datenmodell für die Benutzerverwaltung (siehe Abb. 2.6) sieht vor, Benutzer (*User*) oder Benutzergruppen (*UserGroup*) zu gruppieren. Beide Entitäten werden unter einer gemeinsamen Oberklasse gekapselt, so dass sie einer Berechtigungsgruppe zugeordnet werden können. Hierbei kann es sich um explizite Erlaubnisse (*Assoziationsname: granted*) oder explizite Verbote (*Assoziationsname: revoked*) handeln. Dieses Vorgehen wurde gewählt, um die Berechtigungsverwaltung bequemer zu gestalten. Es sei z. B. eine Gruppe von Personen gegeben, die alle einer bestimmten Berechtigungsgruppe zugeordnet werden. Falls genau eine Person aus dieser Gruppe eine gegebene Berechtigung nicht erhalten soll, so wird eine negative Berechtigung dieser Person auf diese Berechtigungsgruppe hinzugefügt, welche die positive Berechtigung über die Benutzergruppe überschreibt. (Die andere Lösung wäre natürlich, die betreffende Person einfach aus der Benutzergruppe zu entfernen. Allerdings kann es sein, dass diese Benutzergruppe mehreren Berechtigungsgruppen zugeordnet ist, so dass dann die gesamte Struktur der Benutzerverwaltung umgestaltet werden müsste. Dies wäre auf jeden Fall aufwendiger als das Hinzufügen einer einzelnen negativen Berechtigung.) Bei der Verwendung von sowohl positiven als auch negativen Berechtigungen sind Konsistenzregeln für die Berechtigungen untereinander sowie eine Konfliktauflösungsstrategie bereitzustellen, denn es kann nun zwei Berechtigungen geben, die sich gegenseitig widersprechen. Auf die Gestaltung von Berechtigungen für eine konfliktfreie Verwaltung wird in Kapitel 5.1.2 näher eingegangen.

Die Quasar Authorization, wie sie hier vorgestellt wurde, unterliegt, auch auf Grund der Komplexität spezifizierbarer Berechtigungen, einigen Einschränkungen. So werden z. B. die Aufgabenteilung und das statische und dynamische Aktivieren und Deaktivieren von Rollen gemäß des Prinzips der minimalen Rechte nicht unterstützt. In [Nie04] wurde deshalb ein neuer Entwurf für die Quasar Authorization Komponente vorgeschlagen. Dieser beinhaltet eine Sitzungsverwaltung, welche das selektive Aktivieren und Deaktivieren von Rollen beinhaltet.

Die Quasar Authorization eignet sich zwar sehr gut, um Berechtigungen in betrieblichen Anwendungen von der Geschäftslogik entkoppelt zu definieren und zu verwalten. Bei der Quasar Authorization handelt es sich aber „nur“ um eine ADF-Komponente. Die AEF ist hier nicht realisiert.

2.5 Zusammenfassung

Für die Realisierung einer Zugriffskontrolle für betriebliche Anwendungen sind mehrere Schritte notwendig. Zunächst muss ein geeignetes Berechtigungsmodell erstellt werden. Für betriebliche Anwendungen hat sich das rollenbasierte Zugriffskontrollmodell durchgesetzt. Aber auch das ältere, benutzerbestimmbare Zugriffskontrollmodell ist für einige Anwendungen relevant, insbesondere für Community-Anwendungen. Das gewählte Berechtigungsmodell, welches auch eine Mischung aus den beiden obigen Modellen sein kann, definiert bestimmte Entitäten, wie Gruppe, Rolle und Eigentümer, die sich in der Implementierung der Anwendung widerspiegeln müssen, damit auf Basis dieser die

konkreten Berechtigungen für eine gegebene Anwendung definiert werden können. Für die Spezifikation der Berechtigungen ist eine geeignete Berechtigungsbeschreibungssprache auszuwählen oder auszuarbeiten. Für jede Zugriffsanfrage durch ein Subjekt, werden die in der Berechtigungsspezifikation definierten Berechtigungen ausgewertet. Die Auswertung wird zentral durch eine Berechtigungsentscheidungskomponente (ADF-Komponente) vorgenommen. Für die Durchsetzung der Zugriffskontrolle gibt es derzeit noch keine anerkannte Vorgehensweise.

3 Datenzentrierte, dienstbasierte, betriebliche Anwendungen

In diesem Kapitel wird in Abschnitt 3.1 ein Modell für datenzentrierte und dienstbasierte Anwendungen aufgestellt. Es bildet ein breites Spektrum an betrieblichen Anwendungen ab. Die Anwendungsgebiete und Charakteristika betrieblicher Anwendungen, die sich im vorgestellten Modell wieder finden, werden in Abschnitt 3.2 besprochen. Das Kapitel schließt mit Erläuterungen, wie die Elemente der Berechtigungen der ISO/IEC-Norm 10181-3 (vgl. Kap. 2.4.1) in das vorliegende Modell integriert werden können, um dieses um Funktionalitäten der Zugriffskontrolle zu erweitern.

3.1 Modell einer datenzentrierten und dienstbasierten Anwendung

Ausgehend von der allgemeinen Architektur (siehe 3.1.1 – 3.1.3) werden die einzelnen Schichten einer datenzentrierten und dienstorientierten betrieblichen Anwendung vorgestellt (siehe 3.1.4 – 3.1.8).

3.1.1 Allgemeine Architektur

Die in dieser Arbeit betrachteten Anwendungen sind datenzentriert und dienstorientiert. Datenzentriert bedeutet hier, dass die Anwendung auf persistent gespeicherte Ressourcen zugreift. Diese Ressourcen sind die Geschäftsobjekte, auch Assets genannt, die innerhalb der Anwendung verarbeitet werden. Im Weiteren werden die Begriffe Asset und Geschäftsobjekt als Synonyme betrachtet. *Verzeichnis*, *Dokument*, *Person* und *Gruppe* sind Beispiele für fachliche Assettypen, die in einer Dokumentenverwaltung auftreten; *Buch*, *Exemplar*, *Bibliotheksmitarbeiter* und *Kunde* sind fachliche Assettypen eines Bibliothekssystems. Ein Asset ist eine konkrete Ausprägung eines Assettyps, also ein konkretes Verzeichnis, ein konkretes Dokument, usw.

Dienstorientiert bedeutet hier, dass die Anwendung den Benutzern verschiedene Dienste anbietet, die über eine dem Benutzer bekannte Adresse zugreifbar sind. Diese Adresse bildet den anwendungsweit eindeutigen Namen des Dienstes. Die Ausführung eines Dienstes setzt zumeist voraus, dass sich der Client gegenüber der Anwendung authentifiziert hat. Hierfür wird ein Authentifizierungsdienst angeboten. Diesen muss ein Client zuerst aufrufen, bevor er zugriffsgeschützte, für anonyme Clients nicht zugreifbare, Dienste ausführen kann. Der Authentifizierungsdienst nutzt zur Erfüllung seiner Aufgaben eine Authentifizierungskomponente.

Datenzentrierte und dienstorientierte Anwendungen, die die obigen Eigenschaften erfüllen, werden im Weiteren mit *betrieblicher Anwendung* (engl.: information system) bezeichnet.

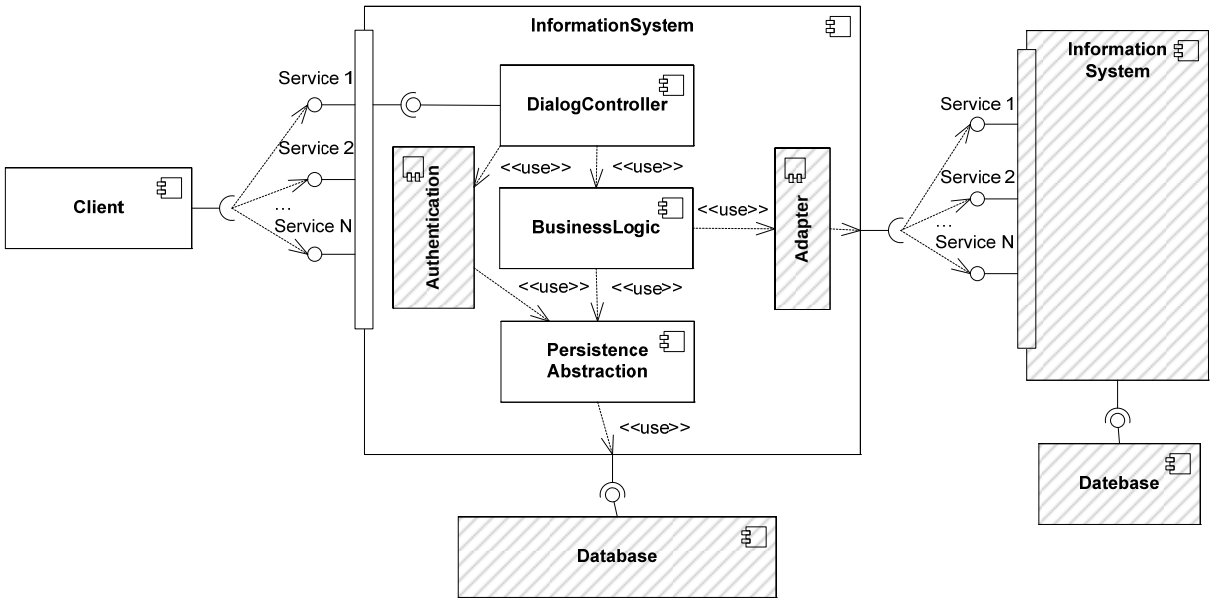


Abb. 3.1: Modell einer betrieblichen Anwendung

Die allgemeine Architektur einer solchen Anwendung sowie die Schnittstellen, die diese zur Verfügung stellt und nutzt, sind in Abb. 3.1 dargestellt. Die Architekturbestandteile, welche in dieser Arbeit nicht näher behandelt werden, sind schraffiert dargestellt. Der Client ist ein Programm, welcher im Namen eines Benutzers oder einer anderen Anwendung läuft. In einer webbasierten Umgebung ist dies z. B. ein Webbrowser, der dem Benutzer eine graphische Oberfläche (engl.: graphical user interface, GUI) zur Verfügung stellt. Der Benutzer selbst ist eine natürliche Person.

Die betriebliche Anwendung bietet dem Client eine Reihe von Diensten an. Über die Dienstschnittstellen kann der Benutzer *(Dienst-)Anfragen* (engl.: service requests) an die betriebliche Anwendung stellen und damit den entsprechenden Dienst anstoßen. Die betriebliche Anwendung greift zur Diensterbringung auf einen persistenten Speicher zu, dessen Daten in Form von Assets innerhalb der Anwendung genutzt werden.

Die betriebliche Anwendung kann über eine Adapterkomponente Dienste anderer Anwendungen aufrufen, die ihrerseits wieder über ihre Adapterkomponente weitere Dienste aufrufen können, usw.. Hierbei fungiert die Adapterkomponente als Client für die aufgerufene Anwendung. Die aufgerufene Anwendung verfügt über eine eigene Authentifizierung. Im Weiteren werden die von einer betrieblichen Anwendung realisierten Schnittstellen zu externen Systemen, seien dies Datenbanken oder andere Anwendungen, nicht weiter betrachtet. Es wird von einer einzelnen Anwendung ausgegangen, deren Dienste von einem Client genutzt werden.

3.1.2 Dienste

Ein *Dienst* (engl.: service) setzt eine fachliche Anforderung einer betrieblichen Anwendung um [KBS05, HHV06]. Er bietet einem Anwender eine wohl definierte Funktionalität über eine Schnittstelle an. Diese Schnittstelle bildet einen Teil eines Vertrags [KBS05, S.59] zwischen dem Aufrufenden und dem Diensterbringer. Wesentliche Eigenschaften eines Dienstes sind, dass dieser zu anderen Diensten lose gekoppelt ist, d. h. unabhängig ausgeführt werden kann, und, dass dieser in einem transaktionalen Kontext abläuft, d. h. ganz oder gar nicht ausgeführt wird [HHV06]. Er führt im einfachsten Fall eine *Basisfunktionalität* auf einem Asset aus. Basisfunktionalitäten sind das Anlegen und Löschen von Assets, das Suchen und Anzeigen von Assets nach bestimmten Such- und Sortierkriterien, sowie das Ansehen und Aktualisieren von Detailinformation zu den einzelnen Assets. Hier wird davon ausgegangen, dass die betriebliche Anwendung in einer objektorientierten Programmiersprache, z. B. in Java, realisiert ist. Dann sind die einzelnen Assets als Klassen realisiert, die Basisfunktionalitäten werden als Methoden angeboten. Die Implementierung dieser Methoden nutzt

die Persistenzabstraktionsschicht, um die Änderungs- und Leseoperationen auf der darunter liegenden Datenbank auszuführen.

Bei der Nutzung einer GUI können die Anfragen über die GUI-Elemente abgesetzt werden. Ansonsten muss der Client die Adresse des Dienstes kennen. So hat der Klick auf einen Button auf der Benutzeroberfläche etwa die Bedeutung des Speicherns der auf dieser Seite eingegebenen Daten oder des Ausführens einer Suchoperation gemäß den eingegebenen Suchkriterien, der Klick auf einen Link ermöglicht etwa das Anzeigen von Detailinformationen zu einem Geschäftsobjekt oder die Navigation zu weiteren vernetzten Inhalten. Die Anfrage enthält neben der Information darüber, welcher Dienst angesprochen werden soll, auch weitere Parameter, welche der Dienst für die Anfragebearbeitung benötigt. Bei einer Suchfunktion bspw. enthalten die Parameter die Such- und Sortierkriterien, nach denen die Suche durchgeführt und das Ergebnis angezeigt werden soll. Handelt es sich bei der Kommunikationsform zwischen (entferntem) Client und der Anwendung um ein zustandsloses Protokoll, so müssen auch Zustandsinformationen über den Client mit in der Anfrage übergeben werden. Hierzu gehört z.B. die Information, dass die Anfrage einer bestimmten Sitzung zugeordnet werden kann. Dies ist insbesondere für Fälle wichtig, in denen mehrere, in einem Workflow zusammenhängende Dienste hintereinander ausgeführt werden. Z. B. könnte es einen Dienst geben, welcher ein Geschäftsobjekt löscht; der Dienst bereitet die Löschaktion vor und lässt den Client in einer neuen Dienstanfrage bestätigen, dass die Löschaktion tatsächlich durchgeführt werden soll. Jeder Dienst selbst kann also als einfache Transaktion (siehe z. B. [KE06]) angesehen werden. Der Dienst wird entweder vollständig oder gar nicht ausgeführt, er führt nur konsistente Änderungen durch, die für jeden Client isoliert ablaufen und speichert die Ergebnisse dauerhaft in einer Datenbank.

Nach erfolgter Anfragebearbeitung bzw. Dienstausführung kann der Dienst eine *Antwort(-nachricht)* (engl.: response) zurückliefern, welche an den anfragenden Client zurückgesendet wird. Sie enthält Informationen, welche dem Benutzer präsentiert werden.

3.1.3 Schichtenarchitektur

Die betriebliche Anwendung ist in mehreren Schichten aufgebaut, wobei eine Schicht immer nur die Funktionalitäten nutzen kann, die die jeweils direkt unter ihr liegende Schicht zur Verfügung stellt. Die oberste Schicht ist die Dialogsteuerung. Sie ist für die Darstellung der Inhalte auf der Benutzeroberfläche verantwortlich. Sie kontrolliert außerdem die Reihenfolge, in der die verschiedenen Interaktionsschritte ausgeführt werden, um einen Dienst zu erbringen. Die mittlere Schicht bildet die Geschäftslogikschicht. Sie enthält die von der Anwendung verwalteten Geschäftsobjekte. Die Dialogsteuerung ruft die Methoden der Geschäftsobjekte für die Erbringung eines Dienstes auf. In Analogie zur Model-View-Controller Architektur [RWL96] fasst die Dialogsteuerung den View und den Controller zusammen. Die Geschäftslogikschicht bildet das Model.

Eine weitere Schicht ist die Persistenzabstraktionsschicht. Sie bildet die Schnittstelle zwischen der verwendeten Datenbank und der Geschäftslogikschicht. Sie sorgt dafür, dass die Repräsentation der Daten, wie sie in der Datenbank vorliegen, in eine Repräsentation umgewandelt wird, die dem Modell der Geschäftslogikschicht genügt und umgekehrt.

In der Quasar-Referenzarchitektur für betriebliche Anwendungen [Sie04] fasst die Dialogkomponente die Funktionalitäten des View, dort Präsentation genannt, und die Funktionalitäten des Controllers, dort mit Dialogkern bezeichnet, zusammen. Sie entspricht der Dialogsteuerung dieses Modells. Die Funktionalität der Geschäftslogikschicht und der Persistenzabstraktionsschicht dieses Modells entsprechen der Funktionalität der Anwendungskernkomponente des Quasar-Modells. Ein Asset im vorliegenden Modell entspricht dem Entitätstypen der Datenmodellierung im Anwendungskern von Quasar. Abb. 3.2 stellt die wichtigsten Klassen jeder Schicht des vorliegenden, datenzentrierten und dienstorientierten Modells vor. Sie werden in den folgenden Unterkapiteln näher erläutert. Die Klasse *Asset* im Paket *BusinessLogic* ist schraffiert dargestellt, da es sich hierbei um eine Klasse aus der Persistenzabstraktionsschicht handelt. Konkrete Subklassen von *Asset* gehören der Geschäftslogikschicht an.

Eine konkrete technische Infrastruktur, auf die eine solche betriebliche Anwendung aufbauen kann, ist die Java EE-Architektur (vorher: J2EE-Architektur) von Sun Microsystems. Die Java EE-Architektur bildet ein *Rahmenwerk* (engl.: Framework), welches Richtlinien und Spezifikationen liefert, die

das Design einer mehrschichtigen, serverseitigen, in Java implementierten Anwendung vorgeben, also genau den Anforderungen an das hier aufgestellte Modell genügt. Außerdem unterstützt Java EE Persistenzmechanismen, Transaktionen und weitere Mechanismen, welche im Umfeld betrieblicher Anwendungen wichtig sind. In der Java EE-Architektur wird die Präsentationskomponente durch Servlets oder Java ServerPages gebildet. Die Geschäftslogikkomponente wird in so genannten Enterprise Beans implementiert. Eine Einführung in die Java EE-Architektur findet sich in [BCE+06].

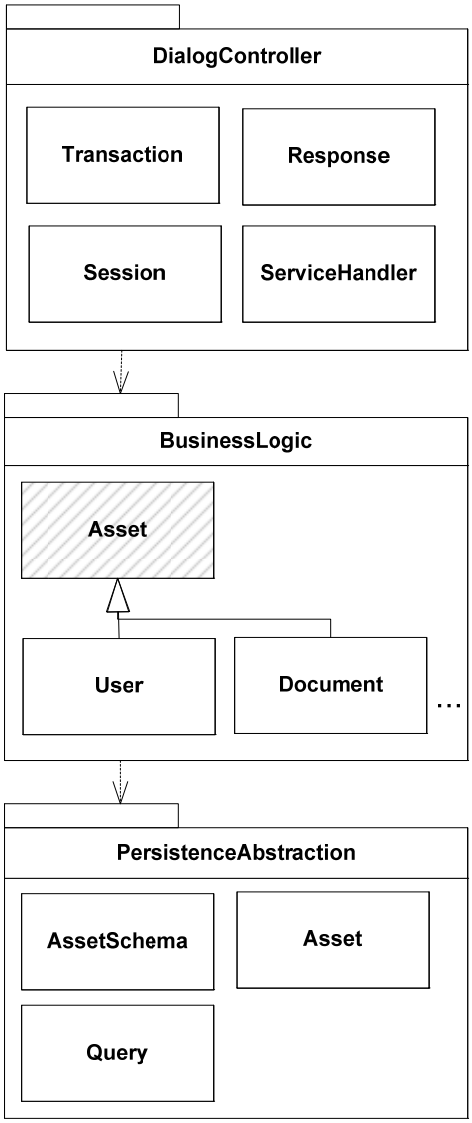


Abb. 3.2: Wichtigste Klassen in den Schichten des Modells

3.1.4 Die Dialogsteuerung

Die oberste Schicht des hier vorgestellten Modells bildet die Dialogsteuerung. Die Aufgabe der Dialogsteuerung ist es, die Client-Sitzung zu identifizieren, bzw. eine anonyme Sitzung zu erstellen, falls der Client noch keine besitzt. Eine anonyme Sitzung ist eine Sitzung, die für nicht-authentifizierte Clients bereitgestellt wird. Diese Clients dürfen dann nur nicht-zugriffsbeschränkte Dienste nutzen. Jedem Client wird eine Sitzung mit einer eindeutigen Sitzungs-ID zugeordnet. Insbesondere existiert für authentifizierte Clients ein Abbild im persistenten Speicher, ein Objekt vom Typ User, der den authentifizierten Client repräsentiert. Dieses User-Asset wird der Sitzung zugeordnet. Der User ent-

spricht dem in Kapitel 2 definierten Subjekt, so dass alle während der Dienstausführung notwendigen Berechtigungsabfragen mit dem in der Sitzung gespeicherten User als Subjekt durchgeführt werden.

Der Kommunikationsablauf zwischen Client, betrieblicher Anwendung und Objekten der Dialogsteuerung ist in Abb. 3.3 in einem UML-Sequenzdiagramm [OMG05] dargestellt. Die betriebliche Anwendung kann als Server angesehen werden, welcher ständig auf Clientanfragen wartet. Der Client stellt eine Anfrage an einen bestimmten Dienst, indem er eine Anfrage an den Server stellt, der diese an die Dialogsteuerung weiterreicht. Die Anfrage enthält den Namen des Dienstes. Evtl. werden weitere Parameter mitgeliefert. Zum einen ist dies eine bestehende Sitzungs-ID, zum anderen sind dies weitere Parameter, die für die Ausführung des spezifischen Dienstes notwendig sind. Die Kommunikationsverbindung zwischen dem Client und der Anwendung sei derart, dass sie nicht durch Dritte abgehört oder abgefangen werden kann, so dass sichergestellt ist, dass die Anwendung tatsächlich mit dem Client kommuniziert, der sich authentifiziert hat und dass umgekehrt der Client sicher sein kann, mit der richtigen Anwendung in Verbindung zu stehen. Die Sitzungs-ID sei fälschungssicher und eindeutig einem Client zuordenbar.

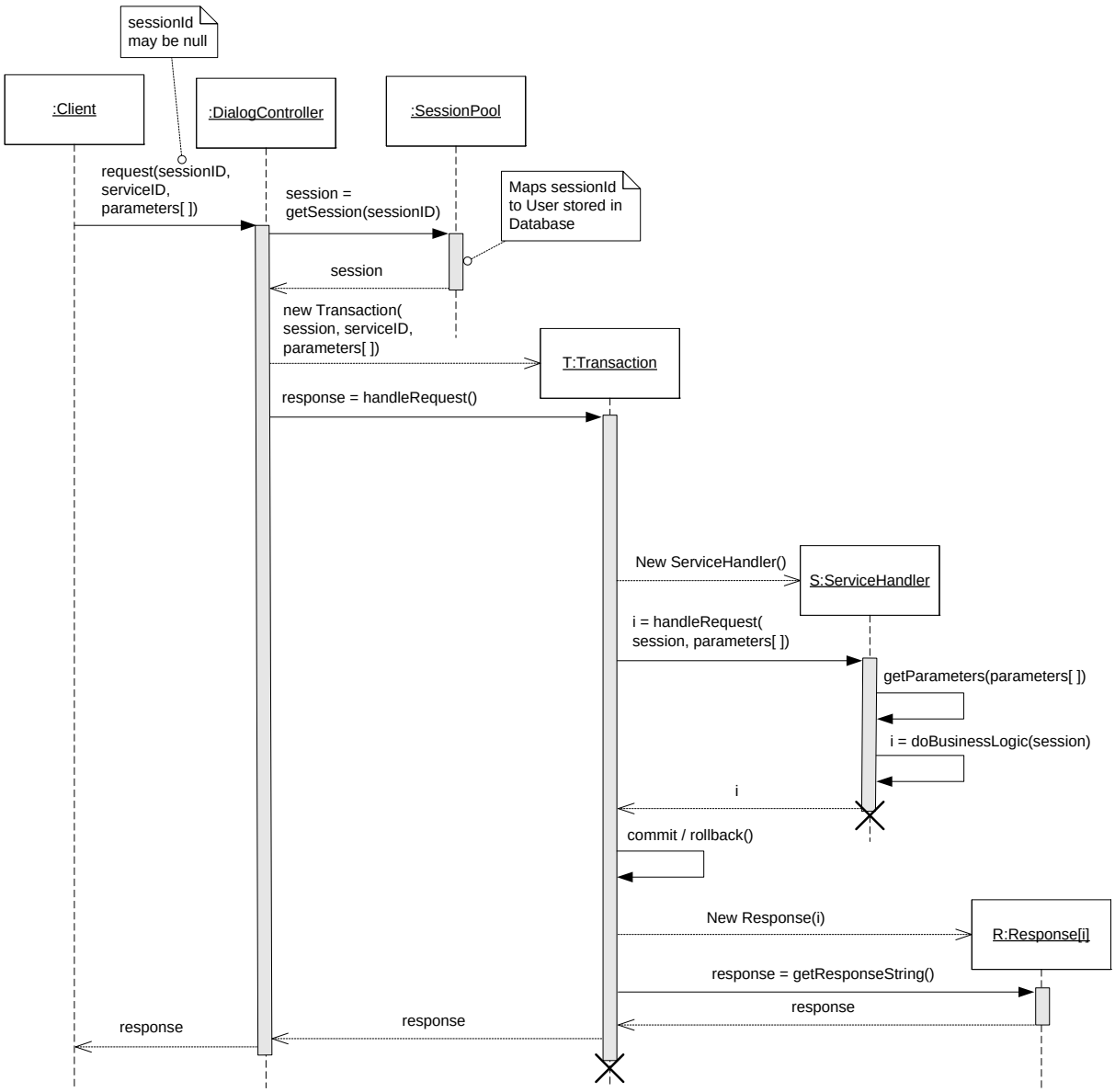


Abb. 3.3: Anfrageverarbeitung in der Dialogsteuerung

Trifft eine Anfrage ein, so wird zunächst durch den *DialogController* die Sitzung identifiziert oder eine neue Sitzung erzeugt. Danach wird zur Abarbeitung der Anfrage ein separater, leichtgewichtiger Prozess innerhalb des Programms der betrieblichen Anwendung gestartet. Dadurch können viele Anfragen gleichzeitig bearbeitet werden. Der leichtgewichtige Prozess hat hier transaktionalen Charakter und wird dementsprechend in der Abbildung als *Transaktion* bezeichnet. Der Dienst, den ein Client anfragt, wird entweder vollständig oder gar nicht ausgeführt. Im letzteren Fall erhält der Client eine Fehlermeldung zurück. Die Transaktion leitet die Anfrage an den entsprechenden Dienstbearbeiter, in den Abbildungen mit *ServiceHandler* bezeichnet, weiter. Für die Dienstausführung wird ein neues Objekt des angesprochenen Dienstbearbeiters instantiiert, so dass derselbe Dienstbearbeiter parallel für andere Clients ausgeführt werden kann. Die Dienstbearbeitung läuft isoliert für jeden Dienstaufruf, so dass ein Client die Illusion hat, der alleinige Benutzer der Anwendung zu sein. Der Dienstbearbeiter selbst liest die evtl. mitgelieferten Parameter aus und führt die eigentliche Dienstleistung aus. Die Ausführung dieser ist hier als Methode *doBusinessLogic( )* angegeben. Die *doBusinessLogic( )* greift für die Diensterbringung auf andere Methoden der Geschäftslogikschicht zu. Die Diensterbringung kann aber muss nicht mit dem Aufruf eines einzigen Dienstbearbeiters beendet sein. Sie kann auch mehrere Dienstbearbeiter umfassen. Der Fall einer zweiten Dienstbearbeiterinstantiierung nach Ausführung des ersten Dienstbearbeiters ist in Abb. 3.4 gezeigt. Der Dienstbearbeiter, welcher zu starten ist, wird aus dem Rückgabewert der *doBusinessLogic( )* entnommen. Es können auch mehr als zwei Dienstbearbeiter bei einer Diensterbringung beteiligt sein.

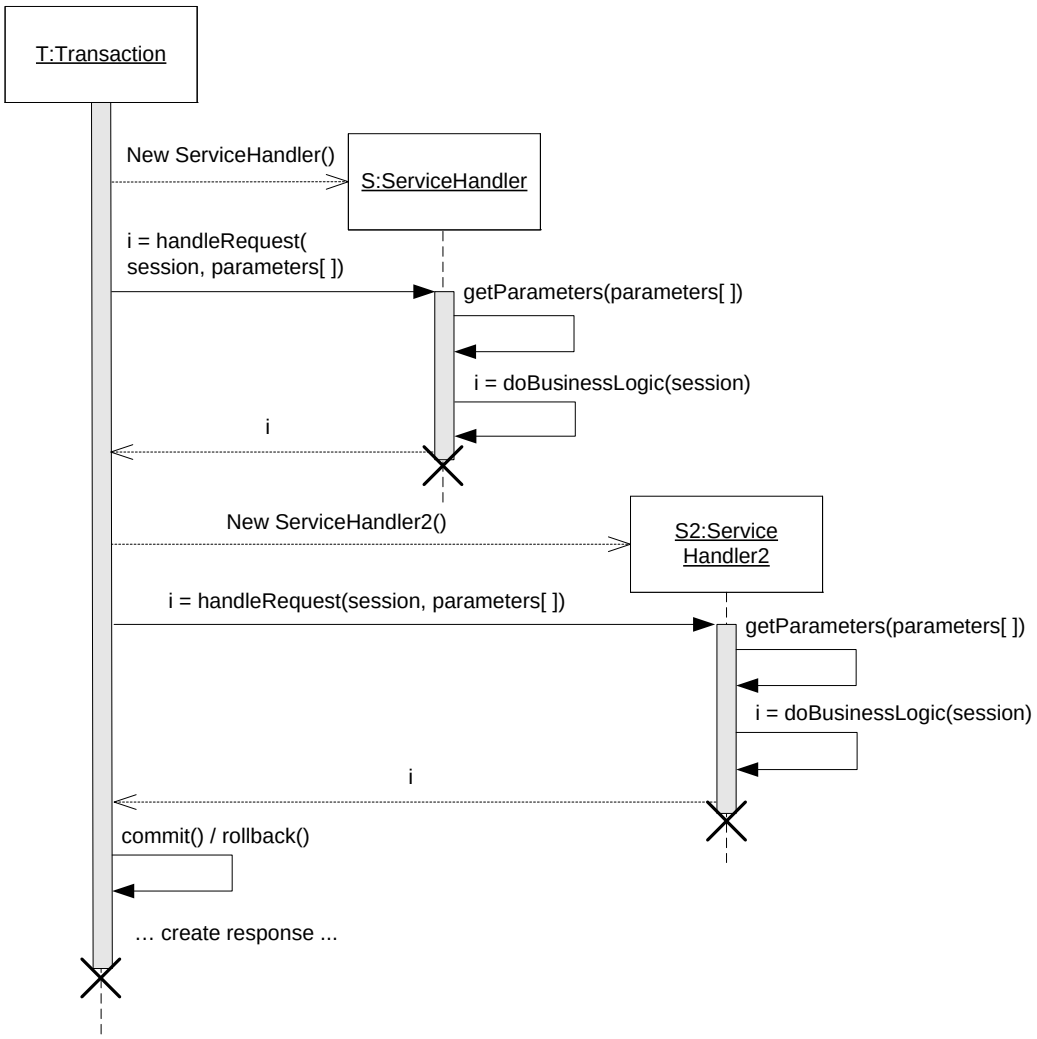


Abb. 3.4: Diensterbringung über mehrere Dienstbearbeiter

Die gesamte Diensterbringung kann erfolgreich oder nicht erfolgreich abschließen. Bei erfolgreichem Abschluss werden alle Änderungen an bestehenden Assets und alle neu erstellten Assets dauerhaft im persistenten Speicher abgelegt. Bei nicht erfolgreichem Abschluss wird keine der Änderungen in den persistenten Speicher übernommen.

Die Ausführung des Dienstes schließt immer mit der Erstellung einer Rückantwort an den Client, in den Abbildungen mit *Response* bezeichnet, ab. Je nach Verlauf der Dienstausführung werden verschiedene Dienstantworten erstellt. Dieser Sachverhalt ist in den Abb. 3.3 und 3.4 schematisch dadurch dargestellt, dass die *doBusinessLogic( )* und damit die *handleRequest( )* verschiedene *int*-Werte zurückgibt. Anhand des int-Wertes entscheidet die Transaktion, ob und welche Antwortnachricht zu erstellen ist oder ob und welcher weitere Dienstbearbeiter aufgerufen wird. Die *doBusinessLogic( )* liest während ihrer Ausführung Assets aus dem persistenten Speicher aus. Auf die in der jeweils letzten *doBusinessLogic( )* ausgelesenen Assets kann das Response-Objekt zugreifen. Innerhalb der Erstellung der Antwort können auch lesende Aufrufe von Methoden der Geschäftslogik getätigt werden. Das Erstellen der Antwort findet in diesem Modell nach einem *commit( )* oder *rollback( )* der Transaktion statt, da sonst eventuell Sperren auf Teile des persistenten Speichers sehr lange gesetzt bleiben müssen. Die Erstellung der Antwort enthält nur noch lesende Zugriffe auf den persistenten Speicher, so dass die Konsistenz der gespeicherten Daten nicht beeinflusst wird. Sehr wohl können bei dieser Modellierungsvariante aber Lese-Anomalien auftreten. Die lesenden Zugriffe beschränken sich auf die Abfrage einzelner Eigenschaften konkreter Asset-Objekte oder auf Filteranfragen auf Mengen von Assets. Die Response muss von dem Client wieder interpretiert werden können. Für die Response eignet sich also ein maschinenlesbares Format, wie etwa XML (siehe [BPS+06]). Ist der Client ein Webbrowser, so kann die Response im HTML-Format zurückgeliefert werden.

Das Zusammenspiel der Klassen der Dialogsteuerung ist noch einmal in einem UML-Klassendiagramm [OMG05] (siehe Abb. 3.5) dargestellt, welches die Beziehungen der einzelnen Klassen untereinander verdeutlicht.

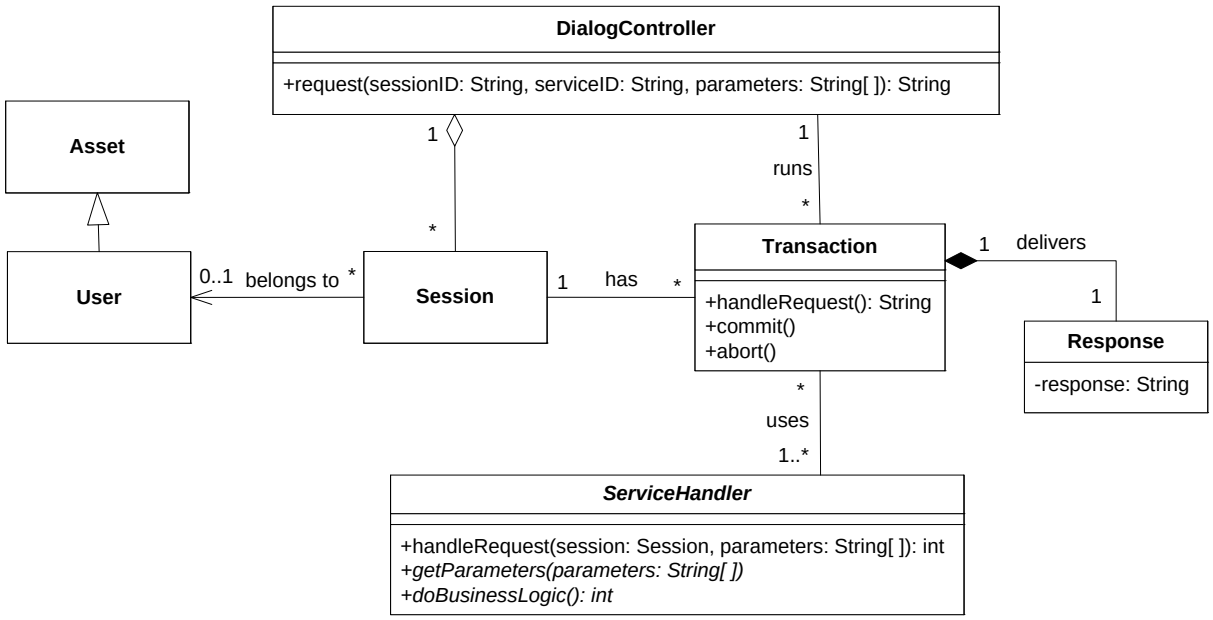


Abb. 3.5: Klassendiagramm der Dialogsteuerung

3.1.5 Die Geschäftslogik

Die Geschäftslogikschicht operiert auf den Assets. Sie enthält für jeden Assettyp einen Container, welcher die Assets dieses Typs verwaltet. Jeder Container ist ein Singleton. Er muss nur einmal pro

Geschäftsobjekttyp vorhanden sein. Die Assets selbst haben jeweils eine eindeutige, fälschungssichere ID, welche nur einmal verwendet wird, d. h. IDs von einmal gelöschten Assets werden nicht wieder verwendet, da sonst Verwechslungsgefahr zwischen Assets besteht. Die konkreten Assets und Container genügen einem Metamodell. Dieses ist der Persistenzabstraktionsschicht zugeordnet, die in Unterkapitel 3.1.6 besprochen wird.

Abgesehen von den Geschäftsobjekten, welche in einer objektorientierten Umgebung in der Anwendung als Klassen vorliegen, besteht für die Geschäftslogik kein Modell. Es werden keine weiteren Annahmen über die Struktur des Quellcodes der Geschäftslogik getroffen. Insbesondere gehört die Methode *doBusinessLogic( )* des ServiceHandlers auch zur Geschäftslogik dazu.

3.1.6 Das Metamodell der Persistenzabstraktion

Die Persistenzabstraktionsschicht kapselt den Zugriff auf den darunter liegenden persistenten Speicher. Außerdem stellt sie ein Metamodell zur Verfügung, welches beschreibt, wie die Assets und die Container für die einzelnen Assettypen aussehen. Das Metamodell ist an MOF [OMG02] angelehnt und wurde aus [Bue07] entnommen. Das *AssetSchema* ist die Oberklasse für alle Container. Es enthält zum einen die Beschreibung der Attribute, die den Assettyp definieren. Dies sind im Datenmodell die *Property*-Klassen. Eine Property repräsentiert eine Eigenschaft eines Assets, die als primitiver Datentyp, also z. B. als int, Boolean oder String, modelliert werden kann. Im Weiteren werden die Begriffe Attribut und Property als Synonyme betrachtet. Außerdem werden alle Datentypen als primitiv bezeichnet, die nicht vom Typ eines Assets sind. Die nicht-primitiven Datentypen stellen binäre Beziehungen zwischen Assets dar. Sie werden nicht als Felder von Klassen modelliert, sondern als unabhängige Klassen. Beziehungen werden hier als Assoziationen (*Associations*) dargestellt. Jede binäre Assoziation hat zwei Assoziationsenden, die die Wertigkeit (Multiplizität) der an der Assoziation beteiligten Assettypen beschreibt. Gängige Wertigkeiten sind 0..1, 1, 1..* und *. * bezeichnet hierbei eine Wertigkeit, die keine, eine oder beliebig viele Elemente enthält. Assoziationen, bei denen mindestens ein Assoziationsende mit einem positiven Integerwert beschriftet ist, dies ist z. B. für 1 und 1..* der Fall, werden hier als *Muß-Assoziationen* oder *Pflichtassoziationen* bezeichnet. Es handelt sich hierbei um Assoziationen, die für jede konkrete Instantiierung des UML-Klassendiagramms der Anwendung als UML-Objektdiagramm eine Ausprägung als Referenz zwischen den jeweiligen Instanzen der Assettypen haben müssen. Assoziationen, bei denen beide Assoziationsenden mit 0..1 oder * beschriftet sind, werden hier als *Kann-Assoziationen* bezeichnet. Sie können, aber müssen nicht, in einem UML-Objektdiagramm der Anwendung vorhanden sein.

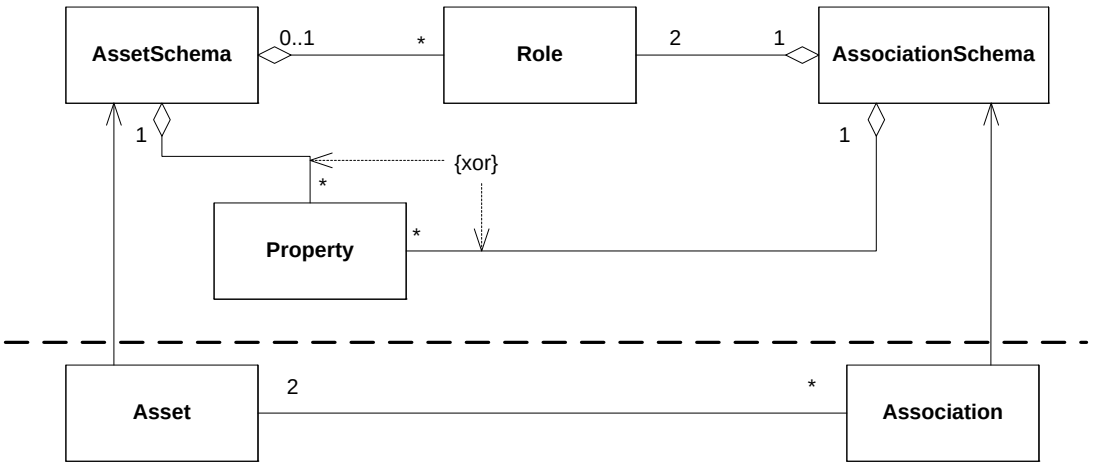


Abb. 3.6: Metamodell der Geschäftslogik

Eine Assoziation wird durch ein *AssociationSchema* beschrieben. Ein Assoziationsschema beschreibt die Wertigkeit der Assoziation und identifiziert die Rollen, die die Assoziation ausmachen. In diesem Metamodell sind zwei verschiedene Typen von Assoziationsschemata vorgesehen, die 1..*-Assozia-

tion und die *..*-Assoziation (siehe Abb. 3.7). Hierüber werden sowohl Muß- als auch Kann-Assoziationen modelliert, d. h. sowohl 0..1-1-Beziehungen als auch 1-1-Beziehungen werden über die 1..*-Assoziation modelliert. In diesem Metamodell werden nur binäre, keine ternären oder höherwertigeren Assoziationen modelliert. Von daher definiert ein AssociationSchema genau zwei *Role*-Klassen. Diese geben der Assoziation zwei Rollennamen, einen für jedes Assoziationsende. Die Rolle einer Assoziation kann genau einem AssetSchema zugeordnet werden. Falls das genaue AssetSchema, auf das die Rolle verweist, nicht definiert ist, kann es keinem AssetSchema zugewiesen werden. Assoziationen können selbst wieder durch primitive Eigenschaften erweitert werden. In einem UML-Klassendiagramm würde man dies als Assoziationsklasse modellieren. In diesem Metamodell sind die weiteren Eigenschaften einfach wieder Property-Klassen.

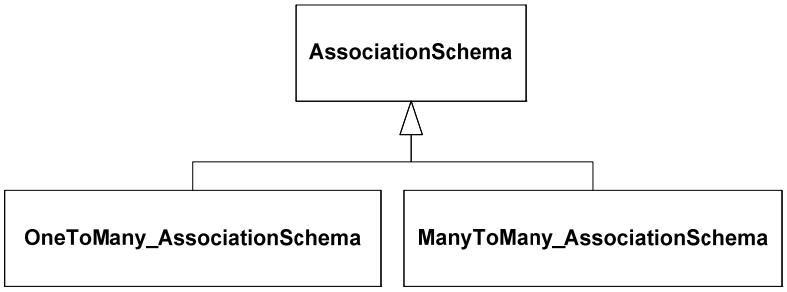


Abb. 3.7: Assoziationstypen

Abb. 3.8 stellt einige wichtige Methoden der AssetSchema- und Asset-Klassen dar. Das AssetSchema enthält Methoden zur Verwaltung des Lebenszyklus, also *createAsset( )*- und *remove( )*-Methoden für die Assets, die dem jeweiligen AssetSchema genügen. Es stehen ebenfalls Methoden zur Verfügung, welche ein (*getAsset(…)*) oder alle (*getAssets( )*) Assets dieses Schemas zurückliefern oder nach bestimmten Suchkriterien Assets suchen (*queryAsset(…)*). Die Assets selbst enthalten eine Vielzahl von Methoden, um ihre Properties zurückzugeben (*getProperty(…)*) oder zu setzen (*setProperty(…)*), sowie die Assoziationen zu anderen Assets über den Rollennamen anzulegen (*createAssociation(…)*) und zu löschen (*removeAssociation(…)*).

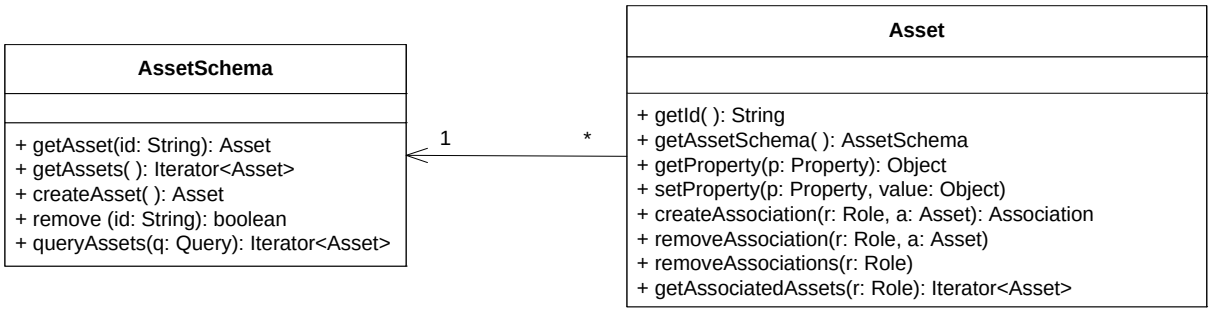


Abb. 3.8: Methoden von Asset und AssetSchema

Konkrete AssetSchemas und Assets der Geschäftslogik sind Subklassen der Klassen AssetSchema und Asset aus dem Metamodell. Wird z. B. in der Anwendung ein Geschäftsobjekt vom Typ User verwaltet, so gibt es eine Klasse Users, welche AssetSchema erweitert, sowie eine Klasse User, welche Asset erweitert. Die Methoden zum Holen, Anlegen und Löschen von Assets sind bereits generisch im AssetSchema implementiert. Eine konkrete Containerklasse definiert die Properties des Geschäftsobjekttyps sowie verschiedene, konkrete Suchmethoden. Sie kann auch Methoden zum Anlegen und Löschen von Geschäftsobjekten implementieren, greift in ihrer Realisierung aber auf die von AssetSchema bereitgestellten Methoden zurück. Eine konkrete Asset-Klasse enthält Methoden zum Holen und Setzen der Properties und zum Anlegen und Löschen von Beziehungen zu anderen Assets.

Für die Realisierung dieser Funktionalität greift die konkrete Asset-Klasse ausschließlich auf die durch die Oberklasse Asset bereitgestellten Methoden zurück.

Die Geschäftslogik nutzt nun die konkreten Assets und AssetSchemas, indem sie sich über einen, genau einmal in der Anwendung vorhanden *AssetSchemaManager* zunächst das AssetSchema holt. Von dort aus kann auf konkrete Assets zugegriffen werden. Im Folgenden wird ein Beispiel für den Zugriff auf ein User-Objekt gegeben (siehe Abb. 3.9). Die Implementierung der konkreten Assets greift auf die im Metamodell realisierten Methoden zu, und kommuniziert so mit der Persistenzabstraktionsschicht.

```
AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
Users users = asm.getUsers();
User u = users.getUser(/* some id */);
```

Abb. 3.9: Zugriff auf ein User-Objekt

3.1.7 Abbildung des Metamodells auf ein relationales Datenbankschema

Das hier beschriebene Metamodell greift wiederum auf Klassen zu, die die Abbildung auf den darunter liegenden Persistenzspeicher vornehmen (siehe Abb. 3.10). Hier gehen wir davon aus, dass es sich um eine relationale Datenbank handelt. Die Abbildung eines objektorientierten Datenmodells, welches Assets und ihre Beziehungen untereinander modelliert, auf eine relationale Datenbank kann nach einem stereotypen Muster erfolgen. Für jeden Geschäftsobjekttyp wird eine Datenbanktabelle mit entsprechenden Spalten für die Attribute (im Metamodell: Property) reserviert. Dies wird durch eine *Container*-Klasse dargestellt. Jedes Asset wird auf eine Tabellenzeile abgebildet. Diese ist in der Abbildung als *Content*-Klasse dargestellt. Assoziationen zwischen Assets werden, falls es sich um eine 1-zu-*-Beziehung handelt als Fremdschlüssel dargestellt. Falls es sich um eine *-zu-*-Beziehung handelt, kann eine Beziehungstabelle angelegt werden. Diese ist wiederum eine *Container*-Klasse. Das relationale Datenmodell verfügt über keine Konstrukte, die eine Vererbung ausdrücken. Es gibt allerdings einige Mechanismen, um Spezialisierungs- und Generalisierungsbeziehungen zu realisieren [KE06, Kap. 4.17]. So kann z.B. ein Geschäftsobjekttyp mit all seinen Unterklassen in einer Tabelle mit entsprechend vielen Spalten für die Attribute dargestellt werden. Je nach Geschäftsobjekt, welches jeweils eine Tabellenzeile einnimmt, können nicht vorkommende Attribute auf *null* gesetzt werden.

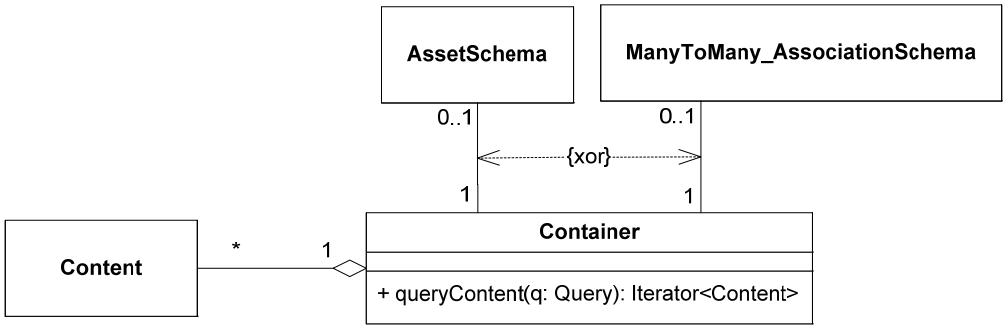


Abb. 3.10: Abbildung des Metamodells auf Klassen, die die Abbildung auf die Datenbank vornehmen

3.1.8 Die Anfragesprache der Persistenzabstraktion

Die Implementierung der Methoden der Metamodellklassen AssetSchema und Asset nutzt die von der Persistenzabstraktion zur Verfügung gestellte Anfragesprache um Änderungen persistent zu speichern und Daten aus der Datenbank auszulesen.

Die Anfragesprache, *Queries* genannt (siehe Abb. 3.11), die hier verwendet wird, erlaubt es, Anfragen über die Assets, die in genau einem AssetSchema verwaltet werden, an die Datenbank zu stellen. Anfragekriterien können auf den Properties der Assets definiert werden. Diese Kriterien sind die üblichen Operatoren, wie „<“, „<=“, „==“ usw. Mehrere solcher Anfragekriterien können mit den logischen Operatoren „und“, „oder“ und „nicht“ verknüpft werden. Die eigentliche Datenbankanfrage erfolgt über den entsprechenden *Container* und die Methode *queryContent( )*, welche ein Set an *Content*-Objekten zurückliefert, von denen jedes auf genau ein Asset abgebildet werden kann.

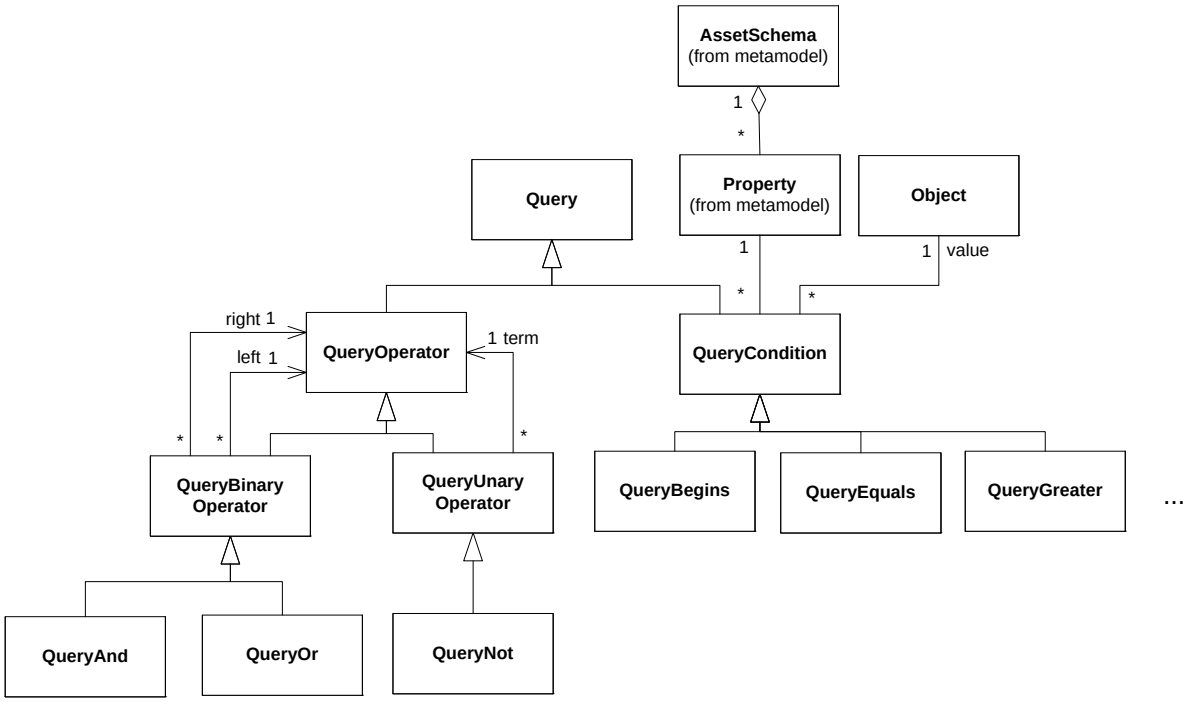


Abb. 3.11: Die Queries-API der Persistenzabstraktion

Die Queries, die hier formuliert werden können, sind also Filteranfragen auf AssetSchemas. Filteranfragen werden ausschließlich in den Suchmethoden der AssetSchemas implementiert. Sie können sowohl innerhalb der Dienstdurchführung, also innerhalb der oben beschriebenen *doBusinessLogic( )*-Methode, als auch während der Erstellung der Response (siehe Abb. 3.12) auftreten. Innerhalb der *doBusinessLogic( )* werden außerdem Methoden benutzt, um zwischen den Beziehungen der einzelnen Assets zu navigieren, z. B. können ausgehend von einem Verzeichnis alle dort enthaltenden Dateien geholt werden. Für Suchanfragen, die über eine bestehende Assoziation gestellt werden können, wird ausschließlich die *getAssociatedAssets( )*-Methode verwendet.

- `doBusinessLogic()` greift über konkrete Assets und AssetContainer zu auf:
  `AssetSchema: getAsset(), getAssets(), queryAssets(), createAsset(), remove()`
  `Asset`: alle Methoden

- `Response()` greift über konkrete Assets und AssetContainer zu auf:
  `AssetSchema: getAsset(), getAssets(), queryAssets()`
  `Asset: getId(), getProperty(), getAssociatedAssets()`

Abb. 3.12: Mögliche Aufrufe der Methoden des Metamodells über die Methoden konkreter Assets und AssetSchemas

3.2 Aufgaben betrieblicher Anwendungen

Das hier vorgestellte datenzentrierte und dienstorientierte Modell ist auf ein breites Spektrum von Anwendungssystemen anwendbar, die alle unter der Kategorie „betriebliche Anwendung“ klassifiziert werden können. Im folgenden Unterkapitel wird zunächst auf die Anwendungsgebiete betrieblicher Anwendungen eingegangen, wie sie in der gängigen Wirtschaftsinformatikliteratur beschrieben werden. Im Anschluss wird gezeigt, dass alle diese Systeme dieselben Charakteristika aufweisen, die durch das oben beschriebene Modell abgedeckt werden können.

3.2.1 Anwendungsgebiete

Softwaretechnische Lösungen, welche die operativen Prozesse und Führungsaufgaben innerhalb eines Unternehmens unterstützen, sind die *betrieblichen Anwendungen* oder genauer *betrieblichen Anwendungssysteme* eines Unternehmens. So gibt es Administrations- und Dispositionssysteme, welche die operativen Prozesse und Routineaufgaben innerhalb eines Unternehmens unterstützen. Planungs- und Kontrollsysteme werden in den höheren Ebenen einer Firma eingesetzt und unterstützen die Führungskräfte bei ihrer Entscheidungsfindung [FSV01, SH02]. Es gibt eine Reihe von betrieblichen Anwendungssystemen, die als Querschnitts- oder branchenneutrale Systeme bezeichnet werden. Das Personal- und Rechnungswesen zählt z. B. zu dieser Kategorie [FSV01, SH02], obwohl es hier sicherlich auch Details wie etwa Urlaubsverwaltung oder Reisekostenabrechnung gibt, die in einer Branche oder gar in einer Firma individuell gehandhabt werden.

Das Gegenstück zu den branchenneutralen Systemen sind die branchenspezifischen Anwendungssysteme. Im Industriesektor gibt es z.B. Systeme für die Produktionsplanung und -steuerung, welche Angebote bearbeiten und überwachen, Kundenanfragen und Kundenreklamationen bearbeiten und weiterleiten, Aufträge erfassen und prüfen, die Lieferung und den Eingang von Waren überwachen, den Lagerbestand verwalten, Kundenrechnungen versenden, etc. Im Dienstleistungssektor werden immaterielle Wirtschaftsgüter in Transport, Verkehr, Banken-, Hotel- und Gaststättengewerbe angeboten. Aufgaben von betrieblichen Anwendungen im Dienstleistungsgewerbe sind z. B. das Informieren der Kunden jederzeit über Preise, Fahrpläne, Angebote, …, das Buchen von Reisen, Flügen und Eintrittskarten für Veranstaltungen etc. Im Güterverkehrsbereich kann eine Trackingfunktionalität angeboten werden, so dass der Kunde darüber informiert werden kann, wo sich seine Ware gerade befindet. Natürlich werden betriebliche Anwendungssysteme auch im Gesundheitssektor, etwa für das Verwalten von Patientenakten, benötigt. Schließlich werden betriebliche Anwendungen im ECommerce-Sektor für das Abwickeln elektronischer Geschäftsabschlüsse bei Online-Diensten, Auktionsplattformen u. v. m. herangezogen [FSV01].

Betriebliche Anwendungen werden sowohl im Front-Office zur Kommunikation mit den Kunden als auch im Back-Office für Verwaltungs- und computergestützte Zusammenarbeit zwischen Mitarbei-

tern eines Unternehmens eingesetzt. So gibt es z. B. Community-Software-Plattformen, hierzu zählen Weblog-Software und Bookmarking-Tools, welche den Informationsaustausch und das schnelle Wiederfinden und Aufsuchen von Information zwischen Mitarbeitern innerhalb eines Unternehmens erleichtern. Zu den Kommunikationsaufgaben des Front-Office gehören sowohl das Einbeziehen von Kunden, welche aktuelle Informationen über das Internet abrufen und Anfragen stellen können, als auch von Lieferanten, welche geschäftsinterne Informationen über einen geschützten Zugang abrufen und einstellen können.

3.2.2 Charakteristika

Alle oben beschriebenen Anwendungen verwalten große Datenbestände von Geschäftsobjekten. Die Art der Geschäftsobjekte unterscheidet sich je nach Anwendung. So werden etwa im Personalwesen Informationen, wie Anschrift, Gehalt, etc. über alle Mitarbeiter eines Unternehmens gespeichert. Systeme für die Produktionsplanung und -steuerung verwalten u. a. große Mengen an Lieferaufträgen und führen Buch über die Waren, welche sich aktuell im Lager befinden. Betriebliche Anwendungen im Dienstleistungssektor präsentieren dem Kunden Informationen jeglicher Art. Die Hauptaufgabe betrieblicher Anwendungen besteht also in der Datenerfassung, -verarbeitung und -präsentation.

Datenerfassung, -verarbeitung und -präsentation

Für die Datenerfassung, d.h. dem Anlegen neuer Geschäftsobjekte, ist es nützlich, eine für den Benutzer angenehme, ergonomische Benutzeroberfläche bereitzustellen. Dieses ist über einen eingabemaskenorientierten Ansatz möglich. Der Benutzer kann dort in verschiedene Formularfelder die gewünschten Daten zu einem einzelnen Geschäftsobjekt einfügen. Für die Datenbearbeitung**,** d.h. dem Aktualisieren der Attribute der Geschäftsobjekte, ist eine ähnlich aussehende Benutzeroberfläche geeignet. Hier werden dem Benutzer für ein bestimmtes Geschäftsobjekt die jeweiligen im System gespeicherten Daten angezeigt. Diese können dann von dem Benutzer geändert werden. Ein Beispiel für eine Maske zum Aktualisieren eines Geschäftsobjekts ist in Abb. 3.13 gezeigt. Für die Datenpräsentation werden dem Benutzer (Such-)Ergebnisse, Reports u. ä. in Form von Listen angeboten. Ein Beispiel für eine Liste von Verzeichnissen ist in Abb. 3.14 gegeben.

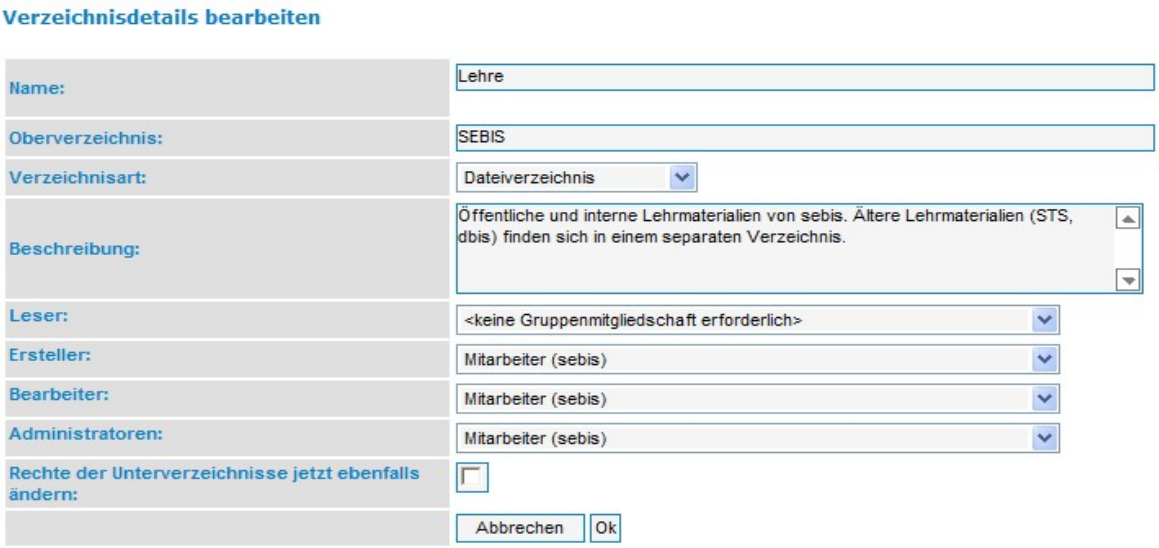


Abb. 3.13: Eingabemaske zum Editieren der Eigenschaften eines Verzeichnisses, Quelle: http://wwwmattes.in.tum.de

Für die Datenerfassung, -verarbeitung und -präsentation eignet sich ein Anfrage-Antwort-orientiertes Kommunikationsprotokoll. Der Client sendet eine Anfrage an die betriebliche Anwendung und wartet eine Antwortnachricht ab, im maskenorientierten Ansatz kann dies z. B. eine HTML-Seite sein.

Unterverzeichnisse | Neu anlegen | Alle löschen |

Name	Größe in MB	Dokumente	Verzeichnisse	Bemerkung
BIRU	293,8454	92	12	Vorlesung: Betriebliche Informationssysteme und ihre Rolle in Unternehmen
DBIS	194,0078	1073	313	Praktikum: "Datenbanken und Informationssysteme" (aktuelles Semester)
Doktorandenseminar	0,0014	1	0	Seminar: Doktorandenseminar sebis
GFSU	36,7279	88	18	Gründung und Führung kleiner softwareorientierter Unternehmen
Hauptseminar	12,9734	45	11	
JASS 2005	29,5279	47	5	
MMCM	73,7463	3011	223	Vorlesung: Multimedia-Datenbanken und Content-Management
Oberseminar	0,1171	10	0	Oberseminar sebis
Planung	4,042	33	7	Planung der Lehre für sebis und die Wirtschaftsinformatik
Proseminar	85,9891	125	7	Proseminar
PSEBIS	21,2407	86	11	Hauptstudiumspraktikum: Software Engineering betrieblicher Informationssysteme
SEBA_Bachelor	97,0804	92	9	Software Engineering für betriebliche Anwendungen Bachelorkurs
SEBA_Master	21,2797	28	2	
SITM	167,2441	534	11	Strategisches IT Management
SoftArch	50,9621	66	6	Vorlesung: Software Architectures
Winfo3	99,1991	313	52	Vorlesung: Einführung in die Wirtschaftsinformatik 3
<lokal>	0,1028	13		
Summe	**1188,087**	**5657**	**703**	

Abb. 3.14: Liste von Verzeichnissen, Quelle: http://wwwmatthes.in.tum.de

Basisfunktionalitäten und Diensttypen

Eine Benutzeroberfläche bietet also verschiedene Funktionalitäten an, welche auf den in der Anwendung gespeicherten Geschäftsobjekten ausgeführt werden können. Obwohl die Funktionalitäten anwendungsabhängig sind, so lassen sich doch allgemeine Funktionalitäten festmachen, welche unabhängig von den verwalteten Geschäftsobjekten und dem Anwendungsgebiet der betrieblichen Anwendung bestehen.

Diese Basisfunktionen sind das Anlegen, Ansehen, Aktualisieren und Löschen von Geschäftsobjekten sowie deren Suche. Außerdem sind Geschäftsobjekte untereinander vernetzt. Eine gute Benutzeroberfläche bietet einfache Navigationsmöglichkeiten zu miteinander verbundenen Geschäftsobjekten an. Bspw. besteht eine Warenbestellung aus mehreren Artikeln. Es muss also möglich sein, sich zu einer Bestellung die einzelnen Artikel anzeigen zu lassen und auch von einem konkreten Artikel auf die Bestellung zu schließen, zu der er gehört. Typische Elemente der Benutzeroberfläche über die Funktionalitäten angeboten werden, sind Buttons, Links und Eingabe einer URL, welche eindeutig eine Funktionalität, das heißt einen Dienst, identifiziert. Dargestellt werden die Geschäftsobjekte entweder einzeln für eine Detailansicht, für das Anlegen und Aktualisieren eines Geschäftsobjekts, oder in Listenform für Auflistungen und Suchergebnisse. Die Anzeige der Listen und der Detailinformation

kann für verschiedene Benutzer so konfiguriert werden, dass jeder nur die Attribute eines Geschäftsobjekts sehen kann bzw. von der Existenz eines Geschäftsobjekts erfährt, wenn er dazu berechtigt ist.

Eine Basisfunktionalität lässt sich auf eine der beschriebenen Methoden der Geschäftslogikschicht abbilden. Für das Anlegen eines Assets steht die Methode *AssetSchema.createAsset( )* und für das Löschen die Methode *AssetSchema.remove( )* zur Verfügung. Assoziationen zwischen Assets werden über *Asset.createAssociation( )* und *Asset.removeAssociation( )* angelegt und wieder entfernt. Das Lesen der Assets und ihrer Properties erfolgt über *AssetSchema.getAsset( )* und die entsprechenden *Asset.getProperty( )*-Methoden. Miteinander über Assoziationen vernetzte Assets werden über *Asset.getAssociatedAssets( )* gelesen. Das Aktualisieren von Geschäftsobjekteigenschaften erfolgt über die jeweilige *Asset.setProperty( )*-Methode. Suchfunktionen werden über *AssetSchema.queryAssets( )* ausgeführt. Die Anzeige von Suchergebnislisten kann während der Erstellung der Antwortnachricht generiert werden.

Jeder Dienst, den eine betriebliche Anwendung zur Verfügung stellt, kann nun die Ausführung einer oder mehrerer Basisfunktionalitäten beinhalten. Von daher gibt es in der Regel für jeden Geschäftsobjekttyp, den eine Anwendung verwaltet, vier Dienste: Das *Anlegen* (engl.: Create, C), das *Lesen* (engl.: Read, R), *Aktualisieren* (engl.: Update, U) und *Löschen* (engl.: Delete, D). Darüber hinaus kann es zu jedem Geschäftsobjekttyp spezifische Funktionalitäten geben, welche durch einen Dienst zur Verfügung gestellt werden. Bei einer Dokumentenverwaltung könnte dies z.B. das Sperren eines Dokuments für exklusiven Zugriff, das Ablegen mehrerer Versionen zu einem Dokument oder das Verschlagworten eines Dokuments sein. Diese anwendungsspezifischen Dienste lassen sich wieder auf eine Basisfunktionalität oder eine Kombination aus mehreren Basisfunktionalitäten abbilden. Weiterhin gibt es für jeden Geschäftsobjekttyp ein oder mehrere *Such*-Dienste (engl.: Query, Q) wie oben beschrieben. Die fünf *Diensttypen* und ihre Zuordnung zu den Basisfunktionalitäten sind in unten stehender Tabelle (siehe Tab. 3.1) noch einmal aufgeführt.

Tab. 3.1: Zuordnung von Basisfunktionalitäten zu Diensttypen einer betrieblichen Anwendung

Basisfunktionalität	**Diensttyp**
`AssetSchema.createAsset(): Asset` `Asset.createAssociation(r: Role, a: Asset): Association`	Create (C)

`AssetSchema.remove(id: String): boolean`
`Asset.removeAssociation(r: Role, a: Asset)`
`Asset.removeAssociations(r: Role)`
	Delete (D)

`AssetSchema.getAsset(id: String): Asset`
`AssetSchema.getAssociatedAssets(r: Role) : Iterator<Asset>`
`Asset.getProperty(p: Property): Object`
	Read (R)

`Asset.setProperty(p: Property, value: Object)`	Update (U)
`AssetSchema.getAssets() : Iterator<Asset>` `AssetSchema.queryAssets(q: Query) : Iterator<Asset>`	Query (Q)

Einfache Algorithmen

In der Regel sind die Algorithmen, welche die Daten bearbeiten und darstellen, einfach. Bei der Neueingabe von Daten, z.B. Personaldaten, ist beispielsweise häufig lediglich eine Integritätsprüfung der eingegebenen Daten durchzuführen. So können bspw. das Geburtsdatum und die Postleitzahl auf Plausibilität hin überprüft werden. Beim Bearbeiten und Anzeigen bestehender Daten müssen, wenn überhaupt, nur simple Formeln berechnet werden, wie etwa das Aufaddieren von Gehältern oder das Ausrechnen des Alters aus dem Geburtsdatum. Im Gegensatz zu anderen Bereichen, in denen

Software unterstützend eingesetzt wird, wie etwa bei Simulationssoftware oder beim Rendering von Bilddateien, ist die Komplexität der zu programmierenden Logik eher gering.

Geschäftsprozesse

Software, welche in Betrieben zur Arbeitsunterstützung eingesetzt wird, unterstützt und optimiert immer einen bestimmten Geschäftsprozess / Arbeitsablauf (engl.: workflow). Eine zu erledigende Aufgabe wird in mehrere Teilschritte untergliedert, welche den gesamten Arbeitsablauf abbilden. Diese Teilschritte sind meist sehr einfach. Solche Teilschritte können z. B. sein, das Ein- oder Ausgeben von Daten über eine Benutzeroberfläche, die Überprüfung eingegebener Daten auf ihre Richtigkeit und Integrität, das Schreiben von Daten in eine Datenbank oder das Auslesen von Daten aus einer Datenbank unter Verwendung einer Anfragesprache.

Ein Geschäftsprozess besteht also aus mehreren Interaktionsschritten. Ein Interaktionsschritt kann im Modell auf die Ausführung eines Dienstes abgebildet werden.

3.3 Zusammenfassung

In diesem Kapitel wurde ein Modell einer datenzentrierten und dienstbasierten Architektur, welches sich auf ein breites Spektrum betrieblicher Anwendungen anwenden lässt, vorgestellt. Hierbei wird davon ausgegangen, dass im Falle der Vernetzung, also der verteilten Diensterbringung, jede Anwendung ihre eigenen Berechtigungen überprüft und nur über die vorgesehenen Schnittstellen mit anderen Anwendungen kommuniziert.

Für jedes Subjekt besteht eine Abbildung auf ein Geschäftsobjekt innerhalb der Anwendung. Diese speziellen Geschäftsobjekte sind im Weiteren Assets vom Typ *User*. Mit einem User kann hierbei eine natürliche Person verbunden sein oder eine andere Anwendung, je nachdem, wer das Clientprogramm, dies kann jetzt auch eine betriebliche Anwendung sein, ausführt. Die betriebliche Anwendung stellt einem Client verschiedene Dienste zur Verfügung. Die meisten dieser Dienste unterliegen einer Zugriffsbeschränkung, so dass es notwendig ist, dass sich der Benutzer vor Nutzung eines Dienstes bei der Anwendung authentifiziert. Hierfür steht ein spezieller Authentifizierungsdienst bereit, der dem Client eine Sitzung mit einer eindeutigen ID zuweist. Diese Sitzungs-ID kann für nachfolgende Anfragen verwendet werden, um so mehrere zusammengehörige Dienste, d. h. einen ganzen Geschäftsprozess, ausführen zu können.

In dem hier gewählten Verteilungsmodell ist immer diejenige Anwendung bzw. derjenige Client, der die entsprechende Anwendung direkt aufgerufen hat, der User, der für die Berechtigungsüberprüfungen herangezogen wird. Die aufgerufene Anwendung verfügt über einen eigenen Transaktionskontext und ist für ihre eigene Berechtigungsüberprüfung zuständig. Im Fremdsystem ist also die aufrufende Anwendung der User und damit das Subjekt für alle zu prüfenden Berechtigungen.

Falls innerhalb eines verteilten Systems die User global bekannt sind, so kann auch ein Berechtigungskontext zwischen den Anwendungen weitergereicht werden, so dass eine Anwendung im Namen eines anderen Users agiert. Das hier aufgestellte Modell bildet keine Einschränkung bezüglich des Subjekts, für das eine Berechtigungsüberprüfung innerhalb einer Anwendung stattfinden soll.

Die in Kapitel 2.4.1 vorgestellte konzeptuelle Sicherheitsarchitektur kann wie folgt in dem hier beschriebenen Modell abgebildet werden. Der Initiator der ISO/IEC-Norm entspricht dem Subjekt und damit dem User der betrieblichen Anwendung. Ein Target ist ein Asset oder eine Property eines Assets. Pro Dienstausführung, also pro Anfrage, kann auf mehrere Targets zugegriffen werden. Die ADF kann als eigenständige Komponente realisiert werden. Diese wird dann Regeln enthalten, die für jede Dienstanfrage ausgewertet werden, um zu bestimmen, ob der Zugriff auf die gewünschten Targets erlaubt ist. Die ADF-Komponente muss also auf die Geschäftsobjekte der Anwendung zugreifen können. Die Überprüfung, also die AEF, wird im Quellcode der Dialogsteuerung und der Geschäftslogik, genauer in den ServiceHandler-Klassen und den Response-Klassen, die ja auf die Targets zugreifen, stattfinden. Die ADF-Komponente wird genauer in Kapitel 5 erläutert, die AEF in Kapitel 6.

4 Anforderungsanalyse und Vorgehensmodell

Der Leistungsumfang von Sicherheitslösungen ist immer auf die Abwendung konkreter Schadensszenarien ausgerichtet. Im Falle der Zugriffskontrolle lassen sich alle Schadensszenarien finden, wenn die Anzahl der Subjekte und Objekte endlich ist. In dieser Arbeit werden Modelle und Techniken für eine effiziente und lückenlose Zugriffskontrolle in betrieblichen Anwendungen entwickelt. In Abschnitt 4.1 befassen wir uns mit den Schutzzielen und möglichen Bedrohungen, die durch die hier zu entwickelnden Modelle und Techniken abgedeckt werden können und sollen, und erläutern das hier zugrunde gelegte Bedrohungsmodell. Das Bedrohungsmodell ist ein Ergebnis der Analysephase der Sicherheitsanforderungsanalyse. Neben dem Bedrohungsmodell, welches die Kapazitäten eines möglichen Angreifers darstellt, ist es auch notwendig festzustellen, wie hoch die Ausdrucksmächtigkeit der zu realisierenden Berechtigungen für betriebliche Anwendungen sein muss. In Abschnitt 4.2 werden die Ergebnisse der Analyse der Zugriffskontrollanforderungen bestehender betrieblicher Anwendungen aufgezeigt, um daraus allgemeine Zugriffskontrollanforderungen abzuleiten. Abschnitt 4.3 beschreibt das Vorgehensmodell, welches hier eingesetzt wird, um sowohl die Spezifikation der Berechtigungen, als auch die Durchsetzung der Zugriffskontrolle innerhalb der Anwendung durchzuführen. Das Vorgehensmodell realisiert die in 4.2 aufgestellten Anforderungen und deckt die Design- und Implementierungsphase des Softwareengineering-Prozesses ab. In dieser Arbeit betrachten wir Java-basierte Anwendungen. Für Java gibt es einige Bibliotheken und Frameworks, welche Mechanismen für die Zugriffskontrolle bieten. In Abschnitt 4.4 werden diese beschrieben und auf ihre Eignung für die Verwendung zur Realisierung einer effizienten und lückenlosen Zugriffskontrolle nach den hier entwickelten Anforderungen untersucht. Das Kapitel schließt mit einer Diskussion alternativer Ansätze und Vorgehensmodelle.

4.1 Schutzziele und Bedrohungsmodell

Die Schutzziele der betrieblichen Anwendung, die hier verfolgt werden sollen, sind die *Vertraulichkeit* und die *Integrität* der verwalteten Geschäftsobjektdaten. Es wird also davon ausgegangen, dass die schützenswerten Informationen in Form von Geschäftsobjekten vorliegen und persistent in einer Datenbank gespeichert werden. Volatile Daten, wie sie etwa während einer Sitzung anfallen, dies können Authentifizierungsdaten und Benutzerdaten der aktuellen Benutzersitzung sein, werden hier nicht weiter betrachtet. Wie wir schon in Kapitel 2.1.1 gesehen haben, kann die Zugriffskontrolle einen Beitrag leisten, um diese Schutzziele zu erreichen. Natürlich handelt es sich bei der Zugriffskontrolle nicht um die einzige mögliche Maßnahme, so dass für eine umfassende Sicherheit noch andere, hier nicht betrachtete Maßnahmen, wie etwa die Authentifizierung und die Verschlüsselung von Daten zur Unterstützung der Vertraulichkeit oder der Einsatz von digitalen Signaturen zur Feststellung der Integrität, eingesetzt werden müssen.

Bedrohungen auf ein IT-System lassen sich auf drei Ebenen klassifizieren: der Anwendungsebene, der Protokollebene und der Ressourcenebene [EA06]. Bedrohungen auf Anwendungsebene nutzen Implementierungsfehler und konzeptuelle Schwachstellen aus. Eine fehlende oder unzureichende

Zugriffskontrollüberprüfung gehört zu den Schwachstellen auf dieser Ebene. Angriffe auf Protokollebene beziehen sich auf die genutzten Kommunikationsprotokolle wie etwa TCP. Sie nutzen Schwachstellen der Protokollimplementierung aus, um fehlerhafte Pakete in die Kommunikation einzuschleusen. Hierdurch wird bspw. der SYN-Flood Angriff realisiert, der zu einer Dienstverweigerung seitens des Opfers führt. Bei Angriffen auf der Ressourcenebene wird ebenfalls auf ein Denial-of-Service abgezielt, indem z. B. die Rechen- und / oder Speicherkapazitäten des Opfers aufgebraucht werden oder durch eine „fork bomb" sehr viele neue Prozesse auf dem Zielrechner generiert werden, so dass der Dienstgeber nicht mehr seine eigentlichen Aufgaben erfüllen kann. Bedrohungen auf den unteren beiden Ebenen werden in dieser Arbeit nicht weiter betrachtet. Von daher springen wir jetzt wieder auf die Anwendungsebene.

Clients, die im Namen eines (gutartigen) Benutzers oder eines bösartigen Angreifers laufen, sollen nur über vordefinierte Schnittstellen auf die Anwendung zugreifen können, um die dort implementierten Dienste anzustoßen. In unserem Bedrohungsmodell gehen wir nicht davon aus, dass ein Angreifer direkt auf die Datenbank oder auf Objekte der Geschäftslogik zugreifen kann. Die Integrität der Daten kann daher durch die Implementierung der Dienste gewährleistet werden, d.h. jede Ausführung eines Dienstes kann das System von einem integren Zustand in einen neuen integren Zustand überführen und der Benutzer kann ausschließlich über die gegebenen Dienste mit der Anwendung kommunizieren. Die Idee der ausschließlichen Verwendung von Prozeduren, die den integren Zustand einer Anwendung vor und nach jedem Prozeduraufruf gewährleisten, wurde bereits im Zugriffskontrollmodell von Clark und Wilson [CW87] beschrieben. Diese Prozeduren werden dort mit „Transformationsprozeduren" bezeichnet. Eine Transformationsprozedur sorgt dafür, dass alle Benutzereingaben in konsistente Datenelemente (engl.: constrained data items) umgewandelt werden, welche den konsistenten Zustand der Anwendung gewährleisten, oder von der Prozedur nicht angenommen werden. Alle Operationen, die auf den konsistenten Datenelementen ausgeführt werden können, sorgen ebenfalls dafür, dass die Konsistenz aller in der Anwendung gespeicherten und verwendeten Daten erhalten bleibt.

Die Vertraulichkeit der gespeicherten Geschäftsobjekte kann dadurch gewährleistet werden, dass pro Dienstaufruf eine Zugriffskontrollprüfung durchgeführt wird, so dass nur autorisierte Benutzer die für sie freigegebenen Dienste aufrufen können bzw. die Antwort auf eine Anfrage nur solche Information enthält, die der entsprechende Nutzer auch lesen darf.

Versteckte Kanäle (engl.: side channel), die zum Verlust der Vertraulichkeit führen (können), werden in dieser Arbeit nicht behandelt. Versteckte Kanäle sind Kanäle, über die ungewollt Information fließt, die (eventuell) indirekte Aussagen über geschützte Information liefern. Versteckte Kanäle verbergen sich u. a. hinter zu ausführlichen Fehlermeldungen. Gegeben sei ein Szenario, in dem ein Benutzer ein neues Dokument anlegen möchte, er aber die Nachricht erhält, dass ein Dokument dieses Namens bereits existiert. Daraus kann der betroffene Benutzer schließen, dass ein für ihn nicht sichtbares Dokument dieses Namens im System vorhanden ist. Verdeckte Informationsflüsse werden im Bereich der Informationsflusskontrolle näher untersucht [Man01].

Weiterhin hat ein potentieller Angreifer keinen Zugriff auf den Java-Bytecode, so dass er kein Bytecode-Rewriting betreiben kann. Diese Art der Bedrohung wird in der Java Security Architecture [GED03] behandelt.

Szenarien, die sich außerhalb der Kontrolle des IT-Systems bewegen, werden ebenfalls nicht weiter betrachtet. So könnten z. B. Benutzer ihre Authentifikationsdaten freiwillig, versehentlich oder absichtlich an Dritte weitergeben. *Social Engineering* Angriffe zielen darauf ab, Personen geschickt zu manipulieren, die Zugriff auf oder Informationen über IT-Systeme haben, die für einen Angreifer interessant sind, um wichtige Informationen von ihnen zu erhalten. Die Abwehr dieser Art von Angriffen kann nicht auf technischer Ebene gelöst werden, sondern nur über geeignete Schulungsmaßnahmen potentieller Zielpersonen realisiert werden, die das Bewusstsein für solche Art Angriffe erhöhen. Social Engineering Angriffe sind ausführlich in [Mit02] beschrieben.

Die in dieser Arbeit betrachteten Bedrohungen beziehen sich also nur auf Bedrohungen der Anwendungsebene, welche auf Verlust der Vertraulichkeit von persistent gespeicherten Geschäftsobjekten abzielen. Diesen Bedrohungen kann mit Maßnahmen der Zugriffskontrolle beim Zugriff auf einen Dienst und / oder bei Auslieferung der Antwort sowie mit einer geeigneten Berechtigungsverwaltung entgegen gewirkt werden.

Es sei an dieser Stelle noch einmal angemerkt, dass nur dann ein auf allen Ebenen sicheres System entwickelt werden kann, wenn eine umfangreiche Betrachtung aller möglichen Bedrohungen und eine

Realisierung von Sicherheitsmaßnahmen auf allen Ebenen, auch z. B. auf physischer Ebene, stattfindet. Konzentriert man sich bei der Entwicklung eines sicheren Systems nur auf einen Aspekt, kann dies gefährlich sein, da man sich dann leicht in Sicherheit wähnt, wo keine ist [SS75]. Diese Arbeit liefert also nur einen Teilbeitrag zur Gesamtsicherheit eines Systems.

4.2 Analyse bestehender Anwendungen

Um herauszufinden, welche Berechtigungen für betriebliche Anwendungen relevant sind, wurden zunächst bestehende Anwendungen auf ihre Anforderungen bezüglich der Zugriffskontrolle untersucht und die Ergebnisse verallgemeinert. Für die Analyse wurden eine Dokumentenverwaltung als Beispiel für ein typisches Querschnittssystem, eine Bankanwendung zur Konten- und Kundenverwaltung (siehe [Sch05, Hup05]) als Beispiel für ein branchenspezifisches Anwendungssystem und eine Weblog-Community als Beispiel für eine Community-Software (siehe http://www.21publish.de) herangezogen. Für alle drei Arten von Anwendung konnten hierbei ähnliche Zugriffskontrollanforderungen extrahiert werden. Zunächst werden in 4.2.1 Teile der untersuchten Dokumentenverwaltung herausgegriffen, um beispielhaft die Zugriffskontrollanforderungen aufzuzeigen. In 4.2.2 werden dann die allgemeinen Anforderungen vorgestellt.

4.2.1 Beispielanwendung „Dokumentenverwaltung“

Das hier vorgestellte Beispiel ist eine vereinfachte Darstellung des Datenmodells der Dokumentenverwaltung, so wie es im infoAsset Broker (http://www.infoasset.de), Version 2.4, momentan implementiert ist (siehe Abb. 4.1). Es handelt sich hierbei um eine gewachsene Anwendung, die sich gut für eine Studie über Berechtigungen eignet. Sie enthält sowohl benutzerbestimmbare als auch rollenbasierte Berechtigungen. Der hier vorgestellte Ausschnitt der Dokumentenverwaltung dient als durchgängiges Beispiel für diese Arbeit. An ihr werden in späteren Kapiteln die in dieser Arbeit entwickelte Berechtigungsbeschreibungssprache sowie der Algorithmus zur Einbettung der Zugriffsdurchsetzung in den Quellcode demonstriert.

Die Dokumentenverwaltung bildet eine Verzeichnisstruktur ab. Jedes Verzeichnis (*Klasse: Directory*) kann sowohl Dokumente (*Klasse: Document*) als auch weitere Unterverzeichnisse enthalten (*Rollennamen: parent, subdirectories*). Weiterhin können die Benutzer (*Klasse: User*) Mitgliedschaften (*Klasse: Membership*) in verschiedenen Gruppen (*Klasse: Group*) haben.

Dokumente bestehen zum einen aus ihren Zitatinformationen wie etwa Titel (*Attribut: title*) und Version (*Attribut: version*) und zum anderen aus der Datei selbst. Die Datei wird durch ihren Namen (*Attribut: fileName*) und die physische Datei selbst dargestellt (nicht im Datenmodell enthalten). Auf Dokumente können logische Sperren gesetzt werden (*Klasse: AssetLock*). Das Sperren eines Dokuments erlaubt dem Sperrenden (*Rollenname: locking_user*) einen exklusiven Schreibzugriff auf das Dokument.

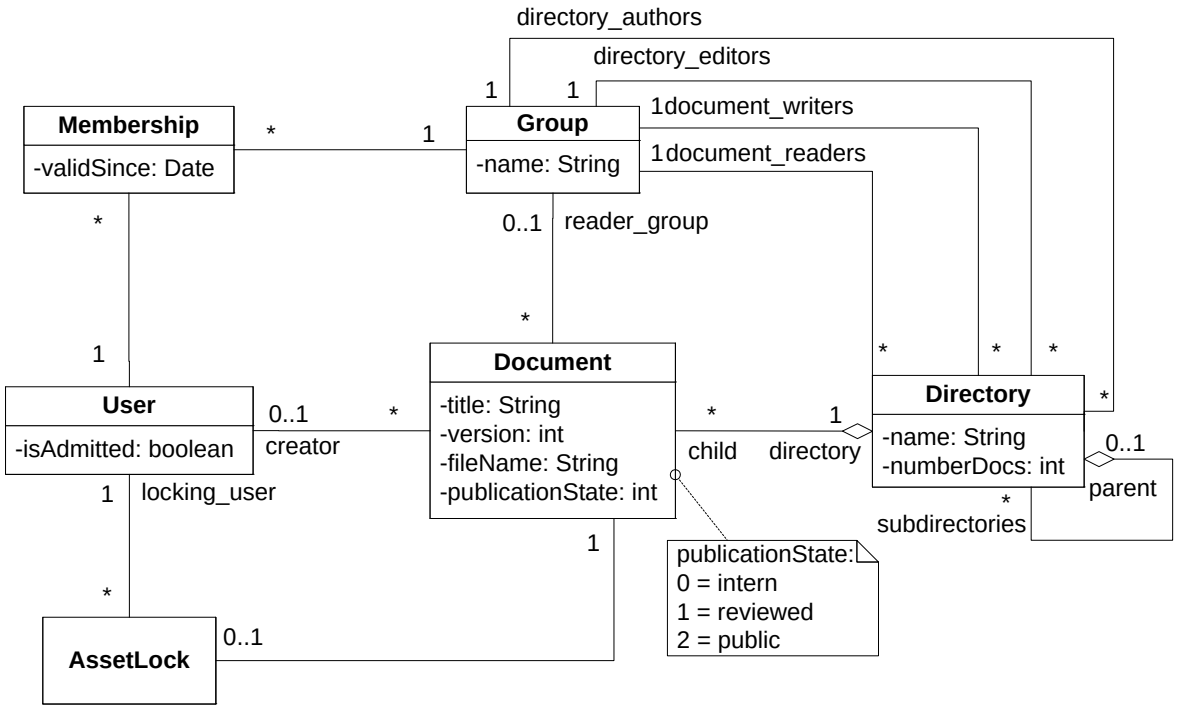


Abb. 4.1: Datenmodell der Dokumentenverwaltung

Einige Dienste, die die Dokumentenverwaltung anbietet, sind in Tabelle 4.1 zusammen mit ihren Diensttypen aufgelistet. Die nicht selbsterklärenden Dienste werden im Folgenden kurz erläutert.

Tab. 4.1: Dienste der Dokumentenverwaltung

Dienst	**Diensttyp**
Neues Dokument einstellen	C (Document)
Datei hochladen	C (Document)
Dokument löschen	D (Document)
Datei löschen (Dokument mit Zitatinformationen bleibt bestehen)	U (Document)
Dateivorschau ansehen	R (Document)
Dokumentzitatinformation ansehen	R (Document)
Dokumentzitatinformation aktualisieren	U (Document)
Datei umbenennen	U (Document)
Datei durch andere Datei ersetzen	U (Document)
Dokumentüberprüfung abschließen (Dokumentstatus auf „reviewed“ setzen)	U (Document)
Dokument veröffentlichen (Dokumentstatus auf „public“ setzen)	U (Document)
Dokumente (nach verschiedenen Suchkriterien) suchen	Q (Document)
Exklusive Bearbeitung eines Dokuments beginnen	C (AssetLock)
Exklusive Bearbeitung eines Dokuments beenden	D (AssetLock)
Neues (Unter-)Verzeichnis anlegen	C (Directory)
Verzeichnis mit allen Dokumenten und Unterverzeichnissen löschen	D (Directory)
Verzeichnisdetails ansehen	R (Directory)
Verzeichnisdetails aktualisieren	U (Directory)
Verzeichnis mit Dateisystem synchronisieren	C, R, U, D (Directory, Document)
Verzeichnisse (nach verschiedenen Suchkriterien) suchen	Q (Directory)

Die Dokumentenverwaltung verwaltet Dokumente mit ihren zugehörigen Verzeichnisstrukturen. Dokumente können in die Dokumentenverwaltung eingestellt werden, indem entweder zunächst die Zitatinformation (hier: Titel und Version) generiert wird (*Dienst: Neues Dokument einstellen*), dann gibt es noch keine physische Datei, oder eine bestehende Datei in das System hochgeladen wird (*Dienst: Datei hochladen*), dann wird ein neues Dokument für die Datei angelegt. Neben den Diensten, welche Standardfunktionen wie Aktualisieren und Löschen durchführen, gibt es Dienste, welche einen Publikationsworkflow abbilden. Dokumente sind nach ihrer Erstellung zunächst im Status „intern". Sie können dann nach ihrer Überprüfung auf „reviewed" gesetzt werden (*Dienst: Dokumentüberprüfung abschließen*) und schließlich publiziert werden (*Dienst: Dokument veröffentlichen*). Weiterhin gibt es Dienste zum Sperren (*Dienst: Exklusive Bearbeitung eines Dokuments beginnen*) und Entsperren (*Dienst: Exklusive Bearbeitung eines Dokuments beenden*) von Dokumenten. Das Setzen einer Sperre geschieht durch das Anlegen eines Sperrobjekts (*Klasse: AssetLock*). Ist ein Dokument gesperrt, so kann diejenige Person, die die Sperre gesetzt hat, alle Dienste, die auf diesem Dokument angeboten werden, exklusiv nutzen. Damit keine Blockade durch nie wieder entsperrte Dokumente zustande kommt, ist es sinnvoll, wenn in der realisierten Berechtigungsverwaltung neben der Person, welche die Sperre gesetzt hat, auch Mitglieder der Administratorengruppe die Berechtigung haben, diese Sperre aufzuheben. Weiterhin besteht die Möglichkeit, die Dokumente und Verzeichnisse direkt in das Dateisystem einzustellen bzw. direkt dort zu löschen und dann mit den in der Dokumentenverwaltung dargestellten Dokumenten und Verzeichnissen zu synchronisieren (*Dienst: Verzeichnis synchronisieren*).

Die Berechtigungen, die für einen Dienst gelten, lassen sich aus den Berechtigungen auf den Basisfunktionalitäten kombinieren. Zur Wiederholung: Basisfunktionalitäten sind das Anlegen (C) und Löschen (D) von Assets und Assoziationen zwischen den einzelnen Assets, das Suchen (Q) nach Assets, sowie das Ansehen (R) und Aktualisieren (U) von Detailinformation zu den einzelnen Assets. Die Basisfunktionalitäten lassen sich nicht exklusiv einem einzigen Diensttyp zuordnen. So greift z. B. der Dienst, welcher ein neues Dokument anlegt auch lesend auf das Verzeichnis zu, in dem das neue Dokument eingestellt werden soll. Die zugriffsberechtigten Subjekte für diese Basisfunktionalitäten sind in der Tabelle 4.2 für das Asset *Document* dargestellt.

Tab. 4.2: Berechtigungen für die Basisfunktionalitäten des Assettyps "Document"

Basisfunktionalität	**Zugriffsberechtigte Subjekte**

C:
`Documents.createAsset( )`
`Document.createAssociation(…)`
	Mitglied der Verzeichniseditorengruppe ODER Mitglied der Verzeichnisautorengruppe ODER Mitglied der Dokumentschreibergruppe ODER Administrator

D:
`Documents.remove(…)`
`Document.removeAssociation(…)`
	((Dokumenteigentümer ODER Mitglied der Verzeichniseditorengruppe ODER Mitglied der Verzeichnisautorengruppe ODER Mitglied der Dokumentschreibergruppe) UND Dokument nicht gesperrt) ODER Administrator

R:
`Documents.getAsset(…)`
`Documents.getAssociatedAssets(…)`
	(Dokumenteigentümer ODER Mitglied der Dokumentschreibergruppe) ODER ((Mitglied der Dokumentlesergruppe ODER Mitglied der Lesergruppe) UND Dokument ist öffentlich) ODER Administrator

R:
`Document.getProperty(„title“)`
`Document.getProperty(„version“)`
`Document.getProperty(„fileName“)`
`Document.getProperty(„publicationState“)`
	(Dokumenteigentümer ODER Mitglied der Dokumentschreibergruppe) ODER ((Mitglied der Dokumentlesergruppe ODER Mitglied der Lesergruppe) UND Dokument ist öffentlich) ODER Administrator

U:
`Document.setProperty(„title“)`
`Document.setProperty(„version“)`
`Document.setProperty(„fileName“)`
	((Dokumenteigentümer ODER Mitglied der Dokumentschreibergruppe) UND Dokument nicht gesperrt) ODER (Dokument gesperrt UND Momentaner Bearbeiter) ODER Administrator

U: `Document.setProperty(„publicationState“)`	(Dokument intern UND Mitglied der Verzeichnisautorengruppe) ODER (Dokument überarbeitet UND Mitglied der Verzeichniseditorgruppe)
Q: `Documents.queryAssets(…)`	Authentifizierter Benutzer

Die Suche nach Dokumenten kann zwar für alle authentifizierten Subjekte möglich sein. Die Anzeige der Ergebnisse hängt aber von den Leseberechtigungen auf den einzelnen Dokumenten im Suchergebnis ab. Die für einen vollständigen Dienst zugriffsberechtigten Subjekte ergeben sich aus der Schnittmenge der für jede einzelne im Dienst vorkommende Basisfunktionalität zugriffsberechtigten Subjekte.

4.2.2 Nutzung von Zugriffsprinzipien für die Definition von Berechtigungen

Für die untersuchten Anwendungen ist zu beobachten, dass es pro Asset immer nur eine eingeschränkte Anzahl an *Zugriffsprinzipien* gibt, die etwas darüber aussagen, wann auf das Asset zugegriffen werden darf. Diese Zugriffsprinzipien beschreiben den Subjekt-Objekt Teil einer Berechtigung anhand von Klassen, Assoziationen und Property-Werten im Datenmodell der Anwendung unabhängig von der Aktion. Die Zugriffsprinzipien für die Dokumentenverwaltung sind in Tabelle 4.3 gegeben. Um eine Berechtigung zu definieren, werden für jede Aktion zugehörige Zugriffsprinzipien mit den logischen Operatoren UND, ODER und NICHT kombiniert, wie oben dargestellt. In der Weblog-Community und in der Bankanwendung lassen sich ebenfalls ähnliche Zugriffsprinzipien definieren, welche sich auf dieselbe Arten klassifizieren lassen.

Tab. 4.3: Zugriffsprinzipien der Dokumentenverwaltung

Zugriffsprinzip	**Art**
Dokumenteigentümer	Eigentümerbezogen
Authentifizierter Benutzer	Gruppenbezogen
Administrator	Gruppenbezogen
Mitglied der Verzeichniseditorengruppe	Assetbezogen (Assoziation)
Mitglied der Verzeichnisautorengruppe	Assetbezogen (Assoziation)
Mitglied der Dokumentschreibergruppe	Assetbezogen (Assoziation)
Mitglied der Dokumentlesergruppe	Assetbezogen (Assoziation)
Mitglied der Lesergruppe für ein Dokument	Assetbezogen (Assoziation)
Momentaner Bearbeiter (hält AssetLock zu dem Dokument)	Assetbezogen (Assoziation)
Dokument gesperrt (es existiert ein AssetLock zu dem Dokument)	Assetbezogen (Assoziation)
Dokument ist überprüft	Assetbezogen (Property)
Dokument ist veröffentlicht	Assetbezogen (Property)

Es lassen sich drei Arten von Zugriffsprinzipien ausmachen. Zunächst gibt es wie bei DAC eigentümerbezogene Berechtigungen. Dieses führt zum *eigentümerbezogenen Zugriffsprinzip*. In der Dokumentenverwaltung z. B. hat der Ersteller eines Dokuments spezielle Rechte auf diesem Dokument. Dieses Zugriffsprinzip wird im Datenmodell durch eine Assoziation zwischen *User* und *Document* festgestellt, die die Erstellerbeziehung ausdrückt. Weiterhin gibt es *gruppen-* bzw. *rollenbezogene Zugriffsprinzipien*. Diese sind unabhängig von dem Asset, auf welches zugegriffen werden soll. Mitglieder verschiedener Gruppen, wie etwa Mitglieder einer Administratorengruppe, haben in der Dokumentenverwaltung z. B. verschiedene Berechtigungen auf Dokumenten oder Verzeichnissen. Um diese Art von Berechtigungen abbilden zu können, müssen Gruppen bzw. Rollen im Datenmodell abgebildet werden (siehe RBAC-Modell in Abb. 2.2). Hier sind Gruppen als Klasse *Group* modelliert, denen Berechtigungen zugeordnet werden. Drittens gibt es *assetbezogene Zugriffsprinzipien*. Diese beschreiben eine von dem betreffenden Asset ausgehende Relation zu anderen Assets, z. B. Gruppen, oder eine das Asset betreffende Eigenschaft. In der untersuchten Dokumentenverwaltung können den Benutzern Berechtigungen pro Verzeichnis oder pro Dokument gewährt werden. Es ist eine relativ starre Zuordnung verankert, was die Rechte pro Verzeichnis und pro Dokument betrifft. Jedem Verzeichnis sind über Assoziationen, die ausschließlich Berechtigungszwecken dienen, genau vier Gruppen zugeordnet (*Rollennamen: directory_authors, directory_editors, document_writers, document_readers*). Diese entsprechen verschiedenen Rollen in Bezug auf ein Verzeichnis: Verzeichnisautoren, Verzeichniseditoren, Dokumentschreiber, Dokumentleser. Jede dieser Gruppen hat unterschiedliche Ausführungsrechte und Leserechte auf dem entsprechenden Verzeichnis und auf die im Verzeichnis enthaltenen Dokumente. Da pro Verzeichnis genau vier Gruppen angegeben werden, welche jeweils bestimmte Rechte auf dem Verzeichnis und den im Verzeichnis enthaltenen Dokumenten haben, vererben sich die Gruppenzuordnungen von Oberverzeichnissen nicht auf die Unterverzeichnisse. Jedem Dokument ist zusätzlich eine Lesergruppe zugeordnet (*Rollenname: reader_group*), die dann die *document_reader*-Gruppe überschreibt. Offensichtlich ist es notwendig, Leserechte pro Dokument und nicht nur pro Verzeichnis festlegen zu können. Weitere assetbezogene Zugriffsprinzipien werden über die Eigenschaften von Assets ausgedrückt. So können zum Beispiel Lesezugriffe auf ein Dokument davon abhängen, in welchem Status sich das Dokument befindet, oder ob der User authentifiziert (*Attribut: isAdmitted*) ist.

4.2.3 Objektorientiertes Datenmodell als Grundlage für die Definition von Zugriffsprinzipien

Es werden hier Anwendungen betrachtet, deren zu persistierende Geschäftsobjekte objektorientiert modelliert werden. Das Datenmodell bildet eine statische Gesamtsicht auf die in der Anwendung verwalteten Geschäftsobjekttypen und deren Beziehungen untereinander. Es enthält implizit Klassen, Assoziationen und Properties, welche auch oder ausschließlich für die Definition von Berechtigungen benötigt werden oder kann einfach um diese erweitert werden. Deshalb scheint es geeignet, Berechtigungen, oder zumindest die Zugriffsprinzipien für die Berechtigungen, direkt auf dem Datenmodell zu definieren. Hieraus ergeben sich einige Vorteile. Zum einen sind dann die Berechtigungen direkt aus dem Datenmodell ablesbar, so dass keine zusätzliche Dokumentation notwendig ist. Dieses ist möglich, da man davon ausgehen kann, dass das objektorientierte Datenmodell intuitiv verständlich ist. Bei der Introspektion, welche z.B. von Programmiersprachen wie Java unterstützt wird, geht man genau von der Annahme aus, dass eine Anwendung so implementiert werden kann, dass sie gleichzeitig als Dokumentation dient. In [Bue07] wird genau dieses Ziel verfolgt. Zum anderen müssten bei einer Änderung des Datenmodells die Berechtigungen zwangsläufig auch angepasst werden, so dass die Berechtigungsstruktur in sich konsistent bleibt. Mit einer deklarativen Spezifikation von Berechtigungen auf dem Datenmodell gehen zusätzlich einige Visualisierungsmöglichkeiten einher, die dem Administrator bei der Rechteverwaltung behilflich sein können, da sie die Übersichtlichkeit definierter Berechtigungen erhöhen. (Diese werden ausführlich in Kapitel 5 betrachtet). Als Nachteil der Spezifikation von Berechtigungen auf dem Datenmodell lässt sich sicherlich sagen, dass die Ausdrucksmächtigkeit bezüglich definierbarer Nebenbedingungen eingeschränkt ist. So kommen im statischen Modell etwa keine Uhrzeiten oder Orte vor, von denen aus auf die Anwendung zugegriffen wird. Aus unserer Anforderungsanalyse geht aber hervor, dass diese Art

der Berechtigungen in den hier untersuchten Typen von betrieblichen Anwendungen nicht notwendig ist.

4.2.4 Weitere Anforderungen an die Zugriffskontrolle betrieblicher Anwendungen

Neben der oben beschriebenen Ausdrucksmächtigkeit der Zugriffskontrolle in Bezug auf die Geschäftsobjekte der Anwendung gibt es noch weitere Anforderungen, die aus der Analyse der drei erwähnten betrieblichen Anwendungen hervorgegangen sind. Abbildung 4.2 fasst die Anforderungen zusammen.

Anforderungen an die Effizienz und Lückenlosigkeit der Berechtigungsüberprüfungen

- Die Durchsetzung der Zugriffskontrolle sollte keine unnötigen Ressourcen in Anspruch nehmen
  - Minimierung redundanter Überprüfungen
  - Minimierung der Anzahl an Überprüfungen
- Vollständigkeit und Lückenlosigkeit der Berechtigungsüberprüfungen
  - Es gibt keine Ausführungspfade der Anwendung, die keiner Zugriffskontrolle unterliegen
  - Die Berechtigungsüberprüfung ist korrekt, d. h. sie überprüft die richtigen Berechtigungen

Anforderungen an die Realisierung der Zugriffskontrolle

- Klare Trennung zwischen ADF und AEF
- Unterstützung einer deklarativen Berechtigungsbeschreibungssprache
- Unterstützung von Query Rewriting für die effiziente Suche

Anforderungen an die Ausdrucksmächtigkeit modellierbarer Berechtigungen

- Definition von sowohl rollenbasierter als auch benutzerbestimmbarer Zugriffskontrolle
- Definition von Berechtigungen auf Geschäftsobjektebene
  - Definition von geschäftsobjektspezifischen Berechtigungen
  - Definition feingranularer Nebenbedingungen auf Attributen der Geschäftsobjekte
- Definition von Berechtigungen auf Attributebene
- Darstellung einer zweistufigen Berechtigungshierarchie: Berechtigungen auf Attributen von Geschäftsobjekten überschreiben Berechtigungen auf den dazugehörigen Geschäftsobjekten
- Definition möglicher Vererbung von Berechtigungen über Geschäftsobjekttypgrenzen hinweg

Anforderungen an die Konfigurierbarkeit der Response

- Konfiguration der Response, falls Benutzeroberfläche verwendet wird
  - Möglichkeit des Ausblendens von Links / Buttons
  - Möglichkeit der Fehleranzeige bei Dienstaufruf über Link / Button / Dienst-URL
- Konfiguration der Response für R- und Q-Dienste
  - Möglichkeit des Ausblendens von zugriffsgeschützten Teilen der Response
  - Möglichkeit der Fehleranzeige für zugriffsgeschützte Teile der Response

Anforderungen an die Autorisierung unterstützter Geschäftsprozesse

- Berücksichtigung von Workflowabhängigkeiten
  - Festsetzen impliziter Rechte: Schreiben impliziert Lesen

Abb. 4.2: Anforderungen an die Zugriffskontrolle betrieblicher Anwendungen

Anforderungen an die Effizienz und Lückenlosigkeit der Berechtigungsüberprüfungen

Berechtigungsüberprüfungen sind mit kostspieligen Datenbankanfragen verbunden. Werden sehr viele Berechtigungsüberprüfungen durchgeführt, so sinkt die Performanz der Anwendung. Dies ist insbesondere dann auffällig, wenn sehr viele Benutzer die Anwendung gleichzeitig nutzen wollen, denn der Datenbankzugriff bildet den Flaschenhals, über den alle Dienstanfragen geleitet werden müssen. Von daher sollte die Anzahl der Berechtigungsüberprüfungen pro Dienstausführung minimiert werden und redundante Überprüfungen, − das sind Überprüfungen, deren Ergebnis bereits bekannt ist −, vermieden werden. Finden nur wenige Berechtigungsüberprüfungen statt, so muss sichergestellt werden, dass tatsächlich immer alle Berechtigungen überprüft werden und dass es kein Ausführungspfad innerhalb der Anwendung gibt, der nicht durch die Zugriffskontrolle abgedeckt wird, d. h. das Vollständigkeitsprinzip muss eingehalten werden. Die durchgeführten Berechtigungsüberprüfungen sollten selbstverständlich auch die richtigen Berechtigungen, also die Berechtigungen die tatsächlich für die Dienstausführung benötigt werden, überprüfen.

Anforderungen an die Realisierung der Zugriffskontrolle

Die ADF und die AEF sollten gemäß der Sicherheitsarchitektur für die Zugriffskontrolle (siehe 2.4.1) voneinander getrennt werden, so dass beide Architekturbausteine unabhängig voneinander entwickelt und gewartet werden können. Um dies zu realisieren, ist es wünschenswert, eine deklarative Berechtigungsbeschreibungssprache zu unterstützen. Beim Durchführen von Suchoperationen auf den Assets der Anwendung sollte es möglich sein, die Suchanfrage automatisch so mit den Leseberechtigungen auf den einzelnen Assets zu verknüpfen, dass nur solche Assets im Suchergebnis enthalten sind, die der jeweilige Benutzer auch lesen darf. Das Umschreiben der Suchanfragen auf der Datenbank fördert die Effizienz der Berechtigungsüberprüfung. Diese Funktionalität ist auch für die Realisierung einer Limited View-Strategie (siehe S. 44) geeignet.

Anforderungen an die Ausdrucksmächtigkeit modellierbarer Berechtigungen

Eine Lösung für die Definition von sowohl rollenbasierter als auch benutzerbestimmbarer Zugriffskontrollelemente, sowie für die Definition von assetabhängigen Berechtigungen wurde bereits mit den Zugriffsprinzipien vorgestellt.

Es ist allerdings nicht nur notwendig, Berechtigungen auf Geschäftsobjekte zu definieren. Für einige Geschäftsobjekte ist es sogar notwendig, den Zugriff auf einzelne Attribute speziell zu beschränken. Diese Berechtigungen können wieder mit Hilfe der Zugriffsprinzipien realisiert werden. Es muss möglich sein, eine zweistufige Berechtigungshierarchie darzustellen, in der die spezifischeren Berechtigungen auf den Attributen die Berechtigung auf dem Geschäftsobjekt überschreiben. Dieses ist natürlich nur für Lese- und Schreibrechte möglich. In der Dokumentenverwaltung gibt es z. B. einen Dienst „Datei umbenennen“. Dieser benötigt nur den Schreibzugriff auf das Dateinamen-Attribut eines Dokuments. Der Dienst „Dokumentzitatinformation bearbeiten“ hingegen benötigt Schreibzugriff auf alle Attribute des Dokuments, die seine Metadaten darstellen.

Weitere Berechtigungshierarchien können sich auch zwischen den Berechtigungen auf den Geschäftsobjekten ergeben. So könnte eine Leseberechtigung auf ein Verzeichnis auf alle Unterverzeichnisse und die in den Verzeichnissen enthaltene Dokumente vererbt werden. Es muss also möglich sein, solche Containerstrukturen darzustellen und die entsprechenden Berechtigungsvererbungen definieren zu können.

Die Anforderungen an die Ausdrucksmächtigkeit modellierbarer Berechtigungen werden durch eine Berechtigungsspezifikationssprache und die ADF realisiert. Die zweistufige Berechtigungshierarchie impliziert, dass Konsistenzregeln zwischen definierten Berechtigungen eingehalten werden. Die Berechtigungen auf Attributebene sollten die Berechtigungen auf Geschäftsobjektebene nur einschränken. Idealerweise sollte es ein Werkzeug geben, welches die Einhaltung dieser Konsistenzregeln überprüft. Auf die Konsistenzanalyse wird in dieser Arbeit im Rahmen des Ausblicks eingegangen.

Anforderungen an die Konfigurierbarkeit der Response

Der Benutzer greift nicht direkt auf die Geschäftsobjekte zu, sondern interagiert über Dienstaufrufe mit der betrieblichen Anwendung. Hieraus ergeben sich einige Anforderungen an die Präsentation der Antwortnachricht, also der Response. Es gibt zwei Möglichkeiten, dem Benutzer die Inhalte auf der Benutzeroberfläche, bzw. die Inhalte der Response, zu präsentieren. Diese werden als Sicherheitsmuster *Limited View* und *Full View With Errors* bezeichnet [SNL06]. Beim Limited View werden alle Informationen ausgeblendet, die der jeweilige Benutzer nicht sehen soll. So können bei Verwendung einer Benutzeroberfläche Links und Buttons, auf die der Benutzer nicht zugreifen darf, ausgeblendet werden. Für alle Antwortnachrichten können die Ergebnislisten für den Diensttyp Q: „Geschäftsobjekte suchen“ diejenigen Zeilen und Spalten ausblenden, die der jeweilige Benutzer nicht sehen darf. Für den Diensttyp R können zugriffsbeschränkte Property-Werte eines Geschäftsobjekts für bestimmte Benutzer ausgeblendet werden. Beim Full View With Errors wird eine vollständige Sicht auf die Antwortnachricht gegeben. Bei Verwendung einer Benutzeroberfläche sind alle Buttons und Links auf der Seite vorhanden. Entweder wird erst beim Klick auf diese Oberflächenelemente eine Fehlermeldung angezeigt oder die entsprechenden Oberflächenelemente sind anders, z. B. ausgegraut, dargestellt, um dem Benutzer zu verdeutlichen, dass ihm die entsprechenden Dienste nicht zur Verfügung stehen. Für die Suchergebnislisten ergibt sich, dass hier zwar alle Zeilen und Spalten angezeigt werden, aber die Elemente, die der Benutzer nicht sehen darf, werden statt mit dem Inhalt mit einer Fehlermeldung versehen. Ein Beispiel für eine Suchergebnisanzeige mit diesem Sicherheitsmuster ist in Abb. 4.3 gegeben. Für den R-Dienst kann statt der Fehlermeldung für die Anzeige eines Leseverbots auf ein Attribut einfach eine leere Anzeige ohne weitere Erklärung für den Benutzer erstellt werden.

Es gibt sowohl für die eine als auch für die andere Variante Für- und Gegenargumente. Für die Limited View-Variante spricht, dass dem Benutzer eine Oberfläche präsentiert werden kann, die genau die Bedienmöglichkeiten bietet, die der Benutzer für die Erfüllung seiner Aufgaben benötigt. Andererseits wird dem Benutzer bei einer Fehlkonfiguration seiner Berechtigungen nicht auffallen, dass es Dienste gibt, für die er keine Berechtigung hat. Dieses würde bei der Full View With Errors-Variante schneller bemerkt werden. Allerdings könnte es bei dieser Variante auch zu einer Frustration bei der Bedienung des Systems kommen, da viele Bedienelemente angezeigt werden, die der Benutzer nicht bedienen kann. Es sollte möglich sein, die Benutzeroberfläche einer betrieblichen Anwendung je nach Kundenwunsch nach einer dieser beiden Varianten zu konfigurieren.

Die Anforderungen an die Benutzeroberfläche bzw. an die Gestalt der Antwortnachricht werden durch ein Zugriffskontrollframework realisiert, welches die AEF enthält.

Anforderungen an die Autorisierung unterstützter Geschäftsprozesse

Zwischen einigen Diensten gibt es Abhängigkeiten, die durch den Workflow impliziert werden. Ein Workflow, welcher ein Geschäftsobjekt bearbeitet, setzt sich häufig aus zwei Diensten zusammen. Der erste Dienst zeigt das gewünschte Geschäftsobjekt mit all seinen Attributwerten an, der zweite Dienst nimmt die vom Benutzer eingegebenen geänderten Werte für die Attribute des besagten Geschäftsobjekts an und speichert diese persistent. Dem U-Dienst, der ein Geschäftsobjekt aktualisiert, geht also ein R-Dienst, der ein Geschäftsobjekt anzeigt, voraus. Ein Benutzer, der das Recht zum Aktualisieren der Attribute hat, sollte also auch das Recht zum Ansehen dieser Attribute haben, da er sonst den Editierungs-Workflow nicht durchführen kann. Idealerweise sollte ein entsprechendes Werkzeug die Spezifikation einer Berechtigungspolicy unterstützen, indem es diese Abhängigkeiten in einer Berechtigungsspezifikation kennt und die Berechtigungen auf Konsistenz prüfen kann. Auf die Konsistenzprüfung wird im Ausblick der Arbeit eingegangen.

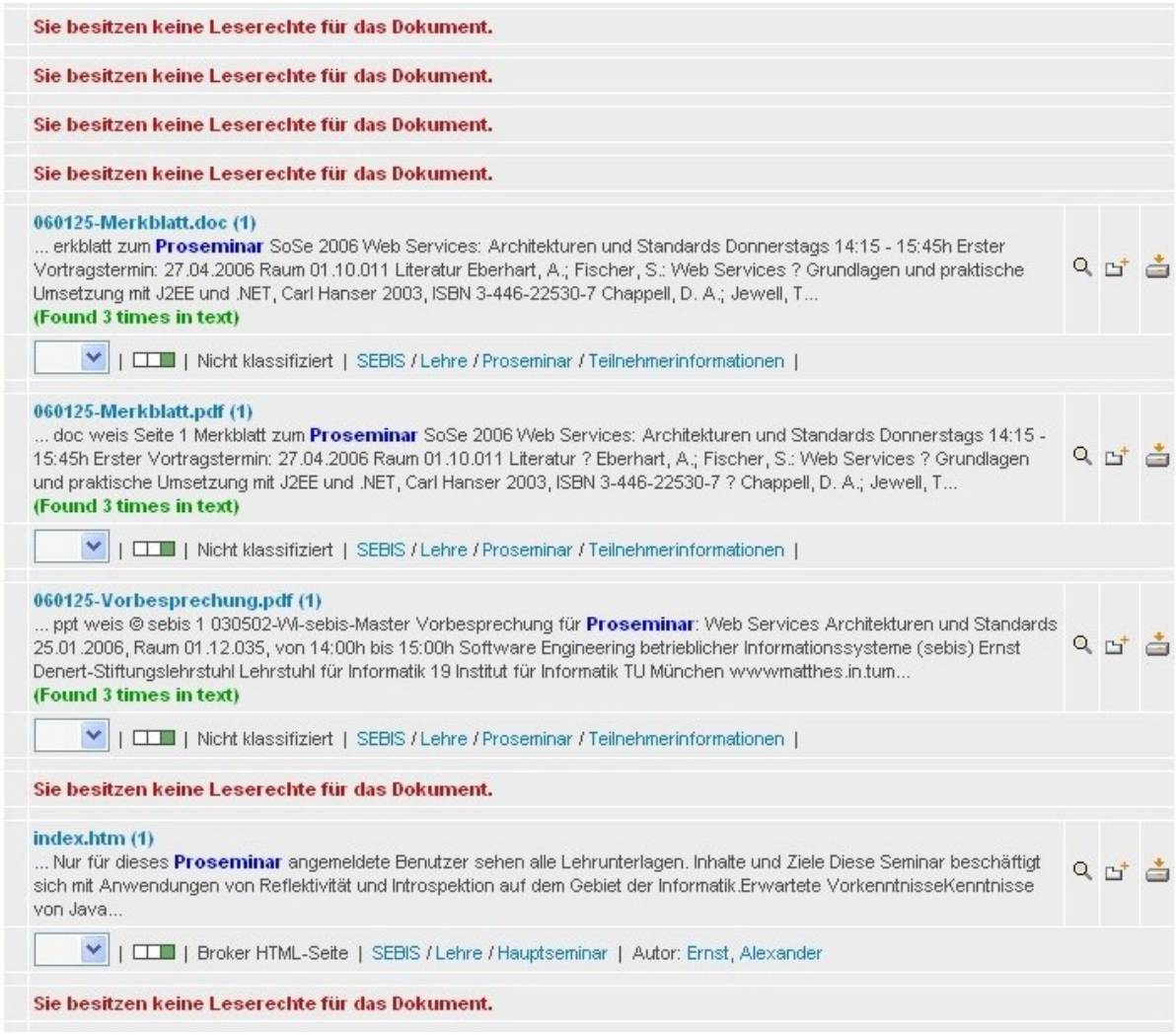


Abb. 4.3: Beispiel für eine Ergebnisliste einer Dokumentensuche nach dem Sicherheitsmuster *Full View With Errors*, Quelle: http://wwwmatthes.in.tum.de

4.3 Vorgehensmodell

Im Folgenden wird ein Vorgehensmodell vorgestellt, nach dem vorgegangen wird, um die oben genannten Anforderungen an die Zugriffskontrolle zu realisieren. Unser Ziel ist es, den Anwendungscode einer betrieblichen Anwendung von dem Zugriffskontrollcode zu entkoppeln. Auf diese Weise kann sich der Entwickler der betrieblichen Anwendung auf den funktionalen Code konzentrieren. Ebenso soll es möglich sein, die Zugriffskontrollpolicy einer bestehenden betrieblichen Anwendung so zu konfigurieren, dass sie an verschiedene Kundenwünsche bezüglich der Zugriffskontrolle angepasst werden kann. Für die Konfigurierbarkeit ist es notwendig, die Berechtigungsspezifikation von der Berechtigungsdurchsetzung zu trennen (siehe 2.4.1).

In Abb. 4.4 wird das allgemeine Vorgehensmodell beschrieben. Auf der rechten Seite ist die betriebliche Anwendung schematisch dargestellt. Clients, hier als Browserfenster dargestellt, greifen ausschließlich über Dienstschnittstellen auf die betriebliche Anwendung zu. Jeder Dienst führt eine Transformationsprozedur, hier als Transaktion bezeichnet, aus und greift auf die in der Anwendung verwalteten Geschäftsobjekte zu, um diese zu lesen und zu schreiben. Die Geschäftsobjekte werden persistent in einer darunter liegenden Datenbank gespeichert.

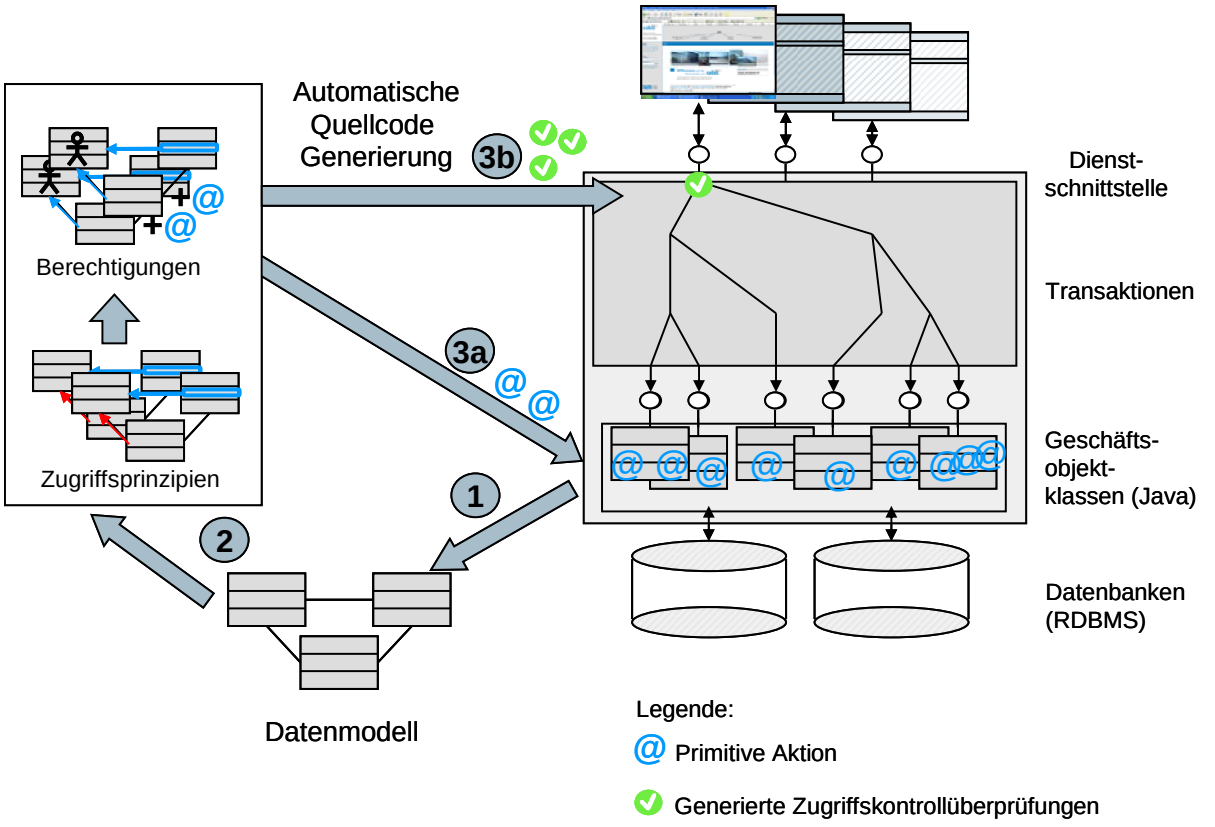


Abb. 4.4: Vorgehensmodell zur Einbettung von Zugriffskontrollüberprüfungen in eine betriebliche Anwendung

In dem Vorgehensmodell muss zunächst ein konsolidiertes Datenmodell bestehen, auf Basis dessen die Zugriffskontrollpolicy definiert werden kann. Das Datenmodell beschreibt die persistent gespeicherten Ressourcen, die Assettypen und ihre Beziehungen untereinander. Es kann, wie in [Bue07] beschrieben, durch Techniken der Introspektion aus einer bestehenden Anwendung ausgelesen werden. Wir gehen davon aus, dass uns das Datenmodell zur Verfügung steht (Schritt 1). Auf Basis des Datenmodells werden nun die Berechtigungen für die Anwendung spezifiziert (Schritt 2).

Das Vorgehen bei der Definition der Berechtigungen der betrieblichen Anwendung ist in Abb. 4.5 schematisch dargestellt. Wie wir gesehen haben, basieren Zugriffsberechtigungen auf Geschäftsobjekte auf einigen wenigen Zugriffsprinzipien (siehe 4.2.2). Zugriffsprinzipien werden nun im Datenmodell auf den Attributen und Relationen der Assets beschrieben. Eine Berechtigung beschreibt, welches Subjekt, also welche Benutzer, eine bestimmte Aktion, diese kann lesend oder schreibend sein, auf einem Asset durchführen darf. Die Relation zwischen dem jeweiligen Asset und dem Asset, welches das Subjekt darstellt, in der Abbildung als Strichmännchen gezeichnet, wird als Zugriffsprinzip modelliert. Die Zugriffsprinzipien werden der Anwendung als Quellcodeklassen zur Verfügung gestellt. Da es für jede betriebliche Anwendung nur eine eingeschränkte Anzahl verschiedener Zugriffsprinzipien gibt, ist es möglich, all diese bereit zu stellen. Die Berechtigungen werden aus konkreten Zugriffsprinzipien kombiniert. Kombinierte Zugriffsprinzipien werden der Anwendung wiederum als Quellcodeklassen zur Verfügung gestellt. Die Berechtigungen können je nach Anforderung der betrieblichen Anwendung konfiguriert und angepasst werden. Berechtigungen werden ausschließlich aus den bereitgestellten Zugriffsprinzipien erzeugt und sind somit unabhängig von der funktionalen Implementierung der Dienste und der Geschäftslogik der betrieblichen Anwendung. Die Berechtigungen bestehen neben einem zusammengesetzten Zugriffsprinzip aus einer Aktion. Die Aktion ist in der Abbildung als @-Zeichen dargestellt. Die Anzahl möglicher Aktionen ist, genau wie die der Zugriffsprinzipien, ebenfalls beschränkt. Die Dienste der betrieblichen Anwendung bedienen die Basisfunktionalitäten Anlegen, Löschen, Lesen und Aktualisieren auf den einzelnen Geschäftsobjekten. Diese lassen sich auf primitive Aktionen der Geschäftsobjekte abbilden (siehe 3.2.2).

Für verschiedene Auslieferungen einer betrieblichen Anwendung können verschiedene Zugriffsprinzipien und verschiedene Zuordnungen von Zugriffsprinzipien zu primitiven Aktionen vorgenom-

men werden, so dass dieselbe betriebliche Anwendung mit unterschiedlichen Berechtigungen ausgestattet werden kann. Kapitel 5 beschäftigt sich im Detail mit der Definition von Berechtigungen, die aus Zugriffsprinzipien und Aktionen aufgebaut sind, und der hier realisierten ADF.

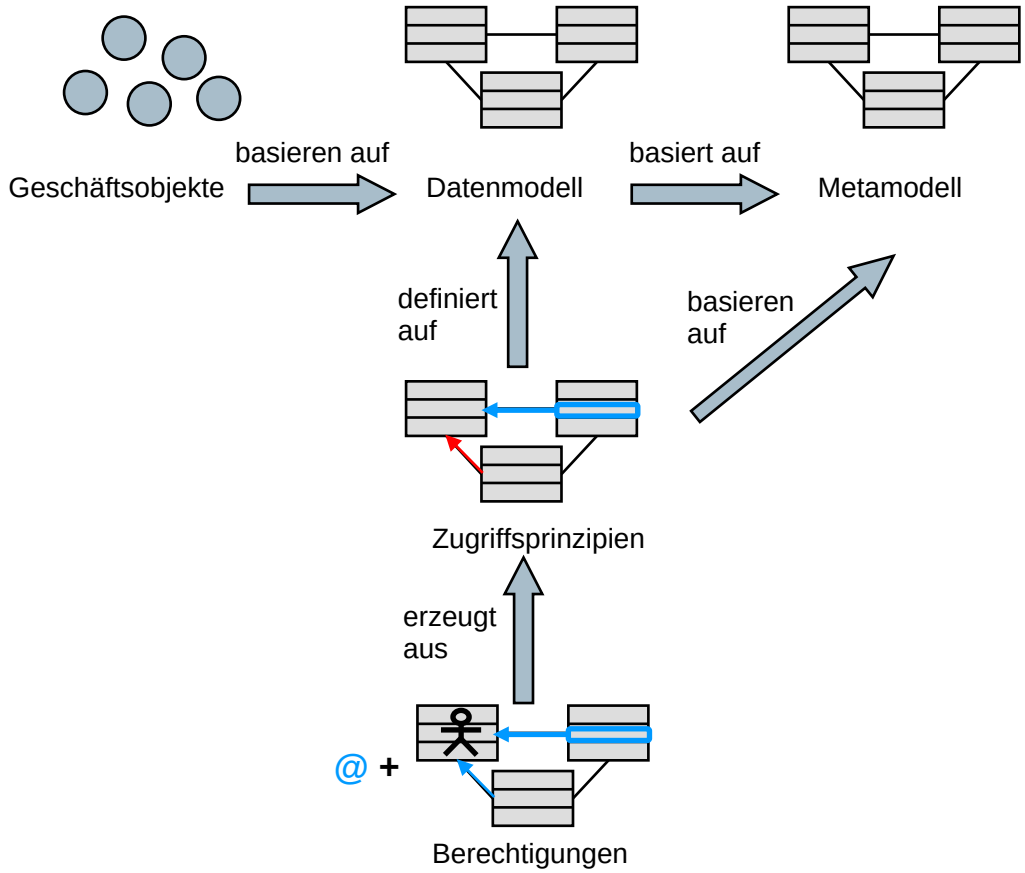


Abb. 4.5: Definition von Berechtigungen

Die Integration der Zugriffsdurchsetzung erfolgt nun in zwei weiteren Teilschritten (siehe Abb. 4.4). Zunächst wird der Quellcode annotiert, indem zu jedem Geschäftsobjekttyp und zu jeder Aktion eine Annotation erstellt wird, die die zutreffenden Zugriffsprinzipien enthält (Schritt 3a). Jetzt stehen die Berechtigungsinformationen im Quellcode der Anwendung zur Verfügung. Und es kann für jeden Dienst eine Zugriffsdurchsetzungsfunktion generiert und an der vorgesehenen Stelle in das Framework der Anwendung eingebettet werden (Schritt 3b). Damit die Einbettung der AEF auf systematische Art und Weise erfolgen kann, ist es notwendig, ein Framework zu erstellen, welches dedizierte Eingriffspunkte vorsieht, in die der Quellcode für die Zugriffskontrolldurchsetzung eingefügt werden kann. Nur wenn diese Eingriffspunkte vorhanden sind, ist es möglich, die Zugriffskontrolle konfigurierbar und vom Anwendungscode unabhängig zu gestalten. Kapitel 6 beschreibt, wie ein solches Framework aussehen kann und wie der Quellcode für die Berechtigungsdurchsetzung aus der Spezifikation der Berechtigungen automatisch generiert werden kann.

Die Vorteile des hier vorgestellten Vorgehensmodells bestehen nun darin, dass die Einbettung der Zugriffskontrollfunktionalität sowohl für die Neukonzeption als auch für eine nachträgliche Konfiguration der Anwendung möglich ist. Durch die Unabhängigkeit von funktionalem Code und Code für die Berechtigungsüberprüfung ist eine unabhängige Änderbarkeit sowohl des funktionalen Codes als auch der Berechtigungspolicy gegeben. Wird die Implementierung eines Dienstes geändert oder die Policy umgestaltet, so kann der Code für die Zugriffsdurchsetzung durch ein automatisches Werkzeug neu generiert werden. Dadurch, dass alle Informationen im Quellcode der Anwendung zur Verfügung stehen, ist keine externe Konfigurationsdatei notwendig, so dass die Konsistenz von Berechtigungspolicy, der spezifizierten Berechtigungen und der Durchsetzung derselben einfach gewährleistet werden

kann. In der Realisierung der Zugriffsdurchsetzung wird angestrebt, pro Dienst einen festen Ankerpunkt im Quellcode der Dienstimplementierung vorzusehen, in den der Quellcode für die Zugriffsdurchsetzung hinein generiert wird. Dieses Vorgehen ist dann möglich, wenn die Dienste so strukturiert sind, dass alle Zugriffskontrollanforderungen zu Dienstbeginn feststellbar sind. Da die einzelnen Dienste, wie wir später noch sehen werden, aber relativ einfach und stereotyp realisierbar sind, ist dieses Vorgehen gerechtfertigt.

4.4 Analyse bestehender Java-Bibliotheken und -Frameworks

Die in dieser Arbeit betrachteten betrieblichen Anwendungen sind in Java implementiert. Von daher wird untersucht, ob die Mechanismen, die Java bietet, verwendet werden können, um die oben beschriebenen Anforderungen an die Zugriffskontrolle sowie das oben beschriebene Vorgehensmodell effektiv zu realisieren. Es gibt sowohl Bibliotheken für die Zugriffskontrolle als auch ganze Frameworks. Der Einsatz eines bestehenden Java-Frameworks würde bedeuten, dass bestehende Anwendungen nicht einfach um Zugriffskontrollfunktionalität erweitert werden können, sondern dass diese in dem jeweiligen Framework realisiert sein muss. Von daher werden die bestehenden Frameworks daraufhin untersucht, ob die dort implementierten Funktionalitäten für eine eigene Realisierung übernommen werden können. Die Zugriffskontrolle besteht aus den beiden getrennten Teilen Berechtigungsspezifikation (ADF) und Berechtigungsdurchsetzung (AEF). Die Betrachtungen beziehen sich sowohl auf die ADF als auch auf die AEF.

4.4.1 Java Security Architecture (JSA)

Die *Java Security Architecture (JSA)* [GED03] ist eine Sammlung von Java-APIs, die es ermöglicht, Rechte mit Code zu assoziieren. Das Ziel ist es, den Rechner vor der Ausführung von bösartigem bzw. nicht-vertrauenswürdigem Code, welcher aus einer unbekannten Quelle stammt, zu schützen. Ausführbarer Code kann auf Basis seiner Herkunft oder auf Basis seiner Signatur als vertrauenswürdig oder nicht-vertrauenswürdig eingestuft werden. Von daher liegt hier ein ganz anderes Bedrohungsszenario vor, als bei der Zugriffskontrolle auf Geschäftsobjekte einer betrieblichen Anwendung.

Die JSA schützt Systemressourcen, wie Drucker, Netzwerk, Dateien, Ports, ... Hierfür gibt es einige vorgefertigte Berechtigungsklassen, z. B. *FilePermission* für den Zugriff auf Dateien und *SocketPermission* für das Öffnen von Kommunikationsverbindungen [Sun05]. Die Berechtigungen werden in einer externen Datei mit dem Namen *java.policy* konfiguriert.

```
grant codeBase "file:${java.home}/lib/ext/*" {
      permission java.security.AllPermission;
};
```

Abb. 4.6: Beispiel eines Berechtigungseintrags in der java.policy-Datei

Wie in Abb. 4.6 zu sehen ist, kann pro Quelle angeben werden, welche Berechtigungen die von dieser Quelle stammenden Java-Klassen haben sollen. Weiterhin ist es möglich, Zugriffserlaubnisse von der Signatur einer Code-Quelle abhängig zu machen. Im obigen Beispiel hat die lokale Bibliothek Vollzugriff auf alle Systemressourcen. Jeder Code ist mit einer Default-Policy verknüpft, welche eine minimale Menge an notwendigen Berechtigungen darstellt. Mobiler Code hat z. B. die Berechtigung, eine Kommunikationsverbindung mit dem Rechner herzustellen, von dem er heruntergeladen wurde.

Der *SecurityManager*, dieser ist eine Java-Klasse, überprüft vor dem Zugriff auf geschützte Ressourcen bzw. auf die geschützten Methoden der Ressourcen automatisch die Berechtigung. Entwickler

können eigene Berechtigungsklassen definieren, allerdings müssen sie dann selbst dafür Sorge tragen, dass diese Berechtigungen auch überprüft werden, d. h., die AEF muss der Entwickler selbst programmieren. Wenn das Bedrohungsmodell keinen bösartigen Code enthält, ist der SecurityManager meistens überflüssig [SNL06]. Aus diesem Grund ist die Java Security Architecture als solche nicht für die in dieser Dissertation betrachtete anwendungsbezogene Zugriffskontrolle geeignet.

4.4.2 Java Authentication and Authorization Services (JAAS)

Die *Java Authentication and Authorization Services (JAAS)* erweitern die Java-Plattform um Authentifizierungs- und Autorisierungsdienste. Zusätzlich zur Code-Herkunft und Signatur kann die Berechtigung von Code auch auf Basis der Subjekte, die diesen Code ausführen, definiert werden. Das Hauptaugenmerk von JAAS liegt allerdings auf der Authentifizierung. Es können verschiedene Module, die *Login Modules*, zur Authentifizierung in das JAAS-Framework eingebunden werden. Ein Login Module ist eine separate Komponente, welche den Loginvorgang durchführt und der eigentlichen Anwendung das authentifizierte Subjekt in Form von Prinzipal-Objekten zurückliefert. Ein Login Module ist ein *Pluggable Authentication Module* [PAM95], eine separate Komponente, die per Anwendungskonfiguration austauschbar ist, ohne die Anwendung neu kompilieren zu müssen.

Genau wie die JSA benutzt JAAS auch eine externe Policy-Datei, um Berechtigungen zu spezifizieren.

```
grant codeBase "http://bar.com", signedBy "bar",
   Principal bar.Principal "duke" {
   permission java.io.FilePermission "/cdrom/duke/-", "read";
}
```

Abb. 4.7: Beispiel eines Berechtigungseintrags für JAAS in der java.policy-Datei, Quelle: [LGK+99]

In Abb. 4.7 ist ein Beispiel für einen Eintrag in eine Policydatei gegeben. Hier muss es für jeden Prinzipal einen Eintrag geben, welcher die gewünschten Berechtigungen enthält. Ein Subjekt, also ein Benutzer, kann mehrere Prinzipale besitzen, z. B. kann ein Benutzer außer seiner Benutzer-ID auch Mitgliedschaften in mehreren Gruppen und Rollen haben. Diese Art der Berechtigungsspezifikation ist nicht für große Systeme geeignet [LGK+99]. Die Aktionen, im obigen Beispiel die Aktion *read*, werden als String eingegeben, so dass keine Prüfung auf Korrektheit oder Sinnhaftigkeit der Einträge zur Compilezeit gegeben ist.

Um die JAAS-Autorisierung nutzen zu können, muss die Authentifizierung über JAAS erfolgen, damit Subjekte und Prinzipale vorhanden sind. Die Klasse *Subject* repräsentiert ein authentifiziertes Subjekt. Der Zugriff auf geschützte Ressourcen erfolgt über die Methode

```
Subject.doAs(final Subject subject,
   final java.security.PrivilegedExceptionAction action),
```

welche den laufenden Code mit einem Subjekt assoziiert und auf Basis dieses Subjekts über die Berechtigung entscheidet, ob der Code, der sich hinter der Aktion *action* verbirgt, ausgeführt werden darf. Die Action-Klasse enthält eine *run( )*-Methode, welche den Zugriff auf eine geschützte Ressource kapselt [Sun01].

Das Ziel dieser Dissertation ist es, Autorisierungsüberprüfungen auf eine systematische Art und Weise in den Quellcode einzubetten. Die Verwendung von JAAS hat hier keine Vorteile, da zum einen JAAS für die Authentifizierung verwendet werden muss, um die *Subject*-Klasse verwenden zu können, und zum anderen die geschützten Aktionen zwar in Klassen gekapselt werden müssen, der Aufruf der Berechtigungsüberprüfung aber immer noch vom Programmierer selbst angestoßen werden muss. Hierdurch ergeben sich keine Vorteile bezüglich der Quellcode-Struktur.

4.4.3 Java 2 Platform, Enterprise Edition (J2EE) / Java Platform, Enterprise Edition 5 (Java EE 5)

Java EE 5 (vorher: J2EE) bietet eine komponentenbasierte Architektur für mehrschichtige, verteilte Anwendungen. Diese Architektur ist in Schichten aufgebaut und realisiert eine Client-Server Architektur. Der Client kann entweder ein *thin client* sein, dann besteht er aus einem Browser und HTML-Seiten, oder ein *fat client*, dann enthält er selbst weitere Anwendungslogik. Auf Serverseite werden mehrere Schichten unterschieden, eine Webschicht, eine Businessschicht und eine Datenbankschicht. Die Webschicht besteht im Wesentlichen aus Java ServerPages oder Servlets, enthält also den Quellcode für die Präsentation und Dialogsteuerung, die Businessschicht verwaltet Enterprise JavaBeans (EJBs), bildet also die Geschäftslogik ab. Es gibt mehrere Arten von EJBs. Hier seien die SessionBean zur Verwaltung des Workflows und das Entity, welches Geschäftsobjekte abbildet, erwähnt.

Java EE 5 bietet Zugriffskontrollfunktionalität sowohl für die Webschicht als auch für die Businessschicht an. Diese kann im J2EE-Rahmenwerk entweder deklarativ spezifiziert oder programmatisch implementiert werden. In der neueren Version Java EE 5 ist eine dritte Möglichkeit dazugekommen, das Spezifizieren von Berechtigungen über Annotationen im Quellcode.

```
<security-constraint>
      <web-resource-collection>
            <web-resource-name>retail</web-resource-name>
            <url-pattern>/acme/retail/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>CLIENT</role-name>
      </auth-constraint>
</security-constraint>
```

Abb. 4.8: Auszug aus einem Deploymentdeskriptor zur Deklaration von Zugriffsrechten auf Webressourcen, Quelle: [BCE+06, S. 940]

```
<method-permission>
      <role-name>employee</role-name>
      <method>
            <ejb-name>EmployeeService</ejb-name>
            <method-name>*</method-name>
      </method>
</method-permission>
```

Abb. 4.9: Auszug aus einem Deploymentdeskriptor zur Deklaration von Zugriffsrechten auf EJBs, Quelle: [BCE+06, S. 904]

Die deklarative Spezifikation wird, ähnlich wie bei JSA und JAAS, über eine externe, statische Datei vorgenommen, den Deploymentdeskriptor. Für die aus (dynamischen) HTML-Seiten bestehende Präsentation kann hier für verschiedene URL-Muster angegeben werden, welche Rollen berechtigt sind, eine bestimmte http-Methode auszuführen. Für EJBs können Ausführungsrechte für Rollen pro EJB-Methode vergeben werden. In Abb. 4.8 ist ein Ausschnitt aus einem Deploymentdeskriptor

gezeigt, welcher einer bestimmten Rolle, hier der Rolle *CLIENT*, Get- und Post-Zugriffe auf alle Webressourcen im Verzeichnis */acme/retail/* erlaubt. Abb. 4.9 spezifiziert, dass die Rolle *employee* alle Methoden der EJB *EmployeeService* ausführen darf.

Die Ausdrucksmächtigkeit dieser statischen Deklaration der Zugriffskontrolle ist stark eingeschränkt, da hier nur Ausführungsrechte für Rollen auf Webressourcen oder EJB-Methoden angegeben werden können. Komplexe, zustandsabhängige Nebenbedingungen, wie etwa der aktuelle Zustand einer EJB oder Elemente der benutzerbestimmbaren Zugriffskontrolle, wie die Zugriffserlaubnis aufgrund der Eigentümerbeziehung zu der EJB, können nicht dargestellt werden.

Für diese Art von Zugriffskontrollüberprüfung muss die *programmatische* Zugriffskontrolle verwendet werden. Hierfür stellt Java EE 5 zwei Methoden bereit, um den aktuellen Prinzipal des Sitzungsbenutzers festzustellen und zu überprüfen, ob dieser eine bestimmte Rolle innehat. Mit Hilfe dieser Methoden kann dann innerhalb des Anwendungscodes die Zugriffskontrollüberprüfung programmiert werden. Die Ausdrucksmächtigkeit der programmatischen Zugriffskontrolle ist jetzt höher, da innerhalb des Anwendungscodes auch Zugriff auf dynamische Zustandsinformation, Variablenbelegungen und die übergebenen Parameter möglich ist.

Eine dritte Art der Spezifikation von Berechtigungen geschieht mit Hilfe von Annotationen im Quellcode. Die Annotationen bieten dieselben Möglichkeiten der Zugriffsspezifikation wie die statische Zugriffskontrolle mit dem Unterschied, dass zugriffsberechtigte Rollen direkt im Quellcode vor der jeweiligen Bean-Methode angegeben werden. Deklarationen im Deploymentdeskriptor überschreiben Deklarationen, die mit Annotationen vorgenommen wurden.

Werden sowohl deklarative bzw. annotationsbasierte als auch programmatische Zugriffskontrollüberprüfungen verwendet, so ergibt sich ein zweistufiges Verfahren. Zunächst wird über die deklarativ definierte oder annotationsbasierte Zugriffspolicy überprüft, ob ein Prinzipal einen Prozess bzw. einen bestimmten Dienst oder eine bestimmte Geschäftslogikmethode ausführen darf. Die Ausdrucksmächtigkeit ist hier relativ gering, da nur eine Rollenzugehörigkeit des Prinzipals überprüft werden kann. Es ist für diese einfache Art von Berechtigung möglich, die Information, also die Mitgliedschaft für jede existierende Rolle, statisch im Voraus zu generieren, so dass sie bei einem Dienstaufruf abgefragt werden kann. Als zweites kann programmatisch überprüft werden, ob ein Prinzipal aufgrund von erst zur Laufzeit bekannter Zustandsinformation den Dienst oder die Methode auch zu Ende ausführen darf oder ob die Ausführung vorher durch eine Ausnahmebehandlung wegen fehlender Zugriffsrechte abgebrochen wird. Hier können komplexe Bedingungen und aktuelle Parameterwerte überprüft werden. Die Zugriffserlaubnis kann hier nicht statisch berechnet und zur Verfügung gestellt werden. Ein Vorteil der zweistufigen Vorgehensweise ist sicherlich die Effizienz, denn die Information über Rollenzugehörigkeit kann bereits in jeder Sitzung zur Verfügung stehen, so dass keine kostspielige Datenbankanfrage zur Berechtigungsüberprüfung eines jeden Dienst- oder Methodenaufrufs notwendig ist. Die Anforderungen an die Benutzeroberfläche, wie etwa das automatische Ein- und Ausblenden von Links, ist so nur eingeschränkt für den deklarativen Teil der Berechtigungsspezifikation realisierbar. Die programmatisch implementierten Berechtigungen werden eventuell erst spät innerhalb der Ausführung der Geschäftslogik überprüft. Komplexe Berechtigungen müssen weiterhin vom Entwickler implementiert werden.

Als Fazit lässt sich festhalten, dass die statische Definition über einen Deploymentdeskriptor, wie sie hier vorgenommen wird, ungeeignet ist, um komplexe Berechtigungen zu spezifizieren. Weiterhin werden die Rollennamen im Deploymentdeskriptor als Strings angegeben, so dass keine statische Überprüfung auf die Richtigkeit dieser Strings stattfinden kann. Die programmatische Zugriffskontrolle wird wieder vom Entwickler selbst geschrieben, so dass hier keine Hilfestellung bei der Überprüfung der Korrektheit der Implementierung gegeben ist.

4.4.4 Acegi Security

Acegi Security [Ace06] ist ein Open-Source Projekt, welches Authentifizierungs- und Autorisierungsmechanismen für Enterprise Software bietet. Acegi Security setzt auf dem Spring Framework [Spr06] auf. Dies ist ein auf J2EE basierendes Framework, welches zum Ziel hat, die Verwendung von J2EE zu vereinfachen. Das Spring Framework bietet Model-View-Controller-Funktionalität und kann mit

mehreren Persistenzmechanismen verwendet werden. Außerdem kann jedes Objekt innerhalb von Spring um Funktionalitäten der aspektorientierten Programmierung [Kic96] erweitert werden.

Bei Verwendung von Acegi Security können, anders als in JAAS, die Autorisierungsmechanismen unabhängig von der gewählten Authentifizierung benutzt werden. J2EE-Lösungen sind nicht portabel. Bei Wechsel des Application Servers müssen die Sicherheitseinstellungen neu konfiguriert werden. Bei Acegi Security ist dies nicht der Fall.

Acegi Security bietet Zugriffskontrollfunktionalität ähnlich wie J2EE auch auf der Präsentationsebene, also der Webschicht, und der Geschäftslogikebene. Zusätzlich können pro Geschäftsobjekt, hier als *Domain Object* bezeichnet, Zugriffsberechtigungen definiert und durchgesetzt werden. Berechtigungen pro Geschäftsobjekt werden als Zugriffskontrollliste (siehe 5.2.1) für dieses Objekt realisiert.

Interessant sind die Autorisierungsmöglichkeiten auf der Geschäftslogikebene. Jede Methode eines Geschäftsobjekts kann sowohl vor als auch nach ihrem Aufruf auf die nötigen Autorisierungen geprüft werden. Technisch gesehen wird hierfür ein Around-Advice aus der aspektorientierten Programmierung verwendet. Eine Überprüfung nach der Ausführung der Methode ist dann sinnvoll, wenn eine Überprüfung vorher nicht stattfinden konnte. Gegeben sei der Fall, dass die Methode eine Liste von Geschäftsobjekten zurückgibt. Dann können aus dieser Liste alle diejenigen Einträge entfernt werden, die der entsprechende Benutzer nicht lesen darf, bevor diese Liste in der Präsentationsschicht bearbeitet und das Ergebnis dem Benutzer dargestellt wird. In einer Konfigurationsdatei kann festgelegt werden, welche Rollenmitgliedschaften für die Ausführung welcher Methode bzw. für den Zugriff auf welches Geschäftsobjekt notwendig sind. Die eigentliche Zugriffsüberprüfung vor dem Methodenaufruf übernimmt ein *AccessDecisionManager*, die Überprüfung nach der Ausführung der Methode ein *AfterInvocationManager*. Diese beiden Klassen haben Zugriff auf das zu überprüfende Objekt, dieses kann ein Methodenaufruf (*MethodInvocation*-Objekt) oder bspw. eine Liste von Geschäftsobjekten sein, so dass auch feingranulare, zustandsabhängige Berechtigungsüberprüfungen implementiert werden können.

Das Besondere an Acegi Security ist, dass Zugriffsberechtigungen pro Domain Object festgelegt werden können. Hierfür werden Zugriffskontrolllisten verwendet. Ein Zugriffskontrolllisteneintrag spezifiziert über einen Integerwert, um welche Berechtigung(en) es sich bei diesem Eintrag handelt. Weiterhin wird der Benutzer bzw. die Rolle angegeben, für die dieser Eintrag gültig ist. Optional kann innerhalb einer hierarchischen Beziehung das Vaterobjekt dieses Domain Objects angegeben werden. Dieses ist nützlich, um im Fall von Vererbung von Berechtigungen, alle für ein Domain Object gültigen Berechtigungen leichter zu finden. Die Zugriffskontrolllisten werden persistent in der Datenbank gespeichert. Durch das Zugriffskontrolllistenkonzept ist es möglich, sehr feingranulare Berechtigungen anzugeben.

Acegi Security hat ein ausgearbeitetes Autorisierungskonzept. Die Verwaltung von einer Zugriffskontrollliste pro Geschäftsobjekt kann unter Umständen sehr datenintensiv sein, wenn eine Anwendung viele verschiedene Geschäftsobjekte beinhaltet. Weiter oben haben wir bereits gesehen, dass Berechtigungen für betriebliche Anwendungen nicht einzeln pro Geschäftsobjekt definiert werden, sondern von einer Relation zwischen dem aufrufendem Subjekt und dem Geschäftsobjekttyp abhängen. Die Verwendung von Zugriffskontrolllisten ist oft nicht ideal, denn die Zugriffskontrolllisten müssen für jedes neue Geschäftsobjekt der Anwendung berechnet und gespeichert werden, so dass viel Speicherplatz benötigt wird. Sind die Zugriffskontrolllisten lang, so ist die Zugriffskontrolle aufwendig durchzuführen. Die Zugriffsinformation lässt sich effizienter erhalten, indem nur einige wenige Zugriffsprinzipien verwaltet werden.

4.5 Vergleich mit anderen Ansätzen

Zum Abschluss des Kapitels werden einige interessante, alternative Vorgehensmodelle sowohl für die Spezifikation als auch für die Durchsetzung der Zugriffskontrolle vorgestellt und mit dem hier gewählten Vorgehensmodell verglichen.

4.5.1 Sicherheitsengineering – SecureUML

Die Idee von *SecureUML* [LBD02, BDL03] besteht darin, Softwaremodellierungssprachen mit Sprachen, in denen Sicherheitsanforderungen spezifiziert werden können, zu vereinen. In diesem Fall wird die Modellierungssprache UML [OMG05] um Konstrukte erweitert, mit denen sich Zugriffskontrollanforderungen ausdrücken lassen. SecureUML verfolgt dabei die Idee der Model Driven Architecture (MDA) [OMG03]. Da es sich hier um die Modellierung und Generierung von Code für Sicherheitsüberprüfungen handelt, wurde dieser Ansatz unter dem Namen „Model Driven Security" publiziert.

In SecureUML wird, genau wie bei der MDA, zunächst ein Modell des zu entwickelnden Systems erstellt. Hierbei kann es sich z. B. um ein UML-Klassendiagramm oder ein UML-Zustandsdiagramm handeln. Daraufhin werden die Anforderungen an die Zugriffskontrolle modelliert. Im vorliegenden Ansatz wird das klassische RBAC, erweitert um Nebenbedingungen für Berechtigungen, herangezogen. Der Entwickler modelliert Benutzer und Rollen, bestimmt, welche Modellelemente schützenswerte Ressourcen darstellen und welche Aktionen zugriffsgeschützt sind. Ein Visio-Template, welches die Modellierungselemente von SecureUML enthält, ist zum freien Download von der Firma FoundStone [FS03] erhältlich.

Das Modell wird direkt in Quellcode umgewandelt. Es wurde sowohl eine automatische Codegenerierung für J2EE als auch für .NET realisiert. Im Fall von J2EE sind die Ressourcen die Enterprise JavaBeans und die Aktionen Methodenaufrufe, sowie Lese- und Schreibaktionen auf Attributen der Beans. Der Codegenerierungsprozess generiert automatisch Deploymentdeskriptoren sowie Code zur Zugriffskontrollüberprüfung zu Beginn einer jeden EJB-Methode. In Abb. 4.10 ist ein Beispiel für generierten Quellcode dargestellt. Der EJBContext *ctxt* ist ein Objekt, welches Zustandsinformationen der Ausführungsumgebung, wie etwa die Rollen, die der momentane Sitzungsbenutzer innehat, enthält.

```
if (! (ctxt.isCallerInRole("Supervisor")||
   ((ctxt.isCallerInRole("User") || ctxt.isCallerInRole("Supervisor"))
   && ctxt.getCallerPrincipal.getName().equals(getOwner()) )
)) throw new AccessControlException("Access denied.");
```

Abb. 4.10: Durch SecureUML generierter Quellcode einer EJB-Methode, Quelle: [BDL03]

Der Ansatz von SecureUML eignet sich für die Integration der Zugriffsdurchsetzung bei der Neuentwicklung einer Anwendung. Für die nachträgliche Integration von Berechtigungsüberprüfungen ist er nicht geeignet, da hier zuerst Quellcode für die Berechtigungsüberprüfung in die Methodenrümpfe der Geschäftsobjekte generiert wird, bevor die Methodenrümpfe mit dem Quellcode der Anwendungslogik gefüllt werden. Weiterhin wird jede geschützte Methode mit Berechtigungscode versehen, so dass innerhalb der Abarbeitung eines Dienstes unter Umständen wiederholt dieselben Überprüfungen (mit bereits bekanntem Ergebnis) durchgeführt werden. Diese Strategie kann nur dann effizient sein, wenn zusätzlich eine Caching-Strategie realisiert wird, die wiederholte Überprüfungen vermeidet. In dem in der vorliegenden Arbeit gewählten Ansatz wird angestrebt genau eine Berechtigungsüberprüfung zu Beginn der Dienstausführung durchzuführen, so dass mehrfache Überprüfungen derselben Berechtigung vermieden werden. Die Ausdrucksmächtigkeit der Berechtigungen ist in beiden Fällen vergleichbar. Im Ansatz von SecureUML werden die Berechtigungen außerhalb des Quellcodes in einem Deploymentdeskriptor definiert. In dem hier vorliegenden Ansatz dient der Quellcode der Anwendung als einzige Dokumentationsquelle. Die Berechtigungen, bzw. die Bestandteile der Berechtigungen liegen in dem in dieser Arbeit entwickelten Vorgehensmodell als Klassen oder Annotationen vor, so dass die Möglichkeit besteht, bestehende Berechtigungen durch introspektive Techniken wieder aus der Implementierung zu gewinnen.

4.5.2 Spezifikation von Berechtigungen – Model Checking

Zhang, Ryan und Guelev haben einen Formalismus entwickelt, um Lücken bzw. versteckte Kanäle in Zugriffskontrollspezifikationen aufzudecken [ZRG05]. Hierbei werden Zugriffskontrollregeln in dem so genannten RW-Formalismus, es werden nur Lese- und Schreibzugriffe modelliert, spezifiziert. Zugriffskontrollregeln können neben den gängigen Berechtigungen auch Delegation von Zugriffsrechten enthalten. In dem implementierten Model Checking System wird getestet, ob es Agenten, das sind die Subjekte, gelingen kann, ein bestimmtes Ziel zu erreichen. Dieses kann ein legitimes Ziel sein, dann wird getestet, ob in dem spezifizierten Zugriffskontrollsystem Agenten ihre Aufgaben erfüllen können. Es kann sich aber auch um ein illegitimes Ziel handeln, dann wird getestet, ob die Zugriffspolicy hintergangen werden kann, und wie die Strategie aussieht, um dieses Ziel zu erreichen. Häufig ist es überraschenderweise möglich durch Kooperation mehrerer Agenten, die Zugriffspolicy außer Kraft zu setzen.

In [ZRG05] ist folgendes Beispiel gegeben, welches hier zur Verdeutlichung kurz beschrieben wird. In einem Angestellteninformationssystem könnten z. B. Manager verschiedene Bonusse an Mitarbeiter, aber nicht an andere Manager, verteilen. Diese Aufgabe ist den Direktoren vorbehalten. Nur der Mitarbeiter selbst und die Manager können den Bonus eines bestimmten Mitarbeiters lesen. In dem System können Manager von ihrer Position als Manager zurücktreten. Die Frage ist nun, ob es einem Manager *m1* möglich ist, den Bonus von Manager *m2* zu setzen. Dieses ist mit folgender Strategie möglich: Wenn m2 zuerst von seinem Managerposten zurücktritt, kann m1 ihm einen Bonus setzen.

In diesem Model Checking-Ansatz, welcher nur die ADF abdeckt, wird die Problemstellung der versteckten Kanäle behandelt. Diese ist verwandt zu der Problemstellung der Informationsflusskontrolle. Das im oben referenzierten Artikel beschriebene Modell beinhaltet lediglich vier Agenten. Von daher stellt sich die Frage, ob das Verfahren auch für große Systeme geeignet ist oder ob aufgrund des im Model Checking bekannten Problems der Zustandsexplosion das Testen großer Systeme nicht durchführbar ist. Obwohl die betrachtete Problemstellung sehr interessant ist, kann sie nur als mögliche Erweiterung der in dieser Arbeit behandelten Problemstellung betrachtet werden, da sie in den Bereich der Informationsflusskontrolle (siehe Kap. 7), welche hier nicht näher betrachtet werden soll, gehört.

4.5.3 Durchsetzung der Zugriffskontrolle – Aspektorientierung

Sicherheitsanforderungen können gemeinhin nur schwierig in objektorientierte Anwendungen eingebettet werden, da sich deren Realisierung nur schlecht in zuständige Klassen oder Komponenten kapseln lässt [KIL+97, VBC01]. Sicherheitsaspekte sind global für die gesamte Anwendung gültig. Während sich die ADF in einer einzelnen Komponente kapseln lässt, ist die Durchsetzung der Zugriffskontrolle eine Angelegenheit, welche sich in objektorientierten Anwendungen durch den gesamten Quellcode der Anwendung ausbreitet. Zwar kann eine Klasse geschrieben werden, welche Methoden für die Berechtigungsüberprüfung bereitstellt. Der Aufruf dieser Methoden wir aber an vielen Stellen im Quellcode stattfinden, wann immer eine sicherheitskritische Anweisung ausgeführt wird. Auf diese Weise wird die Lesbarkeit und Wartbarkeit des rein funktionalen Quellcodes beeinträchtigt, da dieser mit nicht-funktionalem Quellcode vermischt wird. Nicht-funktionale Anforderungen im Allgemeinen und Sicherheitsanforderungen im Speziellen werden als *cross-cutting concerns*, also als Querschnittsfunktion (siehe 2.1.2) bezeichnet, die orthogonal zum funktionalen Code der Anwendung liegen. Von daher bietet es sich an, die Durchsetzung der Zugriffskontrolle durch den Einsatz von aspektorientierter Programmierung zu implementieren.

In [Roe04] wurde untersucht, inwieweit aspektorientierte Ansätze bei der Integration von Berechtigungsprüfungen in bestehende, in Java geschriebene, betriebliche Anwendungen geeignet sind. Hierfür wurde unstrukturierter Geschäftslogik-Code um Aspekte der Zugriffskontrolle erweitert. Als Ergebnis der Arbeit ist festzuhalten, dass durch die Verwendung von Aspektorientierung die Übersichtlichkeit des Quellcodes zwar erhöht werden konnte, mit der Auslagerung von Berechtigungsprüfungen in Aspekte jedoch eine große Anzahl an Kontextinformationen eingesammelt werden musste, deren Bereitstellung die Komplexität der eigentlichen Berechtigungsüberprüfung überschritt. Das Auf-

finden geeigneter JoinPoints war hier eine Herausforderung, da an diesen alle notwendigen Parameter, die für die Berechtigungsüberprüfung benötigt werden, vorliegen müssen. Weiterhin wurde im Rahmen der Untersuchung AspectJ [KLM+01] als Implementierungssprache gewählt. Es zeigte sich, dass AspectJ im praktischen Einsatz sehr viel Speicherplatz verbraucht. Ebenso wird durch AspectJ Bytecode generiert, so dass es schwierig ist, das Ergebnis des Weaving-Prozesses auf Korrektheit zu überprüfen. Debugging des durch den AspectJ-Precompiler erzeugten Codes war leider nicht möglich.

Die Verwendung von Aspekten scheint im Framework Acegi Security (siehe 4.4.4) jedoch gut gelungen. Da die Aspekte hier an vordefinierten Stellen als Filter von Eingabewerten bzw. Ausgabewerten verwendet werden. Jedoch wird hier die Berechtigungsüberprüfung auf Ebene der Geschäftsobjektmethoden realisiert, nicht auf Ebene der Dienste, wie in dieser Arbeit angestrebt.

5 ADF – Spezifikation von Policies für betriebliche Anwendungen

Dieses Kapitel beschäftigt sich mit der Spezifikation von Berechtigungen und der Definition einer für betriebliche Anwendungen geeigneten Berechtigungsbeschreibungssprache, auf deren Basis Zugriffskontrollentscheidungen getroffen werden können. Es beginnt in Abschnitt 5.1 mit der Definition und einer eingehenden Analyse von Berechtigungen und Policies. In Abschnitt 5.2 werden verschiedene Policysprachen vorgestellt und bezüglich ihrer Ausdrucksmächtigkeit und Eignung als Policysprache für betriebliche Anwendungen diskutiert. In Abschnitt 5.3 wird die neu entwickelte Berechtigungsspezifikationssprache PathExpressions vorgestellt, die speziell auf Beschreibung und Auswertung von Berechtigungen ausgelegt ist, die für betriebliche Anwendungen relevant sind.

5.1 Berechtigungen und Policies

Dieses Unterkapitel startet in Abschnitt 5.1.1 mit einer Reihe von Definitionen, die für die Spezifikation einer Berechtigung benötigt werden. In Abschnitt 5.1.2 werden dann Eigenschaften von Berechtigungen besprochen. In Abschnitt 5.1.3 werden Modellierungsmöglichkeiten der drei Hauptelemente einer Berechtigung, Subjekt, Objekt und Aktion, erörtert. Anschließend wird auf den Begriff der Policy eingegangen. Begrifflichkeiten, die im Zusammenhang mit Policies auftreten, werden in Abschnitt 5.1.4 definiert. Abschnitt 5.1.5 beschreibt die Charakteristika von Policies und deren Auswertungsalgorithmen. Das Unterkapitel schließt mit einer Aufstellung von allgemeinen Designprinzipien für Policysprachen.

5.1.1 Definition des Begriffs „Berechtigung“ und seiner Elemente

Zentrales Element der Zugriffskontrolle und der Rechteverwaltung sind die Berechtigungen. Synonyme für den Begriff „Berechtigung“ sind *Recht*, *Privileg* und *Autorisierung*. Eine Berechtigung drückt ein positives (oder negatives) Zugriffsrecht eines Subjekts auf ein Objekt aus. Berechtigungen lassen sich im einfachsten Fall als Tripel {Subjekt, Objekt, Aktion} beschreiben, das auf einen Wahrheitswert abgebildet wird [Nie04, RBK+91]. Der Wahrheitswert sagt etwas darüber aus, ob die Berechtigung gewährt oder verweigert wird.

f: Subjekt x Objekt x Aktion → [wahr, falsch] (**1**)

Subjekt und *Objekt* wurden als Begriffe bereits in 2.1.2 eingeführt. Im Kontext von betrieblichen Anwendungen ist das Subjekt derjenige Benutzer einer betrieblichen Anwendung, welcher die dazugehörige Sitzung initiiert hat. Dieser wird im Folgenden auch als Sitzungsbenutzer (engl.: session

user) bezeichnet. Objekte einer betrieblichen Anwendung sind typischerweise die verwalteten Geschäftsobjekte. Ist eine feinere Granularitätsstufe gewünscht, so können auch einzelne Attribute von Geschäftsobjekten als zu schützende Objekte betrachtet werden. Auf den Objekten werden geschützte Operationen, die *Aktionen*, ausgeführt. Eine Aktion kann hierbei entweder eine primitive oder eine komplexe Operation sein. Primitive Operationen werden hier auch als *Basisaktionen* bezeichnet. Typischerweise gibt es für jedes Objekt, gleich ob es sich um Objekte einer betrieblichen Anwendung, eines Betriebssystems, eines Dateisystems, einer Datenbank oder einer anderen Anwendung handelt, Basisaktionen, die weitgehend typunabhängig sind. Diese Basisaktionen sind das Lesen, Aktualisieren, Anlegen und Löschen von Objekten. Allerdings haben diese Basisaktionen je nach Anwendungskontext eine andere Semantik. Im Kontext von betrieblichen Anwendungen bezeichnen wir diese Aktionen in Anlehnung an die in 3.2.2 definierten Diensttypen im Folgenden mit *Create (C)* für Anlegen, *Read (R)* für Lesen, *Update (U)* für Aktualisieren und *Delete (D)* für Löschen. Weiterhin gibt es Aktionen, die nur auf spezifische Objekttypen angewendet werden können. Diese Operationen auf Geschäftsobjekte einer betrieblichen Anwendung lassen sich auf die vier eben definierten Basisaktionen oder eine Kombination aus diesen Aktionen zurückführen. Bspw. lässt sich auf Geschäftsobjekten vom Typ Dokument die Aktion *Move* ausführen, welche das Dokument vom Ursprungsverzeichnis in ein neues Verzeichnis verschiebt. Diese Aktion kann in zwei Teilaktionen, Löschen des Dokuments im Ursprungsverzeichnis und Anlegen des Dokuments in einem neuen Verzeichnis, zerlegt werden. Auch, wenn technisch gesehen, eventuell nur ein Zeiger umgelegt wird, so wird doch der ursprüngliche Zeiger von seiner Ausgangsposition in eine andere Position umgebogen. Das Resultat ist dasselbe, als hätte man das gesamte Dokument gelöscht und wieder neu angelegt.

Aktionen lassen sich in *seiteneffektfreie* und *seiteneffektbehaftete* Aktionen unterteilen. Seiteneffektfreie Aktionen verändern den Systemzustand nicht. Hierunter sind alle lesenden Aktionen zu verstehen. Seiteneffektbehaftete Aktionen ändern den Systemzustand. Das Aktualisieren, Anlegen und Löschen fallen hierunter.

Um die Ausdrucksmächtigkeit von Berechtigungen zu erhöhen, können neben dem Tripel {Subjekt, Objekt, Aktion} noch weitere Systemelemente zur Bestimmung von Zugriffsrechten genutzt werden. Diese weiteren Elemente können globale Systemzustandswerte wie z.B. eine Uhrzeit, oder objektabhängige Zustände, wie z.B. Attributwerte von Objekten sein. Sie werden auch als *Randbedingungen* oder *Nebenbedingungen* bezeichnet. In der vorliegenden Arbeit werden nur objektabhängige Nebenbedingungen betrachtet. Externe Nebenbedingungen, wie Uhrzeiten und Orte traten in den untersuchten betrieblichen Anwendungen (siehe Kap. 4) nicht auf.

f: Subjekt x Objekt x Aktion x Nebenbedingung → [wahr, falsch] (**2**)

Ein Beispiel für eine Berechtigung mit einer zustandsabhängigen Nebenbedingung wäre die folgende (hier als Klartext formuliert): „Ein Benutzer darf ein Dokument sehen, wenn dieses im Publikationszustand ‚veröffentlicht' ist". Der Publikationszustand eines Dokuments könnte durch ein Attribut ‚publicationState' des Assettyps Dokument modelliert sein. Zustandsabhängige Nebenbedingungen beschreiben also einen Objekt-Zustand.

Der Ort, von dem aus ein Subjekt auf eine Anwendung zugreift, ist keine statische Eigenschaft eines Subjekts, sondern ein volatiles Datum. Es ist aber möglich, zu dem Zeitpunkt, zu dem ein Subjekt sich bei einer Anwendung authentifiziert oder zu dem eine Zugriffsüberprüfung stattfindet, ein Attribut des betreffenden Subjekts mit ortsbezogenen Informationen zu befüllen, so dass auf diese Weise eine externe Nebenbedingung als zustandsabhängige Nebenbedingung modelliert werden kann.

Prinzipiell ist dieselbe Vorgehensweise auch mit der aktuellen Uhrzeit möglich. Hierbei werden sich bei der Berechtigungsüberprüfung aber Ungenauigkeiten ergeben, denn die Uhrzeit ist eine hochdynamische Variable, deren Aktualität schon zu dem Zeitpunkt verloren gegangen ist, zu dem sie als Zugriffszeitpunkt in bspw. einem Attribut eines Subjekts gespeichert worden ist.

Ein weiteres Element von Berechtigungen ist die *Verpflichtung* (engl.: *obligation*). Hier geht es darum, dass das Fällen der Zugriffsentscheidung, ob für ein anfragendes Subjekt die Durchführung einer Aktion auf ein Objekt erlaubt ist oder nicht, andere Vorgänge innerhalb der Anwendung auslöst. Es könnte z. B. eine Anforderung an eine betriebliche Anwendung geben, dass jeder Zugriffsversuch protokolliert wird. Andere Verpflichtungen könnten sein, bei jedem Zugriffsversuch auf ein Objekt, den Eigentümer dieses Objekts zu benachrichtigen. Verpflichtungen spielen eine große Rolle in An-

wendungen, die hohe Privatheitsanforderungen erfüllen müssen. Im Umfeld medizinischer Anwendungen bspw. wird das Vertrauen der Benutzer in eine elektronische Datenverwaltung ihrer Krankenakte erhöht, wenn diese darüber benachrichtigt werden, welcher Arzt wann auf ihre Daten zugegriffen hat. Damit können Personen, deren private Daten in einem System gespeichert sind, überprüfen, ob ihre Daten lediglich zweckgebunden verwendet werden. Auf Verpflichtungen wird im Rahmen des Ausblicks eingegangen.

5.1.2 Charakteristika und Auswertungsalgorithmen von Berechtigungen

Berechtigungen können anhand ihrer Art, ihres Vergabemodus und ihres Weitergabemodus charakterisiert werden [Rud99]. Die Art eines Rechts kann seine Polarität und seine Priorität sein. Die *Polarität* eines Rechtes ist entweder positiv, dann drückt die Berechtigung eine Zugriffserlaubnis aus, oder negativ, dann drückt die Berechtigung eine Verweigerung aus. Die Verwendung von sowohl positiven als auch negativen Rechten innerhalb der Berechtigungspolicy einer Anwendung verringert zwar die Anzahl der gesamten für eine Anwendung zu spezifizierenden Berechtigungen, erhöht aber die Komplexität der Berechtigungsverwaltung umso mehr.

Dieses sei an einem Beispiel erläutert. Gegeben sei eine Berechtigung *b1*, die allen Mitgliedern einer Gruppe *g* einen bestimmten Zugriff, die Ausführung von Aktion *a*, auf ein Objekt *o* gewähren. Weiterhin sei eine zweite, negative Berechtigung *b2* gegeben, die Person *p*, die Mitglied von *g* ist, die Ausführung der Aktion *a* auf dem Objekt *o* explizit verbietet. Jetzt gibt es zwei Berechtigungen, die auf dieselbe Aktion, dasselbe Objekt und dieselbe Person angewendet werden können. Für Person *p* kommen die zwei Berechtigungen zu jeweils unterschiedlichen Berechtigungsergebnissen. Es liegt also ein Konflikt vor.

Ein Konflikt liegt immer dann vor, wenn es zwei oder mehrere Berechtigungen gibt, die auf dieselbe Zugriffsanfrage angewendet werden können, aber zu verschiedenen Ergebnissen bezüglich der Zugriffserlaubnis kommen. Ein Konflikt sollte durch die Definition von *Prioritäten* aufgelöst werden. Ein *starkes Recht* ist ein Recht, welches nicht durch ein Recht mit anderer Polarität überschrieben werden darf, ein *schwaches Recht* hingegen darf überschrieben werden [Rud99, RBK+91].

Für unser kleines Beispiel ist *b1* ein schwaches Recht, welches durch das starke Recht *b2* anderer Polarität überschrieben wird. Jetzt ist der Konflikt aufgelöst und Person *p* ist der Zugriff auf Objekt *o* verwehrt.

Dasselbe Berechtigungsergebnis kann man auch erhalten, indem man nur Berechtigungen einer Polarität definiert. Eine Möglichkeit wäre z. B. für jedes Mitglied, außer für Person *p*, eine positive Berechtigung zu definieren. Nach dem Verbotsprinzip (siehe 2.1.3) würde dies implizit eine Berechtigungsverweigerung für *p* bedeuten. Die Anzahl der benötigten Berechtigungen wäre hier sehr viel höher als bei der Variante mit der negativen Polarität. Allerdings entfällt die Konfliktbehandlung.

Eine weitere Möglichkeit wäre, falls das verwendete Berechtigungsmodell Gruppenhierarchien unterstützt, die Gruppe *g* in zwei Gruppen aufzuspalten, eine Gruppe *g1*, die alle Personen außer *p* enthält und eine Gruppe *g2*, die nur *p* enthält. Die Gruppe *g* wäre jetzt eine Obergruppe, die sich aus *g1* und *g2* zusammensetzt. Die Zugriffsrechte könnten jetzt mit Hilfe der Gruppen *g1* und *g2* definiert werden. Es gibt also immer Möglichkeiten, die Berechtigungen so zu definieren, dass nur Berechtigungen einer Polarität verwendet werden brauchen. Die Verwendung von Berechtigungen verschiedener Polaritäten und die Verwendung von Prioritätsregeln machen die Berechtigungsverwaltung unter Umständen sehr unübersichtlich.

Wenn die Anzahl der zugriffsgeschützten Entitäten sehr groß ist, wird es sehr aufwendig, für alle diese Entitäten Zugriffsrechte zu definieren. Der *Vergabemodus* eines Rechts gibt nun an, wie ein Zugriffsrecht definiert worden ist. Bei einem *expliziten Recht* wurde ein Recht direkt definiert. *Implizite Rechte* hingegen sind Rechte, die aus anderen Rechten abgeleitet werden [Rud99, RBK+91]. In einer betrieblichen Anwendung ist es z. B. häufig der Fall, dass die Daten eines Geschäftsobjekts aktualisiert werden. Der Workflow läuft so ab, dass die ursprünglichen Daten des Geschäftsobjekts gelesen und dem Benutzer angezeigt werden, bevor er diese verändern kann. Ein Schreibrecht für die Daten eines Geschäftsobjekts impliziert also ein Leserecht auf denselben Daten [Nie04].

Ein weiteres Beispiel, in dem häufig implizite Rechtevergabe stattfindet, sind hierarchisch strukturierte Ressourcen, wie etwa Dateien in einem Verzeichnis. Ein explizit definiertes Leserecht auf einem

Verzeichnis könnte hierbei ein Leserecht für alle in diesem Verzeichnis enthaltenen Dateien implizieren.

Für implizite und explizite Rechte gelten dieselben Komplikationen wie für Regeln mit unterschiedlichen Prioritäten. Denn es ist manchmal erwünscht, dass implizit definierte Regeln wieder durch andere explizite Regeln überschrieben werden können. So sollen etwa die Rechte eines Verzeichnisses auf alle seine Unterverzeichnisse vererbt werden, außer für ein spezielles Unterverzeichnis, das anderen Zugriffsrechten gehorchen soll. Diese Möglichkeit des Überschreibens von vererbten Rechten ist z. B. im Betriebssystem Windows vorgesehen [DPS03, Win32]. Implizite Rechte sind also schwache Rechte während explizite Rechte, starke Rechte sind.

Der *Weitergabemodus* eines Rechts gibt an, ob das Subjekt, welches ein Zugriffsrecht besitzt, dieses an andere Subjekte weitergeben darf. Dieses Vorgehen ist auch unter dem Namen *Delegation* bekannt und tritt in benutzerbestimmbaren Zugriffskontrollmodellen (siehe 2.3.1) auf. In der vorliegenden Arbeit wird die Delegation von Rechten nicht weiter behandelt. Sie trat in den untersuchten betrieblichen Anwendungen nicht auf. In betrieblichen Anwendungen ist die Rechteverwaltung zentral und systembestimmt gesteuert und ein Benutzer kann nur in einem geringen Umfang Zugriffsrechte auf die ihm gehörenden Objekte vergeben. Die Möglichkeit der Weitergabe von Berechtigungen ist nur möglich, soweit dies durch das System vorgesehen ist.

5.1.3 Klassifizierung und Modellierung von Berechtigungen

Orthogonal zu den obigen Charakteristika können Berechtigungen nach weiteren Eigenschaften klassifiziert werden. Zum einen lassen sich Berechtigungen danach qualifizieren, welche Elemente – Subjekt, Objekt, Aktion – sie beinhalten. Zum anderen können Berechtigungen, je nachdem welche Aktionen innerhalb eines Systems sie schützen sollen, in Kategorien eingeteilt werden.

Berechtigungen beinhalten immer mindestens ein Element vom Typ Subjekt, Objekt oder Aktion. Diese drei Elemente können die Menge aller möglichen Subjekte, Objekte und Aktionen der Anwendung aber in verschiedenen Detaillierungsgraden beschreiben. Die Menge aller Subjekte, die auf eine Anwendung zugreift, kann auf einer groben Granularitätsstufe in überschneidungsfreie Untermengen aufgeteilt werden. Die Subjekte werden dann nach einer sie identifizierenden Eigenschaft sortiert, z. B. können Subjekte authentifiziert oder nicht authentifiziert sein, ein gültiges Zertifikat besitzen oder kein gültiges Zertifikat besitzen. Berechtigungen dieser Granularitätsstufe sind dann z. B. von folgender Form: „Ein Subjekt s darf die Aktion a auf Objekt o ausführen, wenn s authentifiziert ist.“ oder „Ein Subjekt s darf die Aktion a auf Objekt o ausführen, wenn s ein gültiges Zertifikat besitzt.“. Eine feinere, nicht unbedingt mehr überschneidungsfreie Aufteilung der Subjekte wird in dem rollenbasierten Berechtigungsmodell realisiert. Hier werden Subjekte verschiedenen Rollen zugeordnet. Berechtigungen können jetzt aufgrund der Rollenzugehörigkeit eines Subjekts eine Aktion auf einem Objekt gewähren oder nicht gewähren. Eine Berechtigung dieser Form wäre z. B.: „Ein Subjekt s darf eine Aktion a auf einem Objekt o ausführen, wenn das Subjekt in der Rolle r ist.“. Berechtigungen, bei denen die Subjekte aufgrund ihrer Eigenschaften oder Rollenzugehörigkeit identifiziert werden, sind *identitätsbasierte* Berechtigungen. Im Gegensatz dazu gibt es *identitätsunabhängige* Berechtigungen, die Berechtigungen nur aufgrund von Eigenschaften der Objekte und / oder Aktionen beschreiben.

Die Objekte der Berechtigungen können ebenfalls in verschiedenen Granularitätsstufen modelliert werden. Innerhalb einer betrieblichen Anwendung sind die Objekte, für die Berechtigungen vergeben werden, die Geschäftsobjekte der Anwendung. Eine grobgranulare Einteilung der Geschäftsobjekte würde diese nach ihrem Geschäftsobjekttyp einteilen. Es macht keinen Sinn, Berechtigungen über Geschäftsobjekttypgrenzen hinweg zu deklarieren, da die Berechtigungsanforderungen häufig vom Typ des Geschäftsobjekts und von der Art der Aktionen, die auf diesem Geschäftsobjekttyp ausführbar sind, abhängig oder vollständig objektunabhängig sind. Falls es doch eine Berechtigungsanforderung gibt, die für zwei oder mehrere Geschäftsobjekttypen dieselbe Subjekt- und dieselbe Aktionsmenge beinhaltet, so kann man einfach zwei verschiedene Berechtigungen verwalten, eine für jeden Geschäftsobjekttyp. In einer Dokumentenverwaltung (vgl. Kap. 4.2.1) könnte z. B. der Zugriff auf Gruppen- und Mitgliedschaftsobjekten denselben Zugriffsbeschränkungen unterliegen. In beiden Fällen könnte es nur den Administratoren gestattet sein, Geschäftsobjekte dieses Typs anzulegen und wieder zu löschen. Statt nur eine Berechtigung für das Anlegen und Löschen von Gruppen und

Mitgliedschaften zu verwalten, sollten der Übersichtlichkeit halber vier Berechtigungen verwaltet werden, eine für das Anlegen von Gruppen, eine für das Löschen derselben, eine für das Anlegen von Mitgliedschaften und eine für das Löschen dieser. Im Allgemeinen reicht diese grobe Modellierung der Objekte aber nicht aus, um den Anforderungen an eine Berechtigungsverwaltung für betriebliche Anwendungen zu genügen. Stattdessen ist es gewünscht, den Zugriff auf Geschäftsobjekte weiter einzugrenzen, indem der Zugriff auch von Eigenschaften der Geschäftsobjekte abhängt. Bspw. könnte die Zugriffserlaubnis, z.B. das Lesen auf einer Datei, davon abhängen, wann die Datei in das System gestellt wurde oder ob die Datei sich in einem Bearbeitungsprozess durch einen Moderator befindet. Eine detailliertere Modellierung von Rechten geht also einher mit der Definition von Nebenbedingungen auf Objekten.

Die Modellierung von Objekten kann sich auch auf einzelne Attribute eines Geschäftsobjekts beziehen. Am besten kann dies am Beispiel von Personendaten erklärt werden. Bei personenbezogenen Daten können einige Daten öffentlich sein, wie etwa Name und Vorname, andere Daten nur einem begrenzten Personenkreis zugänglich sein, wie etwa Adresse und Telefonnummer, und wieder andere Daten nur von ausgewählten Personen eingesehen und geändert werden, wie etwa das Gehalt. Der Attributzugriff kann durch Bedingungen präziser definiert werden. Die Definition von Berechtigungen auf Attribute ist in der objektorientierten Modellierung die feingranularste und aufwendigste Art, Berechtigungen zu modellieren. Im praktischen Einsatz ist es sehr aufwendig, für alle Attribute eines Geschäftsobjekts eigene Berechtigungen zu verwalten. Ein Mittelweg wäre hier, die Attribute in disjunkte Klassen einzuteilen und die Berechtigung auf diese Klassen zu vergeben. Ansonsten ist die Anzahl der zu administrierenden Berechtigungen sehr hoch.

Bei der Modellierung der Objekte in einer Berechtigungsverwaltung für eine betriebliche Anwendung gibt es also zwei Möglichkeiten. Bei der ersten Möglichkeit sind Objekte Geschäftsobjekte der Anwendung. Die zweite Möglichkeit besteht darin, Objekte als Attribute von Geschäftsobjekten zu modellieren. Mit Attributen sind hier nur „echte“ Attribute gemeint, also Attribute primitiven Datentyps. Attribute nicht primitiven Datentyps werden als Assoziationen zwischen Geschäftsobjekten dargestellt (vgl. Kap. 3.1.3). Assoziationen können wieder als eigenständige Objekte in einer Berechtigung auftauchen. Für Assoziationen gibt es zwei Aktionen, das Anlegen und das Löschen derselben (siehe Abb. 3.8, 3.11 und Kap. 4.2.1). Diese Aktionen können unabhängig mit Berechtigungen belegt werden, falls es sich um Kann-Assoziationen handelt. Handelt es sich um Pflichtassoziationen, so sollte das Anlegen und Löschen mit denselben Rechten verknüpft sein, wie das Anlegen und Löschen des zu der Assoziation gehörenden Geschäftsobjekts. In der Dokumentenverwaltung gibt es z. B. das Recht zum Anlegen eines Dokuments. Immer, wenn das Dokument angelegt wird, wird auch eine Assoziation zum dazugehörigen Verzeichnis und zu seinem Ersteller angelegt. Daher sollten die Berechtigungen zum Anlegen dieser beiden Assoziationen mit der Berechtigung zum Anlegen eines Dokuments übereinstimmen. Die Zuordnung von einem Dokument zu seiner Lesergruppe hingegen kann unabhängig erfolgen. Daher kann die Berechtigung zum Anlegen und Löschen dieser Assoziation auch unabhängig von der Berechtigung zum Anlegen eines Dokuments formuliert werden. Die Berechtigung zum Zugriff auf ein assoziiertes Geschäftsobjekt sollte so sein, wie die Berechtigung auf das jeweilige Geschäftsobjekt selbst. Die beiden beschriebenen Möglichkeiten bilden eine hierarchische Ordnung auf modellierten Objekten. Im Allgemeinen überschreiben spezifische Rechte die allgemeinen Rechte, in diesem Fall überschreiben Rechte, die auf Attributen definiert sind, die Rechte, die auf ganzen Geschäftsobjekten definiert sind.

Berechtigungen beinhalten immer das Tripel {Subjekt, Objekte, Aktion}. Sind die Objekte für eine Berechtigung nicht weiter spezifiziert, bezieht sich eine Berechtigung also auf die Menge aller Objekte, so handelt es sich um eine *objektunabhängige* Berechtigung. Ein Beispiel für eine objektunabhängige Berechtigung wäre: „Ein Subjekt, welches authentifiziert ist, darf alle Aktionen auf allen Objekten ausführen.“ Berechtigungen, die die Objekte weiter klassifizieren sind *objektabhängig*. Fast alle Berechtigungen sind objektabhängig.

Ebenso wie die Subjekte und die Objekte können natürlich auch die Aktionen einer Berechtigung *aktionsabhängig*, mit einer grobgranularen oder feingranularen Modellierung der Aktionen, oder gar nicht, d. h. *aktionsunabhängig*, modelliert werden. Typische Aktionen für die Berechtigungsmodellierung sind die Basisaktionen Create, Read, Update und Delete (siehe 5.1.2). Die Dienste, die auf Geschäftsobjekten ausgeführt werden, lassen sich auf diese vier Basisaktionen bzw. Kombination aus diesen Basisaktionen zurückführen.

In einem objektorientierten System möchte man häufig Aktionen auf einer Ebene definieren, die mehr Kontextinformationen bereitstellt als die Ebene der Basisaktionen. Es sollen also anwendungsspezifische Aktionen auf Methodenebene definiert werden, so dass eine Aktion jeweils mit dem Ausführen einer Geschäftsobjektmethode korrespondiert. Die Ausführung von Methoden enthält wieder die vier genannten Operationen auf der Datenebene. In der vorliegenden Arbeit werden keine anwendungsspezifischen Aktionen als Aktionen für die Berechtigungen betrachtet, denn die Implementierung einer Methode kann wieder andere Methoden aufrufen. Stellt eine Methode eine Aktion einer Berechtigung dar, so kann diese nun wiederum andere Aktionen anderer Berechtigungen beinhalten. Bei diesem Vorgehen ist sehr große Aufmerksamkeit bei der Vergabe der Berechtigungen gefordert, damit die Berechtigungen konsistent sind und die anwendungsspezifischen Methoden auch tatsächlich ausgeführt werden können.

Es können auch Methoden geschützt werden, die nicht das Datenmodell verändern, sondern den Zugriff auf anderweitige Aktionen schützen. Ein Beispiel hierfür wäre das Versenden einer Email, das Aufbauen einer Kommunikationsverbindung oder das Drucken. In der vorliegenden Arbeit werden nur persistente Daten zugriffsgeschützt.

Ebenso wie die Objekte können auch die Aktionen weitere Nebenbedingungen enthalten. Z. B. könnten Aktionen nur für bestimmte Parameterwerte erlaubt sein. Man stelle sich hierfür das System als eine Zustandsmaschine vor, die nur bestimmte Änderungen zulässt. Befindet sich das System in einem Zustand, so sind nur bestimmte Folgezustände erlaubt, d. h. das Ausführen einer Aktion ist nur für bestimmte Parameter der Aktion möglich.

Allgemein gilt, dass die Modellierung der Aktionen abhängig ist von der Modellierung der Objekte. Sind die Objekte Geschäftsobjekte der Anwendung, so kann es sich bei den Aktionen um Methoden der Geschäftsobjekte handeln. Sind die Objekte Attribute von Geschäftsobjekten, so sind für Attribute primitiven Datentyps nur die Aktionen R und U möglich. Für Assoziationen sind die Operationen „Assoziation anlegen“ (eine C-Aktion) und „Assoziation löschen“ (eine D-Aktion) denkbar. Ebenfalls ist es möglich, die Dienste einer Anwendung als Objekt für eine Berechtigung zu betrachten. Die Aktionen sind dann auf der Geschäftsprozessebene definiert. Die Ausführung eines Geschäftsprozesses beinhaltet wieder Rechte auf der Ebene der Geschäftsobjektmethoden. Bezüglich der Verwaltung der Berechtigungen innerhalb der Anwendung gilt, dass mit steigender Granularität der Berechtigungen auch der Verwaltungsaufwand steigt.

Weiterhin lassen sich zwei Kategorien von Berechtigungen unterscheiden, die administrativen Rechte und die anwendungsspezifischen Rechte. *Administrative Rechte* sind Rechte, die das Verwalten der anwendungsspezifischen Rechte ermöglichen, während die *anwendungsspezifischen Recht* die Zugriffe auf die geschützten Ressourcen regeln.

Zu den administrativen Rechten gehört zum einen die Benutzerverwaltung mit dem Erstellen und Löschen von Personen und Gruppen. Personen und Gruppen sind ja die Subjekte der Berechtigungen. In einer rollenbasierten Berechtigungsverwaltung (siehe 2.3.2) werden Personen und Benutzergruppen verschiedenen Rollen zugeordnet, so dass ebenfalls die Rollenverwaltung zu den Tätigkeiten gehört, für die administrative Rechten gelten. Zu guter Letzt müssen natürlich auch die Rechte der Anwendung selbst verwaltet werden, d. h. die Zuordnung von Objekt, Aktion und Subjekt. In einer systembestimmten und rollenbasierten Berechtigungsstruktur sind die administrativen Rechte einer kleinen Gruppe von Administratoren vorbehalten. In einem System mit benutzerbestimmbarer Zugriffskontrolle haben die Eigentümer der jeweiligen Ressourcen das Recht, entsprechende Zugriffsrechte zu definieren, die Administration ist dezentral.

5.1.4 Definition des Begriffs „Policy“

Eine *Policy* ist ein Plan, den eine Organisation hat, um ihre Ziele zu erreichen [Hos92]. Eine Policy stellt somit Richtlinien und Verhaltensregeln auf, die für eine Organisation bindenden Charakter haben. Die einzelnen Regeln, aus denen eine Policy aufgebaut ist, können z. B. Verpflichtungen der Mitarbeiter und Managementaufgaben umfassen. Eine *Sicherheitspolicy* ist ein Plan einer Organisation, ihre Sicherheitsziele zu erreichen. Sie umfasst Maßnahmen wie Authentifizierung, Autorisierung, Auditing, Backup und Systemwiederherstellung. Eine Sicherheitspolicy, die Autorisierungsziele beschreibt, heißt *Autorisierungspolicy*. Sie ist aus Regeln aufgebaut, die jeweils einer Berechtigung

entsprechen. Der Begriff *Regel* ist hier also ein Synonym für Berechtigung. Eine Regel enthält Elemente für die Spezifikation von Subjekten, Objekten, Aktionen und Nebenbedingungen und kann einige oder alle der oben diskutierten Charakteristika (siehe 5.1.2), wie etwa eine Polarität, aufweisen.

Der Teil der Sicherheitspolicy, welcher in einem System implementiert werden kann, ist die *automatisierte Sicherheitspolicy* [Hos92]. Es ist gängige Vorgehensweise, die automatisierte Sicherheitspolicy deklarativ zu spezifizieren, so dass diese unabhängig von dem Mechanismus existiert, der die Regeln implementiert. Im Fall von Autorisierungspolicies ist der Mechanismus, der die Zugriffsentscheidungen trifft und durchsetzt, unabhängig von der Berechtigungsspezifikation. Auf diese Weise können Berechtigungsspezifikation und Implementierung getrennt modifiziert werden.

Es gibt eine Vielzahl an deklarativen Berechtigungsbeschreibungssprachen. Einige werden weiter unten in Kapitel 5.2 vorgestellt und diskutiert.

Der allgemeine Aufbau und die Integritätsbedingungen, die eine konkrete Policy realisieren muss, werden durch eine *Metapolicy* beschrieben. Eine Metapolicy enthält Metadaten zu einer konkreten Policy. Dies können z. B. eine informelle Beschreibung des Inhalts und des Zwecks der Policy, der Autor bzw. der für die Policy Verantwortliche, der Gültigkeitszeitraum und der Gültigkeitsbereich sein. Die Aufgaben einer Metapolicy sind weiterhin, die Auswertungsreihenfolge und die Prioritäten für die einzelnen Regeln festzulegen, eine Konfliktauflösungsstrategie bei sich widersprechenden Regeln bereitzustellen, sowie Defaultwerte bereitzustellen für Anfragen, die nicht durch die Policy beantwortet werden können. Auf die Aufgaben der Metapolicy wird im folgenden Unterkapitel 5.1.5 eingegangen. Metapolicies können zudem für eine Policy gültige Invarianten und Integritätsbedingungen beschreiben. Eine Metapolicy für die Autorisierungspolicy kann z. B. Einschränkungen folgender Art definieren [JSS97, DDL+01]:

- *Inkompatible Gruppen*: Zwei Gruppen verhalten sich komplementär zueinander und es kann kein Subjekt geben, welches Mitglied in beiden Gruppen ist. Z. B. sind die Gruppen „Staatsbürger“ und „Ausländer“ disjunkt.
- *Inkompatible Rollenzuweisung (Statische Aufgabenteilung*, siehe 2.3.2*)*: Zwei Rollen verhalten sich komplementär zueinander und es kann kein Subjekt geben, welches beide Rollen, gleichzeitig oder hintereinander, aktiviert. Z. B. sind die Rollen „Prüfer“ und „Prüfling“ disjunkt.
- *Inkompatible Rollenaktivierung (Dynamische Aufgabenteilung*, siehe 2.3.2*)*: Zwei Rollen dürfen nicht zur selben Zeit von einem Subjekt aktiviert werden. Dies kann notwendig sein, um zu verhindern, dass eine Person während einer Sitzung zu viele Rechte vereint.
- *Selbstmanagement*: Ein Subjekt soll nicht die Möglichkeit haben, Policies zurückzuziehen, bei denen es als Subjekt auftaucht. Diese Bedingung bezieht sich darauf, dass es Managern in einem Unternehmen nicht möglich sein soll, sich von Aufgaben zurückzuziehen, die sie erledigen sollen. Diese Metapolicy bezieht sich nicht nur auf die Autorisierung, sondern auf allgemeine Managementaufgaben, die durch Policies beschrieben werden.

Die Forcierung der hier erwähnten Invarianten durch Metapolicies kann unter Umständen sehr aufwendig sein. Insbesondere gibt es Integritätsbedingungen, für deren Überprüfung auf dynamische Information wie die aktuelle Benutzersitzung innerhalb einer Anwendung zugegriffen werden muss. Für die inkompatible Rollenaktivierung muss z. B. überprüft werden, welche Rollen ein gegebener Benutzer in welcher Sitzung gerade aktiviert hat. Der Mechanismus, welcher die Entscheidung trifft, ob ein Benutzer eine Rolle aktivieren darf, muss also sehr viele Zustandsinformationen verwalten und sollte auch darüber informiert werden, wann ein Benutzer eine Sitzung geschlossen hat, um die Information über diese Sitzung verwerfen zu können.

5.1.5 Charakteristika von Policies

Die Spezifikation einer Autorisierungspolicy ist korrekt, wenn sie konsistent und vollständig ist [JSS97]. Eine Spezifikation ist dann vollständig, wenn für jede Zugriffsanfrage eine Zugriffsentscheidung getroffen werden kann. Um die Anzahl der zu definierenden Policies zu verringern, ist es gebräuchlich, eine Defaultpolicy zu definieren, die anzuwenden ist, wenn es keine Policy gibt, die auf in einer Zugriffsanfrage enthaltenen Subjekte, Objekte und / oder Aktionen zutrifft. Bei dieser Policy handelt es sich um eine *geschlossene Policy*, wenn die Rechteverwaltung nach dem Verbotsprinzip (vgl. Kap. 2.1.3) realisiert ist. Eine *offene Policy* realisiert das Erlaubnisprinzip. In der vorliegenden

Arbeit wird eine geschlossene Policy favorisiert, da diese Auswirkung möglicher Auslassungen in der Berechtigungsspezifikation nicht dazu führt, dass versehentlich zu viele Rechte an ein Subjekt vergeben werden.

Eine Spezifikation ist dann *konsistent*, wenn für jede Zugriffsanfrage nur eine Art von Zugriffsentscheidung möglich ist. Es sei an dieser Stelle angemerkt, dass es innerhalb einer Anwendungspolicy Autorisierungsregeln geben kann, die bei ein- und derselben Zugriffsanfrage zu verschiedenen Zugriffsentscheidungen kommen. Die Metapolicy muss also einen *Regelauswertungsalgorithmus* definieren, der diese Konflikte auflöst.

Für die Wahl des Regelauswertungsalgorithmus gibt es verschiedene Möglichkeiten [SL02, GM+03]:

- *Regeln negativer Polarität überschreiben Regeln positiver Polarität*: Dieser Auswertungsalgorithmus spezifiziert, dass die Priorität negativer Regeln immer höher ist als die Priorität positiver Regeln.
- *Regeln erhalten explizit zugewiesene Prioritäten*: Jede Regel erhält eine spezifische Priorität. Bei diesem Verfahren ist darauf zu achten, dass es keine zwei Regeln unterschiedlicher Polarität aber gleicher Priorität geben darf, die auf dieselbe Zugriffsanfrage anwendbar sind, da die Policyspezifikation dann inkonsistent ist und eine Zugriffsentscheidung nicht möglich ist. Wenn die Regeln per Hand zugewiesen werden, ist es bei einem großen Regelsatz sehr schwierig, für eine Regel die jeweils beste Priorität auszuwählen.
- *Regeln mit der geringsten Distanz haben Vorrang*: Bei diesem Auswertungsalgorithmus wird ein Distanzmaß festgelegt, welches sich auf die Objekte oder andere Attribute bezieht. Z. B. könnte, falls es mehr als eine Regel gibt, die auf eine Zugriffsanfrage zutrifft, diejenige genommen werden, deren Objekt in der Vererbungshierarchie dem in der Zugriffsanfrage auftauchenden Objekt am nächsten ist. Andere Distanzmaße können sein, dass bei mehreren zutreffenden Regeln diejenige genommen wird, die am letzten geändert wurde, oder diejenige eines bestimmten Autors vorgezogen wird.
- *Spezifischere Regeln überschreiben unspezifische Regeln*: Dieser Auswertungsalgorithmus ähnelt dem Auswertungsalgorithmus mit Distanzmaß. Allerdings werden hier alle Elemente einer Regel, also Subjekte, Objekte und Aktionen, für die Bestimmung der spezifischsten Regel herangezogen. Eine Regel ist dann spezifischer, wenn sie auf eine kleinere Menge an Subjekten, Objekten oder Aktionen zutrifft. Dieser Auswertungsalgorithmus unterliegt allerdings einigen Einschränkungen, denn falls z. B. die Menge der Subjekte kleiner aber die Menge der Objekte größer ist, so lässt sich nicht ohne Weiteres feststellen, welche Regel spezifischer sein soll [LS99].
- *Die erste anwendbare Regel entscheidet*: Dieser Auswertungsalgorithmus weist allen Regeln ihrer Reihenfolge nach eine implizite Priorität zu. Dieser Auswertungsalgorithmus ist ähnlich zu dem „Regeln erhalten explizit zugewiesene Prioritäten“, nur, dass hier keine Konflikte auftreten, falls es in der Policyspezifikation inkonsistente Regeln gibt. Die Auswertung wird ja beendet, sobald eine zutreffende Regel gefunden wurde. Man beachte außerdem, dass es immer mindestens eine anwendbare Regel gibt, wenn die Anwendungspolicy eine Defaultpolicy enthält.
- *Nur genau eine Regel darf anwendbar sein*: Dieser Auswertungsalgorithmus geht Konflikten aus dem Weg, indem postuliert wird, dass es keine zwei sich widersprechenden Regeln geben darf.

5.1.6 Designprinzipien für Policysprachen

Im Folgenden werden einige Designkriterien diskutiert, die eine Policysprache erfüllen sollte, um nicht nur die gewünschte Ausdrucksmächtigkeit sondern auch eine gute Administrierbarkeit zu gewährleisten.

Skalierbarkeit: Die Policysprache sollte so strukturiert sein, dass sie auch für große Anwendungen mit einer Vielzahl an Berechtigungen eingesetzt werden kann. Ein *hierarchischer Aufbau* der Policysprache verringert die Gesamtanzahl und damit die Komplexität zu definierender Regeln. Zunächst sollte eine Defaultpolicy bestehen, die eine einheitliche Zugriffsentscheidung für alle Zugriffsanfragen bereitstellt. Diese Defaultpolicy kann dann durch Policies für spezielle Subjekte und Objekte überschrieben werden. Im Fall von betrieblichen Anwendungen sind die Geschäftsobjekte die Objekte, die

innerhalb einer Berechtigung modelliert werden. Die unter 4.2.4 diskutierte Ausdrucksmächtigkeit erfordert, dass pro Geschäftsobjekttyp Berechtigungen definiert werden können. Diese Geschäftsobjektpolicy wird durch eine Attributpolicy verfeinert, d. h. überschrieben, denn insbesondere Lesezugriffe auf einzelne Attribute von Geschäftsobjekten werden aufgrund von höheren Vertraulichkeits- und Privatheitsanforderungen noch einmal strenger behandelt, als der Zugriff auf ein Geschäftsobjekt. Eine andere Form der hierarchischen Strukturierung wird in Ponder (siehe 5.2.3) realisiert. Es können Templates definiert werden, aus denen konkrete Policies abgeleitet werden können. Skalierbarkeit kann auch erleichtert werden, indem die Bestandteile einer Berechtigung wieder verwendet werden können, um neue Berechtigungen zu definieren. Mit Hilfe von Zugriffsprinzipien (siehe 4.2.2) kann diese Art der Wiederverwendung gewährleistet werden. Es können bestehende Zugriffsprinzipien als Artefakte bereitgestellt und neu kombiniert werden, um eine Berechtigung abzubilden.

Erweiterbarkeit: Es soll einfach möglich sein, neue Regeln und Policies hinzuzufügen sowie neue Sprachelemente einzufügen. Bspw. sollte es einfach möglich sein, eine bestehende Policysprache für die Definition von Berechtigungen so zu erweitern, dass auch weitere Anforderungen wie die Delegation und die Verpflichtungen modelliert werden können. Mit Hilfe der Zugriffsprinzipien ist es möglich, die Erweiterbarkeit einer Policysprache zu gewährleisten. Zum einen sind Zugriffsprinzipien eigenständige Artefakte, die einfach erstellt werden können. Zum anderen kann ein objektorientiertes Design einer auf Zugriffsprinzipien basierenden Policysprache einfach um weitere Klassen erweitert werden, die z. B. Verpflichtungen modellieren.

Einfachheit / Verständlichkeit: Die Policysprache sollte intuitiv verständlich sein. Die Zugriffsentscheidungen sollten für den Administrator der Policysprache und für die Nutzer einer Anwendung einfach nachvollziehbar sein. Die Verständlichkeit und Administrierbarkeit einer Policy wird erhöht, wenn die Zugriffsentscheidung nicht von der Reihenfolge der Regeln innerhalb einer Policy abhängt. Regelauswertungsalgorithmen, die unabhängig von der Reihenfolge der Regeln zu Zugriffsentscheidungen gelangen, sind vorzuziehen. Die Verständlichkeit der Policysprache wird ebenfalls dadurch erhöht, dass die Freiheitsgrade bei der Definition von Regeln eingeschränkt werden. Wenn z. B. pro Regel nur genau eine Ressource, im Falle von betrieblichen Anwendungen nur ein Geschäftsobjekttyp, verwendet werden kann, so ist es einfacher, einen bestehenden Regelsatz zu ändern, als wenn innerhalb einer Regel viele Geschäftsobjekttypen als Objekte auftauchen. Die Verständlichkeit einer Zugriffsentscheidung wird dadurch erhöht, dass auf negative Autorisierung verzichtet wird. Dann treten nämlich keine Polaritätskonflikte innerhalb einer Policy auf.

5.2 Policysprachen

Zum Aufstellen einer Anwendungspolicy werden Berechtigungsbeschreibungssprachen eingesetzt sowie Auswertungsalgorithmien definiert, welche die Anwendungspolicy für eine Zugriffsanfrage auswerten. Die herkömmliche Art und Weise der Rechtedefinition erfolgt über Zugriffskontrolllisten und Zugriffsausweise und implizite Berechtigungsauswertungsmechanismen. Sie wird in Abschnitt 5.2.1 erläutert. Neuere Berechtigungsbeschreibungssprachen sind deklarativ. Sie legen die Zugriffskontrollanforderungen in einem deklarativen Regelwerk ab. Ein Regelauswertungsalgorithmus wertet dann (zur Laufzeit) aus, ob einem Subjekt der gewünschte Zugriff auf ein Objekt erlaubt ist oder nicht. Der Regelauswertungsalgorithmus kann hier explizit und unabhängig von der Definition der einzelnen Berechtigungen angegeben werden. Bei diesem Ansatz ist zu beachten, dass die Rechte nicht physisch direkt mit Objekt oder Subjekt verbunden sind, da die Rechte in einem externen Dokument oder in einer externen Komponente abgelegt werden. D. h., die Regelsätze können nur auf Subjekte bzw. Objekte verweisen. In den Unterkapiteln 5.2.2 bis 5.2.5 werden einige dieser Beschreibungssprachen beschrieben. Das Unterkapitel schließt mit einer Auswertung und Diskussion der untersuchten Policysprachen.

5.2.1 Zugriffskontrolllisten und Zugriffsausweise

Der älteste Mechanismus der Berechtigungsauswertung in einem System ist die Zugriffskontrollliste (engl.: access control list / ACL). Sie wurde und wird insbesondere in Betriebssystemen eingesetzt. In [SS75] werden Implementierungsmöglichkeiten in der Hardware aufgezeigt. Eine ACL speichert bei jedem zugriffsgeschützten Objekt, welche Subjekte welche Aktionen auf diesem Objekt ausführen dürfen. Grundlage der ACL ist die Lampson-Matrix [Lam71]. Die einzelnen Zeilen dieser Matrix beschreiben die Subjekte, die Spalten die Objekte. An den Kreuzungspunkten, also in den Zellen der Matrix, werden die Aktionen, die das entsprechende Subjekt auf dem entsprechenden Objekt ausführen darf, notiert. Bei der Lampson-Matrix handelt es sich um eine spärlich besetzte Matrix. Die Zugriffskontrollliste speichert jetzt nur noch nichtleere Zellen der Matrix und ordnet entsprechende Zellen dem betreffenden Objekt zu. Die Zugriffskontrollliste eignet sich für Systeme, welche eine geringe Anzahl an verschiedenen Subjekten verwalten. Dann ist nämlich die Zugriffskontrollliste kurz und es können effizient die Berechtigungen abgefragt werden, indem einfach in der Liste nachgeschaut wird, ob für das entsprechende Subjekt ein Eintrag vorhanden ist. Da die Zugriffskontrollliste bei dem Objekt gespeichert wird, lassen sich auch Änderungen der Zugriffspolicy einfach umsetzen, indem die entsprechenden Listeneinträge geändert werden. Umgekehrt ist es aufwendig herauszufinden, welche Rechte ein bestimmtes Subjekt besitzt.

Die ACL wird u. a. im Windows Betriebssystem [DPS03, Win32] eingesetzt. Da Windows auch implizite Rechte in Form von Vererbung von Zugriffsrechten auf Unterobjekte zulässt und außerdem positive und negative Polaritäten von Rechten, ist die Berechtigungsauswertung sehr komplex. In Windows sind die Prioritäten durch die Position der ACL-Einträge in der ACL gegeben. Eine Verschiebung eines Eintrags innerhalb der ACL kann das Auswertungsergebnis einer Berechtigungsanfrage verändern. Zugriffskontrolllisten beinhalten also implizite Auswertungsregeln. Die Berechtigungsinformationen sind dezentral bei den einzelnen Objekten gespeichert. Falls sich ACL-Einträge von Objekten auf andere Objekte vererben können, reicht es nicht immer aus, die ACL eines isolierten Objekts zu betrachten, um zu einer Berechtigungsentscheidung zu gelangen.

Für die Rechteadministration innerhalb eines Unternehmens ist es wichtig, dass es einen Gesamtüberblick über alle Subjekte und mit den Subjekten verbundenen Zugriffsrechte gibt. Eine Zugriffskontrollliste bietet hier keine optimale Lösung, da die Verwaltung der ACLs dezentral pro Objekt stattfindet und Hierarchie- und Implikationsstrukturen zwischen verschiedenen ACLs aufwendig zu verwalten sind. Aus diesen Gründen sollten für betriebliche Anwendungen in Unternehmen regelbasierte Zugriffkontrollmechanismen eingesetzt werden, d. h. Mechanismen, die mit der Aufstellung expliziter Regeln arbeiten. Diese werden im folgenden Unterkapitel diskutiert.

Ein zu den ACLs komplementärer Ansatz sind die Zugriffsausweise (engl.: capability). Hier werden, ausgehend von der Lampson-Matrix, die nichtleeren Zellen der Zugriffsmatrix den jeweiligen Subjekten zugeordnet. Dieser Ansatz eignet sich für verteilte Systeme, in denen es eine große Anzahl an Subjekten aber nur wenig verwaltete Objekte gibt, da dann die Capability-Liste relativ kurz ist. Bekannte Herausforderungen für die Zugriffsausweise sind, dass die Zugriffsausweise fälschungssicher implementiert werden müssen, da sie auf Clientseite bei den entsprechenden Subjekten gespeichert werden, und dass es ein effizientes Konzept für die Rechterücknahme geben muss. Da die Zugriffsausweise verteilt sind, können sie nicht auf einfache Art und Weise verändert werden. Um dieses Problem zu lösen, erhalten Zugriffsausweise ein Verfallsdatum, und bei Änderungen der Berechtigungen muss zusätzlich eine zentrale Liste von nicht mehr gültigen Capabilities gehalten werden. Für die in dieser Arbeit betrachteten Anwendungen eignen sich die Zugriffsausweise nicht. Denn genau wie bei den ACLs ist auch bei den Zugriffsausweisen kein Gesamtüberblick über alle im System vorhandenen Berechtigungen gegeben. Dieser Gesamtüberblick kann nur über eine zentrale Speicherung aller Berechtigungen realisiert werden.

5.2.2 Authorization Specification Language

Die Authorization Specification Language (ASL) [JSS97] ist eine regelbasierte Sprache, mit der Zugriffskontrollpolicies ausgedrückt werden können. Sie basiert auf stratifiziertem Datalog [KE06]. Die Spezifikationssprache ist insgesamt sehr flexibel, da hier sowohl Regeln darstellbar sind, die auf einer

offenen Policy basieren als auch solche, die auf einer geschlossenen Policy basieren. In ASL können Berechtigungen sowohl positiver als auch negativer Polarität dargestellt werden. Subjekte können in einer Gruppenhierarchie geschachtelt werden, wobei ein Subjekt Mitglied in mehreren Gruppen sein kann. Ebenso können Rollenhierarchien, ebenfalls mit Mehrfachvererbung, modelliert werden. Der Unterschied zwischen Gruppen und Rollen besteht darin, dass Rollenmitgliedschaften von einem Subjekt aktiviert und deaktiviert werden können, während die Gruppenmitgliedschaften eines Subjekts immer gültig sind. Im Gegensatz zu anderen Berechtigungsbeschreibungssprachen können hier Berechtigungen ausgedrückt werden, die von einer direkten Mitgliedschaft oder einer indirekten Mitgliedschaft in einer Gruppe oder Rolle, abhängen. Indirekte Mitgliedschaft bedeutet hier, dass ein Subjekt über mehrere Hierarchieebenen hinweg einer Gruppe oder Rolle zugeordnet ist. Die Subjekte, die innerhalb einer durch ASL spezifizierten Regel vorkommen, werden hier als Autorisierungssubjekte bezeichnet. Ein Autorisierungssubjekt ist ein Subjekt, eine Gruppe oder eine Rolle. Objekte können bezüglich ihres Typs qualifiziert werden.

ASL definiert sechs verschiedene Arten von Regeln. Zum Ersten können explizite Regeln definiert werden, zum Zweiten gibt es Herleitungsregeln, die aus der Existenz einer bestimmten Regel auf eine neue implizierte Berechtigung schließen. In Abb. 5.1 ist ein Beispiel für eine explizite *cando*-Regel gegeben: ein Autorisierungssubjekt *s* hat eine (positive) Schreiberlaubnis auf Datei *file2*, wenn es direktes Mitglied in der Gruppe *CS-Faculty* ist. Die zweite Regel ist ein Beispiel für eine implizite *dercando*-Regel. Ein Autorisierungssubjekt *s* hat ein positives Zugriffsrecht *a* auf Objekt *o*, wenn *s'* ein solches positives Zugriffsrecht besitzt und *s* direktes Mitglied in *s'* ist.

```
cando (file2, s, +write) ← in (s, CS-Faculty).
dercando (o, s, +a) ← cando (o, s', +a) & in (s, s').
```

Abb. 5.1: Beispiel für eine explizite und eine implizite Regel, Quelle: [JSS97]

Da es sowohl Berechtigungen positiver als auch negativer Polarität geben kann, gibt es eine dritte Art von Regeln, die Metapolicies, welche die Konfliktauflösung spezifizieren. Weiterhin ist es wegen der erlaubten Mehrfachvererbung von Gruppen- und Rollenhierarchien notwendig zu spezifizieren, wie die Rechte, die auf Oberklassen spezifiziert wurden, auf die Unterklassen vererbt werden. Diese Spezifikation wird durch die Herleitungsregeln festgesetzt. Z. B. können alle Regeln auf allen Pfaden von einer Oberklasse zu einem tatsächlichen Subjekt propagiert werden. Dann muss eventuell anschließend die Existenz mehrerer sich widersprechender Regeln durch Konfliktauflösungsregeln auf der untersten Ebene aufgelöst werden. Für die Konfliktauflösung kann es mehrere Ansätze geben, wie bereits unter 5.1.5 erläutert. Der erste wäre: es darf kein Konflikt vorkommen, ansonsten ist die Policy fehlerhaft. Andere Ansätze wären: negative Berechtigungen überschreiben positive, positive überschreiben negative, oder bei der Existenz von sowohl positiver als auch negativer Berechtigung wird eine Defaultpolicy angewendet.

Als Viertes gibt es Regeln, welche die Berechtigungen durchsetzen, d. h. wenn bestimmte explizite, implizite und / oder Regeln existieren, deren Konflikte aufgelöst wurden, werden diese tatsächlich forciert. Diese Durchsetzungsregeln sind also die eigentlichen Regeln, die innerhalb der Anwendung benutzt werden.

Ein fünfter Typ von Regeln ist einer, der während des Betriebs einer Anwendung hinzugefügt wird. Diese Regeln sind eigentlich Fakten. Sie stellen dar, dass eine bestimmte Aktion von einem bestimmten Subjekt mit einer bestimmten Menge aktiver Rollen auf einem bestimmten Objekt durchgeführt worden ist. Das Erstellen dieser Fakten ist dann notwendig, wenn es Regeln gibt, die den Zugriff auf eine Ressource aufgrund der Historie vorher getätigter Aktionen einschränken oder erlauben. Ein solches Berechtigungsmodell ist z. B. durch das Chinese-Wall-Modell [BN89] gegeben. Der letzte Typ von Regeln sind die Integritätsregeln. Durch sie werden Metapolicies, wie etwa das Verbot von inkompatiblen Gruppen, inkompatibler Rollenaktivierung sowie statischer und dynamischer Aufgabenteilung definiert. Es können aber auch anwendungsspezifische Integritätsbedingungen modelliert werden.

ASL ist eine sehr ausdrucksmächtige Sprache, die Möglichkeiten bietet, globale Vererbungs- und Integritätsbedingungen zu definieren. Sprachelemente zur Darstellung komplexer Nebenbedingungen

auf Zuständen / Attributen von Objekten oder Subjekten sind nicht definiert. Verpflichtungen werden in ASL ebenfalls nicht behandelt. Delegation wird nicht explizit erwähnt, es müsste aber möglich sein, diese durch Herleitungsregeln darzustellen.

5.2.3 Ponder

Ponder [DDL+00, DDL+01, SL02] ist eine deklarative, objektorientierte und streng typisierte Spezifikationssprache für die policy-basierte Verwaltung von Netzwerken und verteilten Systemen. Eine Policy ist hier eine Regel, welche eine Auswahl im Systemverhalten darstellt. Ponder unterscheidet vier verschiedene Arten von Policies. Neben den Autorisierungspolicies, die hier näher betrachtet werden, gibt es Delegationspolicies, welche eine (meist zeitbeschränkte) Rechteweitergabe an andere Subjekte modelliert, Informationsfilter, welche Ein- und Ausgabeparameter von Aktionen an Bedingungen geknüpft verändern können, sowie Unterlassungspolicies, welche Regeln aufstellen, die nicht von einer Zugriffsentscheidungskomponente, sondern vom Subjekt selbst überprüft und eingehalten werden. Neben diesen Policies können auch Verpflichtungen spezifiziert werden.

Eine Autorisierungspolicy in Ponder kann von positiver oder negativer Polarität sein. Eine solche Policy spezifiziert eine Berechtigung. Es können Subjekte, Objekte, Aktionen und Nebenbedingungen angegeben werden. Aktionen sind hier Methodenaufrufe auf den Objekten. Nebenbedingungen werden in OCL-Syntax notiert. Es kann sich um Zustandseinschränkungen von Subjekten oder Objekten, um Einschränkungen von Parameterwerten von Aktionen oder um zeitliche Beschränkungen handeln. Um eine Policyverwaltung großer Systeme mit einer Vielzahl an Subjekten und Objekten zu ermöglichen, können Subjekte und Objekte in Gruppen angeordnet werden. In Abb. 5.2 ist ein Beispiel für eine positive Autorisierung gegeben. Alle Mitglieder der Gruppe *NetworkAdmin* können auf allen Objekten vom Typ *PolicyT* in der Domäne *Nregion/switches* die Aktionen *load( ), remove( ), enable( )* und *disable( )* durchführen.

Ponder erlaubt die Wiederverwendung von deklarierten Policies. Hierzu können Autorisierungstypen deklariert werden. Ein Autorisierungstyp entspricht einer Templateklasse, die dadurch instanziiert werden kann, dass ihre Parameter mit konkreten Werten versehen werden. Aus jedem Autorisierungstyp können also mehrere konkrete Autorisierungsinstanzen abgeleitet werden. In Abb. 5.3 ist ein Beispiel für eine solche Templatepolicy mit zwei konkreten Instanzen gegeben. Subjekt und Objekt sind Parameter des Autorisierungstyps *PolicyOpsT*. Zwei konkrete Autorisierungen werden hiervon abgeleitet. Die erste entspricht der in Abb. 5.2 deklarierten Policy.

```
inst auth+ switchPolicyOps {
      subject            /NetworkAdmin;
      target <PolicyT> /Nregion/switches;
      action             load(), remove(), enable(), disable() ;
}
```

Abb. 5.2: Beispiel für eine Autorisierungspolicy, Quelle: [DDL+01]

In Ponder können auch Metapolicies spezifiziert werden, mit denen Selbstmanagement, Trennung von Zuständigkeiten u. ä. definiert werden können. Die Integritätsbedingungen, die eine Metapolicy überprüfen soll, werden, genau wie die Nebenbedingungen als OCL-Ausdruck angegeben.
Für eine weitere Strukturierung können Policies zu Gruppen zusammengefasst werden. Eine Gruppierung von Policies kann hier zwecks Wiederverwendung auch analog zu den Autorisierungstypen als Template angegeben werden. Ebenfalls ist es möglich, Rollen und Rollenhierarchien direkt oder als Templates zu modellieren.

```
type auth+ PolicyOpsT (subject s, target <PolicyT> t) {
      action load(), remove(), enable(), disable();
}

inst auth+ switchPolicyOps =
           PolicyOpsT (/NetworkAdmins, /Nregion/switches);
inst auth+ routersPolicyOps =
           PolicyOpsT (/QoSAdmins, /Nregion/routers);
```

Abb. 5.3: Beispiel für eine Templatepolicy mit zwei Instanzen, Quelle: [DDL+01]

Ponder ist insgesamt eine sehr umfangreiche, flexible und ausdrucksmächtige Sprache zur Spezifikation von Autorisierungspolicies. Die Definition von Regelauswertungsalgorithmen ist nicht Teil der Sprache Ponder. In [SL02] werden von zwei der Autoren von Ponder Auswertungsregeln für Policies diskutiert. Es existiert ein Ponder Toolkit, welches einen Editor für die Erstellung und Verwaltung von Ponder Policies bereitstellt [PRG06].

5.2.4 Binder

Binder [DeT02] ist eine logik-basierte Sprache, eine Erweiterung von Datalog, mit der Sicherheitspolicies ausgedrückt werden können. Die Binder-Sprache enthält, genau wie Datalog, Fakten und Regeln, die aus Fakten aufgebaut sind. Aus den Fakten und Regeln können neue Fakten hergeleitet werden. In Abb. 5.4 ist die erste Aussage eine Regel, die besagt, dass ein Prinzipal *X* die Ressource *resource_r* lesen darf, wenn er Mitarbeiter der Firma *bigco* ist. Die zweite Aussage ist ein Fakt und besagt, dass der Prinzipal *john_smith* Mitarbeiter der Firma *bigco* ist. Insgesamt lässt sich aus diesen beiden Sätzen herleiten, dass *johnt_smith* lesend auf *resource_r* zugreifen darf.

```
can (X, read, resource_r) :- employee(X, bigco).
employee (john_smith, bigco).
```

Abb. 5.4: Beispiel für eine Binder-Regel und einen Binder-Fakt, Quelle: [DeT02]

In einer verteilten Umgebung hat jede Komponente, eine Komponente kann z. B. ein Dienst sein, einen lokalen Kontext, in dem die lokalen Fakten und Regeln abgelegt sind. Je nach Bedarf können die lokalen Regeln exportiert und so anderen Komponenten für ihre Entscheidungsfindung zur Verfügung gestellt werden. Der Export lokaler Aussagen erfolgt über die mit dem privaten Schlüssel des jeweiligen Binder-Kontexts signierten Zertifikate. Beim Import kann das Zertifikat mit dem jeweiligen öffentlichen Schlüssel überprüft werden. Importierte Aussagen unterscheiden sich von lokalen Aussagen darin, dass diese über das Schlüsselwort *says* und den öffentlichen Schlüssel des Exporteurs qualifiziert werden (siehe Abb. 5.5).

```
rsa:3:c1ebab5d says
      employee( john_smith, bigco).
```

Abb. 5.5: Beispiel für eine signierte, importierte Aussage, Quelle: [DeT02]

Die zentrale Idee von Binder besteht darin, über den Austausch von Fakten und Regeln über einen sicheren Kommunikationsweg mit signierten Zertifikaten Vertrauen in die Aussagen anderer Komponenten zu gewinnen um so auch Zugriffsentscheidungen in dezentral organisierten Umgebungen treffen zu können. Insbesondere kann die Herleitung für eine Zugriffserlaubnis auch entfernt vorgenommen werden und diese Herleitung als Beweis in einem Zertifikat exportiert werden, so dass die lokale Komponente nur noch den Beweis überprüfen muss. Für die Binder-Sprache wurde gezeigt, dass es in polynomieller Zeit möglich, eine Zugriffsanfrage zu entscheiden. Außerdem ist die Sprache monoton, d. h. durch Hinzufügen weiterer Aussagen werden hergeleitete Fakten nicht ungültig.

Binder ist eine sehr einfache Berechtigungsbeschreibungssprache, welche auf die Vertrauensmodellierung zwischen Komponenten in verteilten Umgebungen ausgelegt ist. Aus Regeln und Fakten können Zugriffsrechte hergeleitet werden. Eine Zugriffsentscheidung kann in polynomieller Zeit gefällt werden. Die in obigem Artikel angegebenen Regeln beziehen sich ausschließlich auf Rechte mit positiver Polarität. Von daher entfällt eine Konfliktauflösungsstrategie. Obwohl nicht explizit erwähnt, ergibt sich, dass Binder nach einer geschlossenen Policy agiert: Fakten, die nicht hergeleitet werden können, sind Zugriffsverbote. Auf Modellierung globaler Invarianten, wie z. B. der Trennung von Zuständigkeiten oder dem Selbstmanagement wird nicht eingegangen.

5.2.5 Extended Access Control Markup Language

Es gibt einige regelbasierte Beschreibungssprachen, welche auf XML basieren. Genannt seien hier XACL, die „XML Access Control Language“, [HK00] und sein Nachfolger XACML [GM+03], die „eXtended Access Control Markup Language“, ein OASIS Standard, welcher momentan in Version 2.0 verfügbar ist. Während XACL nur zur Darstellung von Zugriffspolicies auf XML-Dokumente geeignet ist, lassen sich über XACML beliebige Ressourcen schützen. Im Weiteren wird deshalb kurz auf XACML eingegangen.

XACML wurde entwickelt, um Berechtigungen, wie sie in einer verteilten Anwendung bzw. einer Anwendungslandschaft existieren, in einem gemeinsamen Format darzustellen. Die Ausdrucksmächtigkeit der Spezifikationssprache muss also so mächtig sein, dass Berechtigungen für beliebige Anwendungen mit ihren verschiedenen Anforderungen abgebildet werden können. Dadurch ist die Struktur von XACML-Dokumenten relativ komplex.

Oberstes Element einer XACML-Spezifikation ist eine Policy. (Eine Policy selbst kann hierbei wieder aus einem Set von Policies aufgebaut sein.) Eine Policy besteht aus einzelnen Regeln, welche neben der Spezifikation von Subjekt und Aktion auch ein Resource-Element enthalten, welches das Objekt darstellt, auf das die Regel angewendet werden soll. Policies selbst, welche nichts anderes als eine Gruppierung von Regeln darstellen, enthalten nun ihrerseits ein Target-Element, welches ebenfalls die Objekte klassifiziert, für die die Policy gelten soll. Damit die Regeln innerhalb einer Policy angewendet werden können, muss sich die Menge der Resource-Elemente einer Regel mit der Menge der Target-Elemente in der Policy überschneiden. XACML eignet sich besonders für die Rechteverwaltung von hierarchischen Objekten, da hier auch XPath-Ausdrücke [BB+06] verwendet werden können, um Ressourcen und Targets zu beschreiben.

Um zu einer Zugriffsentscheidung zu gelangen, werden die Regeln der Policy nach einem in der Policy festgesetzten Algorithmus ausgewertet. Denn in einem Regelsatz kann es immer vorkommen, dass mehrere Regeln auf eine Anfrage zutreffen, diese aber zu widersprüchlichen Ergebnissen kommen. Vordefinierte Algorithmen zur Regelauswertung sind „deny-overrides“, "permit-overrides" und "only-one-applicable". "deny-overrides" besagt, dass Regeln negativer Polarität die Regeln positiver Polarität überschreiben. „permit-overrides“ besagt, dass Regeln positiver Polarität Vorrang vor Regeln negativer Polarität haben. „only-one-applicable“ ist ein Auswertungsalgorithmus, bei dem postuliert wird, dass auf jede Zugriffsanfrage immer nur genau eine Regel zutreffen darf. Eigene Algorithmen können definiert werden. Ist eine Policy aus einem ganzen Satz an Policies zusammengesetzt, so kann durch einen weiteren Algorithmus spezifiziert werden, wie die endgültige Zugriffsentscheidung aus den Zugriffsentscheidungen der einzelnen Policies generiert wird.

Zu dem oben skizzierten Standard-XACML gibt es noch einige Erweiterungen, die Profile, mit denen spezielle Anforderungen abgebildet werden können. Genannt sei hier das RBAC-Profil. Dieses erlaubt es, auch rollenbasierte Berechtigung (vgl. Kap. 2.4.2) darzustellen. Obwohl eine Berechti-

gungsspezifikation immer auf einem Berechtigungsmodell beruht, ist es zur Erstellung einer XACML-Policy nicht notwendig, auf einem vorgeschriebenen Berechtigungsmodell aufzusetzen, da die Spezifikation der Berechtigungspolicy ja auf Basis von Regeln und Auswertungsalgorithmen für diese Regeln erfolgt. Berechtigungen auf hierarchische Ressourcen werden über das Hierarchie-Profil abgedeckt.

Eine Besonderheit von XACML ist die Möglichkeit, Verpflichtungen zu modellieren. Hierdurch ist es möglich, weitergehende Sicherheitsanforderungen, wie z.B. Protokollierung nach jedem Zugriffsversuch, zu modellieren.

Das Format von XACML basiert auf XML. Von daher ist XACML auf die Maschinenlesbarkeit ausgelegt und hat durch die vielen notwendigen Namespace-Deklarationen und -Verweise eine für alle XML-Formate typische aufgeblähte Struktur, die ohne entsprechende Visualisierungs- und Editierungswerkzeuge kaum noch menschenlesbar ist.

XACML-Dokumente werden weder beim Subjekt noch beim Objekt, sondern an einer zentralen (oder mehreren) Stelle(n) abgelegt. Von daher ist besondere Sorgfalt bei der Abbildung der in der XACML-Policy referenzierten Subjekte, Objekte und Aktionen auf tatsächliche, in der Anwendung vorkommende Subjekte, Objekte und Aktionen, zu legen.

5.2.6 Auswertung und Diskussion der Policysprachen

Das Ziel dieser Arbeit ist es, ein Zugriffskontrollframework zu entwickeln, welches die Unabhängigkeit von Berechtigungsspezifikation und Berechtigungsdurchsetzung gewährleistet, so dass die Berechtigungspolicy der Anwendung konfigurierbar ist. Die Verwendung einer Zugriffskontrollliste eignet sich hier nicht, da zum einen die tatsächlichen Berechtigungen in der Anwendung verborgen und nicht explizit zugreifbar sind. Zum anderen enthalten Zugriffskontrolllisten nur Einträge für Subjekte und Aktionen. Die Definition von Nebenbedingungen ist dort nicht vorgesehen. Diese müssten an anderer Stelle im Programmcode ausprogrammiert werden. Von daher ist die Verwendung von Zugriffskontrolllisten für die Realisierung einer Berechtigungsspezifikation betrieblicher Anwendungen ungünstig.

Die Verwendung regelbasierter, deklarativer Spezifikationssprachen hingegen ist geeignet. Zum einen ermöglichen die Regeln eine zentrale Sicht auf alle definierten Berechtigungen innerhalb der Anwendung. Zum anderen sind die deklarativ spezifizierten Regeln leicht und ohne Veränderung der Anwendungslogik der betrieblichen Anwendung änderbar.

Bezug nehmend auf die Anforderungen an die Ausdrucksmächtigkeit modellierbarer Berechtigungen und auf die Anforderungen an die Autorisierung unterstützter Geschäftsprozesse (vgl. Kap. 4.3.1) werden unterschiedliche Anforderungen durch die verschiedenen, oben beschriebenen Sprachen mehr oder minder elegant oder gar nicht gelöst. Die Anforderungen an die Ausdrucksmächtigkeit der Berechtigungen werden über die Sprachelemente, mit denen anwendungsspezifische Regeln definiert werden, abgebildet. Geschäftsprozessabhängigkeiten können durch Herleitung sich implizierender Regeln oder durch Metapolicies berücksichtigt werden.

Obwohl ASL einige sehr gute Elemente enthält, wie z. B. die Herleitung impliziter Regeln, reicht die Ausdrucksmächtigkeit nicht aus, um allen Anforderungen gerecht zu werden. Insbesondere können Nebenbedingungen nur in einem sehr eingeschränkten Maße modelliert werden. Die Modellierung komplexer Nebenbedingungen auf Attributen von Geschäftsobjekten ist hier nicht möglich. Ebenso ist die Modellierung der benutzerbestimmbaren Zugriffskontrolle nicht vorgesehen. Obwohl es natürlich möglich ist, DAC-Strategien durch die rollenbasierte Zugriffskontrolle abzudecken (siehe Ausführungen in 2.3.2), so führt dies zu einer unnötigen Komplexität an Rollen und damit zu einer unnötigen Explosion notwendiger Regeln. Über die Metapolicies der ASL-Sprache können zwar Fehler in der Spezifikation gefunden werden, es ist aber nicht möglich, allgemeine sich implizierende Berechtigungen, wie etwa, dass ein Schreibrecht ein Leserecht impliziert, abzubilden. Insgesamt lässt sich festhalten, dass die Sprache ASL den hier gestellten Anforderungen nicht genügt.

Die Ausdrucksmächtigkeit von Ponder eignet sich, um fast alle gewünschten Anforderungen abzubilden. Komplexe Nebenbedingungen können durch OCL-Ausdrücke modelliert werden. In Ponder ist vorgesehen, dass die modellierten Aktionen Methodenaufrufe auf Objekten sind. Geschachtelte Methoden, d. h. Methoden, die als Aktionen in einer Policy auftauchen, rufen andere Aktionen auf, werden hierbei nicht berücksichtigt. Es ist zwar nicht explizit erwähnt, aber es wird wohl davon ausgegan-

gen, dass solche geschachtelten Aufrufe in der zu schützenden Anwendung nicht vorkommen. Ansonsten können nämlich Regeln definiert werden, die dazu führen, dass kein Subjekt eine bestimmte Aktion ausführen darf, da es nicht genügend Rechte besitzt, die anderen von dieser Aktion abhängigen Aktionen durchzuführen. Die Skalierbarkeit von Ponder ist durch das Konzept von Templatepolicies gegeben, da dadurch Berechtigungsdefinition wieder verwendet werden können. Konfliktauflösungsstrategien und Regelauswertungsalgorithmen sind allerdings nicht Teil der Ponder-Spezifikation.

Die Binder-Sprache von Microsoft verfolgt ein etwas anderes Ziel als das alleinige Deklarieren von Zugriffsrechten. Vor allem soll hier Vertrauen zu Aussagen, die andere Komponenten über Subjekte machen, dadurch erhöht werden, dass diese Aussagen verschlüsselt übergeben werden. Auf die Beschreibung von der Modellierung komplexer Berechtigungen wurde in dem vorliegenden Artikel verzichtet. Sicherlich ist eine solche Modellierung möglich, da Binder eine Erweiterung von Datalog darstellt. Mit Datalog ist es möglich, jegliche Art von Regeln darzustellen. Auf Metapolicies oder die Beschreibung von Workflowabhängigkeiten wurde in dem vorliegenden Artikel ebenfalls nicht eingegangen.

XACML deckt die gestellten Anforderungen an die Ausdrucksmächtigkeit der Berechtigungen ab. Sie wurde für verteilte Anwendungen konzipiert und skaliert daher für große Anwendungen. Eine Berechtigungshierarchie kann über den Auswertungsalgorithmus forciert werden. Für die Modellierung einer Vererbungshierarchie und andere Erweiterungen, wie etwa die Modellierung von RBAC, gibt es in XACML jeweils ein eigenes Profil. XACML enthält neben dem Regelauswertungsalgorithmus keine Elemente, um Metapolicies zu beschreiben. Insbesondere können keine allgemeinen Integritätsbedingungen, wie etwa Workflowabhängigkeiten, definiert werden. Es wären aber Strategien denkbar, mit denen die Abbildung sich implizierender Rechte möglich ist. Beispielsweise können die einzelnen Regeln jeweils nicht nur eine, sondern mehrere Aktionen enthalten. So können zumindest implizit Abhängigkeiten zwischen den Aktionen dargestellt werden.

Eine der Stärken deklarativer, regelbasierter Beschreibungssprachen, wie sie zurzeit verwendet werden, ist gleichzeitig eine ihrer Schwächen. Die spezifizierten Zugriffsregeln werden entkoppelt von der eigentlichen Anwendung erstellt. Diese liegen dann z. B. in einem von der Anwendung unabhängigen Dokument (wie etwa bei XACML) oder einer unabhängigen Komponente vor (wie etwa bei der Quasar Authorization, siehe 2.4.2). Die Subjekte, Objekte und Aktionen, die in den Zugriffsregeln spezifiziert werden, müssen jetzt also auf Subjekte, Objekte und Aktionen innerhalb der Anwendung abgebildet werden und umgekehrt. Werden die Zugriffsregeln unabhängig erstellt, so muss man auf Objektidentifier oder ähnliche Referenztechniken zurückgreifen, um die jeweiligen Subjekte, Objekte und Aktionen von Spezifikation und Anwendung miteinander zu verknüpfen. Hierbei ist eine Typüberprüfung im Spezifikationsdokument oder einer Spezifikationskomponente natürlich nicht möglich, da Objektidentifier nur Zeichen bzw. Strings darstellen. Des Weiteren kann nicht gewährleistet werden, dass verwendete Objektidentifier tatsächlich in der Anwendung existieren. Regeln, die sich auf Anwendungsobjekte beziehen, die zwischenzeitlich gelöscht wurden, sollten automatisch erkannt und gelöscht werden können. Sonst kommt es zu einem Wildwuchs von Regeln, die nicht mehr verwendet werden, aber dennoch vorhanden sind. Im Folgenden wird daher ein Ansatz vorgestellt, in dem definierte Zugriffsregeln direkt mit der Anwendung bzw. mit dem Datenmodell der Anwendung verknüpft werden, die Spezifikation aber trotzdem regelbasiert erfolgt.

5.3 Regelbasierte Spezifikation von Berechtigungen mit PathExpressions

Im Folgenden wird die speziell für betriebliche Anwendungen entwickelte Policysprache PathExpressions vorgestellt. Durch sie werden die in Kapitel 4.2.4 aufgestellten Anforderungen an die Ausdrucksmächtigkeit der Berechtigungen abgedeckt. Zunächst wird in 5.3.1 die Modellierung der Zugriffsprinzipien als Subjekt-Objekt-Teil von Berechtigungen vorgestellt. Kapitel 5.3.2 beschäftigt sich dann mit dem dritten Teil der Berechtigungen, den Aktionen. Die PathExpressions wurden als Proof of Concept in einem Framework für betriebliche Anwendungen, dem infoAsset Broker, Version 2.4, realisiert. In 5.3.3 wird die Implementierung kurz umrissen. Das Kapitel schließt mit einer Auswertung der Ausdrucksmächtigkeit der PathExpressions.

5.3.1 Modell der Zugriffsprinzipien als PathExpressions

Aus der Analyse bestehender betrieblicher Anwendungen ging hervor, dass Berechtigungen auf Assets mit Hilfe von Zugriffsprinzipien beschrieben werden können (siehe 4.2.2). Zugriffsprinzipien lassen sich als Ausdrücke über Pfaden im Datenmodell, den Pfadausdrücken, oder *PathExpressions*, darstellen. PathExpressions sind in [LM05] beschrieben. PathExpressions sind Regeln, die zu Policies gebündelt werden können, wobei, abweichend von üblichen Definitionen, die Aktionen jeweils der Policy (und nicht der Regel) zugeordnet sind. Eine PathExpression beschreibt eine Abbildung, die eine Menge von Objektinstanzen eines Typs auf eine Menge von Objektinstanzen eines (möglicherweise anderen) Typs abbildet. Objektinstanzen sind hier die in der betrachteten Anwendung verwalteten Geschäftsobjekte. Die Auswertung einer PathExpression ist die ADF des Zugriffskontrollframeworks. Die einzig notwendige Anfrage der ADF ist die Frage, ob eine bestimmte Objektinstanz in der Ergebnismenge der Anfrage enthalten ist. Bei dieser Objektinstanz handelt es sich meistens um das Objekt, welches das Subjekt, also den aktuellen Sitzungsbenutzer darstellt. Die ADF ist also eine primitive Anfragesprache auf Elementen des Datenmodells. Die hier notwendige Anfragesprache kann als Teilmenge der Object Query Language (OQL) [ASL89, ODMG98], einer objektorientierten, deklarativen Anfragesprache, die auf der SQL-Syntax basiert, verstanden werden, die lediglich eine Existenz-Query benötigt.

Die Sprachelemente der PathExpressions sind an die Sprachelemente von OQL angelehnt. Da hier nur ein Subset einer Anfragesprache benötigt wird, wurde hier eine kompaktere, an OCL [OMG06] angelehnte Notation für die Pfadausdrücke der Regeln gewählt.

Das Datenmodell der PathExpressions (ohne Aktionen) ist in Abb. 5.6 illustriert und eine schriftliche Notation in Form einer Grammatik (ebenfalls ohne Aktionen) ist in Abb. 5.7 gegeben. Aktionen werden im Folgabschnitt 5.3.2 getrennt behandelt.

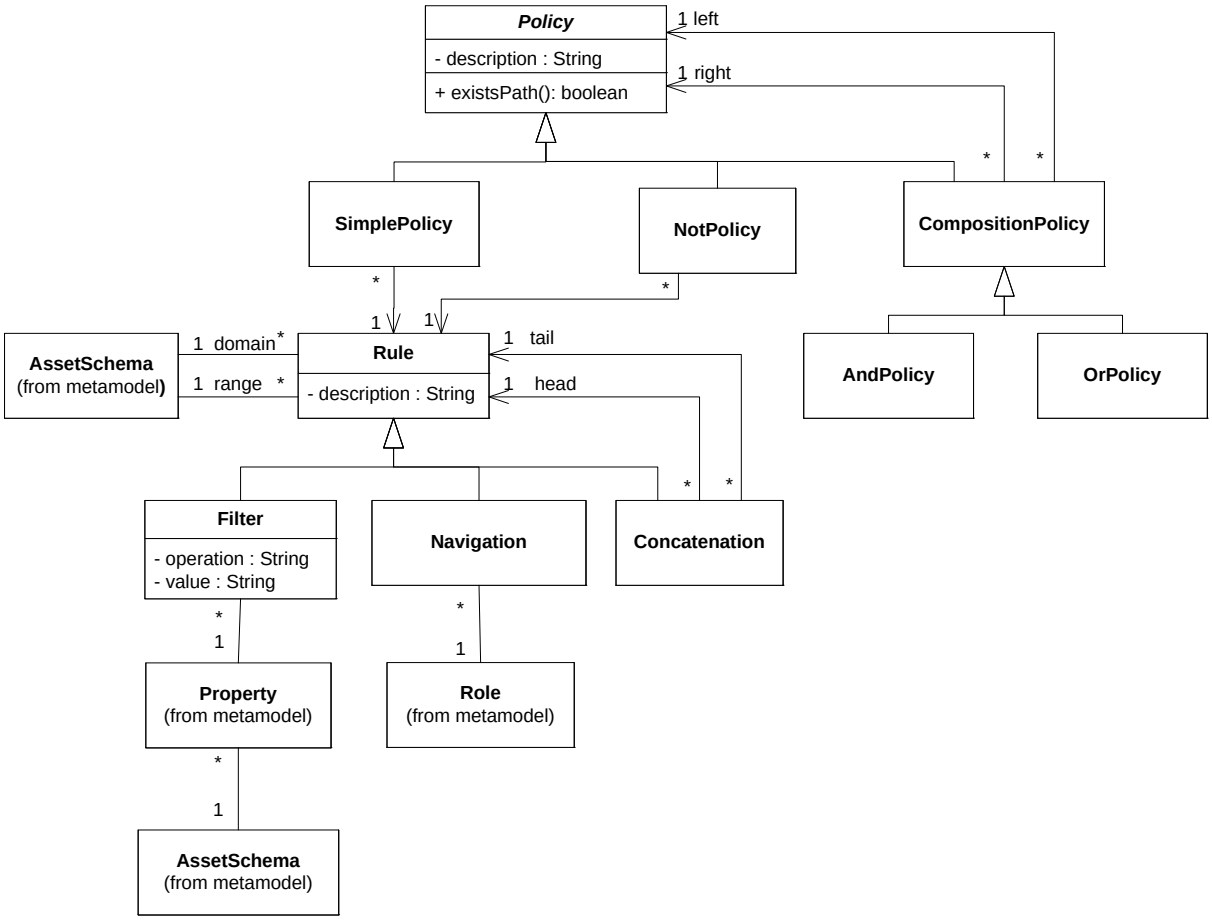


Abb. 5.6: Modell der PathExpressions

```
PolicySet :
   ContextDef ( Policy )*

ContextDef :
   context type :

Policy :
   SimplePolicy | NotPolicy | CompositionPolicy

SimplePolicy :
   PolicyDeclaration PolicyDefinition

NotPolicy :
   PolicyDeclaration PolicyIsEmptyDefinition

PolicyDeclaration :
   let simple_policy_name :

PolicyDefinition :
   boolean = rule_name → includes ( caller ) |
   boolean = rule_name → includes ( object )

PolicyIsEmptyDefinition :
   boolean = rule_name → isEmpty( )

ComplexPolicy :
   let policy_name : boolean = OrPolicy ( AND OrPolicy )*

OrPolicy :
   simple_policy_name OR simple_policy_name |
   simple_policy_name

RuleSet :
   ContextDef ( Rule )*

Rule :
   Filter | Navigation | Concatenation

Filter :
   let rule_name :
      type = self.select(self.attribute_name, operation, value)

Navigation :
   let rule_name : Set(type) = self.role ( .role )*

Concatenation :
   let rule_name : Set(type) = rule_name . rule_name ( . rule_name )*


type : any asset schema type

rule_name : Identifier

policy_name : Identifier

simple_policy_name : policy_name <type> | policy_name

attribute_name : Identifier (name of an attribute of an asset schema)

role : Identifier (name of a role of an asset schema)

object : caller | Identifier (name of asset object)

operation : < | > | <= | >= | == | != | …

value : any value of primitive data type or String
```

Abb. 5.7: Grammatik zur Notation der PathExpressions

Ein Zugriffsprinzip, also eine *Rule*, setzt sich aus primitiven Bausteinen zusammen, die nach bestimmten Regeln miteinander kombiniert werden können. Die primitiven Bausteine sind der *Filter* und die *Navigation*. Diese können mit Hilfe des Konkatenationselements (*Concatenation*) miteinander zu komplexeren Regeln verknüpft werden. Ein Zugriffsprinzip definiert einen Pfad innerhalb des Datenmodells, welcher bei einem oder mehreren Assets eines spezifischen AssetSchemas startet, – dies ist der Urbildbereich der Abbildung, der im Modell durch die Rolle Domain angegeben wird –, und bei Assets eines AssetsSchemas endet, die ebenfalls durch ein AssetSchema repräsentiert werden, – dies ist der Bildbereich, der durch die Rolle *Range* angegeben wird. Jede Regel wird dem AssetSchema zugeordnet, welches die Domain der Regel darstellt. Dieses AssetSchema ist der Kontext der Regel. Da zu jedem AssetSchema mehrere Regeln gehören können, werden diese in einem *RuleSet* zusammengefasst:

```
RuleSet :
   ContextDef ( Rule )*
ContextDef :
   context type :
Rule :
   Filter | Navigation | Concatenation
```

Ein Filter ist eine PathExpression, bei der sowohl der Definitionsbereich als auch der Bildbereich durch Objekte desselben Typs, also desselben AssetSchemas, gebildet werden. Mit Hilfe von Filtern können Mengen von Objekten, welche aus der Auswertung von PathExpressions entstanden sind, zu einem Subset eingeschränkt werden, indem der Wertebereich von Properties eines AssetSchemas begrenzt wird. Filteroperationen können „gleich“, „kleiner“, „kleiner gleich“, „größer“, „größer gleich“, „ungleich“ o. ä. sein. Handelt es sich bei der Eingabe zu einem Filter um ein Set von Objekten, so wird der Filteroperator auf jedes einzelne Objekt im Set angewendet. Ist nur ein einzelnes Objekt im Definitionsbereich so enthält die Ergebnismenge entweder genau dieses Element oder sie ist leer, je nachdem, ob der Filter das Objekt passieren lässt oder nicht. In Anlehnung an OCL [OMG06] kann der Filter wie folgt notiert werden:

```
context type :
   let filter_name : type =
      self.select(self.attribute_name, operation, value)
```

Jeder Filter bezieht sich auf einen Kontext (*context*). Dies ist das Domain-AssetSchema und gehört zur oben angegebenen *ContextDef*. Der Filter wird durch einen Namen eindeutig referenziert. Er selektiert, ausgehend von dem Objekt *self*, sich selbst, falls die Operation *operation* auf dem Attribut, welches durch *attribute_name* beschrieben wird, zum Wert *value* evaluiert. Graphisch im UML-Datenmodell kann der Filter als Kreis um das entsprechende Attribut mit Angabe der Operation und des Vergleichswerts notiert werden.

Ein Beispiel für einen Filter wäre, dass aus einer Menge von User-Objekten diejenigen herausgefiltert werden, deren Attribut *isAdmitted true* ist (siehe Abb. 5.8).

```
context User :
   let AdmittedFilter : User = self.select(self.isAdmitted, =, true)
```

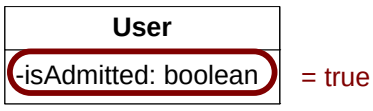


Abb. 5.8: Graphische Notation des AdmittedFilter im UML-Modell

Eine Navigation beschreibt, welche Objekte einer Klasse im Datenmodell über eine Assoziation mit welchen anderen Objekten verbunden sind. Die Navigationsrichtung wird hierbei über eine Rolle beschrieben. Mit einer Rolle wird hier, wie in UML üblich, ein Assoziationsende bezeichnet. Das Ziel-AssetSchema einer Navigation ist dasjenige, dessen Rolle angegeben ist. Rollen sind innerhalb eines UML-Diagramms eindeutig [OMG05]. Fehlt der Rollenname, so kann auch der Name des Ziel-AssetSchemas angegeben werden.

Hat das Assoziationsende, welches durch die Rolle beschrieben wird, die Kardinalität 1, so ist genau ein Objekt, genauer ein Asset, in der Ergebnismenge. Bei der Kardinalität 0..1 könnte das Ergebnis leer sein. Bei Kardinalitäten *, 1...* oder anderen Mengenangaben ist das Ergebnis eine Menge (engl.: collection) von Assets. Nach einem einzelnen Navigationsschritt handelt es sich hierbei um ein Set, da alle Zielobjekte verschieden sind. Dient das Ergebnis eines Navigationsschritts als Eingabe für einen neuen Navigationsschritt, handelt es sich also um eine einfache Ausprägung einer Concatenation, so kann das Ergebnis eine Menge von Sets sein, falls die Eingabe bereits ein Set von Objekten war. In diesem Fall werden alle Sets in einer einzigen Menge zusammengefasst, in der alle Duplikate entfernt werden, so dass wiederum nur ein Set übrig bleibt.

Eine Navigation wird notiert, indem zunächst ihr Kontext, also der Domain, angegeben wird. Eine Navigation hat einen eindeutigen Namen und beschreibt die Anwendung einer Rolle *role_name* auf Objekte des Typs *self*. Das Ergebnis ist ein Set, da es sich auch um eine mengenwertige Assoziation handeln kann. Ist auch die Definition von anonymen Navigationen erlaubt, so kann das Ergebnis der Anwendung der ersten Navigation auf eine weitere Navigation angewendet werden, die wiederum auf eine Navigation angewendet wird, und so fort. Die beliebig oft wiederholte Anwendung ist hier mit ( )* gekennzeichnet. Dies dient lediglich dazu, Schreibarbeit bei der Definition der Navigationen zu sparen und die Übersichtlichkeit bei der Definition der Regeln zu erhöhen.

```
context type :
   let simple_policy_name : Set(type) = self.role_name ( . role_name )*
```

Ein Beispiel für eine Navigation wäre, ausgehend von einem Dokument-Objekt über die Rolle *creator* seinen Eigentümer ausfindig zu machen. Graphisch kann die Navigation durch einen Pfeil notiert werden, der ausgehend von der Domain über die in der Navigation angegebene Assoziation zum Range-AssetSchema zeigt (siehe Abb. 5.9).

```
context Document :
   let Creator : Set(User) = self.creator
```

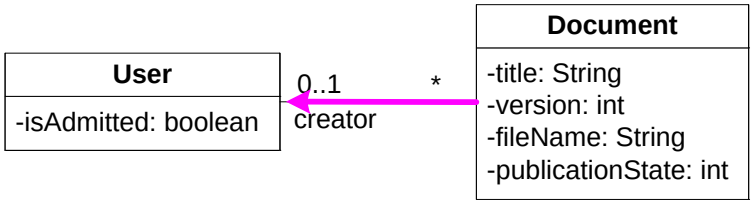


Abb. 5.9: Graphische Notation der Creator-Navigation

Eine Verbindung (*Concatenation*) verbindet zwei Pfade miteinander, um einen einzelnen längeren Pfad zu bilden. Hierbei muss das AssetSchema, welches das Ende des Kopfpfades (*Rollenname: head*), also den Bildbereich des Kopfpfades, bildet, mit dem AssetSchema übereinstimmen, welches den Beginn des Endpfades (*Rollenname: tail*), also den Definitionsbereich des Endpfades, bildet. Die Notation für eine Verbindung startet wieder mit dem Kontext, in dem die Verbindung definiert wird. Sie verknüpft mehrere Regeln miteinander. Falls die Regeln zum selben Kontext gehören oder global eindeutige Namen haben, können die einfachen Regelnamen verwendet werden. Ansonsten müssen die Regelnamen durch Voranstellen ihres jeweiligen Kontexts qualifiziert werden. Wir gehen im Weiteren davon aus, dass alle Regelnamen global, also anwendungsweit, eindeutig sind. Im

Datenmodell besteht eine Verbindung aus genau zwei Regeln. Ist die Verwendung anonymer Regeln, d. h. von Regeln ohne Namen, erlaubt, so kann eine Verbindung, abweichend vom Datenmodell, auch durch die Angabe einer ganzen Kette von Regeln gegeben sein, hier notiert durch ( )*.

```
context type :
  let concat_name : Set(type) = rule_name . rule_name (.rule_name )*
```

Ein Beispiel für einen konkatenierten Pfad wäre, ausgehend von dem AssetSchema *Group*, zunächst die Gruppe zu finden, deren *name*-Attribut „*Administrator*“ ist, um dann über die Assoziationen *Membership* und *User* die Mitglieder dieser Administratorengruppe zu identifizieren. Um diese Regel zu definieren, benötigen wir zunächst die zwei Hilfsregeln *AdminFilter* und *Member*, aus denen dann die Verbindung *MemberAdmin* definiert werden kann. Graphisch wird die Concatenation einfach notiert, indem die einzelnen Bestandteile der Regel durch dieselbe Farbe dargestellt werden und die gerichteten Pfade zu einem durchgehenden Pfeil zusammengefasst werden (siehe Abb. 5.10).

```
context Group:
   let AdminFilter : Group =
      self.select(self.name, =, „Administrator“)
   let Member : Set(User) = self.Membership.User
   let MemberAdmin : Set(User) = AdminFilter.Member
```

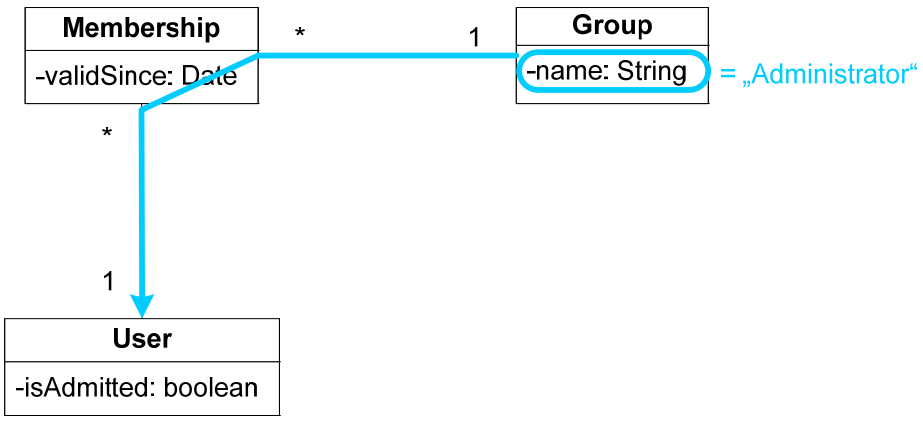


Abb. 5.10: Graphische Notation der MemberAdmin-Concatenation

Policies setzen sich aus Regeln zusammen. Genau wie bei der Definition der Regeln wird auch jede Policy einem Kontext zugeordnet. Der Kontext einer Policy ist dasjenige AssetSchema, bei dem alle assetbezogenen Zugriffsprinzipien, die die Policy ausmachen, starten. Da es für jedes AssetSchema mehrere Zugriffspolicies geben kann, werden diese zu einem *PolicySet* zusammengefasst:

```
PolicySet :
   ContextDef ( Policy )*
Policy :
   SimplePolicy | NotPolicy | CompositionPolicy
```

Eine einfache Policy (*SimplePolicy*) wird in diesem Modell durch eine einzige Regel beschrieben. Eine Policy wird zu „wahr“ oder zu „falsch“ evaluiert. Um die Policy evaluieren zu können, müssen ein konkretes Geschäftsobjekt aus der Menge des Domain-AssetSchemas als Startobjekt des Pfades, der durch die zur Policy gehörenden Regel beschrieben wird, und ein konkretes Geschäftsobjekt aus der Menge des Range-AssetSchemas als Endobjekt gegeben sein. Nun kann eine Berechtigung subjektabhängig oder subjektunabhängig sein. Von daher kann eine Policy, die ja ein Zugriffsprinzip

und damit den Subjekt-Objekt-Teil der Berechtigung darstellt, auch subjektabhängig oder subjektunabhängig sein. Ist die Policy subjektabhängig, so evaluiert sie zu „wahr“, falls der Sitzungsbenutzer als Endobjekt, auf den die Policy angewendet wird, in der Menge der Subjekte ist, die durch die Regel der Policy beschrieben werden. Ist die Policy subjektunabhängig, so beschreibt die Regel eine Menge von Geschäftsobjekten eines beliebigen Typs. In diesem Fall evaluiert die Policy zu wahr, wenn die Menge der Geschäftsobjekte, die durch die Regel der Policy beschrieben werden, ein vor Auswertung der Policy anzugebendes konkretes Geschäftsobjekt als Endobjekt enthält. In Anlehnung an OCL kann das bool’sche Ergebnis einer Policyauswertung im Fall von subjektabhängigen Policies mit →*includes(caller)* und im Fall von subjektunabhängigen aber assetbezogenen Policies mit →*includes(object)* angegeben werden. Das Objekt *caller* ist hierbei das Objekt, welches den Sitzungsbenutzer darstellt; das Objekt *object* wäre ein zum Auswertungszeitpunkt der Policy bekanntes Endobjekt. Die Notation einer Policy ist wie folgt:

```
context type :
   let policy_name<type> :
      rule_name → includes (caller) | rule_name → includes (object)
```

Ein Beispiel für eine Policy im Sinne der PathExpressions wäre, dass ein Benutzer eine (noch nicht näher bestimmte) Aktion auf einem Dokument ausführen darf, falls er der Eigentümer ist.

```
context Document :
    let DocOwner<Document> :  Creator → includes (caller)
```

In manchen Fällen ist es notwendig, eine Art negativer Policy definieren zu können. Die Policy beschreibt dann die Abwesenheit eines Pfades. Dieses wird durch die *NotPolicy* ausgedrückt. Die entsprechende Notation ist hier in Anlehnung an OCL →*isEmpty()*.

```
context type :
   let policy_name<type> : rule_name → isEmpty()
```

Die Funktion *isEmpty( )* wird zu wahr evaluiert, falls der Pfad, auf den die Funktion angewendet wird, nicht existiert, ansonsten evaluiert die Funktion zu falsch. Z. B. könnte ein Zugriff auf ein Dokument erlaubt sein, falls es nicht gesperrt ist. In diesem Fall müsste, startend bei der Klasse *Document*, die Navigation entlang des *AssetLocks* ergeben, dass diese konkrete Assoziation nicht existiert, damit der Zugriff erlaubt ist. Graphisch kann die NotPolicy dargestellt werden, indem das Schlüsselwort *empty* an der Pfeilspitze der dargestellten Policy notiert wird (siehe Abb. 5.11).

```
context Document:
  let lock : Set(AssetLock) = self.AssetLock
  let unlockedDocument<Document> :  lock → isEmpty()
```

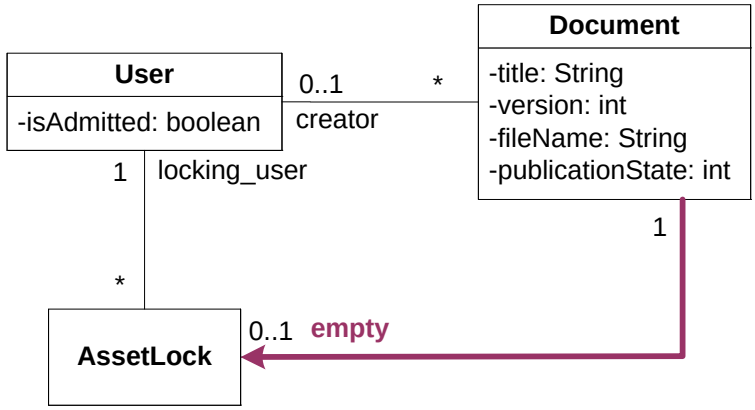


Abb. 5.11: Graphische Notation der unlockedDocument-Policy

Meistens reicht die Definition einer einfachen Policy nicht aus, um die Zugriffskontrollanforderungen abzubilden. Es ist daher möglich, die verschiedenen Policies miteinander zu kombinieren. Wie aus der Analyse sowohl der hier betrachteten Beispielanwendungen (vgl. Kap. 4.2) als auch durch die Betrachtung realer Anwendungen [Hup05] hervorgeht, ist die natürliche Verknüpfung von Policies die Oder-Verknüpfung, d. h. in Anwendungen kommt es am häufigsten vor, dass die Ausführung einer Aktion auf einem Objekt mehreren, auf verschiedene Arten klassifizierbaren Subjekten erlaubt ist. Eine Einschränkung der Subjektmenge, d. h. eine Und-Verknüpfung, kommt nur sehr selten vor. Ein Beispiel für eine Oder-Verknüpfung von Policies wäre z. B.: „Der Zugriff auf ein Dokument ist erlaubt, falls es sich beim Benutzer um den Eigentümer des Dokuments oder um ein Mitglied der Administratorengruppe handelt.“. Und-Verknüpfungen von subjektabhängigen Regeln, wie z. B.: „Der Zugriff auf ein Dokument ist allen Benutzern erlaubt, die sowohl Eigentümer des Dokuments als auch Mitglieder der Administratorengruppe sind.“, kommen in den untersuchten Anwendungen nicht vor und es wird davon ausgegangen, dass diese allgemein nur selten auftreten.

```
ComplexPolicy : policy_name : boolean = OrPolicy [ AND OrPolicy ]*
OrPolicy : SimplePolicy | SimplePolicy OR SimplePolicy
```

Komplexe Policies werden in einer Normalform notiert, in der die Oder-Verknüpfungen geklammert werden, so dass diese vor den Und-Verknüpfungen ausgewertet werden. Falls für eine komplexe Policy die Und-Bindung stärker sein soll als die Oder-Bindung, kann unter Anwendung des Distributivgesetzes die komplexe Policy in diese Normalform gebracht werden.

Im Folgenden ist ein Beispiel für eine Oder-Policy und die Definition ihrer einfachen Bestandteile gegeben:

```
context Group:
  let Admin : boolean  = MemberAdmin → includes (caller)

context Document:
  let DocOwner<Document> : boolean :  Creator → includes (caller)
  let DocumentAccessPolicy: boolean = Admin OR DocOwner<Document>
```

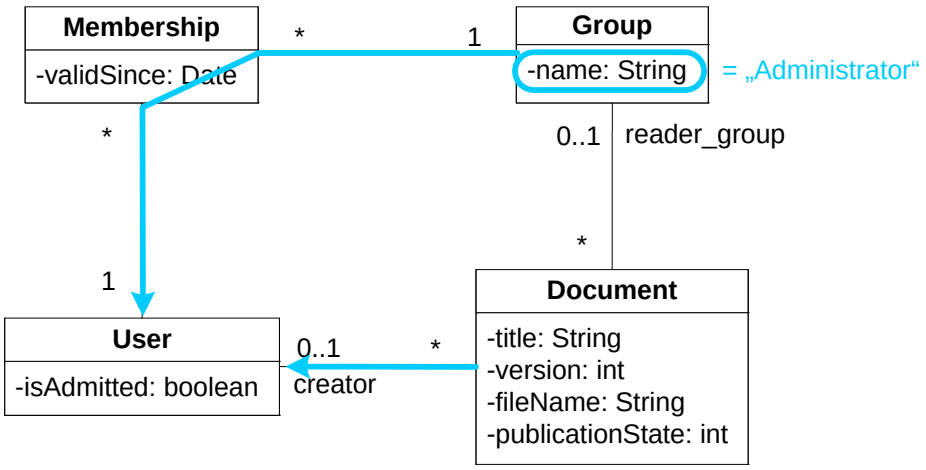


Abb. 5.12: Graphische Notation einer Oder-Policy

Graphisch kann eine Oder-Policy notiert werden, indem die einfachen Policies, aus denen sie aufgebaut ist, in derselben Farbe dargestellt werden (siehe Abb. 5.12).

Zum Abschluss sei eine der selten vorkommenden Policies aufgezeigt, die eine Und-Verknüpfung enthalten: „Der Zugriff auf ein Dokument sei erlaubt, falls der Benutzer Eigentümer des Dokuments oder Mitglied der Administratorengruppe ist, und zusätzlich das Dokument nicht gesperrt ist.“ (siehe Abb. 5.13).

```
context Document:
let DocumentAccessPolicy :
  boolean = (Admin OR DocOwner<Document>) AND
                                    unlockedDocument<Document>
```

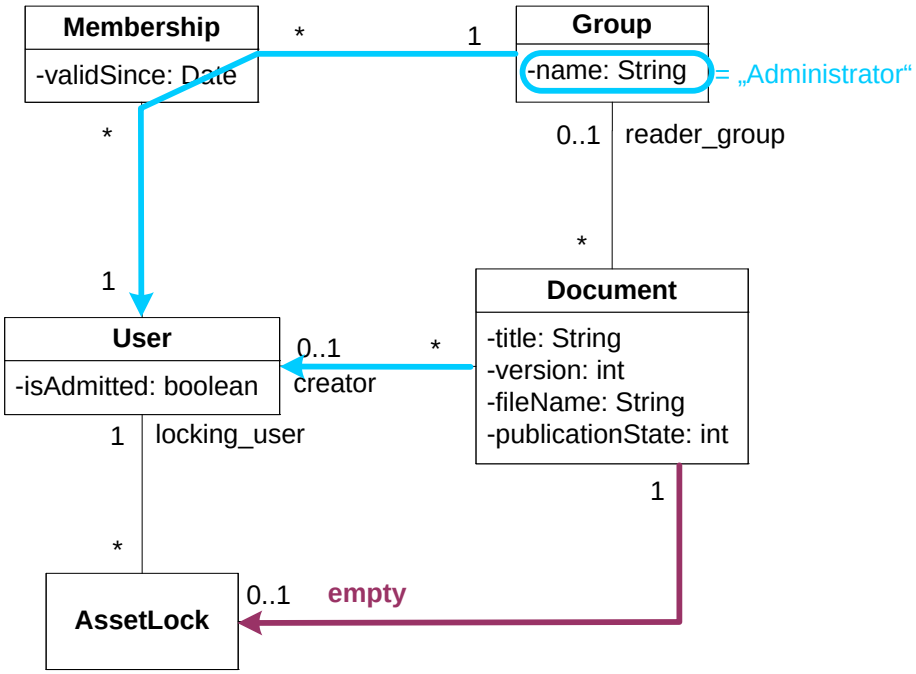


Abb. 5.13: Graphische Notation für eine komplexe Policy mit Und-Verknüpfung

Die graphische Notation lebt von der Vielfalt darstellbarer Farben, da jeder Oder-Policy eine eigene Farbe zugeordnet werden kann. Die Und-Verknüpfung zwischen den Policies wird also dadurch impliziert, dass die beiden Policies in unterschiedlichen Farben angegeben sind. Wenn die einzelnen Policies immer in der Normalform notiert werden, dann ist die graphische Darstellung immer eindeutig, da die Reihenfolge der einzelnen Policies, wie sie in der schriftlichen Notation gegeben ist, für das Auswertungsergebnis keine Rolle spielt. Die graphische Notation dient der einfachen und intuitiven Visualisierung.

5.3.2 Modellierung der Aktionen

Typische Basisaktionen, welche für die Modellierung von Berechtigungen benötigt werden, sind Create, Read, Update Delete und Query (vgl. 5.1.1 und 5.1.3). Im Datenmodell beziehen sich diese Basisaktionen im Fall von Assets auf das Anlegen (*Create Asset*), Lesen (*Read Asset*), Aktualisieren (*Update Asset*), Löschen (*Delete Asset*) und Suchen (Query Assets) derselben. Attribute von Geschäftsobjekten können im Allgemeinen nur gelesen und aktualisiert werden. Es stehen also die beiden Basisaktionen *Read Property* und *Update Property* für die Properties zur Verfügung. Weiterhin modelliert das Datenmodell explizit Assoziationen zwischen Geschäftsobjekten. Deshalb gibt es die drei weiteren Basisaktionen „Assoziation anlegen“ (*Create Association*), „Assoziation löschen“ (*Delete Association*) und „Assoziation navigieren“ (*Read Association*). Diese Basisaktionen korrespondieren direkt mit den in Kapitel 3.2.2, Tabelle 3.1, vorgestellten Basisfunktionalitäten auf den Assets. Eine Basisaktion entspricht also einer im Datenmodell der Anwendung zur Verfügung stehenden Methode. Der Abstraktionsgrad dieser Aktionen ist im Prinzip ungeeignet, um Berechtigungen zu definieren (siehe 5.1.3) allerdings gut geeignet für eine Berechtigungsbündelung mittels statischer Quellcodeanalyse (siehe Kapitel 6).

Eine Möglichkeit für die Definition von Aktionen sind die Geschäftsobjektmethoden. Jedes Geschäftsobjekt hat eine Methode zum Anlegen und zum Löschen desselben. Anders als die primitiven Basisaktionen Create Asset und Delete Asset enthalten diese Anwendungsfunktionen noch weitere Basisaktionen. Das Anlegen eines Geschäftsobjekts geht immer mit dem Anlegen von Muß-Assoziationen zu anderen, bereits bestehenden Geschäftsobjekten einher. Weiterhin werden beim Anlegen Attribute gesetzt. Ähnliches gilt für das Löschen, beim Löschen werden auch Assoziationen wieder entfernt, um die Integrität des Datenmodells zu wahren. Neben den Muß-Assoziationen gibt es die Kann-Assoziationen, die auch einzeln betrachtet als Aktionen für Berechtigungen herangezogen werden können, denn sie können unabhängig gesetzt und entfernt werden.

Leseaktionen gibt es für zwei verschiedene Abstraktionen des Objekts, welches gelesen wird. Zum einen handelt es sich um das Geschäftsobjekt, zum anderen um ein einzelnes Attribut eines Geschäftsobjekts. Es handelt sich hierbei also um eine gestufte Berechtigung. Die Aktion Lesen eines Geschäftsobjekts kann nun entweder bedeuten, dass die ID des Geschäftsobjekts gelesen werden darf, also dass die Existenz des Geschäftsobjekts bekannt sein darf, oder sie kann bedeuten, dass auch alle Daten, also alle Attribute des Geschäftsobjekts, gelesen werden dürfen. Hier impliziert das Leserecht eines Geschäftsobjekts, dass auch alle seine Attribute gelesen werden dürfen. Ein Leserecht auf Attributebene überschreibt hierbei das Leserecht auf Geschäftsobjektebene für dieses Attribut. Unter den Attributen eines Geschäftsobjekts werden hier explizit nur die Attribute primitiven Datentyps verstanden, denn die Attribute nicht-primitiven Datentyps sind Assoziationen, die auf andere Geschäftsobjekte verweisen, für die ein eigenes Leserecht existierten sollte. Die Geschäftsobjektmethoden, die ein Geschäftsobjekt zurückliefern oder ein Attribut auslesen, bestehen im Normalfall nur aus der einen primitiven Basisaktion Read Asset bzw. Read Property. Deshalb ist die Modellierung der Aktionen auf Geschäftsmethodenebene in diesem Fall gleichzusetzen mit der Modellierung der Aktionen auf der Ebene der primitiven Basisaktionen. Die Lese-Berechtigungen auf den Attributen sollten aus Konsistenzgründen so gewählt sein, dass sie die Leseberechtigung auf ein Geschäftsobjekt höchstens einschränken.

Für Update Property gilt Ahnliches wie für das Lesen. Die Geschäftsobjektmethoden, die ein Attribut setzen, bestehen im Normalfall nur aus der Basisaktion zum Aktualisieren dieses Attributs. Hier fallen die Modellierungsebenen Geschäftsobjektmethode und Basisaktion also wieder zusammen. Auch hier sollte zur Wahrung der Konsistenz das Schreibrecht auf die Attribute höchstens

eingeschränkter sein als das Leserecht auf einem Geschäftsobjekt. Denn um Daten eines Geschäftsobjekts aktualisieren zu können, ist es zumindest im rollenbasierten Berechtigungsmodell üblich, dieses auch lesen zu dürfen bzw. von seiner Existenz in Kenntnis gesetzt zu werden.

Das Modell für die Aktionen sieht nun in Analogie zur gerade geführten Diskussion vor, Aktionen zu Clustern aus primitiven Basisaktionen zu gruppieren. Hierbei kann es z. B. für das Anlegen eines Dokuments einen Cluster geben, welcher sich aus den Basisaktionen *Create Document* zum Anlegen des Dokuments selbst, *Create Document_User-Association* zum Anlegen der Eigentümerbeziehung zum aktuellen Sitzungsbenutzer, *Create Document_Directory-Association* zum Referenzieren auf das das Dokument beinhaltende Verzeichnis und einigen *Update Property*-Aktionen zum Initialisieren der Metadaten zusammensetzt. Das Anlegen eines Geschäftsobjekts und das Anlegen zugehöriger Muß-Assoziationen sowie das Setzen von Anfangswerten für diejenigen Attribute, die einen Wert haben müssen, der von null verschieden ist, sollte zur selben Policy gehören, damit sichergestellt ist, dass diese Aktionen auch zusammen innerhalb eines Dienstes ausgeführt werden dürfen. Das entsprechende Datenmodell für die modellierten Aktionen ist in Abb. 5.14 gegeben.

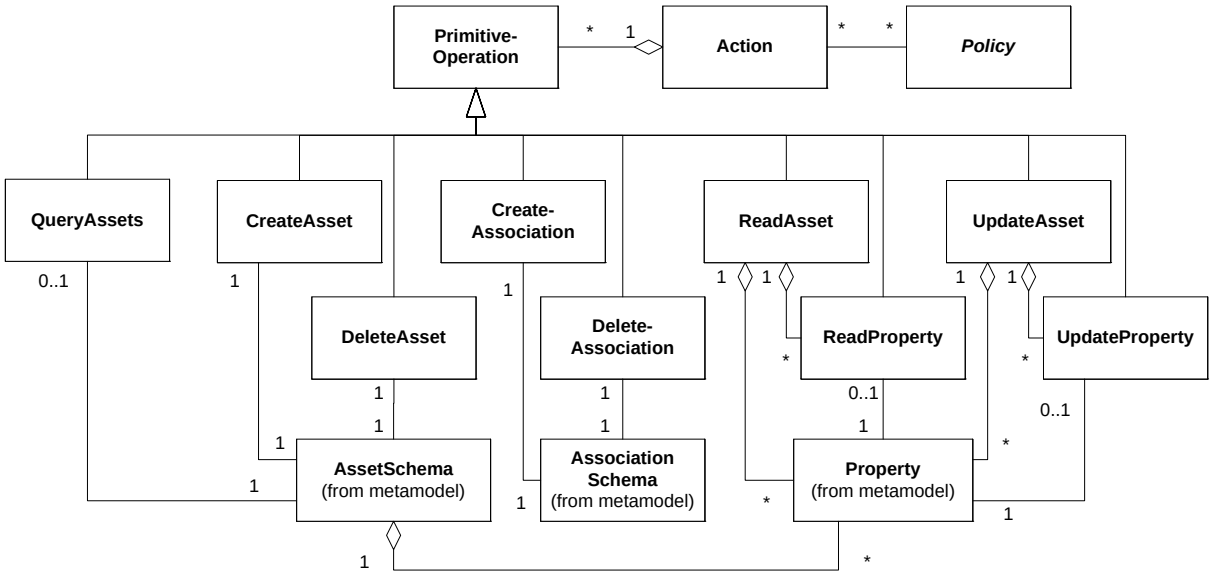


Abb. 5.14: Datenmodell der Aktionen

Später werden wir den Quellcode einer betrieblichen Anwendung statisch analysieren, um für jeden Dienst eine kumulierte Berechtigungsdurchsetzungsfunktion zu generieren (siehe Kapitel 6). Wird eine primitive Basisaktion im Quellcode gefunden, so wird ihr die Berechtigung zugewiesen, die der Aktion, zu der diese Basisaktion gehört, zugeordnet ist. Von daher muss die Aufteilung der primitiven Basisaktionen in Clustern überschneidungsfrei sein. Innerhalb eines Dienstes müssen nicht immer alle primitiven Basisaktionen eines Clusters vorkommen. Dies sei an einem Beispiel erläutert. Die Aktion „Dokument anlegen“ könnte z. B. die oben erläuterten primitiven Basisaktionen beinhalten. Die Aktion „Dokument löschen“ hingegen beinhaltet die primitiven Funktionen *Delete Document* zum Löschen des Dokuments selbst, *Delete Document_Directory-Association* zum Löschen der Beziehung zwischen dem Dokument und seinem Verzeichnis und *Delete Document_User-Association* zum Löschen der Eigentümerbeziehung. Gegeben sei jetzt ein Dienst „Dokument verschieben“, welcher ein bestehendes Dokument von seinem ursprünglichen Verzeichnis in ein neues Verzeichnis verschiebt. Die Implementierung dieses Dienstes wird die zwei Basisaktionen *Delete Document_Directory- Association* zum Löschen der ursprünglichen Beziehung zum Verzeichnis sowie *Create Document_Directory-Association* zum Erstellen einer Beziehung zu einem neuen Verzeichnis enthalten aber nicht die anderen primitiven Basisaktionen aus „Dokument anlegen“ und „Dokument löschen“. Es ist dennoch sinnvoll, die Berechtigung zum Verschieben des Dokuments aus den Berechtigungen „Dokument anlegen“ und „Dokument löschen“ zusammenzusetzen.

5.3.3 Implementierung und Regelauswertung der PathExpressions

Die gewählte Implementierung der Regeln und Policies modelliert diese als ineinander geschachtelte Klassen. Die Klassen werden lediglich als Datencontainer verwendet. Sie implementieren einfache, standardisierte *get( )*-Methoden, über die ihre Daten ausgelesen werden können. Dadurch kann eine introspektive Struktur der implementierten Regeln und Policies erreicht werden, denn die *get( )*-Methoden können statisch leicht analysiert werden. Sie enthalten keinen algorithmischen Code, sondern bestehen nur aus einem Return-Statement, in welchem statische Datenwerte oder neu konstruierte Rule- bzw. Policy-Klassen zurückgeliefert werden. Die in Abb. 5.15 gezeigten Klassen sind alle abstrakt. Die konkreten Implementierungen der Regeln müssen die kursiv dargestellten Methoden implementieren.

Jede einzelne Regel steht als Klasse zur Verfügung, so dass sie als Bestandteil anderer Regeln eingesetzt werden kann. Aus einer Regelklasse kann unter Nutzung der Anfragesprache Queries (vgl. Kap. 3.1.8) automatisch eine Datenbankanfrage generiert werden, indem die Filter in Select-Queries auf einer *Container*-Klasse und die Navigationen in Join-Queries über mehrere *Container*-Klassen übersetzt werden. Eine Select-Query wird in der Anfragesprache Queries über verschiedene *Query-Conditions* ausgedrückt, Queries über mehrere Container hinweg können über die QueryAnd realisiert werden. Aufgrund der einfachen Struktur der Übersetzung von *AssetSchemas* und *AssociationSchemas* auf *Container* bzw. Datenbanktabellen und -spalten, ist es auf stereotype Art und Weise möglich, die Datenbankanfragen mit Hilfe der Select- und Join-Queries zu erstellen. Bei der Übersetzung einer Regel in eine Datenbank-Query wird das Besuchermuster [GHJ+95] angewendet, welches durch die baumartige Regelstruktur navigiert, um die Query zu erstellen.

Genau wie die Regeln stehen auch die Policies als Klassen zur Verfügung. Auch sie sind abstrakt. Die konkreten Implementierungen der Policies müssen die *getRule( )*-Methode bzw. die *getLeft( )*- und *getRight( )*-Methode implementieren. Die Struktur der Policies ist in Abb. 5.16 dargestellt.

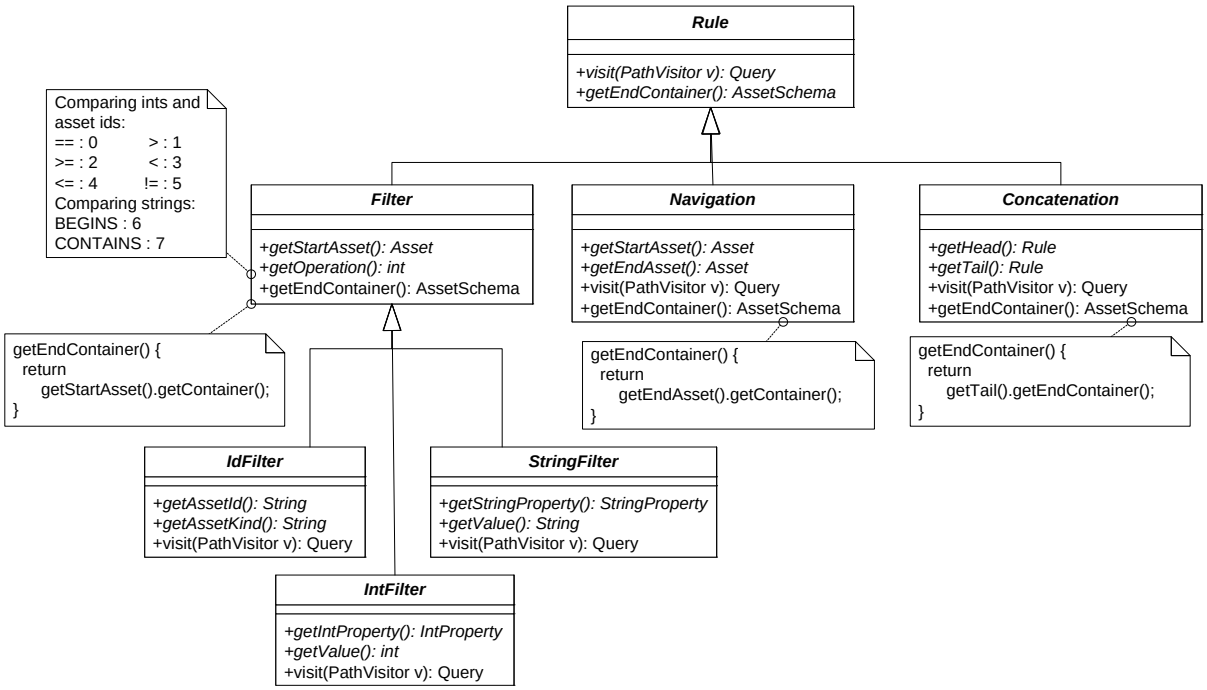


Abb. 5.15: Strukturelles Klassendiagramm der Rule-Implementierung

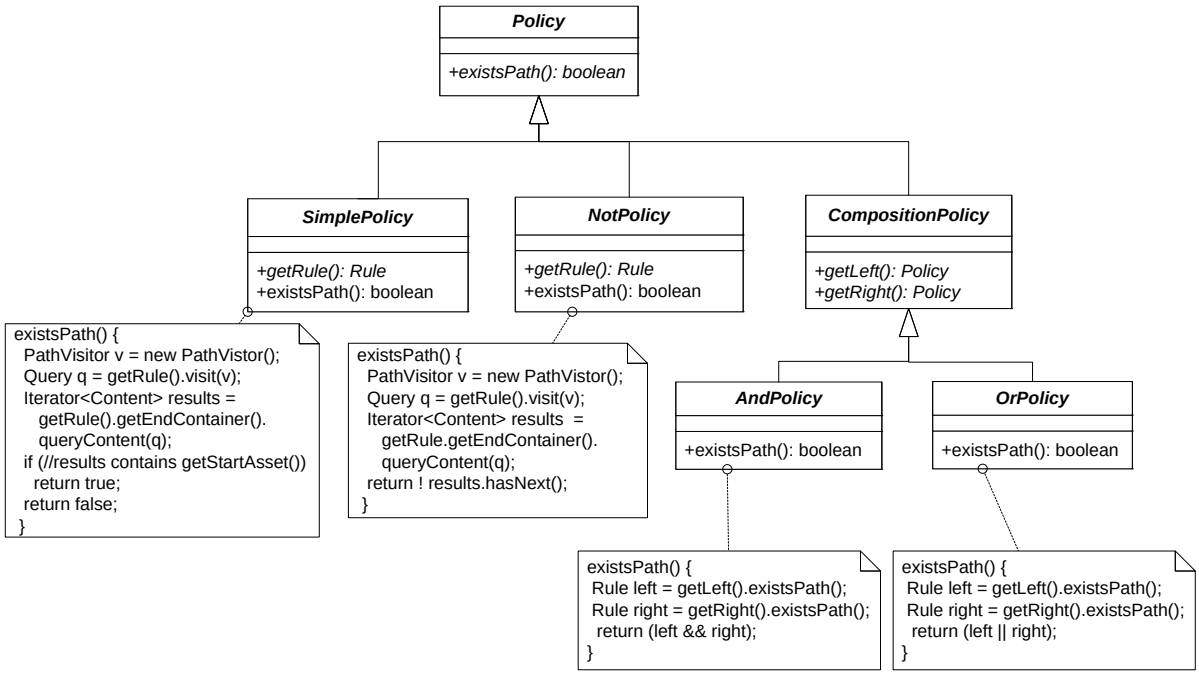


Abb. 5.16: Strukturelles Klassendiagramm der Policy-Implementierung

Zur Verdeutlichung soll hier ein konkretes Beispiel für die Implementierung der einfachen Policy: „Der Zugriff auf ein Dokument ist erlaubt, falls der Benutzer Mitglied der Administratorengruppe ist.“, aufgezeigt werden. Die notwendigen Regeln sind (siehe Seite 77):

```
context Group:
  let AdminFilter : Group =
                    self.select(self.name, =, „Administrator“)
  let Member : Set(User) = self.Membership.User
  let MemberAdmin : Set(User) = AdminFilter.Member
```

Und die einfache Policy ist gegeben durch (siehe Seite 79):

```
  let Admin :  MemberAdmin → includes (caller)
```

Für jede Regel und die Policy werden jetzt jeweils konkrete Klassen angelegt, die von den jeweiligen im Datenmodell der Regeln und Policies vorhandenen abstrakten Klassen erben (siehe Abb. 5.17). Der *AdminFilter* ist als Id-Filter gegeben, da die *Id* der Gruppe der Administratoren statisch bekannt ist. Die *Member-Navigation* enthält zwei Rollen, über die navigiert wird. Die Verbindung *MemberAdmin* schließlich schaltet die beiden Regeln hintereinander. Hierbei wird das Start-Asset auf *null* gesetzt, da der Regelpfad bei dem Asset startet, welches die Administratorklasse darstellt. Das Start-Asset wäre nur dann nicht *null*, wenn geprüft werden soll, ob es sich bei dem bestehenden Objekt vom Typ *Gruppe* um die Administratorengruppe handelt. Die Verbindungsregel und die einfache Policy führen neue abstrakte Methoden ein. Diese sollen das Asset, in diesem Fall den Sitzungsbenutzer, zurückgeben, welcher das Ende des Regelpfades beschreibt. Das konkrete Sitzungsbenutzer-Objekt kann erst zum spät möglichsten Zeitpunkt, dann nämlich, wenn die Policy instantiiert wird, eingesetzt werden.

```
public abstract class AdminFilter extends IdFilter {
  public String getAssetKind(){ return Groups.ASSET_KIND; }
  public String getAssetId(){ return Groups.ADMINISTRATOR; }
  public int getOperation() { return Filter.EQUALS; }
}
public abstract class Member extends Navigation {
  public Role[] getRoles() {
    return new Role[] {
      MEMBERSHIP_GROUP.getManyRole(),
      MEMBERSHIP_USER.getOneRole()};
}}
public abstract class MemberAdmin extends Concatenation {
  public Rule getHead() {
    return new AdminFilter() {
      public Asset getStartAsset() { return null; }
  };}
  public Rule getTail() {
    return new Member(){
      public boolean getStartAsset() { return null; }
      public boolean getEndAsset() { return getUser(); }
  };}
  public abstract User getUser();
}
public abstract class Admin_Policy extends SimplePolicy  {
  public Rule getRule() {
      return new MemberAdmin(){
        public boolean User getUser() {
            return getUserAsset(); }
  };};
  public abstract User getUserAsset();
}


SimplePolicy sp = new Admin_Policy() {
  public User getUserAsset() {
    return /*insert user object here*/}
}
boolean isPermitted = sp.existsPath();
```

Abb. 5.17: Quellcodebeispiel für die Admin-Policy

Während die Zugriffsprinzipien und Policies als Klassen innerhalb der Anwendung zur Verfügung gestellt werden, wurde für die Aktionen ein anderer Weg gewählt. Es ist nützlich, die Zugriffsprinzipien und Policies als Klassen in der Anwendung zu haben, da mit den Regel-Klassen Datenbank-Queries formuliert werden können. Die Policies stellen eine Methode zur Verfügung, um die Existenz eines Objekts in einer Menge von Objekten zu prüfen. Statt der reinen Existenz-Query kann die Policy auch eine Methode zur Verfügung stellen, um die Ergebnismenge der Query zurückzugeben. Dann können die Policy-Klassen auch als einfache Query-APIs für Queries fungieren, die innerhalb der Geschäftslogik der Anwendung notwendig sind. Für die Aktionen hingegen gibt es keinen zweiten Anwendungsfall. Von daher wurden diese als Annotationen in die jeweiligen Container-Klassen, die AssetSchemas, integriert, die in diesem Fall auch die Attribute und Assoziationen deklarieren.

Für jedes konkrete AssetSchema und für alle Aktionstypen können Policy-Annotationen definiert werden. Jede Policy-Annotation verweist auf ihre gültigen Zugriffsprinzipien. Die Zugriffsprinzipien liegen als Policy-Klassen vor. Gelten mehrere Policies für eine Aktion, so werden diese durch Kommas getrennt angegeben. Das Komma stellt eine Oder-Beziehung zwischen den einzelnen Policies dar. Die Oder-Beziehung ist die natürliche Verknüpfung von Policies, wie weiter oben bereits diskutiert. Die *@CreatePolicy*-Annotation ist für das Anlegen eines Geschäftsobjekts des Typs, welcher das jeweilige AssetSchema verwaltet, zuständig. Die @CreatePolicy schützt also die Ausführung der

AssetSchema.createAsset()-Methode (vgl. Kap. 3.2.2). Da das Datenmodell keine Unterscheidung zwischen Muß- und Kann-Assoziationen macht, erhält jede Assoziation, die zusammen mit dem Geschäftsobjekt angelegt werden muss, eine zusätzliche *@Create*-Annotation. Die @Create-Annotation besagt also, dass die dazugehörigen Methodenaufrufe von *Asset.createAssociation()* derselben Zugriffbeschränkung unterliegen wie das betreffende *AssetSchema.createAsset()* für das konkrete AssetSchema. Dieselben Regeln gelten für die *@DeletePolicy*-Annotation und die dazugehörigen *@Delete*-Annotationen. Die @DeletePolicy schützt die Methode *AssetSchema.-remove()* und @Delete beschränkt den Zugriff auf die dazugehörigen *AssetSchema.removeAssociation()*-Aufrufe. Für unabhängige Assoziationen gibt es *@CreateIndependent*- und *@DeleteIndependent*-Annotationen. Weiterhin stehen verschiedene *@ReadPolicy*-Annotationen und eine *@UpdatePolicy*-Annotation für das Lesen und Aktualisieren von Attributen zur Verfügung. Diese gelten für *Asset.getProperty()*- und *Asset.setProperty()*-Methodenaufrufe im Quellcode. Wie unter 5.1.3 diskutiert, ist es sinnvoll, Attribute in disjunkten Klassen zu gruppieren, die dann jeweils unterschiedliche Zugriffsberechtigungen aufweisen. Aus diesem Grund gibt es hier, statisch vorgesehen, drei Annotationen für die einzelnen Attribute, die diese disjunkten Klassen abbilden, *@Read*, *@ReadCommon* und *@ReadRestricted*. Die Annotation @Read ist hierbei nicht nur für den Attributzugriff, sondern auch für die Lesezugriffe, die durch die Methoden *AssetSchema.getAsset()* und *AssetSchema.getAssociated-Assets()* durchgeführt werden, zuständig. Die Möglichkeit des Suchens nach Assets kann speziell durch eine *@Query*-Annotation eingeschränkt werden. Das Lesen der Suchergebnisse wird zusätzlich über die vorhandene *@ReadPolicy*-Annotation eingeschränkt. Für das Aktualisieren von Attributen ist eine Einteilung in unterschiedliche Klassen in der Regel nicht notwendig. Das Modell kann aber sehr leicht hierfür erweitert werden. In Abb. 5.18 ist ein Beispiel für die Verwendung der Annotationen gegeben.

Jede Aktion kann eindeutig einer einzigen Regel zugeordnet werden. Von daher ist der Regelauswertungsalgorithmus einfach. Es gibt eine eindeutige Vorrangsregel: Policies auf Properties überschreiben Policies auf Assets. Diese sorgt dafür, dass der hierarchische Aufbau der Regeln korrekt abgebildet wird. Da keine negativen Berechtigungen zugelassen sind, können keine Konflikte bezüglich der Polarität auftreten. Der Auswertungsalgorithmus realisiert eine geschlossene Policy, d. h. Aktionen, deren Zugriffsregeln nicht explizit durch Annotationen im Quellcode definiert werden, sind Zugriffsverbote.

```
@ReadPolicy(DocumentOwner_Policy, Admin_Policy,
  DocumentReader_Policy)
@UpdatePolicy(DocumentOwner_Policy, Admin_Policy)
@CreatePolicy(DocumentWriter_Policy, Admin_Policy)
@DeletePolicy(…)
public class Documents extends AbstractAssetSchema {
  @Read @Update
  public static final IntProperty asPublicationYear …
  …
  @Create @Delete
  public static final OneToMany_AssociationSchema
     DOCUMENT_DIRECTORY …
  …
  @CreateIndependent(Admin_Policy)
  @DeleteIndependent(Admin_Policy)
  public static final OneToMany_AssociationSchema
     PARENTDIRECTORY_SUBDIRECTORIES …
  …
}
```

Abb. 5.18: Verwendung der Annotationen für die Definition von Aktionen

5.3.4 Auswertung der PathExpressions bezüglich Ausdrucksmächtigkeit

Die in diesem Kapitel vorgestellte Policysprache PathExpressions realisiert die Anforderungen an die Ausdrucksmächtigkeit modellierbarer Berechtigungen aus 4.2.4. Rollenbasierte und benutzerbestimmbare Zugriffskontrollelemente werden als Pfade auf dem Datenmodell der Anwendung modelliert. Dies ist möglich, da die Assoziationen zwischen den Klassen im objektorientierten Datenmodell auch Berechtigungsinformationen darstellen. So werden sowohl Eigentümerbeziehungen als auch Rollenbeziehungen, die eine Berechtigung zu einer bestimmten Ressource, also einer bestimmten Klasse im Datenmodell enthalten, als Assoziationen modelliert. Dadurch ist es möglich, diese Assoziationen zu navigieren und eine Anfrage nach der Existenz bestimmter Ausprägungen von Assoziationen zwischen konkreten Instanzen von Objekten zu stellen. Die PathExpressions ermöglichen es, Berechtigungen sowohl auf Geschäftsobjekten als auch auf den Attributen der Geschäftsobjekte zu beschreiben, denn die modellierbaren Aktionen beziehen sich sowohl auf Geschäftsobjekte als auch (für die Lese- und Aktualisierungsaktionen) auf Attribute. Durch die Verwendung von Filterelementen können komplexe Nebenbedingungen definiert werden. Später, bei der Realisierung der AEF werden wir Berechtigungen pro Dienst bündeln. Dies entspricht einer Und-Verknüpfung zwischen allen innerhalb eines Dienstes vorkommenden Berechtigungsauswertungen. Die Vererbung von Berechtigungen über Geschäftsobjekttypgrenzen hinweg kann explizit durch die Regeln einer PathExpression dargestellt werden. Vererbt z. B. ein Dokument das Recht, welches auf dem zu diesem Dokument gehörigen Verzeichnis definiert ist, so kann eine PathExpression definiert werden, die startend beim Dokument über eine Assoziation zum Verzeichnis navigiert und von dort aus den Berechtigungspfad des Verzeichnisses geht. Voraussetzung dafür ist, dass die entsprechende Assoziation, hier zwischen Dokument und Verzeichnis, im Datenmodell existiert. Dies ist in der Regel der Fall, da Vater-Kind-Beziehungen auch in anderen Kontexten der Geschäftslogik verwaltet werden müssen. Nicht möglich ist die Modellierung von Pfaden unbestimmter Länge, z. B. von Schleifen, deren Iterationsanzahl statisch nicht bekannt ist. Eine Berechtigung könnte z. B. lauten, dass der Zugriff auf ein Verzeichnis erlaubt ist, falls es in der gesamten Verzeichnishierarchie ein Verzeichnis gibt, welches diesen Zugriff erlaubt. Für solche Pfade ist es nicht möglich, eine einzelne Datenbankanfrage zu generieren. Es kann aber Datalog [EN02, GUW02] für solche Anfragen genutzt werden. Datalog ist eine deklarative Anfragesprache, die auch Rekursionen enthält. Eine Erweiterung der PathExpressions um Schleifen in Berechtigungsdefinition wäre möglich, obgleich dann die Auswertung der Berechtigung nicht mehr effizient durchführbar ist. Auf Schleifen wird im Ausblick eingegangen.

Die Vorteile der PathExpressions gegenüber anderen Policysprachen sind, dass die Möglichkeit besteht, die Policies zu visualisieren, was die Verständlichkeit modellierter Policies erhöht. Außerdem enthält die Policysprache nur die Elemente, die notwendig sind, um Berechtigungen zu definieren. Die Einfachheit der Policysprache sorgt ebenfalls für eine gute Verständlichkeit. Teile der Spezifikation, d. h. die Zugriffsprinzipien und ihre Bestandteile, können wieder verwendet werden, um neue Berechtigungen zu definieren. Außerdem ist die Policysprache direkt mit den Elementen des in der Anwendung realisierten Datenmodells verwoben, so dass die Konsistenz zwischen Policydefinition und tatsächlich vorhandenen Datenobjekten gegeben ist.

6 AEF – Durchsetzung der Zugriffskontrolle

Für die Durchsetzung der Zugriffskontrolle in einer betrieblichen Anwendung ist es zum einen notwendig, ein Framework bereitzustellen, welches die systematische Einbettung von Code zur Zugriffskontrollüberprüfung erlaubt, zum anderen muss die eingebettete Berechtigungsüberprüfung selbst genau die Berechtigungen abfragen, die für die Ausführung der Programmlogik der Anwendung notwendig sind. Von daher gliedert sich das folgende Kapitel in zwei Hauptbestandteile. In Kapitel 6.1 wird aufgezeigt, wie eine Architektur für die Zugriffskontrolle aussehen kann, die für die fünf verschiedenen Diensttypen einer betrieblichen Anwendung optimale Eingriffspunkte vorsieht, an denen die Berechtigungen überprüft werden können. Die hier vorgestellte Architektur für die Zugriffskontrolle enthält sowohl Eingriffspunkte für die Dienstbearbeitung als auch Eingriffspunkte, die während der Erstellung der Antwortnachricht genutzt werden. Die Geschäftslogik eines Dienstbearbeiters enthält unstrukturierten Quellcode. In Kapitel 6.2 wird dargestellt, wie mittels einer statischen Quellcodeanalyse berechtigungsrelevante Informationen aus der Geschäftslogik extrahiert und (semi)-automatisch die richtigen Berechtigungspolicies für einen Dienstbearbeiter generiert werden können. Das Kapitel schließt mit einer Auswertung der hier vorgestellten Architektur und des hier vorgestellten Algorithmus zur statischen Quellcodeanalyse.

6.1 Sicherheitsarchitektur für die Integration der Zugriffskontrolle in eine betriebliche Anwendung

Nachdem in Abschnitt 6.1.1 die Schwachpunkte bestehender Architekturen bezüglich der Einbettung der AEF diskutiert wurden, wird in Abschnitt 6.1.2 eine allgemeine Architektur für die Zugriffskontrolle vorgestellt. Diese wird in Abschnitt 6.1.3 für die *ServiceHandler*-Klassen und in Abschnitt 6.1.4 für die *Response* verfeinert.

6.1.1 Schwachpunkte bestehender Architekturen bezüglich der Umsetzung der AEF

Die Trennung der Zugriffskontrolle in eine ADF und eine AEF ist ein anerkanntes Design für die Implementierung derselben. Die ADF lässt sich sehr einfach in eine Komponente kapseln. Sie enthält entweder einfach die gesamte Berechtigungsspezifikation oder kann auf die gesamte Spezifikation zugreifen, um Zugriffsentscheidungen zu fällen. Ein systematischer Ansatz für die Realisierung einer AEF fehlt bislang noch. Da es sich bei der Durchsetzung der Zugriffskontrolle um einen Querschnittsaspekt handelt, kann diese nicht einfach in eine Komponente gekapselt werden. Die Aufrufe der AEF sind daher im Quellcode der Anwendung verstreut. Da die Geschäftslogik einer Anwendung in der Regel aus unstrukturiertem Quellcode besteht, ist es schwierig, einen allgemeinen Ansatz für die Durchsetzung der Zugriffskontrolle bereitzustellen.

Momentan existiert kein anerkanntes, systematisches Vorgehen, um die Durchsetzung der Zugriffskontrolle in eine Anwendung einzubetten. In der zurzeit marktführenden betrieblichen Standardsoftware SAP bspw. werden die AEF-Aufrufe vom Programmierer explizit in den Code geschrieben. Der Programmierer ist für die Vollständigkeit und Korrektheit der eingefügten Berechtigungsüberprüfungen zuständig. Ein Beispiel für eine Berechtigungsüberprüfung in der SAP-eigenen Programmiersprache ABAP findet sich in [Wil02].

Die JEE-Architektur (vgl. Kap. 4.4.3) bietet die Möglichkeit, Berechtigungen in einem Deploymentdeskriptor zu deklarieren, der den Zugriff auf Webseiten und auf Methoden von EJBs regelt. Allerdings ist die Ausdrucksmächtigkeit definierbarer Policies stark eingeschränkt. So können nur rein rollenbasierte Berechtigungen dargestellt werden. Die Durchsetzung eigentümerbezogener und objektabhängiger Berechtigungen liegt weiterhin in der Verantwortung des Entwicklers.

Für die Quasar-Referenzarchitektur gibt es eine ADF-Komponente, die Quasar Authorization (vgl. Kap. 2.4.2), allerdings müssen die Schnittstellen der Quasar Authorization wieder projektspezifisch von den Entwicklern aufgerufen werden. Wo und wie die Aufrufe der ADF-Komponente in eine bestehende Architektur einzubetten sind, bleibt auch hier dem Entwickler überlassen. Genauso kann die ADF-Komponente zwar sowohl positive als auch negative Berechtigungsentscheidungen liefern, wie mit diesen umzugehen ist, ist jedoch wieder für jedes Projekt, das die Quasar Authorization nutzt, neu zu entscheiden.

Durch Tests ist ebenfalls keine hundertprozentige Codeabdeckung erreichbar, so dass auch durch dieses Verfahren die Richtigkeit der Ergebnisse der Berechtigungsüberprüfung nicht allumfassend geprüft wird. Durch Tests kann die Abwesenheit von Fehlern nicht gezeigt werden: „Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72].

Wünschenswert wäre ein Ansatz, in dem die Berechtigungsüberprüfungen an vordefinierten Stellen der Anwendung systematisch eingebaut werden können, und mit dessen Hilfe die Korrektheit und Vollständigkeit der implementierten AEF-Aufrufe gewährleistet werden können. Der Entwickler soll hierbei auch von der Implementierung komplexer, identitätsabhängiger und assetbezogener Berechtigungsüberprüfungen entlastet werden. Der Anwendungscode sollte also sauber von dem Berechtigungscode getrennt werden. Dies erleichtert die Wartung und die Änderbarkeit sowohl von anwendungsspezifischem als auch von zugriffskontrollbezogenem Quellcode. Hierfür ist zum einen ein Framework notwendig, das entsprechende Verankerungspunkte (engl. Hooks) für AEF-Aufrufe vorsieht. Zum anderen muss die Korrektheit des eingebetteten Codes zur Zugriffsdurchsetzung automatisch hergeleitet werden können. Ein Framework für die Zugriffskontrolle betrieblicher Anwendungen wird in den folgenden Unterkapiteln beschrieben. Die automatische Quellcodegenerierung für Berechtigungsdurchsetzungsfunktionen wird in Kapitel 6.2 erläutert.

6.1.2 Allgemeine Sicherheitsarchitektur für die Zugriffskontrolle

Die zugriffsgeschützten Aktionen sind die Basisfunktionalitäten auf den konkreten Assets und Asset-Schemas der Anwendung. Diese werden zum einen in der *doBusinessLogic( )*-Methode der einzelnen ServiceHandler-Klassen und zum anderen während der Erstellung der Antwortnachricht aufgerufen (vgl. Abb. 3.11). Daraus folgt, dass in einer Architektur für die Zugriffskontrolle (vgl. Kap. 2.4.1) sowohl die ServiceHandler-Klassen als auch die Response auf die ADF zugreifen müssen (siehe Abb. 6.1), um eine Berechtigungsdurchsetzung, also die AEF, zu realisieren. Der Zugriff auf die ADF erfolgt in speziell für die Zugriffskontrolle vorgesehenen Methoden der ServiceHandler- und der Response-Klassen. Diese Methoden sind in Abb. 6.2 dargestellt. Eine Erläuterung der neu hinzugekommenen Methoden folgt in den anschließenden Unterkapiteln. Eine mögliche Realisierung der ADF wurde im vorangegangen Kapitel besprochen: die *Policy*- und *Rule*-Klassen der Path-Expressions bilden die Zugriffsregeln ab, eine Auswertungsmethode, die Methode *Policy.existsPath()*, wertet Berechtigungsanfragen aus.

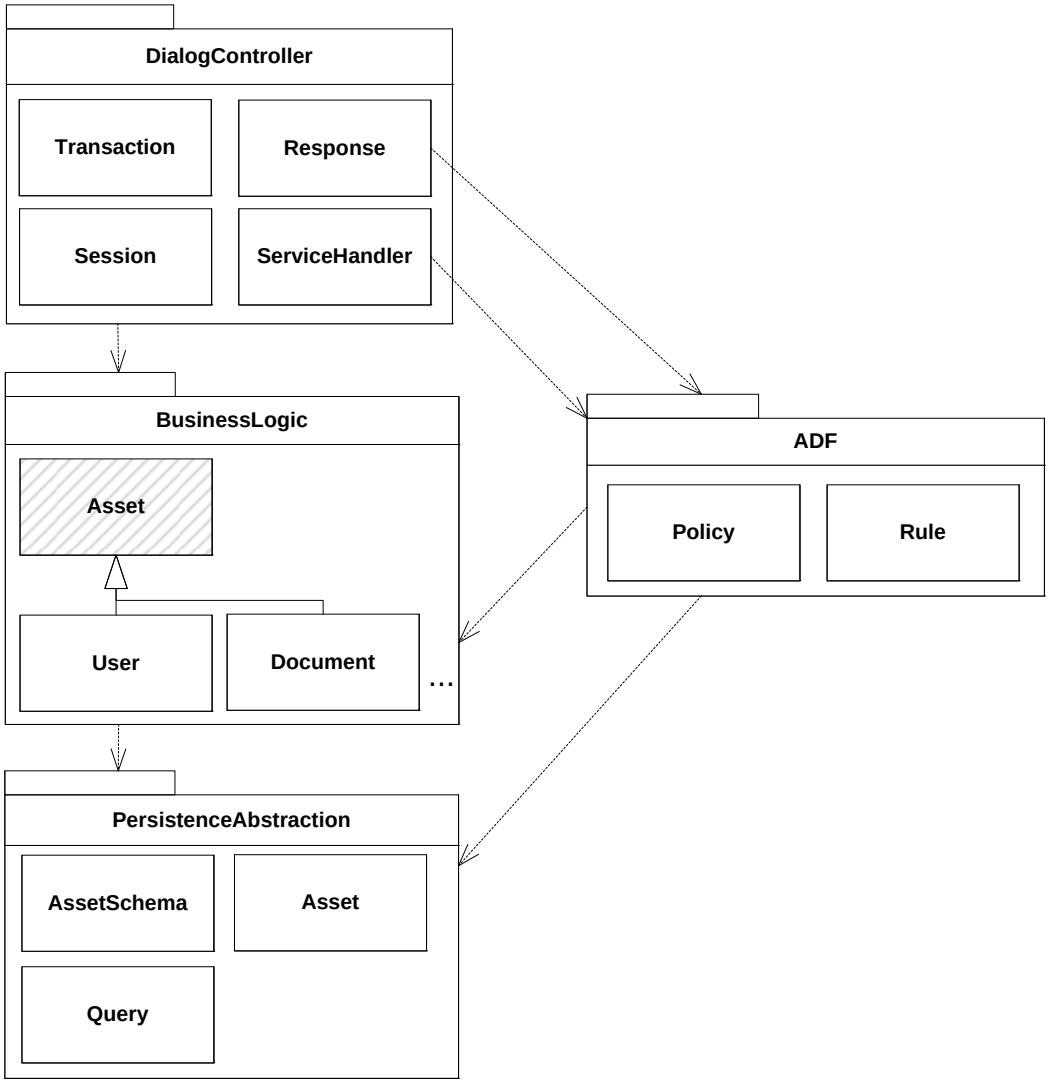


Abb. 6.1: Zugriff auf die ADF-Komponente

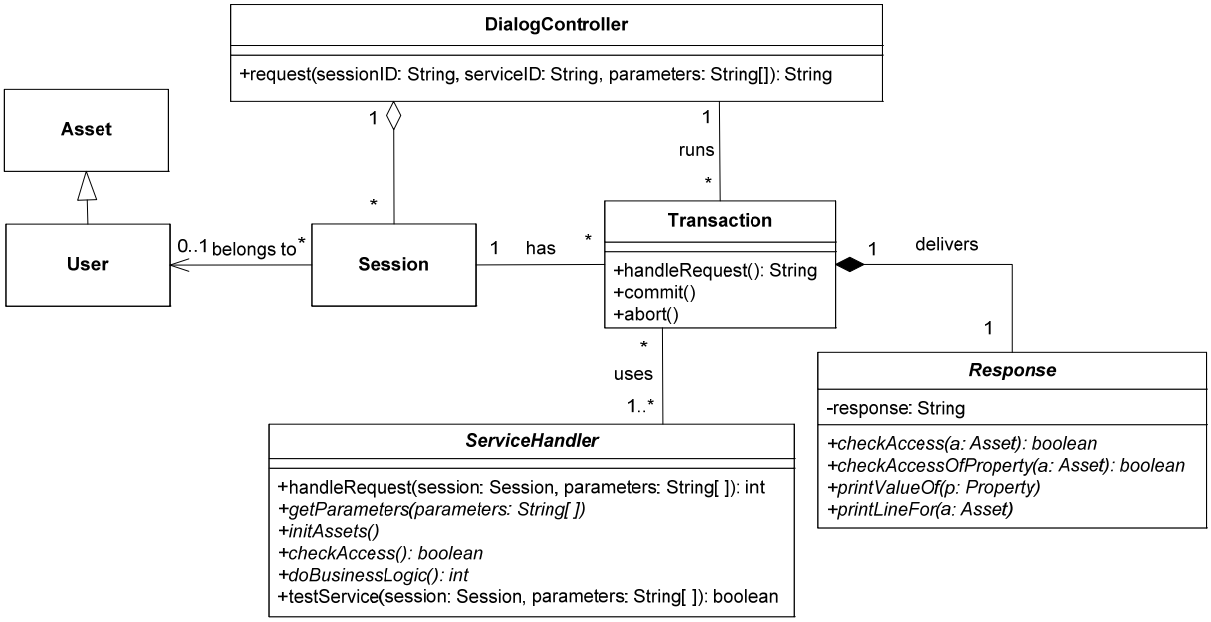


Abb. 6.2: Erweiterung von ServiceHandler und Response um Methoden für die Zugriffskontrolle

6.1.3 Effiziente und lückenlose Zugriffskontrolle für die ServiceHandler-Klassen

Es soll eine effiziente und lückenlose Zugriffskontrolle realisiert werden, welche möglichst wenige Berechtigungsüberprüfungen pro Dienstausführung vorsieht (vgl. Kap. 4.2 und 4.3). Von daher wird angestrebt, jeweils eine gebündelte Berechtigungsüberprüfung pro ServiceHandler-Klasse vorzusehen (siehe Abb. 6.3). Um dies zu realisieren, muss die ursprüngliche Ausführungsmethode des Dienstes, die *doBusinessLogic( )* in drei Teile gespalten werden. Zuerst werden alle Assets und AssetSchemas festgestellt, die während der Dienstausführung benötigt werden. Das Holen dieser Objekte geschieht in der *initAssets( )*-Methode. In einem zweiten Schritt kann dann die eigentliche Berechtigungsüberprüfung für alle in der *initAssets( )*-Methode bereitgestellten Assets und AssetSchemas und für alle während der Dienstausführung aufgerufenen Aktionen stattfinden. Dies ist Aufgabe der *checkAccess( )*-Methode. Die *checkAccess()*-Methode greift auf die ADF zu, indem sie die zu überprüfende Policy erstellt und auswertet. Policies liegen als abstrakte Klassen vor (vgl. Abb. 5.17). Sie benötigen meist noch das Objekt, welches den aktuellen Sitzungsbenutzer darstellt, sowie eventuell weitere Asset-Objekte, damit aus der Policy eine Datenbankanfrage generiert werden kann. Ist das Resultat der Berechtigungsüberprüfung für den aktuellen Sitzungsbenutzer positiv, so wird die eigentliche Dienstausführung, hier wieder mit *doBusinessLogic( )* bezeichnet, ausgeführt. Fällt das Resultat negativ aus, so kann dem Benutzer gleich eine Fehlermeldung zurückgesendet werden, ohne dass die Dienstausführung startet.

Die Realisierbarkeit der Aufspaltung der Geschäftslogikmethode in drei Teile soll im Folgenden für die verschiedenen Diensttypen diskutiert werden. Die fünf Diensttypen C, R, U, D und Q (vgl. Kap. 3.2.2) lassen sich in zwei Gruppen einordnen, die seiteneffektfreien Diensttypen R und Q, und die seiteneffektbehafteten Diensttypen C, U und D.

Zunächst werden die seiteneffektfreien Diensttypen diskutiert. Die R-Dienste für ein Asset sind einfach strukturiert. In Abb. 6.4 ist ein Beispiel eines R-Dienstes gegeben, welcher ein Asset vom Typ *Document* liest. R-Dienste für andere Assettypen sind analog strukturiert. Die ID des zu lesenden Assets wird aus einem mit der Anfrage mitgelieferten Parameter ausgelesen. In der *initAssets( )*-Methode wird das betreffende Asset aus der Datenbank ausgelesen. Der Zugriff auf das Asset erfolgt über das zum Assettyp gehörige AssetSchema, welches wiederum von einem AssetSchemaManager verwaltet wird. (vgl. Kap. 3, Abb. 3.9). Daraufhin kann in der *checkAccess( )*-Methode die Leseberechtigung auf dieses Asset überprüft werden. Im gegebenen Beispiel soll das Lesen eines Dokuments erlaubt sein, falls es sich beim aktuellen Sitzungsbenutzer um den Dokumenteigentümer oder um ein Mitglied der Administratorgruppe handelt. Die eigentliche Geschäftslogik des R-Dienstes ist leer, denn das Lesen der einzelnen Properties erfolgt erst während der Erstellung der Antwortnachricht. Der Zugriffsschutz für Lesezugriffe während der *Response*-Erstellung wird in Unterkapitel 6.1.5 behandelt. Hier wird zunächst nur auf die Dienstausführung innerhalb der *doBusinessLogic( )* eingegangen. Es ist notwendig, das zu lesende Asset bereits in der *initAssets( )*-Methode und nicht erst in der *doBusinessLogic( )* zu holen, wenn die Berechtigungen asset- oder eigentümerbezogen sind, d. h., wenn das konkrete Asset benötigt wird, um die Policy zu erstellen.

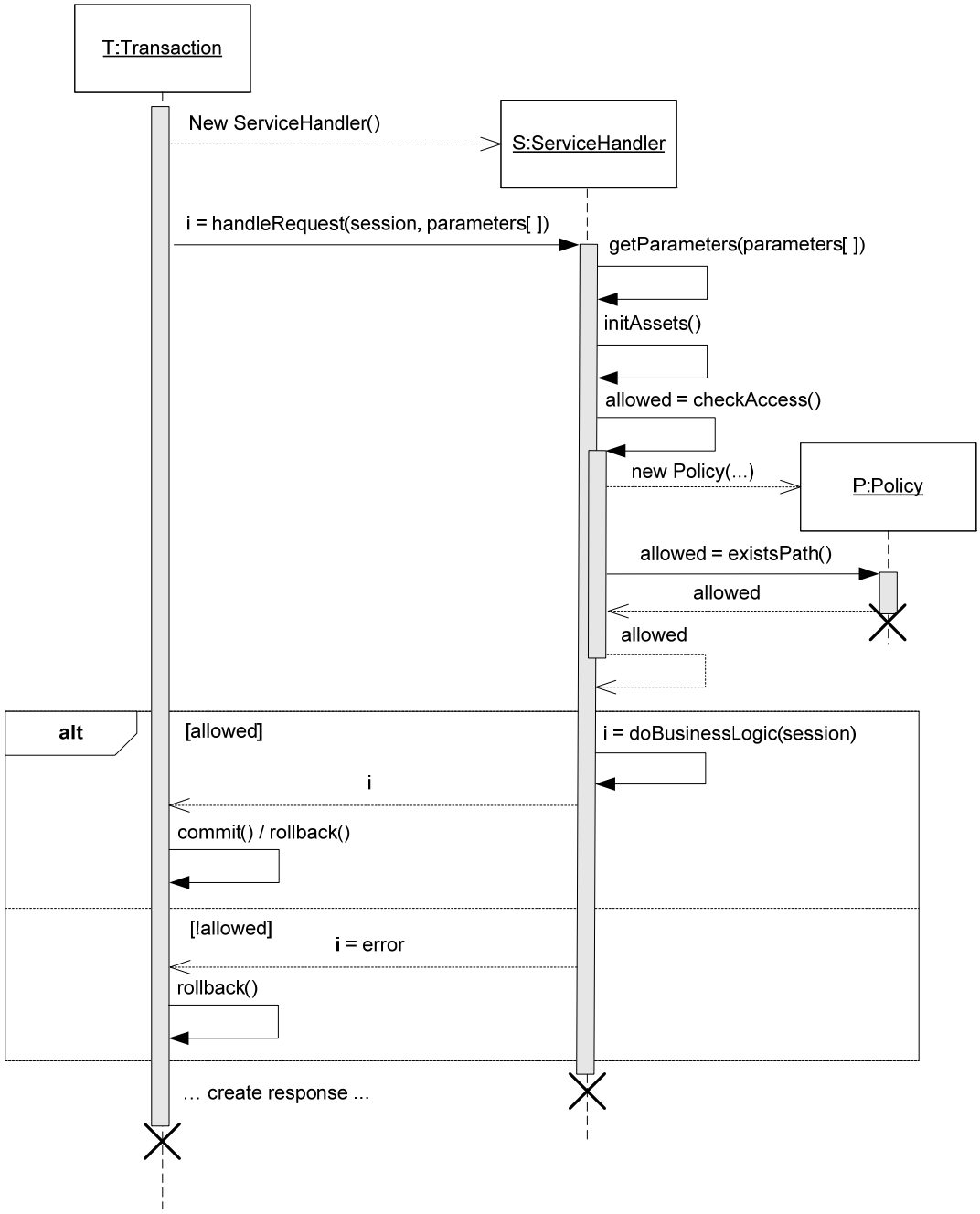


Abb. 6.3: Aufspaltung der doBusinessLogic()-Methode in drei Methoden

```
public class DocumentReadServiceHandler extends ServiceHandler {
  String documentId;
  Document document;
  void getParameters(String[] parameters) {
    documentId = //read id-parameter
  }
  void initAssets( ) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    Documents documents = asm.getDocuments();
    document = documents.getDocument(documentId);
  }
  boolean checkAccess( ) {
    Owner_Policy policy0 = new Owner_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
        public Document getDocumentAsset() {
            return document;
        }
    }
    boolean path0 = policy0.existsPath();
    Admin_Policy policy1 = new Admin_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
    }
    boolean path1 = policy1.existsPath();
    return (path0 || path1);
  }
  int doBusinessLogic(Session session) {  }
}
```

Abb. 6.4: Aufspaltung der Geschäftslogik eines R-Dienstes

Für den Diensttyp Q gibt es zwei Implementierungsmöglichkeiten. In der ersten Variante ist die Geschäftslogik ebenfalls leer und die eigentliche Suche wird erst während der Erstellung der Response durchgeführt. In der zweiten Variante wird die Suche in der Geschäftslogik durchgeführt und der Response die Suchergebnisse zur Verfügung gestellt. Die Policy, welche die Berechtigung für die Sucherlaubnis definiert, sollte unabhängig von spezifischen Assets sein und nur von dem AssetSchema abhängen, auf dem die Suche ausgeführt wird. Berechtigungen, die die Durchführung einer Suche einschränken, sollten also nur aus gruppenbezogenen Zugriffsprinzipien (vgl. Kap. 4.2.2) aufgebaut sein. Das betreffende AssetSchema kann sowohl in der *initAssets( )*- als auch in der *doBusiness-Logic( )*-Methode bereitgestellt werden, da es auf dem *AssetSchema* selbst keine Lesebeschränkung gibt, sondern nur auf den Assets, die es verwaltet. Für beide Implementierungsvarianten des Q-Dienstes ist die Aufspaltung der Geschäftslogik in drei Teile also einfach möglich. Ein Beispiel für einen Q-Dienst ist in Abbildung 6.5 gegeben. Der Suchdienst liefert alle Dokumente innerhalb eines Verzeichnisses *directory* zurück. Für die Ausführung dieses Dienstes werden zwei Berechtigungen benötigt. Die Berechtigung zum Lesen eines Verzeichnisses sei durch die *DirectoryAuthor_Policy*

gegeben, die allen Mitgliedern der Verzeichnisautorengruppe Lesezugriff auf das betreffende Verzeichnis gewährt. Das Durchführen der Suche sei für alle authentifizierten Benutzer erlaubt. Dass es sich bei dem Methodenaufruf *getDocumentsOfDirectory( )* um eine Suche handelt, wird durch eine statische Quellcodeanalyse herausgefunden, die in der Implementierung dieser Methode einen Aufruf der Basismethode *AssetSchema.queryAssets( )* findet.

```
public class DocumentsSearchServiceHandler extends ServiceHandler {
  Documents documents;
  String directoryId;
  Directory directory;
  Iterator result;
  void getParameters(String[] parameters) {
    directoryId = //get directory id
    … //read search parameters from parameters if necessary
  }
  void initAssets( ) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    documents = asm.getDocuments();
    Directories directories = asm.getDirectories();
    directory = directories.getDirectory(directoryId);
  }
  boolean checkAccess( ) {
    NotAnonymous_Policy policy0 = new NotAnonymous_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
    }
    boolean path0 = policy0.existsPath();
    DirectoryAuthor_Policy policy1 =
            new DirectoryAuthor_Policy(){
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
        public Directory getDirectoryAsset() {
            return directory;
        }
    }
    boolean path1 = policy1.existsPath();
    return (path0) && (path1);
  }
  int doBusinessLogic(Session session) {
    //search may also be placed in Response
    result = documents.getDocumentsOfDirectory(dir);
  }
}
```

Abb. 6.5: Aufspaltung der Geschäftslogik eines Q-Dienstes

Die seiteneffektbehafteten Dienste ändern den Zustand der in der betrieblichen Anwendung lebenden Assets. Der U-Dienst setzt und ändert Attribute sowie Assoziationen zu einem spezifischen Asset. Um welches Asset es sich dabei handelt, wird wie beim R-Dienst über einen Anfrageparameter bestimmt. Die Durchsetzung der Berechtigungsüberprüfung findet analog zum R-Dienst nach dem Lesen des zu aktualisierenden Assets statt. In der eigentlichen Geschäftslogik werden alle gewünschten Properties gesetzt. Da die Änderungsberechtigung zum Setzen der Properties häufig für alle Properties identisch ist, ist hier die Bündelung der Berechtigungsüberprüfung vor Ausführung der *doBusinessLogic( )* besonders effizient. In Abb. 6.6 ist ein Beispiel für einen U-Dienst gegeben, welcher den Publikationsstatus eines Dokuments ändert. Das Lesen eines Dokument-Assets und das Setzen von Properties für ein Dokument seien erlaubt, falls der Sitzungsbenutzer Eigentümer des Dokuments oder Mitglied der Administratorgruppe ist. Der U-Dienst kann auch Schleifen enthalten, wenn zusammen mit dem Aktualisieren eines Assets noch andere Assets, die über Assoziationen mit diesem Asset verbunden sind, aktualisiert werden sollen. Die Generierung von Berechtigungsüberprüfungen für solche Schleifen wird im Folgenden im Zusammenhang mit dem D-Dienst diskutiert.

Die Aufspaltung des D-Dienstes kann relativ komplex werden. Durch kaskadierendes Löschen kann es sein, dass im Zuge des Löschens eines Assets, welches wiederum durch einen Parameter in der Dienstanfrage identifiziert wird, eine ganze Reihe abhängiger Assets mitgelöscht werden. In Abb. 6.7 und 6.8 sind zwei Beispiele für unterschiedliche Lösungsmöglichkeiten für die Platzierung der Berechtigungsdurchsetzung für einen D-Dienst gegeben, der eine Gruppe löscht. Das Löschen der Gruppe impliziert das Löschen aller Mitgliedschaftsobjekte zu dieser Gruppe. Ist die Policy zum Löschen von Mitgliedschaftsobjekten gruppenbezogen und nicht eigentümer- oder assetbezogen, so lässt sich eine effiziente Berechtigungsüberprüfung vor Dienstausführung generieren, da diese unabhängig von den ausgelesenen Objekten, in diesem Beispiel den Mitgliedschaftsobjekten, ist. Dieser Fall ist in Abb. 6.7 dargestellt. Das Lesen von Gruppen und Mitgliedschaften sei allen authentifizierten Benutzern möglich. Das Löschen von Gruppen und Mitgliedschaften sei in diesem Beispiel den Administratoren vorbehalten.

Handelt es sich um eigentümer- oder assetbezogene Policies, so muss bei einer Berechtigungsüberprüfung zu Beginn des Dienstes für alle zu löschenden Assets einzeln überprüft werden, ob der Sitzungsbenutzer die Löschberechtigung besitzt. Dazu ist der Quellcode, welcher die abhängigen Assets holt, in der *checkAccess( )*-Methode nachzubilden. Im gegebenen Beispiel sind also der Iterator, der alle Mitgliedschaften zu einer Gruppe holt, und die anschließende while-Schleife zu duplizieren. Innerhalb der while-Schleife kann dann die Berechtigungsüberprüfung stattfinden, wie in Abb. 6.8 dargestellt. Die *Membership_Policy* in der Abbildung sei hier eine assetbezogene Policy, die die Lese- und Löschberechtigung für ein Membership-Objekt überprüft, die *Group_Policy* sei eine assetbezogene Policy, die die Lese- und Löschberechtigung für ein Group-Objekt überprüft. Bei diesem Vorgehen ist die Effizienz der Berechtigungsüberprüfung in Frage zu stellen, da die abhängigen Assets zweimal gelesen werden müssen, einmal für die Berechtigungsüberprüfung und einmal für das eigentliche Löschen innerhalb der Dienstausführung. In einer zweiten Lösungsvariante finden die Berechtigungsüberprüfungen für die abhängigen Assets direkt in der Geschäftslogik statt (siehe Abb. 6.9). Schlägt eine Überprüfung fehl, so muss die Dienstausführung beendet und die Transaktion abgebrochen werden. Bei dieser Variante kann nicht im Vorhinein festgestellt werden, ob eine Dienstausführung für einen Sitzungsbenutzer erlaubt sein wird, so dass die Anforderungen an die Response wie das automatische Ein- und Ausblenden von Links und Buttons (vgl. Kap. 4.2.4 und 6.1.4), falls eine graphische Benutzeroberfläche verwendet wird, nicht realisiert werden kann. Insgesamt müssen die zu löschenden Assets aber nur einmal gelesen werden.

Der C-Dienst legt ein neues Asset mit neuen Beziehungen zu bereits bestehenden Assets an. Die Berechtigung zum Anlegen eines Assets muss unabhängig von dem zu erstellenden Asset selbst sein, da dies ja vor Dienstausführung noch nicht existiert. Der C-Dienst enthält oft einen impliziten U-Dienst, da ja nach dem Anlegen eines Assets meistens auch seine Properties initialisiert werden. Die Berechtigungspolicy muss also dahingehend konsistent sein, dass die Erlaubnis des Anlegens eines neuen Assets auch die Erlaubnis zur Initialisierung seiner Properties mit einbezieht. Die Aufspaltung der Geschäftslogik eines C-Dienstes liest zunächst alle Assets, die nach Dienstausführung mit diesem Asset über Assoziationen verbunden sein sollen. In der *checkAccess( )*-Methode wird sowohl die Erlaubnis zum Anlegen des Assets als auch die Leseberechtigung für die anderen Assets geprüft. In der *doBusinessLogic( )*-Methode kann dann das eigentliche Anlegen mit dem Initialisieren der Properties des neuen Assets stattfinden. In Abb. 6.10 ist ein Beispiel für einen C-Dienst gegeben,

welcher ein neues Dokument anlegt. Innerhalb dieses Dienstes wird ein neues Geschäftsobjekt vom Typ *Document* erstellt. Mit der Erstellung des Dokuments wird auch die Pflichtassoziation *Document_Directory* zwischen dem Dokument und seinem Verzeichnis angelegt. Das Verzeichnis, um das es sich handelt, geht aus einem Parameter der Dienstanfrage hervor. Um das Dokument in besagtem Verzeichnis erstellen zu können, ist es notwendig, sowohl eine Berechtigung zum Erstellen von Dokumenten als auch eine Leseberechtigung für genau dieses Verzeichnis zu besitzen.

Die in diesem Abschnitt gegebenen Beispiele für die einzelnen Diensttypen sind für Demonstrationszwecke bewusst einfach gehalten worden. Reale Dienste, besonders die seiteneffektbehafteten Dienste, können eine wesentlich komplexere Geschäftslogik enthalten. Dem Leser mag auch aufgefallen sein, dass es Berechtigungen gibt, die einander implizieren. So impliziert z. B. die Erfüllung der *DirectoryAuthor_Policy* aus Abb. 6.5, dass der Sitzungsbenutzer sich auch authentifiziert hat, also die *NotAnonymous_Policy* ebenfalls gültig ist, da sonst keine eindeutige Identität des Sitzungsbenutzers festgestellt werden kann, anhand derer überprüft werden kann, ob er Mitglied der Autorengruppe eines Verzeichnisses ist. Soll bei einer automatischen Generierung der Berechtigungsüberprüfungen die implizierten Policies weggelassen werden, so muss der Generierungsalgorithmus die Implikationsbeziehungen zwischen Policies kennen. Das Erkennen dieser Implikationsbeziehungen ist in der vorliegenden Arbeit nicht vorgesehen, könnte aber als Erweiterung bedacht werden. Auf Implikationsbeziehungen wird kurz im Ausblick eingegangen.

```
public class DocumentChangePublicationStateServiceHandler extends
            ServiceHandler {
  String documentId;
  String newPublicationState;
  Document document;
  void getParameters(String[] parameters) {
    documentId = //read id-parameter
    newPublicationState = //read parameter for publication state
  }
  void initAssets( ) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    Documents documents = asm.getDocuments();
    document = documents.getDocument(documentId);
  }
  boolean checkAccess( ) {
    Owner_Policy policy0 = new Owner_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
        public Document getDocumentAsset() {
            return document;
        }
    }
    boolean path0 = policy0.existsPath();
    Admin_Policy policy1 = new Admin_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
    }
    boolean path1 = policy1.existsPath();
    return (path0 || path1);
  }
  int doBusinessLogic(Session session) {
    document.setProperty(
            Documents.publicationState, newPublicationState);
  }
}
```

Abb. 6.6: Aufspaltung der Geschäftslogik eines U-Dienstes

```
public class GroupsDeleteServiceHandler extends ServiceHandler {
  String groupId;
  void getParameters(String[] parameters) {
    groupId = //read id-parameter
  }
  void initAssets( ) {
  }
  boolean checkAccess( ) {
    NotAnonymous_Policy policy0 = new NotAnonymous_Policy() {
      public User getUserAsset() {
        return session_user; //object known by ServiceHandler
      }
    }
    boolean path0 = policy0.existsPath();
    Admin_Policy policy1 = new Admin_Policy() {
        public User getUserAsset() {
            return session_user; //object known by ServiceHandler
        }
    }
    boolean path1 = policy1.existsPath();
    return (path0) && (path1);
  }
  int doBusinessLogic(Session session) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    Groups groups = asm.getGroups();
    Group group = groups.getGroup(groupId);
    Memberships ms = asm.getMemberships();
    Iterator m = ms.getMembershipsOfGroup(group);
    while (m.hasNext()) {
        Membership membership = (Membership) m.next();
        ms.remove(membership.getId());
    }
    groups.remove(group.getId());
  }
}
```

Abb. 6.7: D-Dienst für eine Gruppe nach Aufspaltung der Geschäftslogik für gruppenbezogene Policy

```
public class GroupsDeleteServiceHandler extends ServiceHandler {
  String groupId;
  void getParameters(String[] parameters) {
    groupId = //read id-parameter
  }
  AssetSchemaManager asm;
  Groups groups;
  Group group;
  Memberships ms;
  void initAssets( ) {
    asm = AssetSchemaManager.INSTANCE;
    groups = asm.getGroups();
    group = groups.getGroup(groupId);
    ms = asm.getMemberships();
  }
  boolean checkAccess( ) {
    Iterator m = ms.getMembershipsOfGroup(group);
    while (m.hasNext()) {
        Membership membership = (Membership) m.next();
        Membership_Policy mp = new Membership_Policy() {
          … //object membership needed here
        }
        if(!mp.existsPath()) return false;
    }
    Group_Policy policy0 = new Group_Policy() {
      … //object group needed here
    }
    boolean path0 = policy0.existsPath();
    return path0;
  }
  int doBusinessLogic(Session session) {
    Iterator m = ms.getMembershipsOfGroup(group);
    while (m.hasNext()) {
        Membership membership = (Membership) m.next();
        ms.remove(membership.getId());
    }
    groups.remove(group.getId());
  }
}
```

Abb. 6.8: D-Dienst für eine Gruppe nach Aufspaltung der Geschäftslogik für assetbezogene Policies, Variante 1

```
public class GroupsDeleteServiceHandler extends ServiceHandler {
  String groupId;
  void getParameters(String[] parameters) {
    groupId = //read id-parameter
  }
  AssetSchemaManager asm;
  Groups groups;
  Group group;
  void initAssets( ) {
    asm = AssetSchemaManager.INSTANCE;
    groups = asm.getGroups();
    group = groups.getGroup(groupId);
  }
  boolean checkAccess( ) {
    Group_Policy policy0 = new Group_Policy() {
      … //object group needed here
    }
    boolean path0 = policy0.existsPath();
    return path0;
  }
  int doBusinessLogic(Session session) {
    Memberships ms = asm.getMemberships();
    Iterator m = ms.getMembershipsOfGroup(group);
    while (m.hasNext()) {
        Membership membership = (Membership) m.next();
        //insert access check here
        Membership_Policy mp = new Membership_Policy() {
          … //object membership needed here
        }
        if(!mp.existsPath()) {
          //abort transaction
          return ACCESS_DENIED;
        }
        ms.remove(membership.getId());
    }
    groups.remove(group.getId());
  }
}
```

Abb. 6.9: D-Dienst für eine Gruppe nach Aufspaltung der Geschäftslogik für assetbezogene Policies, Variante 2

```
public class DocumentCreateServiceHandler extends ServiceHandler {
  String directoryId;
  String title;
  … //other values for other properties
  Directory dir;
  void getParameters(String[] parameters) {
    directoryId  = //read id-parameter
    title = // read title-parameter
    … //read other values for other properties
  }
  AssetSchemaManager asm;
  void initAssets( ) {
    asm = AssetSchemaManager.INSTANCE;
    Directories dirs = asm.getDirectories();
    dir = dirs.getDirectory(directoryId);
  }
  boolean checkAccess( ) {
    //use session-user for create access check and
    //use session-user and dir for read access check
  }
  int doBusinessLogic(Session session) {
    Documents docs = asm.getDocuments() ;
    //createDocument() also sets Document_Directory-Association
    Document doc = docs.createDocument(dir);
    doc.setTitle(title);
    … //set other properties
  }
}
```

Abb. 6.10: Beispiel für einen ServiceHandler zum Anlegen eines Dokuments

6.1.4 Realisierung der Anforderungen an die Konfigurierbarkeit der Response bei Nutzung einer Benutzeroberfläche

Bei Nutzung einer graphischen Oberfläche gibt es häufig Buttons und Links, die auf weitere Dienste verweisen. Es soll nun möglich sein, schon bei Auslieferung der Antwortnachricht festzustellen, ob ein bestimmter Sitzungsbenutzer diesen Dienst wird ausführen können. Dies war eine der Anforderungen an die Konfigurierbarkeit der Response (vgl. Kap. 4.2.4). Für die Limited View-Variante kann für nicht-ausführbare Dienste der entsprechende Button bzw. der entsprechende Link gleich ausgeblendet oder deaktiviert werden. Für die Full View With Errors-Variante gibt es hier nichts zu tun, da alle Links und Buttons einfach erhalten bleiben und ganz normal bei Aufruf eines Dienstes die Berechtigungsabfrage erfolgen kann wie in Abb. 6.3 dargestellt. Im Folgenden wird auf die Limited View-Variante eingegangen. Der konzeptuelle Ablauf ist in Abb. 6.11 dargestellt. Zunächst wird der Response-String auf enthaltene Links hin untersucht. Für jeden Link ist herauszufinden, welchen Dienstbearbeiter er anspricht und welche Parameter übergeben werden sollen. Mit diesen beiden Informationen kann der Dienstbearbeiter testweise aufgerufen werden. Hierfür steht die Methode

testService( ) zur Verfügung. Diese liest die mitgelieferten Parameter aus, initialisiert die notwendigen Assets und ruft dann die *checkAccess( )*-Methode auf. Bei positivem Resultat kann der Link in der Antwortnachricht bestehen bleiben, bei negativem Resultat wird der Link gelöscht oder deaktiviert. Die so veränderte Antwortnachricht wird an den Client zurück gesendet.

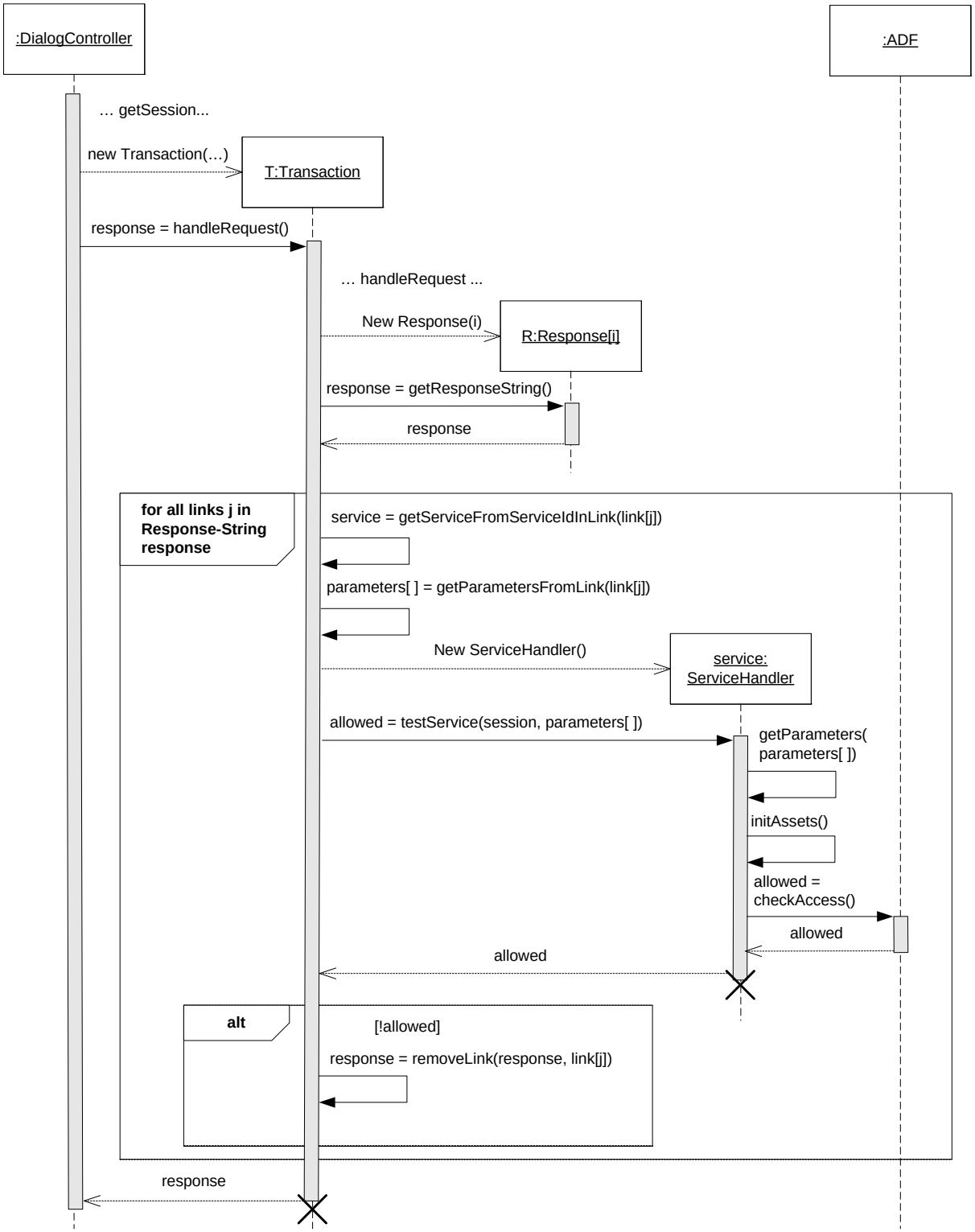


Abb. 6.11: Generisches Ein- und Ausblenden von Links in der Response

6.1.5 Realisierung der Anforderungen an die Konfigurierbarkeit der Response für R- und Q-Dienste

Für die Erstellung der Response eignet sich ein vorlagenbasierter Ansatz (siehe [Weg02]), d. h., es liegen statische Vorlagen für die Antwortnachrichten vor, die zur Laufzeit mit dynamischen Inhalten gefüllt werden. Für die R-Dienste liegt z. B. für jeden Assettyp ein Template vor, in welches die für ein konkretes Asset aktuellen Werte der im Template vorgesehenen Properties eingefügt werden können. Für die Q-Dienste liegen entsprechende Listenvorlagen vor, die es erlauben, beliebig viele Suchergebniszeilen in eine Antwortnachricht zu generieren.

Während der Befüllung der *Response-Templates* mit dynamischen Inhalten finden nur lesende Zugriffe auf Assets und AssetSchemas statt. Anders als bei der Berechtigungsdurchsetzung für die Geschäftslogik der ServiceHandler-Klassen kann hier die Berechtigungsüberprüfung nicht gebündelt werden, denn je nach Berechtigung soll der jeweilige Sitzungsbenutzer verschiedene Sichten auf die Antwortnachricht bekommen. Für einen R-Dienst bekommen verschiedene Sitzungsbenutzer je nach Berechtigungsstatus die Werte unterschiedlicher oder unterschiedlich vieler Properties eines Assets zurückgeliefert. Bei Q-Diensten gilt dasselbe für die Anzahl der zurück gelieferten Suchergebnisse. Für jedes Suchergebnis soll einzeln überprüft werden, ob der jeweilige Sitzungsbenutzer leseberechtigt ist.

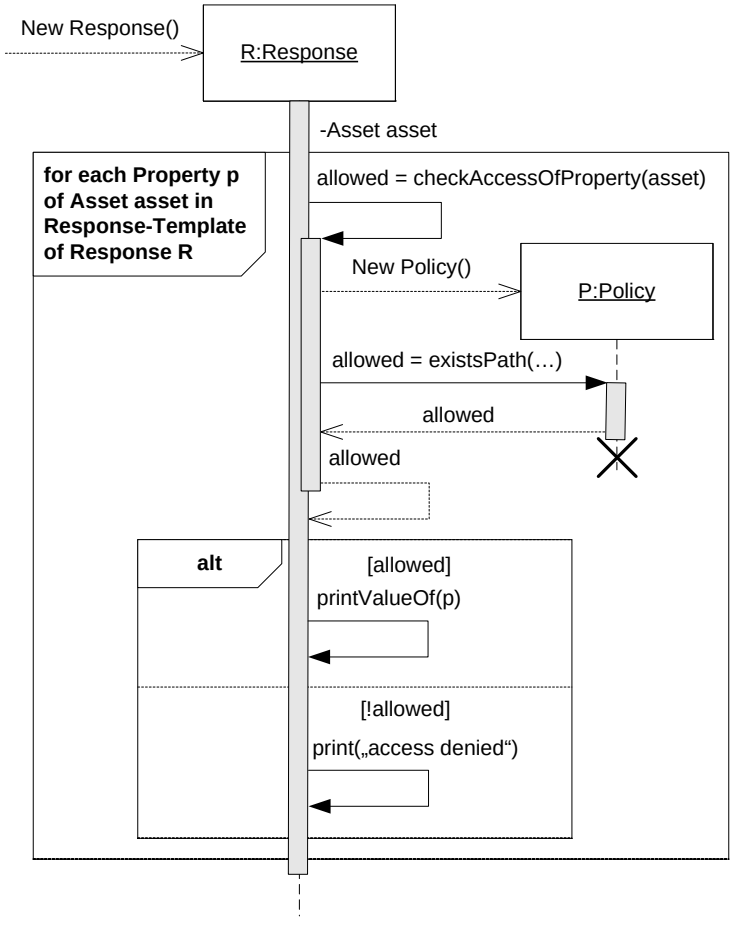


Abb. 6.12: Erstellung der Response für einen R-Dienst

In Abb. 6.12 ist dargestellt, wie die Erstellung der Antwortnachricht für einen R-Dienst aussieht. Der ServiceHandler liest in seiner *initAssets( )*-Methode das Asset aus, dessen Properties gelesen werden sollen. Dieses Asset wird der Response zur Verfügung gestellt. Für jede Property, für die im Response-Template eine dynamische Ersetzung vorgesehen ist, ruft die Response eine *checkAccessOfProperty( )*-Methode auf. Es sind also genauso viele *checkAccessOfProperty( )*-Methoden statisch vorgesehen, wie es unterschiedliche, zu ersetzende Properties gibt. Die Implementierung dieser

Methode erstellt die für die Property gültige Policy. So kann zur Laufzeit die Zugriffsberechtigung abgefragt werden. Um welche Policy es sich handelt, kann einfach durch eine statische Quellcodeanalyse festgestellt werden, denn für die Properties gibt es in den einzelnen AssetSchema-Klassen, die die Eigenschaften des jeweiligen Assettyps definieren, *@Read*-Annotationen, die die jeweils gültigen Read-Policies auflisten, eventuell auch *@ReadCommon*- und *@ReadRestricted*-Annotationen, falls es mehrere Attributcluster mit unterschiedlichen Leseberechtigungen geben soll (vgl. Kap. 5.3.3). Für die Realisierung einer effizienten Zugriffskontrolle ist hier eine Caching-Strategie sinnvoll, da die Zugriffspolicy für viele der darzustellenden Properties identisch sein wird. Auf die Möglichkeit des Cachings wird im Ausblick der Arbeit eingegangen.

Für die Gestaltung der Antwortnachricht gibt es jetzt die zwei Sicherheitsmuster Limited View und Full View With Errors (vgl. Kap. 4.2.4). Im Falle einer Zugriffsverweigerung wird im Sicherheitsmuster Full View With Errors statt des Wertes der entsprechenden Property eine Fehlermeldung ausgegeben. Dieses Szenario ist in Abb. 6.12 dargestellt. Beim Zugriffsverbot im Sicherheitsmuster Limited View wird auf diese Fehlermeldung einfach verzichtet und stattdessen *null* bzw. ein leerer String zurückgeliefert.

Für die Suchdienste gibt es, wie oben beschrieben, zwei Implementierungsmöglichkeiten (siehe Seite 92). Orthogonal dazu können auch die beiden Sicherheitsmuster Limited View und Full View With Errors zum Einsatz kommen. Für die Limited View-Variante ist es von Vorteil, wenn nicht sichtbare Geschäftsobjekte gar nicht erst im Suchergebnis erscheinen. Es bietet sich also an, die Datenbankanfrage so umzuschreiben, dass von den Suchergebnissen der ursprünglichen Query alle diejenigen herausgenommen werden, die der jeweilige Benutzer nicht sehen darf. Um dies zu realisieren, müssen sowohl die Query-API als auch die PathExpressions erweitert werden. Die Query-API wird dazu verwendet, Filteranfragen auf AssetSchemas zu stellen (vgl. Kap. 3.1.8). Eine solche Filteranfrage wurde bislang durch die Basisfunktionalität *AssetSchema.queryAssets(Query q)* realisiert. Solche Filteranfragen können aber ebenfalls mit Hilfe der PathExpressions formuliert werden, indem Filter-Objekte und Navigation-Objekte verwendet werden. Für die Formulierung dieser Filteranfragen wird die Klasse *QueryPolicy* (siehe Abb. 6.13) bereitgestellt. Sie benötigt die eigentliche Datenbank-Query formuliert als Policy sowie die Policy, die die Zugriffsberechtigung ausdrückt. Die Methode *QueryPolicy.getAssets( )*, die ab jetzt anstelle von *AssetSchema.queryAssets( )* verwendet wird, wertet dann die Filteranfrage so aus, dass nur Elemente im Suchergebnis auftauchen, deren Lesezugriff durch die Berechtigungspolicy erlaubt sind.

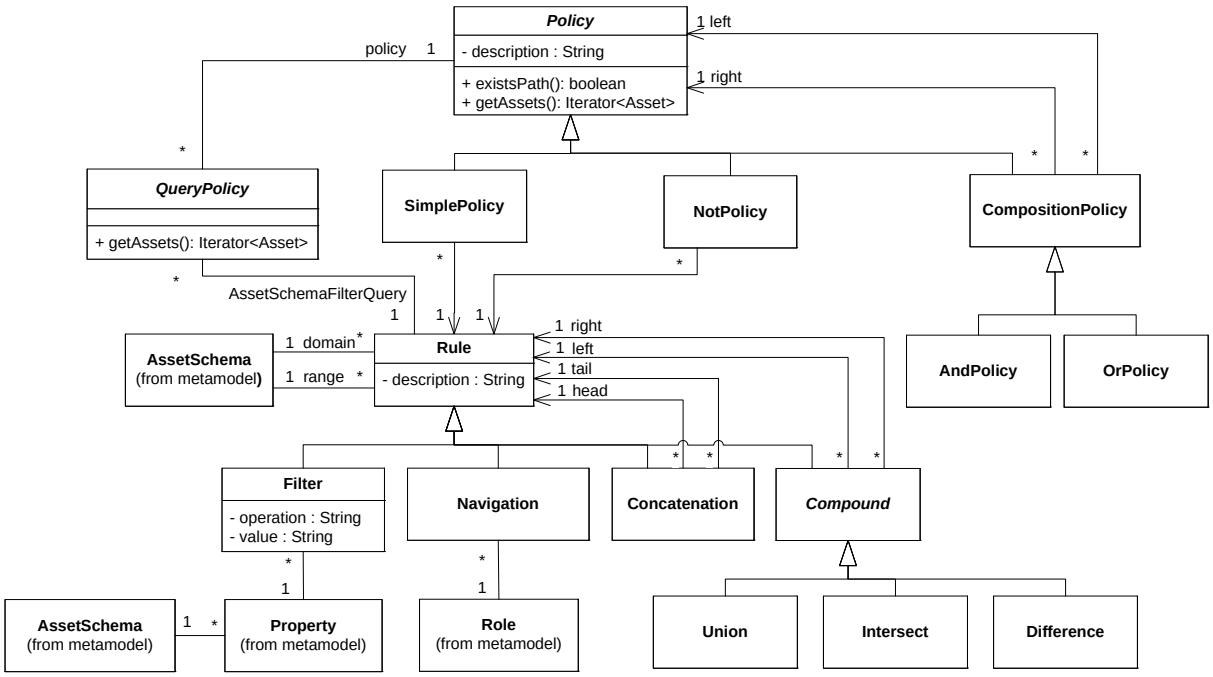


Abb. 6.13: Erweiterung der PathExpressions um Filteranfragen als QueryPolicies

In Abb. 6.14 ist ein Beispiel für eine *QueryPolicy* gegeben, welche alle Dokumente eines Verzeichnisses zurückliefert. Sie besteht aus einer einfachen Navigation, die ausgehend von einem Objekt *dir* vom Typ *Directory* entlang der Assoziation *Document_Directory* zu allen Dokumenten navigiert, die direkt mit *dir* verbunden sind.

```
public abstract class ParentDirectory extends Navigation {
  public Role[] getRoles() {
    return new Role[] {
      DOCUMENT_DIRECTORY.getManyRole();}
}}
public abstract class DocumentOfDirectory_Policy extends QueryPolicy {
  public Rule getRule() {
    return new ParentDirectory() {
        public boolean getStartAsset() { return getDirectory()};
        public boolean getEndAsset() { return null; }
    }
  }
  public abstract Directory getDirectory();
}

public class AssetSchema {
  public Iterator<Asset>
      getDocumentsOfDirectory(Policy p, Directory dir) {
    QueryPolicy qp = new DocumentsOfDirectory_Policy() {
      public Policy getPolicy() {
        return p;
      }
      public Directory getDirectory() {
        return dir;
      }
    }
    return qp.getAssets();
  }
}
```

Abb. 6.14: Beispiel für die Verwendung der neuen Basisfunktionalität *QueryPolicy.getAssets()*

Die Auswertung der Methode *QueryPolicy.getAssets( )* erfolgt über Union-, Intersect- und Difference-Queries. Die Query-API muss also um diese Queries erweitert werden (siehe Abb. 6.15).

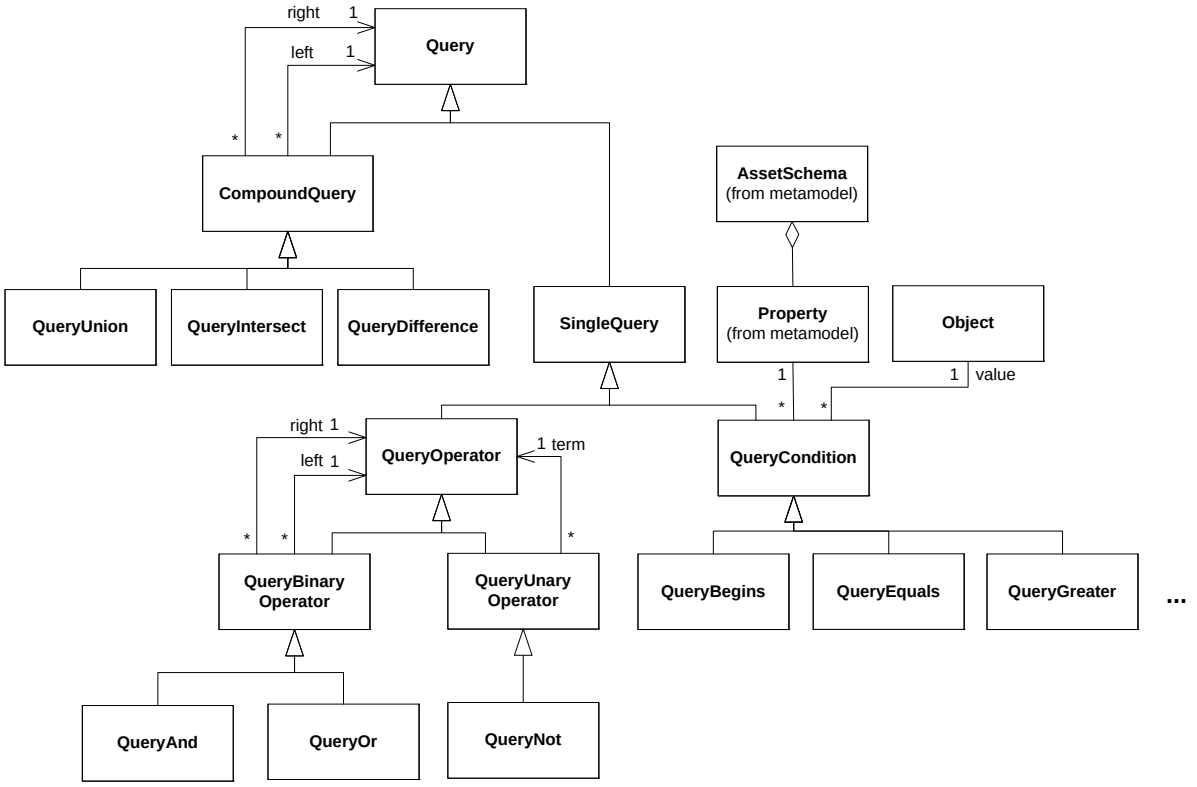


Abb. 6.15: Erweiterung der Query-API um Union-, Intersect- und Difference-Operatoren

Für die Realisierung der Limited View sind mehrere Fälle zu unterscheiden. Zum einen gibt es Berechtigungspolicies, welche über ein eigentümer- oder assetbezogenes, also ein assetabhängiges, Zugriffsprinzip formuliert werden, zum anderen gibt es Policies, welche über ein gruppenbezogenes, also assetunabhängiges, Zugriffsprinzip beschrieben werden. Desweiteren gibt es Mischformen.

Zunächst wird auf die eigentümerbezogenen Berechtigungspolicies eingegangen. Die Regel dieser Berechtigungspolicy ist so gerichtet, dass sie bei Assets vom Typ *User* aufhört. Gegeben sei z. B. eine Berechtigungspolicy, die den Lesezugriff auf Dokumente erlaubt, falls der Sitzungsbenutzer Eigentümer des Dokuments ist, sowie eine Anfrage als *QueryPolicy*, die zu einem gegebenen Verzeichnis alle Dokumente zurückliefert (siehe Abb. 6.16, vgl. Abb. 5.9).

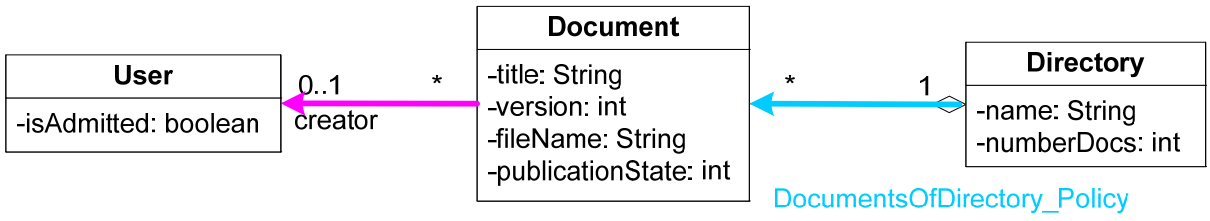


Abb. 6.16: Beispiel für eine eigentümerbezogene Berechtigungspolicy und eine QueryPolicy

Um herauszufinden, welche Dokumente in der Ergebnismenge der Anfrage enthalten sein dürfen, ist es zweckmäßig den Pfad der Berechtigungspolicy umzukehren und beginnend beim Sitzungsbenutzer zu fragen, welche Dokumente dieser sehen darf. Die Schnittmenge dieser Ergebnismenge mit der Ergebnismenge der *QueryPolicy* liefert das gewünschte Ergebnis. Im Folgenden ist die Auswertung mit einer an OCL angelehnten Syntax dargestellt.

```
context Document :
    let Creator : Set (User) = self.creator
context Directory :
    let DocsOfDir : Set (Document) = self.Document
```

wird umgewandelt in ➔

```
context User:
    let Inverse_Creator : Set(Document) = self.Document
context Directory :
    let DocsOfDir: Set (Document) = self.Document
```

wird ausgewertet zu ➔

```
let DocumentsOfDirectory_Policy<Directory>:
Set(Document) = Inverse_Creator→intersection(DocsOfDir)
```

Für die Erstellung der rückwärtsgerichteten Regel muss in die PathExpressions-API eine zusätzliche Methode *reverseVisit( )* eingefügt werden. Momentan wird aus einer PathExpression eine Query gebildet, indem mit Hilfe des Besuchermusters und einer *visit( )*-Methode die Teilregeln einer Regel durchlaufen werden und so schrittweise die Query zusammengesetzt wird. Die *reverseVisit( )*-Methode muss diese Teilregeln jetzt in umgekehrter Reihenfolge durchlaufen. Weiterhin wird die mengenwertige Methode *Policy.getAssets( )* bereitgestellt, die die Menge Assets zurückliefert, die durch die Auswertung einer Query gegeben ist. Des Weiteren wird die PathExpressions-API um eine Intersect-Rule erweitert, die die Schnittmengenbildung der Ergebnismengen zweier ausgewerteter Regeln abbildet. Für die Realisierung dieser *Intersect*-Rule als Datenbankanfrage wird auf die durch die Query-API bereitgestellte Intersect-Query zurückgegriffen.

Für alle anderen einfachen Policies (Objekte vom Typ *SimplePolicy*) mit assetbezogenen Regeln ist analog zu verfahren. Endete die Regel ursprünglich beim Sitzungsbenutzer, so wird die Regel beginnend bei dem Objekt, das den Sitzungsbenutzer darstellt, rückwärts gerichtet ausgewertet, so dass sie eine Menge von Assets des Typs zurückliefert, bei dem die Regel ursprünglich begonnen hat. Endete die Regel bei einem anderen Assettyp, so wird beginnend bei dem entsprechenden Asset dieses Typs, auf das die Regel anzuwenden war, die Regel rückwärts ausgewertet. Die rückwärtsgerichtete Regel und die eigentliche Anfrage werden dann über eine *Intersect*-Rule ausgewertet.

Einfache Policies, die eine Regel beinhalten, die auf gruppenbasierten Zugriffsprinzipien basiert, sind anders zu behandeln. Gegeben sei z. B. eine Berechtigungspolicy, die den Lesezugriff auf Dokumente gewährt, falls der Sitzungsbenutzer Mitglied der Administratorengruppe ist (siehe Abb. 6.17, vgl. Abb. 5.10).

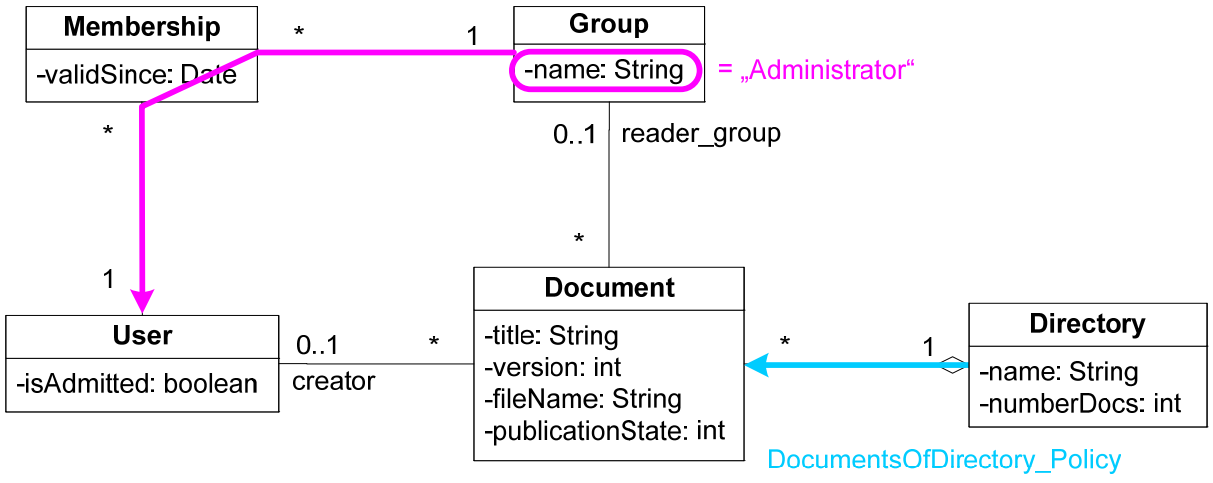


Abb. 6.17: Beispiel für eine gruppenbezogene Berechtigungspolicy und eine QueryPolicy

In diesem Fall können die beiden Regeln unabhängig voneinander ausgewertet werden. Evaluiert die Berechtigungspolicy zu *wahr*, so können ohne Einschränkung alle Dokumente in der Ergebnismenge der QueryPolicy auftauchen, evaluiert die Berechtigungspolicy zu *falsch*, so ist die Ergebnismenge leer und die QueryPolicy braucht nicht mehr ausgewertet werden.

```
context Group:
    let AdminFilter : Group = self.select(self.name, =, „Administrator“)
    let Member : Set(User) = self.Membership.User
    let MemberAdmin : Set(User) = AdminFilter.Member
    let Admin : boolean = MemberAdmin → includes (caller)
context Directory :
    let DocsOfDir : Set (Document) = self.Document
```

wird ausgewertet zu ➔

```
let DocumentsOfDirectory_Policy<Directory>:
  Set(Document) =
    if Admin then
      DocsOfDir
    else
      null
    endif
```

Ein weiterer Typ von Policies ist die NotPolicy, die Regeln beschreibt, deren Pfade nicht existieren dürfen, damit die Policy zu *wahr* evaluiert. Die Kombination einer QueryPolicy mit einer NotPolicy wird durch eine *Difference*-Regel abgebildet, die auf einer *Difference*-Query basiert. Gegeben sei z. B. eine Berechtigungspolicy, die den Lesezugriff auf Dokumente gewährt, falls diese nicht gesperrt sind (siehe Abb. 6.18, vgl. Abb. 5.11). Um diese Anfrage auszuwerten, werden startend bei der Menge aller Assets des Typs, bei dem der leere Pfad endet, in diesem Beispiel also bei allen Assets vom Typ *AssetLock*, alle bestehenden Assoziationen ausgewertet. Die Regel wird also nicht nur rückwärtsgerichtet sondern auch für alle Assets eines Typs ausgewertet. Die →isEmpty()-Policy wird also in eine →allInstances()-Regel umformuliert. Um die Menge der Dokumente herauszufinden, die der Sitzungsbenutzer sehen darf, müssen jetzt von der Ergebnismenge der QueryPolicy alle diejenigen Dokumente abgezogen werden, die durch die so veränderte NotPolicy gegeben sind. Dies wird durch einen Difference-Operator bewerkstelligt. In OCL wird der Difference-Operator mit →excluding() notiert.

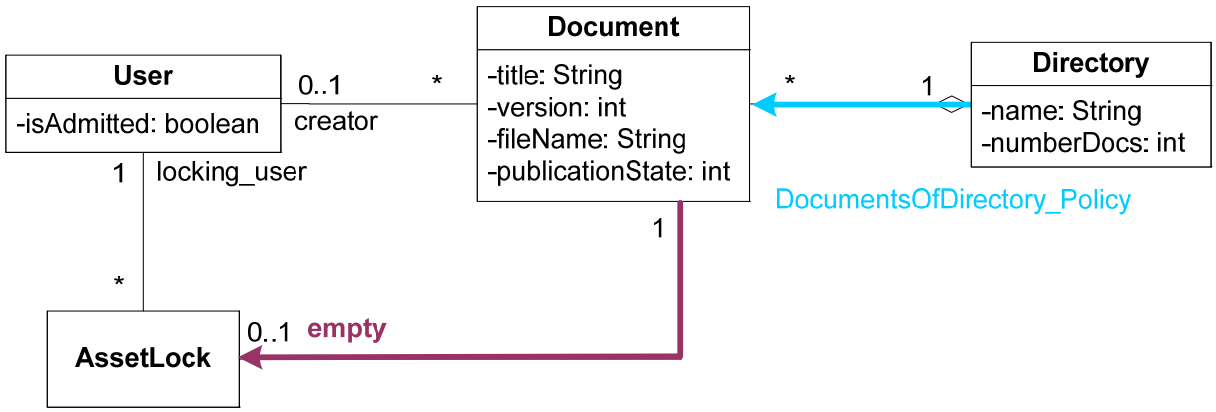


Abb. 6.18: Beispiel für eine Regel mit einem leeren Pfad und eine QueryPolicy

```
context Document:
  let lock : Set(AssetLock) = self.AssetLock
  let unlockedDocument<Document> :  lock → isEmpty()
context Directory :
  let DocsOfDir : Set (Document) = self.Document
```

wird umgewandelt in ➔

```
context AssetLock:
  let not_locked : Set(Document) = AssetLock.allInstances().Document
context Directory :
  let DocsOfDir : Set (Document) = self.Document
```

wird ausgewertet zu ➔

```
let DocumentsOfDirectory_Policy<Directory>:
  Set(Document) = DocsOfDir→excluding(not_locked)
```

Nun bestehen Policies häufig nicht nur aus einer einzelnen Policy sondern aus einer Kombination von Policies. Ein Algorithmus, welcher beliebige Berechtigungspolicies mit einer QueryPolicy verknüpft, wertet die einzelnen Policies gemäß ihrer Klammerung von innen nach außen aus. Hierbei werden eine OrPolicy in einen Union-Operator und eine And-Policy in einen Intersect-Operator umgewandelt. Der Algorithmus ist in Abb. 6.19 als Pseudo-Code dargelegt. Die notwendigen Erweiterungen der PathExpressions-API sind für die Rules in Abb. 6.20 und für die Policies in Abb. 6.21 dargestellt.

```
public abstract class QueryPolicy {
  public abstract Rule getQueryRule();
  public abstract Policy getPolicy();
  public Iterator<Asset> getAssets() {
    Rule queryRule = getQueryRule();
    Policy p = getPolicy();
    if(p instanceof OrPolicy) {
      //analog to implementation of OrPolicy.getAssets()
      return (p.getLeft().getAssets(queryRule) UNION
              p.getRight().getAssets(queryRule));
    }
    if(p instanceof AndPolicy) {
      //amalog implementation of AndPolicy.getAssets()
      return (p.getLeft().getAssets(queryRule) INTERSECT
              p.getRight().getAssets(queryRule));
    }
    if(p instanceof SimplePolicy) {
      return ((SimplePolicy) p).getAssets(queryRule);
    }
    if(p instance of NotPolicy) {
      return ((NotPolicy)p).getAssets(queryRule);
    }
  }
}
public abstract class SimplePolicy {
  public abstract Rule getReverseRule();
  public Iterator<Asset> getAssets(Rule queryRule) {
    if(this is group based policy) {
      if(this.existsPath())
        return queryRule.getEndContainer().queryContent(queryRule.visit());
      else
        return new EmptyIterator();
    }
    if(this is owner based or asset based policy) {
      Intersect intRule = new Intersect(queryRule, this.getReverseRule());
      return queryRule.getEndContainer().queryContent(intRule.visit());
    }
  }
}
public abstract class NotPolicy {
  public abstract Rule getReverseRule();
  public Iterator<Asset> getAssets(Rule queryRule) {
    Difference diffRule = new Difference(queryRule, this.getReverseRule());
    return queryRule.getEndContainer().queryContent(diffRule.visit());
  }
}
```

Abb. 6.19: Auswertungsalgorithmus für eine QueryPolicy als Pseudo-Code

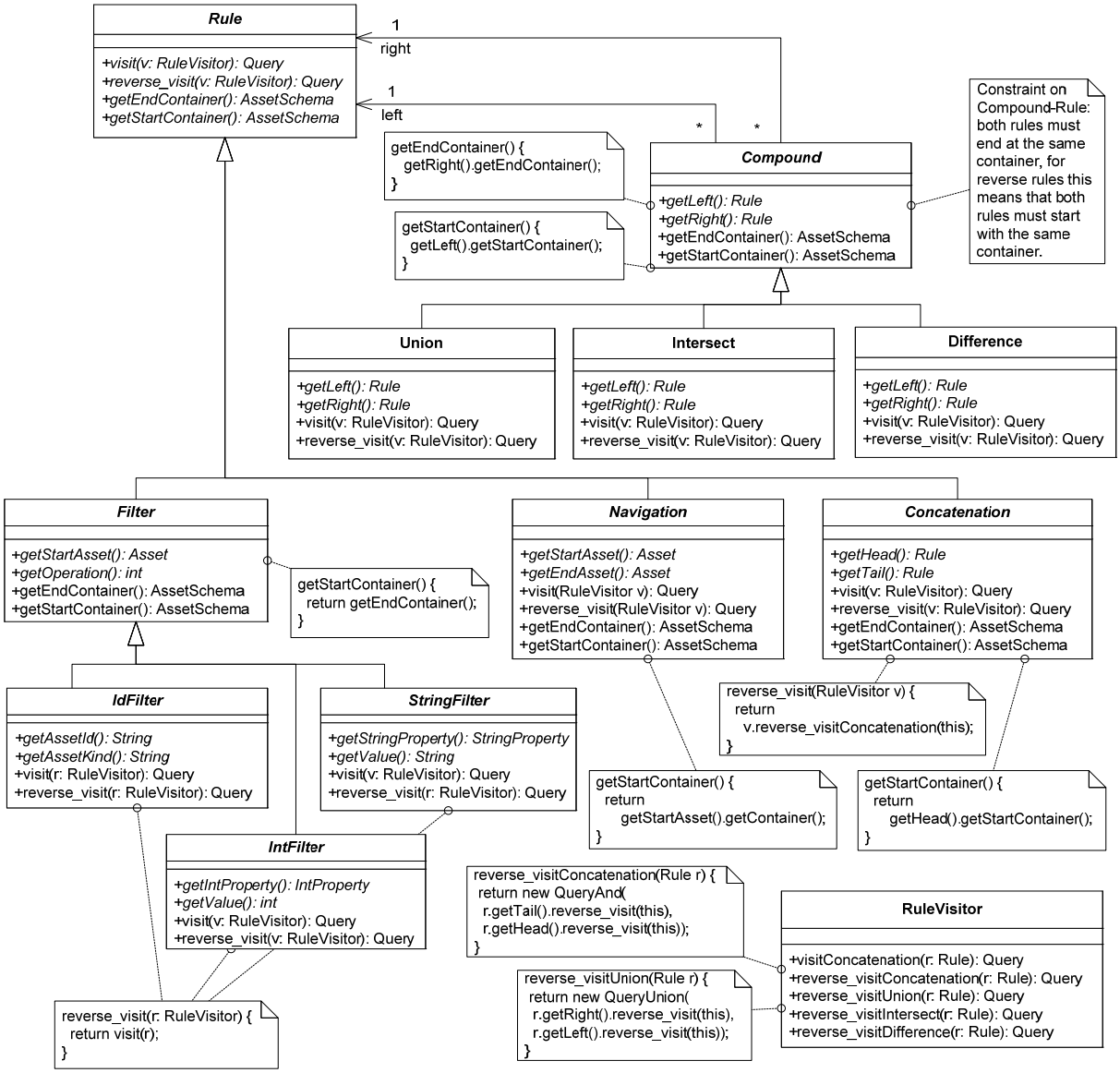


Abb. 6.20: Erweiterungen der Rule-Klassen der PathExpressions

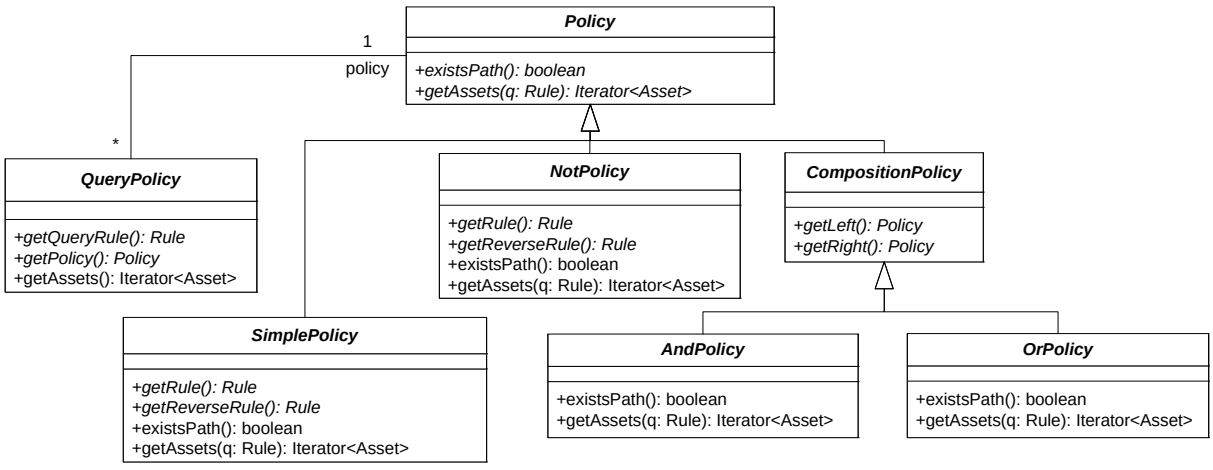


Abb. 6.21: Erweiterungen der Policy-Klassen der PathExpressions

Die Realisierung der Full View With Errors-Variante für die Q-Dienste kann ähnlich erfolgen wie die Realisierung der Full View With Errors-Variante für die R-Dienste. Der konzeptuelle Ablauf der Response-Erstellung ist in Abb. 6.22 dargestellt. Hier gehen wir davon aus, dass die betreffende Query bereits in der *doBusinessLogic( )*-Methode des *ServiceHandlers* ausgeführt wurde und der Response die Suchergebnisse zur Verfügung gestellt werden. Für jedes Asset im Anfrageergebnis wird die Leseberechtigung für den aktuellen Sitzungsbenutzer überprüft. Durch eine statische Quellcodeanalyse kann die betreffende Policy einfach herausgefunden werden, da das AssetSchema, auf das sich die Query bezieht im Vorhinein bekannt ist. Die *@ReadPolicy*-Annotation des AssetSchemas gibt an, welche Policies zu überprüfen sind. Auch für die Darstellung der Suchergebnisse gibt es Vorlagen. Diese sehen z. B. vor, welche Properties der Assets in einer Suchergebniszeile dargestellt werden sollen. Nach der Überprüfung, ob das entsprechende Asset überhaupt gelesen werden darf, ist also noch für alle darzustellenden Properties zu überprüfen, ob diese ausgegeben werden dürfen. Die Überprüfung erfolgt so, wie schon oben für den R-Dienst dargestellt. Eine Caching-Strategie für die Full View With Errors-Variante des Q-Dienstes ist für die Überprüfung der Leseberechtigung auf den einzelnen Assets des Suchergebnisses nur dann sinnvoll, wenn sich die Policy aus gruppenbezogenen Zugriffsprinzipien zusammensetzt, da hier das Ergebnis der Berechtigungsabfrage für alle Assets im Suchergebnis gleich sein wird. Ein Caching der Berechtigungsergebnisse für den im Q-Dienst enthaltenen R-Dienst kann analog zum Caching des R-Dienstes erfolgen.

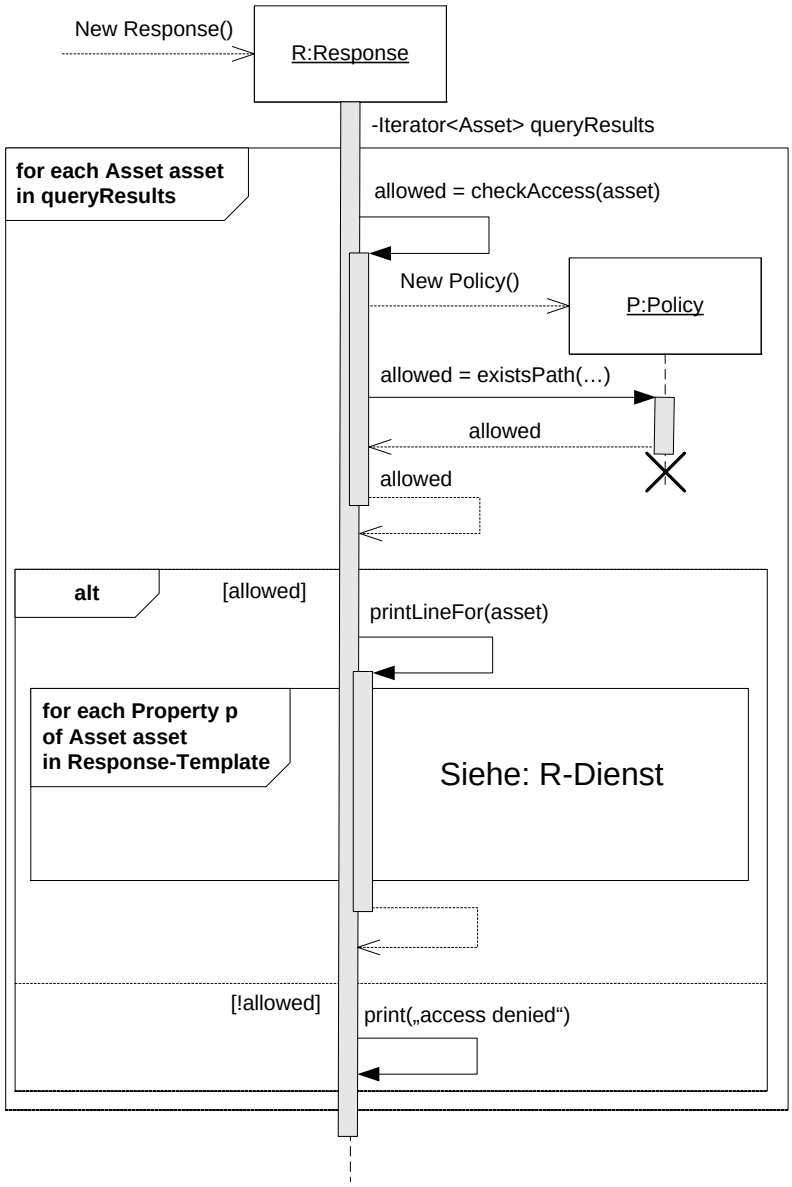


Abb. 6.22: Erstellung der Response für einen Q-Dienst mit Sicherheitsmuster Full View With Errors

6.2 Automatische Quellcodegenerierung für die Durchsetzung der Zugriffskontrolle der Geschäftslogik

Im vorangegangenen Kapitel wurde beschrieben, wie die Anforderungen der Zugriffskontrolle an die Antwortnachricht mit Hilfe einer Sicherheitsarchitektur realisiert werden können. Ebenso wurde die Dreiteilung der Geschäftslogik diskutiert, um pro Dienstbearbeiter eine feste Position im Quellcode zu haben, in der die Berechtigungen durchgesetzt werden können. Dieses Kapitel beschäftigt sich damit, wie die Implementierung der Berechtigungsdurchsetzungsfunktion aus dem Quellcode der Geschäftslogik generiert werden kann. Kapitel 6.2.1 beleuchtet das allgemeine Vorgehen. Die Unterkapitel 6.2.2 - 6.2.4 beschreiben die einzelnen Analyseschritte für eine statische Quellcodeanalyse und die Generierung von Quellcode für die Berechtigungsdurchsetzung in der *checkAccess( )*-Methode einer *ServiceHandler*-Klasse.

6.2.1 Allgemeines Vorgehen

Um für einen *ServiceHandler* eine Berechtigungsdurchsetzungsfunktion bereitstellen zu können, ist es notwendig zu wissen, welche zugriffskontrollrelevanten Anweisungen für die Diensterbringung ausgeführt werden. Es bietet sich an, diese Zugriffskontrollinformationen durch eine statische Quellcodeanalyse bereitzustellen.

In dieser Arbeit werden Java-basierte Anwendungen untersucht. In der Regel ist es ein komplexes Unterfangen, Programme, die in einer komplexen Hochsprache wie Java geschrieben sind, statisch zu analysieren. Die hier zu untersuchende Geschäftslogik ist in mehr oder weniger unstrukturiertem Quellcode geschrieben. Es gibt kein einfaches Modell, um die Geschäftslogik einer betrieblichen Anwendung zu beschreiben. Dennoch bedient jedes Framework eine bestimmte Anwendungsdomäne und verwendet deshalb Abstraktionen, die der Beschreibung der Anwendungsdomäne dienlich sind. Auch im Fall von betrieblichen Anwendungen, die im Wesentlichen fünf verschiedene Diensttypen bzw. Dienste, die sich aus den fünf Diensttypen kombinieren lassen, unterstützen, lassen sich die verschiedenen Dienste unter Zuhilfenahme wiederkehrender Programmiermuster implementieren. Die Menge der im Framework verwendeten Abstraktionen kann also relativ schmal gehalten werden, so dass eine statische Quellcodeanalyse praktisch anwendbar ist. Einfache Beispiele für den Quellcode der Geschäftslogik eines jeden Diensttyps, die die wiederkehrenden Abstraktionen veranschaulichen, wurden bereits in Kapitel 6.1 gegeben. Die einzelnen Vorgehensschritte für die statische Quellcodeanalyse sind in Abb. 6.23 dargestellt. Sie werden in den folgenden Unterkapiteln genau erläutert.

In einem ersten Schritt, wird die Komplexität des zu untersuchenden Quellcodes, der schon unter Verwendung der im Framework bereitgestellten Abstraktionen geschrieben ist, gedrückt, indem er in eine leichter zu analysierende Darstellung umgewandelt wird. Hierfür wird das Framework SOOT [Val00, VHS+99] der McGill-Universität, Montreal, Kanada, eingesetzt. SOOT ist ein Werkzeug für die statische Codeanalyse von Java-Quellcode. Viele gängige Analysealgorithmen für die intraprozedurale und interprozedurale Analyse stehen zur Verfügung. Es bietet aber auch Erweiterungspunkte für die Implementierung eigener Algorithmen an. SOOT arbeitet mit mehreren internen Darstellungen des zu untersuchenden Java-Quellcodes. Die für die Analyse gewählte intermediäre Darstellung ist ein typisierter Drei-Adress-Code. Dieser wird in Unterkapitel 6.2.2 näher erläutert.

In einem zweiten Schritt findet die eigentliche statische Analyse statt. Sie extrahiert aus dem Quellcode die berechtigungsrelevanten Informationen und speichert diese in einer Datenstruktur, die mit *AccessTypes* bezeichnet wird. Am Ende der Analysephase existieren für jeden *ServiceHandler* ein oder mehrere *AccessTypes*. Aus diesen kann zurückverfolgt werden, wie die einzelnen Assets mittels der zur Verfügung stehenden Basismethoden ausgelesen, angelegt, aktualisiert und gelöscht werden, so dass es möglich ist, herauszufinden, welche Policies zur Ausführung eines *ServiceHandlers* notwendig sind. Die Beschreibung der einzelnen Analyseschritte und der *AccessTypes* findet in Kapitel 6.2.3 statt.

In einem dritten Schritt werden die Berechtigungsdurchsetzungsfunktionen generiert. Soweit dies möglich ist, sollen diese in die oben beschriebene *checkAccess( )*-Methode des Frameworks (vgl. Kap. 6.1.3) platziert werden. Der hier beschriebene Algorithmus geht davon aus, dass es möglich ist, alle

Berechtigungsüberprüfungen im Vorhinein abzufragen und generiert diese in die *checkAccess( )*-Methode. Der Schritt der Quellcodegenerierung für die Berechtigungsdurchsetzung wird in Kapitel 6.2.4 erläutert.

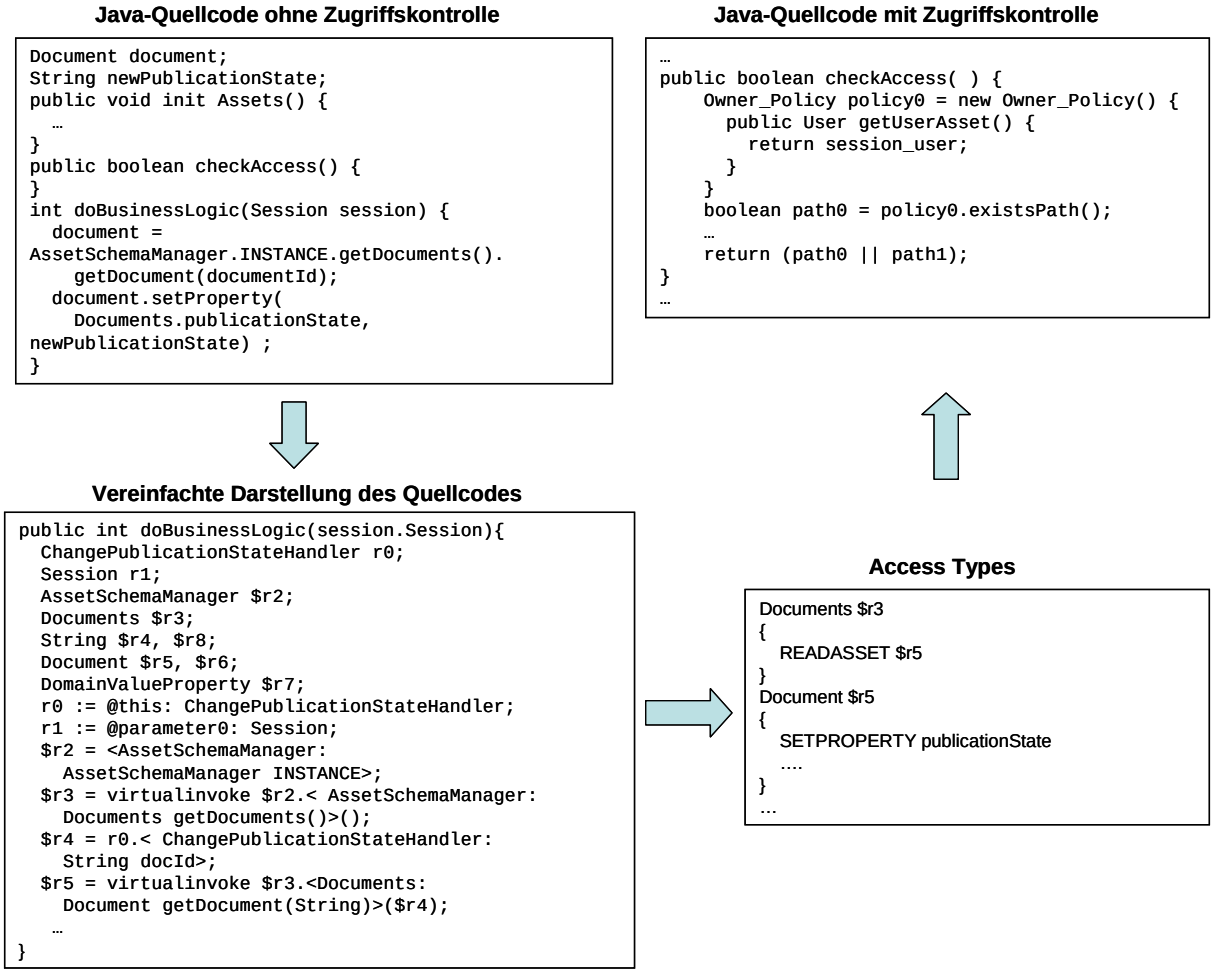


Abb. 6.23: Allgemeines Vorgehen für die Generierung von Berechtigungsdurchsetzungsfunktionen in vom Framework bereitgestellte Methodenrümpfe

6.2.2 Umwandlung des Quellcodes in eine vereinfachte Darstellung

Um die statische Quellcodeanalyse zu erleichtern, wird eine Drei-Adress-Code-Darstellung statt des Original-Java-Quellcodes untersucht. Das SOOT-Framework stellt hierfür die intermediäre Quellcodedarstellung *jimple* [Val00] bereit. Im Folgenden ist ein Beispiel für die Geschäftslogik des *Document-ChangePublicationStateHandler*s (siehe Abb. 6.24) als *jimple*-Code gegeben. Die angegebene jimple-Darstellung wurde dahingehend vereinfacht, dass nicht die gesamte Paketstruktur für jede Variable angegeben wurde, sondern diese gekürzt wurde, um die Lesbarkeit zu erhöhen.

```
Document document;
String newPublicationState;
int doBusinessLogic(Session session) {
  document = AssetSchemaManager.INSTANCE.getDocuments().
    getDocument(documentId);
  document.setProperty(
    Documents.publicationState, newPublicationState) ;
}

➔ Darstellung als jimple-Code:
public int doBusinessLogic(session.Session){
  ChangePublicationStateHandler r0;
  Session r1;
  AssetSchemaManager $r2;
  Documents $r3;
  String $r4, $r8;
  Document $r5, $r6;
  DomainValueProperty $r7;

  r0 := @this: ChangePublicationStateHandler;
  r1 := @parameter0: Session;
  $r2 = <AssetSchemaManager: AssetSchemaManager INSTANCE>;
  $r3 = virtualinvoke $r2.<AssetSchemaManager:
    Documents getDocuments()>();
  $r4 = r0.<ChangePublicationStateHandler: String docId>;
  $r5 = virtualinvoke $r3.<Documents:
    Document getDocument(String)>($r4);
  r0.<ChangePublicationStateHandler: Document document> = $r5;
  $r6 = r0.<ChangePublicationStateHandler: Document document>;
  $r7 = <Documents: DomainValueProperty asDomainValue_status>;
  $r8 = r0.<ChangePublicationStateHandler: String newPublicationState>;
  virtualinvoke $r6.<Document:
    void setProperty(DomainValueProperty, String)>($r7, $r8);
  return 0;
}
```

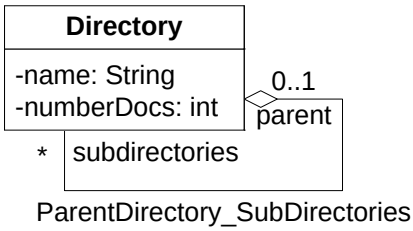
Abb. 6.24: Beispiel für die Umwandlung von Java-Quellcode in jimple-Code

In Java ist es möglich, Methodenaufrufe zu verketten. Im Drei-Adress-Code werden diese Methodenaufrufsequenzen in mehrere einfache Anweisungen herunter gebrochen und für die anonymen Objekte künstliche Variablennamen eingeführt. Jede Methode enthält für die implizite *this*-Variable und für alle Parameter eine lokale Variable, die durch ein vorangestelltes @-Zeichen identifiziert werden und jeweils eine Definitionsanweisung für diese eingeführten Variablen, die mit := notiert wird. Diese Definitionsanweisungen werden in jimple mit *IdentityStatement* bezeichnet. Alle anderen Definitionsanweisungen lokaler Variablen werden mit = dargestellt. Sie entstehen aus dem ursprünglichen Methodeninhalt. Die durch < > gekennzeichneten Identifier sind Felder der untersuchten Klasse.

Durch die Verwendung von jimple-Code reduziert sich die Anzahl zu untersuchender Anweisungstypen innerhalb eines Anweisungsblocks, die benötigt wird, um einfache *ServiceHandler*-Klassen, wie sie in Kapitel 6.1 vorgestellt wurden, statisch zu untersuchen, auf *IdentityStatements*, *AssignStatements*, *InvokeStatements*, *ReturnStatements, IfStatements* und *GotoStatements*. Das IdentityStatement wurde bereits erläutert. Ein AssignStatement ist eine einfache Zuweisung an ein Feld oder

eine lokale Variable. Auf der rechten Seite der Anweisung kann hierbei wiederum ein Feld oder eine lokale Variable stehen, es kann sich aber auch um Konstanten oder Methodenaufrufe handeln. In obigem Beispiel sind die Anweisungen, die die lokalen Variablen $r2, $r3, $r4, $r5, $r6, $r7 und $r8 definieren und die Anweisung, die dem Feld *document* eine lokale Variable zuweist, AssignStatements. Ein InvokeStatement ist ein Methodenaufruf ohne Rückgabewert. In obigem Bsp. ist die Anweisung, die den virtuellen Methodenaufruf auf $r6 darstellt, ein InvokeStatement. Methodenrümpfe mit Rückgabewert werden jeweils durch ein ReturnStatement beendet. Ein ReturnStatement gibt eine lokale Variable, eine Konstante oder *null* zurück.

Der Kontrollfluss von if-Anweisungen und while-Schleifen wird über goto-Anweisungen dargestellt. Die Methodenrümpfe bestehen dann jeweils aus Blöcken, die Anweisungen enthalten, die hintereinander auszuführen sind. Die einzelnen Blöcke sind über die Auswertung von einfachen if-Anweisungen miteinander verknüpft. In der Implementierung der *ServiceHandler* und der Response kommen häufig Iteratoren vor, die mit Hilfe einer *while*-Schleife durchlaufen werden. Die Übersetzung der Definition und des Durchlaufens eines Iterators soll im Folgenden am Beispiel der Basismethode *Asset.getAssociatedAssets( )* gezeigt werden. Gegeben sei ein Stück Quellcode, welches zu einem gegebenen Verzeichnis *dir* alle Unterverzeichnisse ausliest, indem alle mit diesem Verzeichnis über die Assoziation *ParentDirectory_SubDirectories* verbundenen Unterverzeichnisse geholt und in einem Iterator durchlaufen werden (siehe Abb. 6.25 und 6.26). Die while-Schleife in der jimple-Darstellung beinhaltet mehrere Anweisungsblöcke: den Anweisungsblock, der für die Auswertung der while-Bedingung benötigt wird und den Anweisungsblock, der den Schleifenrumpf bildet. In jimple werden die while-Bedingung durch ein *if* und der Kontrollfluss durch ein *goto*-Statement nachgebildet.




```
public class Directory extends Asset {
  public Iterator getSubdirectories() {
    return this.getAssociatedAssets(
      Directories.Parentdirectory_Subdirectories.subdirectories));
  }
}
```

Abb. 6.25: Die Assoziation ParentDirectory_SubDirectories des Assets Directory

➔ Verwendung eines Iterators:

```
dir = …;
Iterator it = dir.getSubdirectories();
while (it.hasNext()) {
  Directory d = (Directory) it.next();
  //use d
}
```

➔ Übersetzung in jimple-Code:

```
  r1 = …;
  r2 = virtualinvoke r1.<Directory: Iterator getSubdirectories()>();
  goto label1;
label0:
  $r4 = interfaceinvoke r2.<Iterator: Object next()>();
  $r5 = (Directory) $r4;
  r3 = (Directory) $r5;
  //use r3
label1:
  $z0 = interfaceinvoke r2.<java.util.Iterator: boolean hasNext()>();
  if $z0 != 0 goto label0;
```

Abb. 6.26: Übersetzung eines Iterators von Java-Quellcode in die jimple-Darstellung

Die Programmlogik erlaubt es, dass deklarierten Variablen mehrfach verschiedene Objekte bzw. primitive Werte zugewiesen werden. Die statische Quellcodeanalyse wird maßgeblich erleichtert, wenn jeder deklarierte Typ nur genau einmal definiert wird. Dieses wird durch die Umwandlung des jimple-Codes in eine Static Single Assignment (SSA) Form (siehe z. B. [Muc97]) gewährleistet. In SOOT hat diese intermediäre Darstellung den Namen *shimple*. Die Umwandlung von Drei-Adress-Code in seine SSA-Form erfordert das Einfügen von Phi-Anweisungen. Phi-Anweisungen werden immer dann benötigt, wenn die Initialisierung einer Variablen mit einem Wert in verschiedenen Kontrollflüssen der Anwendung stattfinden kann und die Variable nach Vereinigung der Kontrollflüsse weiter verwendet wird. Wenn eine lokale Variable zunächst deklariert wird, und dann in verschiedenen Zweigen des Kontrollflusses mit einem Wert oder Objekt belegt wird, führt dies im SSA-Code dazu, dass in den einzelnen Zweigen zwei verschieden benannte, lokale Variablen angelegt werden. Vereinigt sich der Kontrollfluss der verschiedenen Ausführungszweige wieder, so wird durch eine eingefügte Phi-Anweisung deutlich gemacht, dass jede weitere Verwendung der ursprünglichen einzigen Variablen auf mehrere Definitionen zurückzuführen ist. In Abb. 6.27 ist ein Beispiel für die Einführung einer Phi-Anweisung gegeben. In [Muc97] ist ein allgemeiner Algorithmus dargestellt, der Quellcode in seine SSA-Form überführt. Im Weiteren wird davon ausgegangen, dass das shimple-Format, nicht das jimple-Format, für die Analyse verwendet wird.

Ursprünglicher Code	**Code mit Phi-Anweisungen**

`Object o;`
`if ( some condition ) o = …;`
`else o = …;`
`… usage of o …`
	`if ( some condition ) Object o1 = …;`
	`else Object o2 = …;`
	`Object o3 = Phi (o1, o2);`
	`… usage of o3 …`

Abb. 6.27: Beispiel für die Aufspaltung von Variablennamen und die Verwendung eines PhiStatements

```
Statement :
    IdentityStatement | AssignStatement | InvokeStatement |
    ReturnStatement | IfStatement | GotoStatement

IdentityStatement :
    Local := IdentityRef

IdentityRef :
    this type | parameter type

AssignStatement :
    Local = Value; | FieldRef = Immediate;

InvokeStatement :
    InvokeExpression

ReturnStatement :
    return; | return Immediate;

IfStatement :
    if Condition GotoStatement;

Condition :
    boolean expression (statement comparing two Immediates)

GotoStatement :
    goto StatementRef;

StatementRef :
    reference to follow up Statement

Immediate :
    Local | Constant | null

Value :
    FieldRef  | Immediate | InvokeExpression | CastExpression |
    PhiExpression

FieldRef :
    Name

InvokeExpression :
    Local . MethodRef;

MethodRef :
    Name ( ( Parameter )* );

Parameter :
    Immediate

CastExpression :
    ( Name ) Immediate;

PhiExpression :
    Local = phi( List<Local> )

Type :
    Iterator<Type> | String | any primitive Java type |
    Asset | AssetSchema | Association | AssociationSchema |
    Role | Property

Name := Identifier
```

Abb. 6.28: Grammatik der zu untersuchenden Statement-Typen

In Abb. 6.28 ist die Grammatik der sechs zu untersuchenden Anweisungstypen gegeben. Es handelt sich dabei um eine Minimalsyntax, die notwendig ist, um die in Abschnitt 6.1 gegebenen einfachen Dienstbearbeiter zu analysieren. In einer realen Anwendung gibt es noch eine Vielzahl weiterer Anweisungstypen und Java-Syntaxelemente, die hier ignoriert werden. Beispielsweise werden die Ausnahmebehandlung, also Exceptions, throw-Anweisungen und try-catch-Blöcke, für die hier durchgeführte statische Quellcodeanalyse ignoriert. Die Ausnahmebehandlung greift nicht auf zugriffsgeschützte Ressourcen zu, bricht die Dienstausführung ab und ist somit für eine Analyse benötigter Berechtigungen für ein Stück Quellcode uninteressant. Für die statische Analyse wird davon ausgegangen, dass bei Dienstausführung alle Anweisungen durchgeführt werden (könnten), von daher werden try- und catch-Anweisungen nicht weiter berücksichtigt. Desweiteren kann es auch in der oben vorgestellten Minimalsyntax viele Anweisungen innerhalb der Geschäftslogik geben, die für die

Berechtigungskontrolle irrelevant sind und ausschließlich der Erfüllung der Geschäftslogik dienen. So ist z. B. die Bedingung (*Condition*) einer if-Anweisung nicht von Interesse, da sie nur Werte miteinander vergleicht aber keine Aufrufe auf Assets oder AssetSchemas vornimmt. Diese werden in der Drei-Adress-Code-Syntax, falls sie im Originalquellcode innerhalb der if-Bedingung vorkommen, der if-Anweisung vorangestellt. Um eine effiziente statische Quellcodeanalyse durchführen zu können, ist es notwendig, sich auf die Analyse derjenigen Elemente des Quellcodes zu beschränken, die im Zusammenhang mit der Zugriffskontrolle eine Rolle spielen. Alle anderen Quellcodeartefakte brauchen nicht in die Analyse einbezogen werden und der zu entwickelnde Analysealgorithmus kann diese Quellcodeelemente ausblenden. Dieses Ziel kann mit Hilfe der abstrakten Interpretation [NNH99, WM97, Cou01], die es ermöglicht, algorithmisch Informationen aus dem Quellcode zu erhalten, erreicht werden.

6.2.3 Statische Quellcodeanalyse

Die statische Quellcodeanalyse gliedert sich in zwei Phasen, innerhalb derer jeweils mehrere Analyseschritte durchgeführt werden. Die einzelnen Analyseschritte sind in Abb. 6.29 zusammengefasst. Sie werden im Folgenden einzeln erläutert.

Vorbereitungsphase: Intraprozedurale Analyse

1. Festlegen zu untersuchender Pakete der Anwendung
2. Eliminieren von AssignStatements mit CastExpressions
3. Eliminieren doppelter Definitionen von Singletons
4. Auffinden nicht relevanter Anweisungen
5. Auffinden der Quellcodeartefakte, die die Aktionen der Berechtigungen darstellen und Erstellung primitiver AccessTypes
6. Verarbeitung von Anweisungen mit PhiExpressions

Durchführungsphase: Interprozedurale Analyse

1. Zusammenführen von AccessTypes über Methodengrenzen hinweg
2. Erstellen von AccessTypes für die Felder der ServiceHandler-Klassen

Abb. 6.29: Phasen und Analyseschritte der statischen Quellcodeanalyse

Vorbereitungsphase: Intraprozedurale Analyse

In einem ersten Schritt werden die relevanten Pakete der zu untersuchenden Anwendung festgelegt. Diese enthalten den Quellcode, der berechtigungsrelevante Anweisungen aufruft. Diese berechtigungsrelevanten Anweisungen sind die Aktionen der Berechtigungen, die durch die Basisfunktionalitäten auf den Assets und AssetSchemas abgebildet werden, wie in Kapitel 5.3.2 und 5.3.3 beschrieben. Die zu untersuchende betriebliche Anwendung ist in einer Schichtenarchitektur implementiert, d. h. die jeweils höhere Schicht greift zur Erfüllung ihrer Funktionalität auf Klassen und Methoden derselben oder der direkt unter ihr liegenden Schicht zu. Es gibt keine Zugriffe von einer weiter unten liegenden Schicht auf eine weiter oben liegende Schicht. Wird nun der Aufrufgraph (siehe z. B. [Muc97]) eines Dienstbearbeiters gebildet, so ist die Wurzel des Aufrufgraphen durch die *doBusiness-Logic( )*-Methode gegeben. (Hier wird davon ausgegangen, dass eine bestehende Anwendung untersucht wird, deren Struktur so ist, wie in Kapitel 3 beschrieben. Die Aufspaltung der *doBusiness-Logic( )*-Methode, wie in Kapitel 6.1 erläutert, kann durch ein einfaches Refactoring der Anwendung automatisiert stattfinden. Die Assets, welche in der *initAssets()*-Methode definiert werden müssen, werden durch die Quellcodeanalyse festgestellt. In der vorliegenden Arbeit wird kein automatisiertes Umschreiben des Quellcodes angestrebt. Der Algorithmus soll dem Entwickler aber Informationen darüber liefern, welche AssetSchemas und Assets in der *initAssets()*-Methode zu definieren sind. Das

tatsächliche Umschreiben des Quellcodes ist für diese Methode vom Entwickler vorzunehmen. Ist die *initAssets( )*-Methode bereits gefüllt, so kann auch die *handleRequest( )*-Methode eines *Service-Handlers* als Wurzel für den Aufrufgraph verstanden werden.) Von der Wurzelmethode aus werden nur Methoden der Dialogsteuerung aufgerufen, die wiederum auf Methoden der Geschäftslogikschicht zugreifen. Die Geschäftslogikschicht greift auf die Persistenzabstraktion zu, auf der die Basisfunktionalitäten definiert sind. Der Aufrufbaum kann genau dort abgeschnitten werden, wo solch eine Basisfunktionalität aufgerufen wird, da die betriebliche Anwendung so realisiert ist, dass die Basisfunktionalitäten zur Erfüllung ihrer Aufgaben selbst nicht auf andere Basisfunktionalitäten zugreifen, sondern nur auf die Query-API, um eine Datenbankanfrage zu stellen. Die Blätter des Aufrufgraphen sind also zum einen die Basismethoden auf den Asset- und AssetSchema-Klassen. Zum anderen wird für die Realisierung eines Dienstes auch auf die Klassen der Java-Runtime zurückgegriffen. Der Aufruf einer Methode auf einer Klasse der Java-Runtime braucht ebenfalls nicht weiter untersucht werden, da er nur weitere Methoden der Java-Runtime aufruft. Es gibt hier keine Call-back-Methoden, die aus einer Klasse der Java-Runtime heraus eine Klasse der Anwendung aufrufen. Eine Ausnahme bildet hier die Klasse Iterator, da diese Asset-Objekte als Suchergebnisse enthalten können. In Abb. 6.30 ist die Struktur des Aufrufgraphen bildlich dargestellt. Die einzelnen Wurzeln (entry points) sind die Eintrittspunkte für die Dienstausführung. Die Blätter (leaf methods) sind die Aktionen. Von den Blattmethoden aus aufgerufene weitere Methoden können die Tiefe des Aufrufgraphen nur noch erhöhen. Zugriffe von den Blattmethoden oder von unterhalb der Blattmethoden auf höher gelegene Methoden sind in der geforderten Schichtenarchitektur für betriebliche Anwendungen ausgeschlossen. Aus diesen Überlegungen ergibt sich, dass sich die statische Quellcodeanalyse nur über die Pakete der Dialogsteuerung und der Geschäftslogikschicht, sowie innerhalb der Persistenzabstraktion nur über die Klassen Asset und AssetSchema erstrecken muss. Alle weiteren Pakete, insbesondere Pakete der Java-Runtime brauchen nicht weiter untersucht werden. Eine Ausnahme bildet hier nur, wie oben erläutert, die Iterator-Klasse aus dem java.util-Paket.

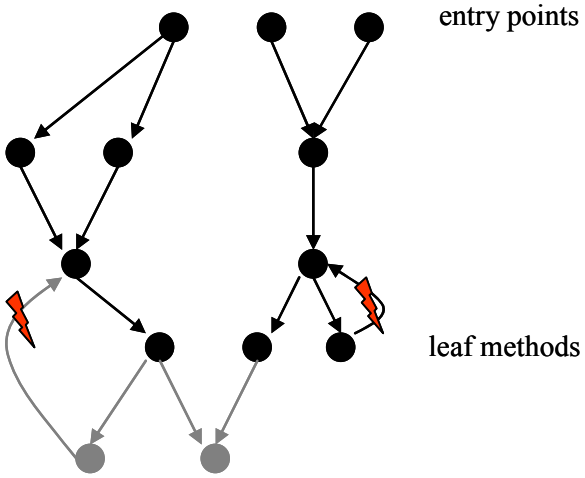


Abb. 6.30: Struktur des Aufrufgraphen der Dienstbearbeiter

Sind die zu untersuchenden Pakete festgelegt, so werden in einem zweiten Schritt, alle Assign-Statements eliminiert, die eine CastExpression enthalten. Die lokale Variable, welche durch die Cast-Expression definiert wird, definiert dann die rechte Seite des vorangegangenen AssignStatements. Hierdurch wird die Anzahl der zu analysierenden Anweisungen verringert. Es ist nicht notwendig, aus der shimple- bzw. der jimple-Darstellung wieder lauffähigen Quellcode zu erzeugen, deshalb kann diese Vereinfachung gemacht werden. Sie wird später das Erstellen der AccessTypes vereinfachen. Ein Beispiel für das Eliminieren der CastExpressions ist in Abb. 6.31 gegeben.

Ursprünglicher Code:

```
$r4 = interfaceinvoke r2.<Iterator: Object next()>();
$r5 = (Directory) $r4;
r3 = (Directory) $r5;
```

➔ Quellcode nach Eliminieren der ersten CastExpression

```
$r5 = interfaceinvoke r2.<Iterator: Object next()>();
r3 = (Directory) $r5;
```

➔ Quellcode nach Eliminieren der zweiten CastExpression

```
r3 = interfaceinvoke r2.<Iterator: Object next()>();
```

Abb. 6.31: Eliminieren von CastExpressions

In einem dritten Schritt werden überflüssige Variablen und AssignStatements entfernt. In realem Quellcode werden zum Zugriff auf Assets häufig verkettete Methodenaufrufe implementiert. Diese holen zunächst den AssetSchemaManager, dann das entsprechende AssetSchema, um über dieses auf die einzelnen Assets zuzugreifen. Bei der Umwandlung dieses Quellcodes werden doppelte Variablen für den AssetSchemaManager und das verwendete AssetSchema angelegt. Es besteht aber das semantische Wissen, dass diese Klassen mit dem Singleton-Muster implementiert sind, d. h. es kann nur eine einzige Instanz geben. Von daher können alle Anweisungen, die den AssetSchemaManager und die AssetSchemas zum wiederholten Mal definieren, aus dem jimple- bzw. shimple-Code eliminiert werden, indem alle Verwendungen der Variablen der wiederholten Definition durch Verwendungen der Variablen der ersten Definition ersetzt werden (siehe Abb. 6.32). Weiterhin können die Definitionen der Singletons immer so verschoben werden, dass sie als erste Anweisungen innerhalb einer Methode auftauchen, da sie von keiner anderen Anweisung abhängen. Hierdurch wird die Analyse des Quellcodes maßgeblich erleichtert.

In einem vierten Schritt können alle Anweisungen aller in den Klassen der Pakete vorhandenen Methodenrümpfe mit Hilfe der abstrakten Interpretation in zwei disjunkte Mengen eingeteilt werden: die Anweisungen, die für die Analyse relevant sind, und die Anweisungen, die irrelevant sind und von daher für die Analyse aus dem zu untersuchenden Quellcode entfernt werden können. Nicht relevant sind alle AssignStatements und InvokeStatements, die einen Methodenaufruf enthalten, dessen Basis-Variable, − diese enthält das Objekt, auf welchem die Methode aufgerufen wird −, in einer Klasse definiert ist, die zu einem Paket gehört, das nicht untersucht wird. Weiterhin sind alle AssignStatements und ReturnStatements nicht von Bedeutung, wenn sie einen Typ definieren bzw. verwenden, der nicht in den zu untersuchenden Anwendungspaketen definiert wird. Alle Zuweisungen, die ein Feld definieren oder verwenden, sind für die intraprozedurale Analyse ebenfalls nicht relevant. Der jimple-Code generiert für jedes Feld, welches innerhalb einer Methode verwendet wird, auch eine lokale Variable, die dieses Feld repräsentiert. Felder werden erst am Ende der interprozeduralen Analyse berücksichtigt und für die intraprozedurale Analyse ignoriert. Weiterhin werden alle Anweisungen ignoriert, die ein neues Objekt erstellen, also einen Konstrukor-Aufruf enthalten. Bei diesen Anweisungen kann es sich nur um Konstruktoren von Objekten der Java-Runtime handeln, aber nicht um die interessanten Objekte vom Typ Asset oder AssetSchema, da ja deren Erstellung in einer Methode von AssetSchema gekapselt wird. Außerdem sind alle AssignStatements nicht von Bedeutung, die eine PhiExpression enthalten, die Variablen von einem Typ der Java-Runtime, z. B. eine int-Variable, zusammenfassen. PhiExpressions, die Asset-Objekte zusammenfassen werden im fünften Schritt des intraprozeduralen Algorithmus behandelt. In Bezug auf Kontrollflussanweisungen werden nur Iteratoren mit anschließenden while-Statements berücksichtigt, welche Suchergebnisse aus Queries abarbeiten. Für if-Anweisungen im Quellcode wird angenommen, dass potentiell alle Kontrollflüsse durch eine Dienstabarbeitung ausgeführt werden können und daher für die Generierung einer Berechtigungspolicy alle berücksichtigt werden müssen. Für die in Kapitel 6.1 angeführten einfachen Dienstimplementierungen ist diese Vereinfachung gerechtfertigt, da hier keine if-Anweisungen vor-

kommen. Falls doch if-Anweisungen im Quellcode vorkommen, ist davon auszugehen, dass für die Ausführung aller Kontrollflusszweige ähnliche Berechtigungen notwendig sind. Ein Dienstbearbeiter, der zwei komplett verschiedene Dienste innerhalb einer *doBusinessLogic( )*-Methode anbietet, z. B. ein Dienstbearbeiter, der je nach übergebenen Parametern ein Geschäftsobjekt anlegt oder ein vorhandenes Geschäftsobjekt aktualisiert, ist in zwei Dienstbearbeiter aufzuspalten. Von Schleifen und rekursiven Methodenaufrufen innerhalb der Programmlogik kann ebenfalls abstrahiert werden. Wenn innerhalb jeden Schleifendurchlaufs bzw. Rekursionsdurchlaufs immer dieselben Assets verarbeitet werden, so ist, wenn eine Aktion auf einem Asset einmalig erlaubt ist, auch das mehrmalige Hintereinanderausführen derselben Aktionen erlaubt. Werden innerhalb eines jeden Schleifen- oder Rekursionsdurchlaufs Assets erstellt, so braucht die Berechtigung zum Erstellen eines Assets auch nur einmalig zu Beginn überprüft werden, da diese ja unabhängig von dem zu erstellenden Asset muss (vgl. S. 96). Das Aktualisieren und Löschen mehrerer Assets findet in while-Schleifen statt. Die Assets werden zunächst über eine Suchanfrage geholt und in einem Iterator gekapselt. Diese Art von while-Schleifen muss für die statische Quellcodeanalyse berücksichtigt werden, insbesondere, weil es notwendig sein kann, die Abarbeitung der while-Schleife in der *initAssets( )*-Methode zu rekonstruieren (vgl. D-Dienst Abb. 6.8, S. 100).

Quellcode zum Zugriff auf Assets:

```
Document doc1 = AssetSchemaManager.INSTANCE.getDocuments().getDocument(docId_1);
Document doc2 = AssetSchemaManager.INSTANCE.getDocuments().getDocument(docId_2);
```

➔Umsetzung im jimple-Code:

```
AssetSchemaManager $r4, $r7;
Documents $r5, $r8;
String $r6, $r9;
Document r3, r4;
$r4 = <AssetSchemaManager: AssetSchemaManager INSTANCE>;
$r5 = virtualinvoke $r4.<AssetSchemaManager: Documents getDocuments()>();
$r6 = …;
r3 = virtualinvoke $r5.<Documents: Document getDocument(String)>($r6);
$r7 = <AssetSchemaManager: AssetSchemaManager INSTANCE>;
$r8 = virtualinvoke $r7.<AssetSchemaManager: Documents getDocuments()>();
$r9 = …;
r4 = virtualinvoke $r8.<Documents: Document getDocument(String)>($r9);
```

➔ jimple-Code nach Kürzung der doppelten Definitionen für die Singletons

```
AssetSchemaManager $r4;
Documents $r5;
String $r6, $r9;
Document r3, r4;
$r4 = <AssetSchemaManager: AssetSchemaManager INSTANCE>;
$r5 = virtualinvoke $r4.<AssetSchemaManager: Documents getDocuments()>();
$r6 = …;
r3 = virtualinvoke $r5.<Documents: Document getDocument(String)>($r6);
$r9 = …;
r4 = virtualinvoke $r5.<Documents: Document getDocument(String)>($r9);
```

Abb. 6.32: Zugriff auf Assets über den AssetSchemaManager

In einem fünften Schritt werden die in den Berechtigungspolicies der einzelnen Assets- und Asset-Schema-Typen definierten Aktionen auf Methodendefinitionen der Anwendung abgebildet. In dem hier vorgestellten Framework bilden alle Methodendefinitionen von Asset und AssetSchema Aktionen, die in einer Berechtigung verwendet werden können. Diese Methodendefinitionen werden im Quellcode entsprechend annotiert, so dass sie durch eine statische Analyse leicht gefunden werden können. In Abb. 6.33 ist ein Beispiel für die annotierten Basismethoden der Klasse AssetSchema gegeben. Die Basismethoden der Klasse Asset werden analog annotiert. Jetzt kann die Codeanalyse alle Anweisungen auffinden, in denen die vorher annotierten Methodendefinitionen verwendet werden. Es kann ein neuer Aufrufgraph gebildet werden, dessen Blätter entweder einer Aktion entsprechen oder aber einen Methodenaufruf enthalten, der für die Analyse irrelevant ist.

```
public abstract class AssetSchema {
  @Action(„READASSET“)
  public Asset getAsset(String id) { … }

  @Action(“READASSET”)
  public Iterator<Asset> getAssets() { … }

  @Action(“CREATEASSET”)
  public Asset createAsset() { … }

  @Action(“DELETEASSET”)
  public boolean remove(String id) { … }

 @Action(“QUERYASSET”)
  public Iterator<Asset> queryAssets(Query a) { … }
}
```

Abb. 6.33: Annotationen der Basismethoden der AssetSchema-Klasse

Die statische Codeanalyse identifiziert jetzt für alle zu untersuchenden Methoden und für alle gefundenen Aktionen, das Objekt, auf dem die Anweisung ausgeführt wird und erweitert dieses Objekt um Typinformationen. Bei diesen Objekten handelt es sich um konkrete Geschäftsobjekte vom Typ Asset und konkrete Containerobjekte vom Typ AssetSchema. Die Typinformationen werden im Folgenden mit *AccessTypes* bezeichnet. Allgemein erweitern AccessTypes das Java-Typsystem um Zugriffsinformationen. AccessTypes, welche Feldzugriffe speichern, wurden in [LT06] behandelt. Dort wurde gezeigt, wie diese in ein Subset von Java einzubetten sind. Um aus den AccessTypes die Berechtigungsdurchsetzungsfunktionen generieren zu können, müssen die AccessTypes so erweitert und verändert werden, dass alle in Kapitel 3 identifizierten Aktionen für die Asset- und AssetSchema-Klassen in den AccessTypes modelliert werden können. Insbesondere müssen Aktionen, die Geschäftsobjekte anlegen und löschen in dieser Datenstruktur protokolliert werden. Die erweiterte Syntax der AccessTypes ist in Abb. 6.34 dargestellt.

```
AccessType :
  Type Name { ( Annotation )* }
Annotation :
  PropertyAnnotation | AssetAnnotation | AssociationAnnotation |
  IteratorNextAnnotation
PropertyAnnotation :
  PropertyActionName ( PropertyName )
AssetAnnotation :
  AssetActionName AccessType
AssociationAnnotation :
  AssociationActionName ( RoleName ) [ [Parameter] ] [Connection]
Parameter :
  AccessType
Connection :
  AccessType
IteratorNextAnnotation:
  NEXT [ Connection ]
PropertyActionName :
  GETPROP | SETPROP
AssetActionName :
  READASSET | CREATEASSET | DELETEASSET | QUERYASSET | NEXT
AssociationActionName :
  READASSET | CREATEASSOC | DELETEASSOC
PropertyName :
  name of property
RoleName :
  association name . role name of association
Type :
  any Java type declared in information system
Name :
  any identifier
```

Abb. 6.34: Syntax der AccessTypes

Jedem im Quellcode vorkommenden Asset- und AssetSchema-Objekt wird ein AccessType zugeordnet. Feldzugriffe auf Felder primitiven Datentyps oder des String-Typs werden in jeweils einer *PropertyAnnotation* festgehalten. Der Name der Aktion ist hier entweder GETPROP für das Lesen einer Property oder SETPROP für das Schreiben desselben. Dieses entspricht den Basismethoden *Asset.getProperty( )* und *Asset.setProperty( )* (vgl. Kap. 3, Abb. 3.8).

Felder nicht-primitiver Datentypen, die selbst wieder ein Geschäftsobjekt oder eine Menge von Geschäftsobjekten darstellen, werden gemäß dem in 3.1.6 vorgestellten Datenmodell für betriebliche Anwendungen als Assoziationen modelliert, d. h. Zugriffe auf die Felder nicht-primitiven Datentyps werden nicht in der PropertyAnnotation sondern in der *AssociationAnnotation* behandelt. Die Basismethode *Asset.getAssociatedAssets( )* wird auf READASSET abgebildet. Hierbei sind der Name der Rolle zusammen mit dem Assoziationsnamen anzugeben, über den das Asset gelesen wird und die lokale Variable, die das oder die gelesenen Assets darstellt, als verbundener AccessType (*Connection*). Die Angabe des Assoziationsnamens und der Rolle sind notwendig, um das oder die gelesenen Assets eindeutig identifizieren zu können. Des Weiteren können unter Angabe des Assoziationsnamens und einer Rolle auch Assoziationen zwischen zwei Geschäftsobjekten angelegt werden. Diese Aktion wird mit CREATEASSOC und dem jeweiligen anderen Asset, dem Parameter-Objekt, zu dem ein *Parameter*-AccessType existieren sollte, notiert. Die Basismethode *Asset.createAssociation( )* entspricht dem *AssociationActionName* CREATEASSOC. Für die Aktion des Löschens einer Assoziation, DELETEASSOC, muss wieder der Name der Assoziation angegeben werden, die gelöscht werden soll. Das mit der Assoziation verbundene Asset bzw. die mit der Assoziation verbundenen Assets auf der anderen Seite der Assoziation können als verbundener AccessType (Connection) angegeben werden. Die

Basisfunktionen *Asset.removeAssociation( )* sowie *Asset.removeAssociations( )* entsprechen DELETEASSOC.

Die *AssetAnnotation* beschreibt mit READASSET das Lesen, mit CREATEASSET das Anlegen und mit DELETEASSET das Löschen eines Geschäftsobjekts, also die Basismethoden *AssetSchema.getAsset( ), AssetSchema.getAssets( )* sowie *AssetSchema.createAsset( )* und *AssetSchema.remove( )* respektive. Die Annotation DELETEASSET erfordert einen Parameter-AccessType, der angibt, welches Asset gelöscht wird. Die Annotationen READASSET und CREATEASSET erfordern die Angabe eines Connection-AccessTypes, der angibt, welches Asset gelesen bzw. angelegt wird. QUERYASSET steht für *AssetSchema.queryAssets( )*. Welches Asset bzw. welche Assets gelesen, angelegt oder gelöscht werden, ist in dem AccessType der jeweiligen Annotation beschrieben. Die AssetAnnotationen werden demjenigen Java-Objekt zugeordnet, welches die Assets des in der Annotation angegebenen Typs verwaltet, also der jeweiligen AssetSchema-Klasse.

Die Annotation NEXT ist zum Festhalten eines *Iterator.next( )*-Aufrufs gedacht und wird für alle Basisfunktionalitäten benötigt, die nicht ein Asset, sondern eine Menge von Assets zurückliefern.

Durch die Angabe der Connections, können nicht nur Zugriffskontrollinformationen festgehalten werden, sondern auch Informationen darüber, wie die einzelnen Assets, die innerhalb der Anwendung deklariert und verwendet werden, miteinander zusammenhängen, d. h. über welche Assoziationen und Rollen die einzelnen Assets miteinander verbunden sind. Diese Information ist notwendig, um komplexe Berechtigungsdurchsetzungsfunktionen zu generieren. Die Struktur der AccessTypes soll im Folgenden an einem Beispiel erläutert werden.

Gegeben sei ein AssetSchema *documents*, welches Assets vom Typ *Document* verwaltet. Weiterhin sei im Datenmodell die Asset-Klasse *Document* über eine Assoziation *Document_Directory* mit einer Klasse vom Typ *Directory* verbunden, die angibt, in welchem Verzeichnis sich das jeweilige Dokument befindet (siehe Abb. 6.35).

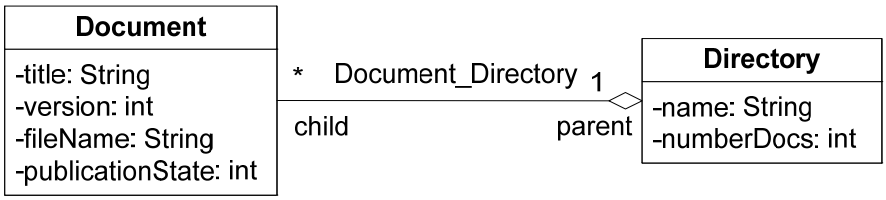


Abb. 6.35: Beispieldatenmodell für ein Verzeichnis und die in einem Verzeichnis enthaltenen Dokumente

Wird jetzt ein neues Dokument angelegt, so wird die Assoziation zu seinem Parent-Directory ebenfalls erzeugt. In dem Parent-Directory wird die Anzahl der enthaltenen Dokumente erhöht (Attribut: *numberDocs*). Im Folgenden ist ein Quellcode Beispiel gegeben (siehe Abb. 6.36), welches einen vereinfachten Auszug aus der Geschäftslogik eines C-Dienstes zum Anlegen eines Dokuments darstellt. Zunächst wird aus den Anfrageparametern, die dem Dienstbearbeiter übergeben werden, bestimmt, in welchem Verzeichnis das neue Dokument erstellt werden soll; danach kann es kreiert werden. Das Erstellen eines neuen Dokuments ist in der AssetSchema-Klasse *Documents* der Geschäftslogikschicht gekapselt. Hier werden alle notwendigen Properties und Assoziationen des neuen Assets gesetzt.

```
public class DocumentCreateServiceHandler extends ServiceHandler {
  String directoryId;
  String title;
  … //other variables for other properties
  void getParameters(String[] parameters) {
    directoryId = …;
    title = …;
    …
  }
  int doBusinessLogic(Session session) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    Directories dirs = asm.getDirectories();
    Directory dir = dirs.getDirectory(directoryId);
    Documents docs = asm.getDocuments();
    Document d = docs.createDocument(dir);
    doc.setProperty(Documents.title, „…“);
    …
  }
}

public class Documents extends AssetSchema {
  …
  public Document createDocument(Directory dir) {
    Document doc = this.createAsset();
    doc.createAssociation(Documents.Document_Directory.parent, dir);
    int numberDocs = dir.getProperty(Directories.numberDocs) + 1;
    dir.setProperty(Directories.numberDocs, numberDocs);
    return doc;
  }
  …
}
```

Abb. 6.36: Beispielquellcode für das Anlegen eines Dokuments

Die einfachen AccessTypes, die im fünften Schritt der Vorbereitungsphase der Quellcodeanalyse erstellt werden sind für den gezeigten Quellcode Ausschnitt in Abb. 6.37 gegeben. Zur Erhöhung der Lesbarkeit werden die Variablennamen, wie sie im Quellcode Ausschnitt benannt wurden, verwendet. Das Erstellen der AccessTypes erfolgt im eigentlichen Algorithmus natürlich auf Basis des jimple- bzw. shimple-Codes und der dort eingeführten Variablennamen. Die lokalen Variablen der Methode *doBusinessLogic( )* enthalten keine AccessTypes mit Ausnahme des Dokuments *d*, dessen Property *title* gesetzt wird. Die lokalen Variablen des Methodenrumpfes der Methode *createDocument( )* enthalten mehrere AccessTypes. Der Methodenaufruf *createAsset( )* korrespondiert mit der Aktion CREATEASSET des AccessTypes. Die Methoden *getProperty( )* und *setProperty( )* korrespondieren mit den Aktionen GETPROP und SETPROP. Es wird jeweils die Property *numberDocs* der lokalen Variable *dir* vom Typ *Directory* gelesen bzw. gesetzt. Des Weiteren wird eine Assoziation zwischen dem bestehenden Verzeichnis *dir* und dem neu hinzugekommenen Dokument *doc* erstellt.

- Einfache AccessTypes für die *doBusinessLogic(Session session)*-Methode:

```
Document d {
  SETPROP title
}
```

- Einfache AccessTypes für die *Documents.createDocument(Directory dir)*-Methode:

```
Documents this {
  CREATEASSET [doc]
}
Document doc {
  CREATEASSOC Documents.Document_Directory.parent [dir]
}
Directory dir {
  GETPROP Directories.numberDocs
  SETPROP Directories.numberDocs
}
```

Abb. 6.37: Einfache AccessTypes für den DocumentCreateServiceHandler aus Abb. 6.36

Als zweites Beispiel werden die AccessTypes, die innerhalb eines Iterators vorkommen, erläutert. Gegeben sei der unter Abb. 6.38 gegebene Quellcode, welcher zu einem gegebenen Verzeichnis *dir* alle Dokumente holt.

```
Directory dir = …;
Documents documents = AssetSchemaManager.INSTANCE.getDocuments();
Iterator<Document> documentsInDir =
    documents.getDocumentsOfDirectory(dir);
while(documentsInDir.hasNext() {
    Document doc = (Document) documentsInDir.next();
   //use doc;
}
```

Abb. 6.38: Beispielquellcode für das Auslesen von Assets aus einem Iterator

Die aus diesem Stück Quellcode extrahierten AccessTypes haben die in Abb. 6.39 gezeigte Gestalt. Hier wurden wieder zur Erhöhung der Lesbarkeit der AccessTypes die lokalen Variablennamen wie sie im Beispielquellcode vorkommen für die AccessTypes genommen, nicht die von jimple bzw. shimple erzeugten Variablennamen. In den AccessTypes wird festgehalten, dass ein *Iterator.next( )*-Aufruf auf einer lokalen Variablen stattfindet.

```
Documents documents { }
Directory dir { }
Iterator<Document> documentsInDir {
  NEXT doc;
}
Document doc {
  //annotations for the usage of doc
}
```

Abb. 6.39: Einfache AccessTypes für einen Iterator

Die statische Codeanalyse wird als rückwärtsgerichtetes Verfahren realisiert, d. h. sie startet bei den Blättern des zu untersuchenden Aufrufbaums und endet bei den Wurzeln. In fünften Schritt der Vorbereitungsphase werden alle Methoden auf die Verwendungen der Aktionen hin untersucht. Für jede identifizierte Aktion, d. h. für jeden Methodenaufruf einer Basisfunktionalität wird das Objekt identifiziert, auf dem die Basisfunktionalität ausgeführt wird. Existiert für die lokale Variable, welche das Geschäftsobjekt repräsentiert noch kein AccessType, so wird ein neuer erstellt und die gefundene Aktion dem AccessType hinzugefügt. Existiert der AccessType bereits, so wird lediglich die neue Aktion hinzugefügt. Einige Aktionen haben weitere AccessTypes als Parameter oder Connections. Existieren diese verbundenen AccessTypes für die jeweilige lokale Variable noch nicht, so müssen diese ebenfalls erstellt werden. Um welche lokalen Variablen es sich handelt, geht eindeutig aus der Auswertung der Anweisung hervor, die die Aktion aufruft. In diesem Analyseschritt wird lediglich eine intraprozedurale Analyse durchgeführt, d. h. jeder Methodenrumpf wird für sich betrachtet und nur, wenn innerhalb einer Methode mehrere Aktionen für ein und dasselbe lokale Objekt gefunden werden, können diese in einem AccessType zusammengefasst werden. In diesem Analyseschritt werden noch keine Felder annotiert, da sie zu einer Klasse und nicht zu einer Methode gehören. In jimple bzw. in shimple werden für jedes Feld, dem impliziten this-Objekt und für alle übergebenen Parameter eigene lokale Variablen erstellt. In diesem Analyseschritt werden diese lokalen Variablen verwendet, um die AccessTypes zu erstellen.

In einem sechsten und letzten Schritt der Vorbereitungsphase werden die Anweisungen mit PhiExpressions, die durch die Verwendung des shimple-Codes in den jimple-Code eingefügt werden, verarbeitet. Nach dem fünften Schritt der Vorbereitungsphase existieren für alle Objekte eines konkreten Asset- oder AssetSchema-Typs einfache AccessTypes. Eine in der intermediären Quellcodedarstellung eingefügte PhiExpression bedeutet, dass sich der Kontrollfluss zweier unterschiedlicher Objekte wieder vereint, für die im ursprünglichen Code nur ein Variablenname vorgesehen war. Bezieht sich die PhiExpression jetzt auf ein Asset- oder AssetSchema-Typ, so werden der AccessType, welcher durch die lokale Variable, die die PhiExpression definiert mit den AccessTypes zusammengefasst, die die PhiExpression zusammenfasst (siehe Abb. 6.40). Für die spätere interprozedurale Analyse, die die AccessTypes über Methodengrenzen hinweg verschiebt, ist es notwendig, sich die Information zu merken, ob ein AccessType zu einer PhiExpression gehört. Da dann auch die in der PhiExpression zusammengefassten AccessTypes verschoben werden müssen.

➔ shimple-Code Fragment

```
Asset a1 = …;
…
Asset a2 = …;
…
Asset a3 = Phi (o1, o2);
…
```

➔ vorhandene einfache AccessTypes

```
Asset a1 {Annotation A}
Asset a2 {Annotation B}
Asset a3 {Annotation C}
```

➔ zusammengefasste AccessTypes

```
Asset a1 {Annotation A, Annotation C} [from phi of a3]
Asset a2 {Annotation B, Annotation C} [from phi of a3]
Asset a3 { }
```

Abb. 6.40: Zusammenfassung der AccessTypes von PhiExpressions

Im Folgenden wird der Algorithmus der Vorbereitungsphase als Pseudocode gegeben (siehe Abb. 6.41). Die Gesamtheit aller Klassen, die für den Algorithmus betrachtet werden, wird hier mit *Scene* bezeichnet. Der Algorithmus annotiert alle Anweisungen, die für die Berechtigungsüberprüfung irrelevant sind, mit einer *is-out-of-scope*-Annotation. Alle Anweisungen, welche Aktionen verwenden, erhalten eine *is-processed*-Annotation. Für die lokalen Variablen, die durch diese Anweisungen definiert werden, werden einfache AccessTypes erstellt. In einem zweiten Durchlauf werden alle PhiExpressions gesucht, deren AccessTypes verschoben und die dazugehörige Anweisung ebenfalls mit einer *is-processed*-Annotation versehen. Für den sich anschließenden interprozeduralen Algorithmus müssen dann nur noch solche Anweisungen betrachtet werden, die noch keine Annotation erhalten haben.

```
//intraprocedural analysis
for (all classes c in scene) {
  for (all methods m in c) {
    for (all statements s in m) {
      if(s.isOutOfScope)
        s.addAnnotation(is-out-of-scope);
      else if (s.containsAction()) {
        local l = findDefinitionLocalOf(s);
        if( !l.hasAccessType())
          l.createEmptyAccessType();
        AccessType at = l.getAccessType();
        //adding an action may involve creation of parameter
        //and connection access types
        at.addAnnotation(s.getAction());
        s.addAnnotation(is-processed);
      }
    }
  }
}
for (all classes c in scene) {
  for (all methods m in c) {
    for (all statements s in m) {
      if(! s.containsAnnotation(is-out-of-scope) &&
           s.containsPhiExpression()) {
         local l = findDefinitionLocalOf(s);
         AccessType at = l.getAccessType();
         for (all local args k in phi expression)
           k.getAccessType().addAnnotationsOf(at);
         s.addAnnotation(is-processed);
      }
    }
  }
}
```

Abb. 6.41: Algorithmen der Vorbereitungsphase als Pseudocode

Durchführungsphase: Interprozedurale Analyse

In der Durchführungsphase werden die in der Vorbereitungsphase aufgestellten einfachen AccessTypes entlang des Aufrufgraphen nach oben geholt bis die lokalen Variablen und schließlich die Felder der *ServiceHandler*-Klassen mit AccessTypes annotiert werden können. AccessTypes, die auf Variablen agieren, die dasselbe Objekt darstellen, können zusammengefasst werden. Es gibt für die interprozedurale, also die methodenübergreifende, Analyse drei Möglichkeiten, über die lokale Variablen zur Laufzeit dasselbe Objekt repräsentieren können:

- Eine Methode wird auf einem Objekt aufgerufen, innerhalb der aufgerufenen Methode handelt es sich hierbei um die lokale Variable, die das implizite this-Objekt identifiziert.
- Ein Methodenaufruf übergibt eine Objektreferenz über die Parameter-Objekte an die aufgerufene Methode. Innerhalb der aufgerufenen Methode gibt es eine lokale Variable für jeden Parameter.
- Eine Methodenimplementierung gibt ein Objekt zurück. Dieses Objekt ist identisch mit dem Objekt, welches bei dem Aufruf der entsprechenden Methode definiert wird.

Die AccessTypes der entsprechenden lokalen Variablen können dann zusammengefasst werden, d. h. die Annotationen der AccessTypes, die im Aufrufgraphen weiter unten definiert sind, können zu den AccessTypes, die weiter oben definiert sind, hinzugefügt werden.
Dieses soll am Beispiel der unter Abb. 6.36 (siehe S. 127) gegebenen *doBusinessLogic( )*-Methode erläutert werden. Die Implementierung der *doBusinessLogic( )*-Methode enthält eine Anweisung, die auf Basis eines konkreten Verzeichnisses *dir* ein neues Dokument *doc* erstellt.

```
Document d = docs.createDocument(dir);
```

Hier wird angenommen, dass zum Analysezeitpunkt bereits AccessTypes für die verwendeten lokalen Variablen *docs* und *dir* existieren. Sollte dies nicht der Fall sein, so können einfach neue leere AccessTypes generiert werden. Die Annotationen der AccessTypes der aufgerufenen Methode können jetzt zu den AccessTypes der aufrufenden Methode hinzugefügt werden. Es ist zu beachten, dass die AccessTypes erst zu einem Zeitpunkt in der statischen Codeanalyse zusammengefasst werden dürfen, zu dem sichergestellt ist, dass sich die AccessTypes der aufgerufenen Methode nicht mehr ändern. Dieser Zeitpunkt ist erreicht, wenn alle Anweisungen der aufrufenden Methode verarbeitet wurden. Der Algorithmus hierzu wird nachfolgend erläutert. Hier soll nur das Zusammenfassen der AccessTypes dargestellt werden.

In dem gegebenen Beispiel entspricht die Basis-Variable *docs* der InvokeExpression *docs.createDocument(dir)* der this-Variablen in der Methodenimplementierung. Die Annotation CREATEASSET (siehe Abb. 6.37, S. 128) kann also zum AccessType der lokalen Variablen *docs* in der *doBusinessLogic( )*-Methode hinzugefügt werden. Der Parameter *dir* entspricht in der Methodenimplementierung einer lokalen Variablen, die ebenfalls mit *dir* bezeichnet ist. Die Annotationen von *dir* innerhalb der *createDocument( )*-Methode können jetzt zu den Annotationen der Varialben *dir* aus der *doBusinessLogic( )*-Methode hinzugefügt werden. Die zu untersuchende Anweisung definiert ein Dokument *d*. Innerhalb der *createDocument( )*-Methode gibt es eine lokale Variable *doc*, die durch das ReturnStatement zurückgegeben wird. Es kann also ein neuer AccessType für die Variable *d* erstellt werden, der die Annotationen aus *doc* erhält. Das Ergebnis der Analyse des Methodenaufrufs *createDocument( )* ist in Abb. 6.42 gegeben. Beim Verschieben der Annotationen ist zu beachten, dass sich die Parameter- und Connection-AccessTypes ebenfalls ändern. Der AccessType für die Variable *dirs* ergibt sich aus dem Zusammenfassen der AccessTypes der Methode *getDirectory( )*.

```
Directories dirs {
  READASSET dir
}
Documents docs {
  CREATEASSET d
}
Document d {
  SETPROP Documents.title
  CREATEASSOC Documents.Document_Directory.parent [dir]
}
Directory dir {
  GETPROP Directories.numberDocs
  SETPROP Directories.numberDocs
}
```

Abb. 6.42: Ergebnis des Zusammenfassens der AccessTypes der Methode createDocument( ) mit den AccessTypes der doBusinessLogic( )

Als zweites soll ein Beispiel für das Hochholen der AccessTypes im Falle der Verwendung eines Iterators gezeigt werden. Gegeben sei ein *DirectoryDeleteDocumentsServiceHandler*, welcher zu einem gegebenen Verzeichnis alle Dokumente holt (siehe Abb. 6.43, vgl. Abb. 6.38), um diese zu löschen. Vor dem Hochholen der AccessTypes gibt es eine unvollständige Darstellung der Access-Types der *doBusinessLogic( )*-Methode, sowie eine vollständige Darstellung der AccessTypes der eigentlichen Suchmethode *documents.getDocumentsOfDirectory( )*, sowie der Methoden *remove( )* und *getAsset( )*, da hier bereits alle Anweisungen verarbeitet werden konnten (siehe Abb. 6.44). Die Variable $r3 der *getDocumentsOfDirectory( )*-Methode entsteht durch die intermediäre Darstellung, die die Anweisung

```
return dir.getAssociatedAssets(…);
```

in die zwei Anweisungen

```
$r3 = dir.getAssociatedAssets(…);
return $r3;
```

zerlegt. Die lokale Variable $r3 der *getAsset( )*-Methode entsteht analog. Nach dem Hochholen der AccessTypes kann rekonstruiert werden, durch die Anwendung welcher Basisfunktionalität der Iterator entsteht. Für die Implementierung eines Algorithmus, der das Zusammenfassen der AccessTypes automatisch durchführen soll, ist es zweckmäßig, sich für jeden AccessType zu merken, ob es sich bei der lokalen Variablen, die er repräsentiert, um die implizite this-Variable, einen und welchen Parameter, oder eine Rückgabevariable handelt. Dafür sind die drei Funktionen *this*, *parameter+Parameternummer* und *return* in eckigen Klammern hinter die jeweiligen Namen der lokalen Variablen angehängt worden.

```
public class DirectoryDeleteDocumentsServiceHandler extends
       ServiceHandler {
  String dirId;
  void getParameters(String[] parameters) {
    dirId = //read id-parameter
  }
  int doBusinessLogic(Session session) {
    Directories directories =
        AssetSchemaManager.INSTANCE.getDirectories();
    Directory dir = directories.getDirectory(dirId);
    Documents documents = AssetSchemaManager.INSTANCE.getDocuments();
    Iterator<Asset> documentsInDir =
        documents.getDocumentsOfDirectory(dir);
    while(documentsInDir.hasNext()) {
      Document doc = (Document) documentsInDir.next();
      documents.remove(doc);
    }
  }
}

public class Documents extends AssetSchema {
  public Iterator<Asset> getDocumentsOfDirectory(Directory dir) {
    return dir.getAssociatedAssets(Documents.Documents_Directory.child);
  }
  public void remove(Document doc) {
    … release asset lock if exists … (ignored for this example)
    remove(doc.getId()); //also removes associations to owner, parent,
                         //reader_group in database
  }
}

public class Directories extends AssetSchema {
  public Directory getDirectory(String dirId) {
    return getAsset(dirId);
  }
}
```

Abb. 6.43: Beispielquellcode für die Verwendung eines Iterators und die Implementierung einer Suchmethode

Einfache AcessTypes nach der intraprozeduralen Analyse:

- für die doBusinessLogic()-Methode:

```
Directories directories {}
Documents documents {}
Directory dir { }
Iterator<Document> documentsInDir {
  NEXT doc
}
Document doc { }
```

- für die getDocumentsOfDirectory()-Methode:

```
Documents this [this] { }
Directory dir [parameter+0] {
  READASSET (Documents_Directory.child) $r3
}
Iterator<Asset> $r3 [return] { }
```

- für die remove()-Methode (soweit gezeigt):

```
Documents this [this] {
  DELETEASSET doc
}
Document doc [parameter+0] { }
```

- für die getAsset()-Methode:

```
Directories this [this] {
  READASSET $r3
}
Directory $r3 [return] { }
```

➔ Hochschieben der AccessTypes der Methoden getDocumentsOfDirectory(), getAsset() und remove() in die doBusinessLogic()-Methode:

```
Documents documents {
  DELETEASSET [doc]
}
Directories directories {
  READASSET dir
}
Directory dir {
  READASSET (Documents_Directory.child) documentsInDir
}
Iterator<Document> documentsInDir {
  NEXT doc;
}
Document doc { }
```

Abb. 6.44: Beispiel für das Zusammenfassen von AccessTypes bei der Verwendung eines Iterators und der Implementierung einer Suchmethode

Jedes Asset wird eindeutig durch seine ID repräsentiert. Diese ID wird einem Dienstbearbeiter als String übergeben, wenn dieses für einen Dienst benötigt wird. In realem Quellcode, wie er in dem untersuchten Framework des infoAsset Broker, Version 2.4, vorkommt, kann die Signatur einer Methode anstelle eines Asset-Objekts auch seine String-ID fordern. Dies hat historische Gründe. Folglich finden innerhalb der Programmlogik viele Umwandlungen von der ID in das dazugehörige Asset und anders herum statt. Für die Umwandlung der ID in das dazugehörige Asset steht die bereits bekannte Methode *AssetSchema.getAsset(String id)* zur Verfügung. Andersherum stellt jedes Asset eine Methode *Asset.getId( )* zur Verfügung, mit Hilfe derer die eindeutige ID des jeweiligen Assets bestimmt werden kann. Um nun eine statische Quellcodeanalyse durchführen zu können, die für jedes innerhalb der Geschäftslogik vorkommende Asset einen AccessType erstellt, ist es notwendig zu wissen, dass es sich bei einer String-Variablen und einer Asset-Variablen um verschiedene Repräsentationen ein und desselben Geschäftsobjekts handeln kann. Die AccessTypes müssen also erweitert werden, um diese Äquivalenzstruktur abzubilden. Die Erweiterung der AccessTypes um die ID-Äquivalenzen sind in Abb. 6.45 als Klassendiagramm dargestellt. In der Vorbereitungsphase des Algorithmus kommt ein weiterer Schritt hinzu, welcher den Quellcode nach dem Vorkommen der Methoden *AssetSchema.getAsset(String id)* und *Asset.getId()* als InvokeExpressions von Assign-Statements sucht. Für die jeweils auf der linken Seite des AssignStatements definierte Variable kann jetzt ein AccessType erstellt werden, falls dieser noch nicht existiert. Dem AccessType der definierten Variable wird dann eine ID-Äquivalenz zu dem AccessType der lokalen Variable, die als Parameter übergeben wird, bzw. zu dem AccessType der Basisvariablen der InvokeExpression hinzugefügt. Die ID-Äquivalenz wird auch in der anderen Richtung notiert.

In dem unten stehenden Klassendiagramm sind ebenfalls die vorher besprochenen Erweiterungen um ein Attribut, welches die Funktion eines AccessTypes innerhalb einer Methode, also ob es sich um die implizite this-Variable, ob es sich um einen Parameter und wenn ja, um welchen, oder ob es sich um eine Variable handelt, die mit einer return-Anweisung zurückgeliefert wird, dargestellt. Die Erweiterung um AccessTypes für die Felder der ServiceHandler-Klassen, in der Abbildung mit *FieldAccess-Type* notiert, ist für den zweiten Schritt der interprozeduralen Analyse notwendig. Der *LocalAccess-Type* ist der bereits bekannte AccessType für eine lokale Variable.

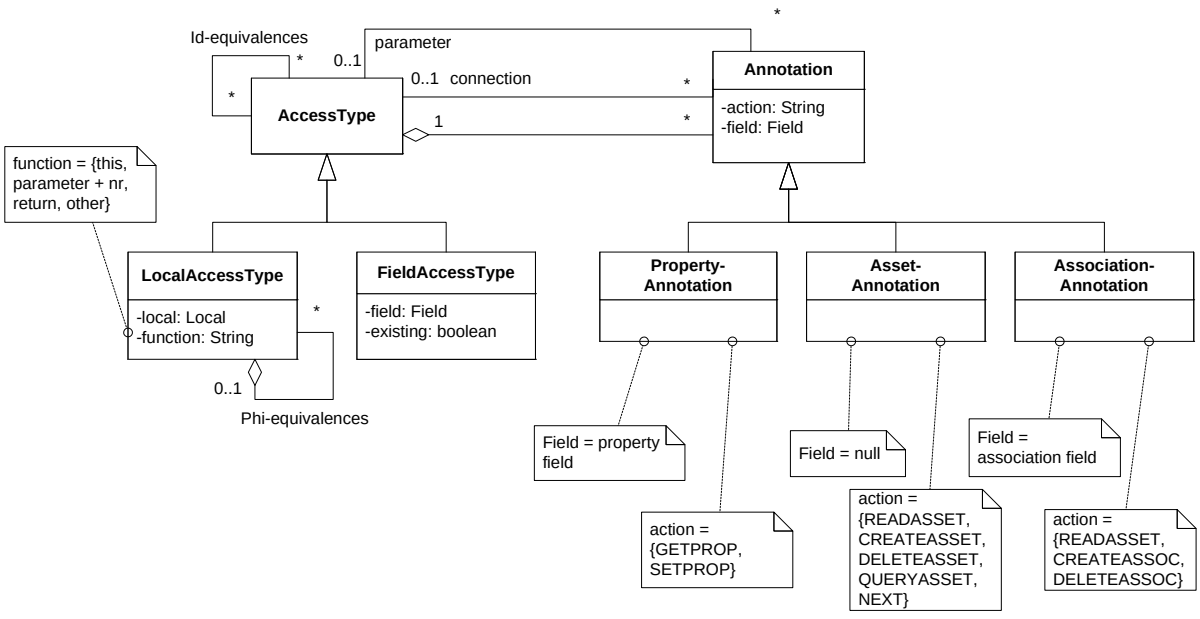


Abb. 6.45: Erweiterung der AccessTypes als Klassendiagramm

Die interprozedurale Analyse muss jetzt nach dem Verschieben der AccessTypes von aufgerufenen Methoden in die aufrufenden Methoden, AccessTypes erkennen, die äquivalent sind. Beim Verschieben der AccessTypes müssen nicht nur die Annotationen, sondern auch die Informationen über die ID-Äquivalenzbeziehungen mit verschoben werden. Zwei äquivalente AccessTypes können zu einem

zusammengefasst werden. Ein AccessType *A* ist zu einem anderen AccessType *B* äquivalent, wenn es einen dritten AccessType *C* gibt, der zwei ID-Äquivalenzen enthält, eine für A und eine für B. Beim Zusammenfassen der AccessTypes ist derjenige AccessType führend, welcher zu der gerade untersuchten Methode gehört. Der aus einer aufgerufenen Methode nach oben verschobene AccessType wird gelöscht, nachdem alle seine Annotationen und ID-Äquivalenzen zum führenden AccessType verschoben worden sind und alle Verwendungen dieses AccessTypes als Connection- oder Parameter-AccessType aktualisiert worden sind.

Ein Beispiel für die Verwendung von Strings, die Asset-IDs repräsentieren, als Parameter von Methoden ist in Abb. 6.46 gegeben. In Abb. 6.47 sind die durch die intraprozedurale Analyse entstehenden AccessTypes für die lokalen Variablen dargestellt. Nach diesem Analyseschritt sind alle Anweisungen der Methode *Directories.getDirectory(String dirId)* verarbeitet, da für alle lokalen Variablen ein AccessTypes erstellt werden konnte. Die Variable *$r3* ist eine künstlich eingeführte Variable, die das *Directory*-Objekt darstellt, für das im Quellcode keine eigene Variable gibt. Die AccessTypes der Methode *Directories.getDirectory(String dirId)* enthalten jetzt einen Hinweis darauf, dass die Variable *dId*, die als Parameter übergeben wird, äquivalent zu der Variablen ist, die von der Methode zurückgegeben wird. Die interprozedurale Analyse kann jetzt mit dem Hochschieben der AccessTypes dieser Methode beginnen. Das Ergebnis ist in Abb. 6.48 gezeigt. Danach ist die Methode *Document.setDirectory(String dirId)* vollständig analysiert, da alle lokalen Variablen einen Access-Type besitzen (mit Ausnahme des AssetSchemaManagers, für den kein AccessType benötigt wird, denn dieser holt nur AssetSchemas und führt nie selbst eine berechtigungsrelevante Aktion durch) und die AccessTypes dieser Methode können weiter nach oben geschoben werden. Durch das Wissen, dass es sich bei dem AssetSchema *Directories* um ein Singleton handelt, können die beiden lokalen AccessTypes *$dirs1* und *$dirs2* zusammengefasst werden zu *$dirs1*. Durch das Hochholen des AccessTypes *dirId* aus der Methode *Document.setDirectory(String dirId)* und das Zusammenfassen mit dem bestehen AccessType *dirId* der Methode *Documents.createDocument(String dirId)* ergibt sich, dass die als ID-äquivalent bezeichneten lokalen AccessTypes der beiden lokalen Variablen *dir* aus den beiden Methoden ebenfalls äquivalent sein müssen. Daraus folgt, dass der Inhalt des AccessTypes der lokalen Variable *dir* in der Methode *Document.setDirectory(String dirId)* mit dem lokalen AccessType *dir* der Methode *Documents.createDocument(String dirId)* verschmolzen werden kann. Das Ergebnis ist in Abb. 6.49 gezeigt. Zum Schluss können die lokalen AccessTypes der Methode *Documents.createDocument(String dirId)*, die jetzt vollständig analysiert ist, mit den AccessTypes der Methode *doBusinessLogic(Session session)* vereint werden (siehe Abb. 6.50). Der lokale Access-Type für das Singleton-Objekt vom Typ *Directories*, der AccessType *$dirs1*, kann nach oben verschoben werden, da die Singleton-Klasse überall bekannt ist.

```
public class DocumentCreateServiceHandler extends ServiceHandler {
  String directoryId;
  String title;
  … //other variables for other properties
  void getParameters(String[] parameters) {
    directoryId = …;
    title = …;
  }
  int doBusinessLogic(Session session) {
    AssetSchemaManager asm = AssetSchemaManager.INSTANCE;
    Documents docs = asm.getDocuments();
    Document d = docs.createDocument(directoryId);
    d.setProperty(Documents.title, „…“);
    …
  }
}

public class Documents extends AssetSchema {
  public Document createDocument(String dirId) {
    Document doc = this.createAsset();
    doc.setDirectory(dirId);
    Directory dir =
      AssetSchemaManager.INSTANCE.getDirectories().getDirectory(dirId);
    int numberDocs = dir.getProperty(Directories.numberDocs) + 1;
    dir.setProperty(Directories.numberDocs, numberDocs);
    return doc;
  }
}

public class Document extends Asset {
  public void setDirectory(String dirId) {
    Directory dir =
      AssetSchemaManager.INSTANCE.getDirectories().getDirectory(dirId);
    this.createAssociation(Documents.Document_Directory.parent, dir);
  }
}

public class Directories extends AssetSchema {
  public Directory getDirectory(String dId) {
     return getAsset(dId);
  }
}
```

Abb. 6.46: Beispiel für die Verwendung von Asset-IDs als Parameter

- AccessTypes für die *doBusinessLogic(Session session)*-Methode:

```
Document d {
  SETPROP (Documents.title)
}
```

- AccessTypes für die *Documents.createDocument(String dirId)*-Methode:

```
Documents this [this] {
  CREATEASSET doc
}
String dirId [parameter+0]{  }
Document doc [return] { }
Directory dir {
  SETPROP (Documents.numberDocs)
  GETPROP (Documents.numberDocs)
}
```

- AccessTypes für die *Document.setDirectory(String dirId)*-Methode:

```
Document this [this] {
  CREATEASSOC (Documents.Document_Directory.parent) [dir]
}
String dirId [parameter+0] { }
Directory dir { }
```

- AccessTypes für die *Directories.getDirectory(String dId)*-Methode:

```
Directories this [this] {
  READASSET $r3
}
String dId [parameter+0] {
  id_equiv: $r3
}
Directory $r3 {
  id_equiv: dId
}
```

Abb. 6.47: AccessTypes nach der intraprozeduralen Analyse

- AccessTypes für die *doBusinessLogic(Session session)*-Methode:

```
Documents docs {  }
Document d {
  SETPROP (Documents.title)
}
```

- AccessTypes für die *Documents.createDocument(String dirId)*-Methode:

```
Documents this [this] {
  CREATEASSET doc
}
String dirId [parameter+0]{
  id_equiv: dir
}
Document doc [return] { }
Directory dir {
  id_equiv: dirId
  SETPROP (Documents.numberDocs)
  GETPROP (Documents.numberDocs)
}
Directories $dirs1 {
  READASSET dir
}
```

- AccessTypes für die *Document.setDirectory(String dirId)*-Methode:

```
Document this [this] {
  CREATEASSOC (Documents.Document_Directory.parent) [dir]
}
String dirId [parameter+0] {
 id_equiv: dir
}
Directory dir {
 id_equiv: dirId
}
Directories $dirs2 {
  READASSET dir
}
```

Abb. 6.48: AccessTypes nach dem Hochholen der AccessType von getDirectory()

- AccessTypes für die *doBusinessLogic(Session session)*-Methode:

```
Documents docs {  }
Document d {
  SETPROP (Documents.title)
}
```

- AccessTypes für die *Documents.createDocument(String dirId)*-Methode:

```
Documents this [this] {
  CREATEASSET doc
}
String dirId [parameter+0]{
  id_equiv: dir
}
Document doc [return] {
 CREATEASSOC (Documents.Document_Directory.parent) [dir]
}
Directory dir {
  id_equiv: dirId
  SETPROP (Documents.numberDocs)
  GETPROP (Documents.numberDocs)
}
Directories $dirs1 {
  READASSET dir
}
```

Abb. 6.49: AccessTypes nach dem Hochholen der AccessTypes aus setDirectory()

```
Documents docs {
  CREATEASSET d
}
Document d {
  SETPROP (Documents.title)
  CREATEASSOC (Documents.Document_Directory.parent) [dir]
}
String directoryId {
  id_equiv: dir
}
Directories $dirs1 {
  READASSET dir
}
Directory dir {
  id_equiv: dirId
  SETPROP (Documents.numberDocs)
  GETPROP (Documents.numberDocs)
}
```

Abb. 6.50: AccessTypes für die doBusinessLogic()-Methode nach der vollständigen interprozeduralen Analyse

Für das Hochholen und Zusammenfassen der AccessTypes wurde ein iterativer Algorithmus entwickelt. In jeder Iteration werden neue, noch nicht verarbeitete Anweisungen mit einer *is-processed*-Annotation annotiert. Ist eine Anweisung mit einer *is-out-of-scope*- (siehe oben) oder einer *is-processed*-Annotation versehen, so bedeutet dies, dass alle Anweisungen, im darunter liegenden, zu untersuchenden Aufrufgraph verarbeitet worden sind. Alle Anweisungen, die nach der intraprozeduralen Phase noch nicht annotiert sind, enthalten einen Methodenaufruf, also eine InvokeExpression. Dies ist durch die vorher beschriebene Definition der irrelevanten Anweisungen gewährleistet. Sind alle Anweisungen innerhalb einer Methode verarbeitet, so können die AccessTypes in die Methoden hochgeschoben werden, die diese Methode verwenden. Hierbei stellen in der aufgerufenen Methode der AccessType, der die this-Variable darstellt und in der aufrufenden Methode der AccessType, der die Basisvariable der gerade untersuchten InvokeExpression bildet, dasselbe Objekt dar und können zusammengefasst werden. Dasselbe gilt für die AccessTypes, deren lokale Variable über einen Parameter übergeben wird und für den AccessType, der über eine InvokeExpression zurückgeliefert wird. Alle anderen AccessTypes werden ohne weiteres Zusammenfassen nach oben kopiert. Nach dem Hochschieben müssen eventuell vorkommende ID-Äquivalenzen verarbeitet, d. h. äquivalente AccessTypes zusammengefasst werden. Am Schluss wird die Methode mit einer *is-processed*-Annotation versehen. Sind alle Methoden innerhalb einer Klasse mit einer Annotation versehen, so können die Klassen ihrerseits wieder eine *is-processed*-Annotation erhalten. Sind alle Klassen der Scene mit einer *is-processed*-Annotation annotiert, so ist der Algorithmus beendet und es wurden für alle lokalen Variablen in den Methoden, die die Wurzeln des Aufrufgraphen bilden, vollständige AccessTypes erstellt. Um zu gewährleisten, dass der Algorithmus endet, müssen alle rekursiven Methodenaufrufe mit einer *is-out-of-scope*-Annotation versehen werden, so dass der zu untersuchende Aufrufgraph keine Schleifen enthält. Der Algorithmus ist in Abb. 6.51 als Pseudo-Code dargestellt.

Nach der interprozeduralen Analyse stehen in der *doBusinessLogic( )*-Methode die AccessTypes für die lokalen Variablen bereit. Falls bereits eine *initAssets( )*-Methode mit Anweisungen gefüllt ist, so stehen auch in dieser Methode AccessTypes für die lokalen Variablen bereit (vgl. S. 121). Die lokalen Variablen, die in der *initAssets( )*-Methode definiert werden, müssen auch der *doBusinessLogic( )*- und später der *checkAccess( )*-Methode zur Verfügung gestellt werden. Deshalb definiert die jeweilige *ServiceHandler*-Klasse Felder für die Objekte, die in *checkAccess( )* definiert werden. Die interprozedurale Analyse schließt also damit, die AccessTypes für diese Felder aus den AccessTypes der lokalen Variablen der Methoden *initAssets( )* und *doBusinessLogic( )* zu generieren. Man beachte hier, dass in der *initAssets( )*-Methode bereits Assets gelesen werden, deren Lesezugriff erst nach dem Lesen in der *checkAccess( )*-Methode geprüft werden kann, wenn es sich bei der Leseberechtigung für diese Assets um asset- oder eigentümerbezogene Berechtigungspolicies handelt. Die Generierung der AccessTypes für die Felder ist sehr einfach. Es brauchen nur die Annotationen der jeweiligen lokalen Variablen in einem AccessType für das betreffende Feld zusammengefasst und die AccessTypes der jeweiligen lokalen Variablen gelöscht werden.

```
//interprocedural analysis for local variables
finished = false;
while(!finished) {
  for (all classes c in scene) {
    if(c.getClassDefinition().containsAnnotation(is-processed)
      continue;
    for (all methods m in c) {
      if(m.getMethodDefinition().containsAnnotation(is-processed)
        continue;
      for (all statements s in m) {
        if(!s.containsAnnotation(is-processed) &&
           !s.containsAnnotation(is-out-ofscope) {
          //due to contruction of algorithms, all statements that are
          //not annotated contain an invoke expression
          InvokeExpression ie = s.getInvokeExpression();
          MethodDefinition calledMD = ie.getMethodDefinition();
          if(calledMD.containsAnnotation(is-processed) {
            //combine access types in this method with access types in
            //called method
            method calledMethod = calledMD.getMethod();
            Local base = ie.getBase();
            Local calledThis = calledMethod.getThisVariable();
            base.getAccessType().
                addAnnotationsOf(calledThis.getAccessType());
            for (int i = 0; i < ie.getNumberOfParameters(); i++) {
              Local param = ie.getParameter(i);
              Local calledParam = calledMethod.getParameterVariable(i);
              param.getAccessType().
                  addAnnotationsOf(calledParam.getAccessType());
            }
            Local return = s.getLeftLocal();
            Local calledReturn = calledMethod.getReturnVariable();
            return.getAccessType().
                addAnnotationsOf(calledReturn.getAccessType());
            move all other access types from called method to this
            method
            reduce id-equivalent access types, update connection- and
            parameter-access types
            s.addAnnotation(is-processed);
          }
        }
      }
      if(all statements annotated)
        m.getMethodDefinition().addAnnotation(is-processed);
    }
    if(all method definitions annotated)
      c.getClassDefinition().addAnnotation(is-processed);
  }
  if(all class definitions annotated)
    finished = true;
}
```

Abb. 6.51: Interprozedurale Analyse des Hochschiebens der AccessTypes als Pseudo-Code

6.2.4 Generierung der Berechtigungsdurchsetzungsfunktionen

Für die Generierung der Berechtigungsdurchsetzungsfunktion aus den AccessTypes ist es notwendig zu wissen, welche AccessTypes für einen ServiceHandler relevant sind. Dadurch, dass Annotationen von AccessTypes wieder andere AccessTypes referenzieren, sei dies über einen Connection- oder einen Parameter-AccessType, entsteht für einen ServiceHandler ein Netz aus zusammenhängenden AccessTypes. Das AccessTypes-Netz für die AccessTypes aus Abb. 6.42 ist in Abb. 6.52 gegeben. Der Einfachheit halber sei hier ein Beispiel ohne ID-Äquivalenzen gegeben. Die Wurzeln dieses AccessTypes-Netzes bilden alle AccessTypes von lokalen Variablen und Feldern der ServiceHandler-Klasse, welche nicht über eine Connection oder einen Parameter anderer AccessTypes referenziert werden. Dieses sind gleichzeitig die AccessTypes der innerhalb des Aufrufbaums eines *ServiceHandlers* definierten AssetSchema-Klassen sowie die AccessTypes der Assets, deren IDs über einen Parameter übergeben und in der *getParameters( )*-Methode ausgelesen werden. Alle anderen AccessTypes, die innerhalb des Aufrufbaums einer ServiceHandler-Klasse auftauchen aber nicht in diesem AccessTypes-Netz sind, sind für die Zugriffskontrolle nicht relevant. Sie wurden bereits durch AccessTypes des AccessTypes-Netzes subsumiert. Dies ist so, da alle Assets, die innerhalb der Diensausführung gelesen oder geschrieben werden, entweder aus den Anfrageparametern stammen oder aber über Assoziationsbeziehungen, die ja auch in den AccessTypes vermerkt sind, miteinander in Verbindung stehen. Es kommt nicht vor, dass Assets aus dem „Nichts“ entstehen. Eine Ausnahme bildet hier die *AssetSchema.queryAssets( )*-Methode, die für die Q-Dienste Anwendung findet. Die Assets, die in dem Suchergebnis auftauchen, werden aber erst während der Erstellung der Response verarbeitet und sind so für die Zugriffskontrollüberprüfung für die *doBusinessLogic( )*-Methode nicht relevant. Eine Lösung für die Platzierung der Zugriffskontrollüberprüfungen der Q-Dienste wurde in Kapitel 6.1 behandelt.

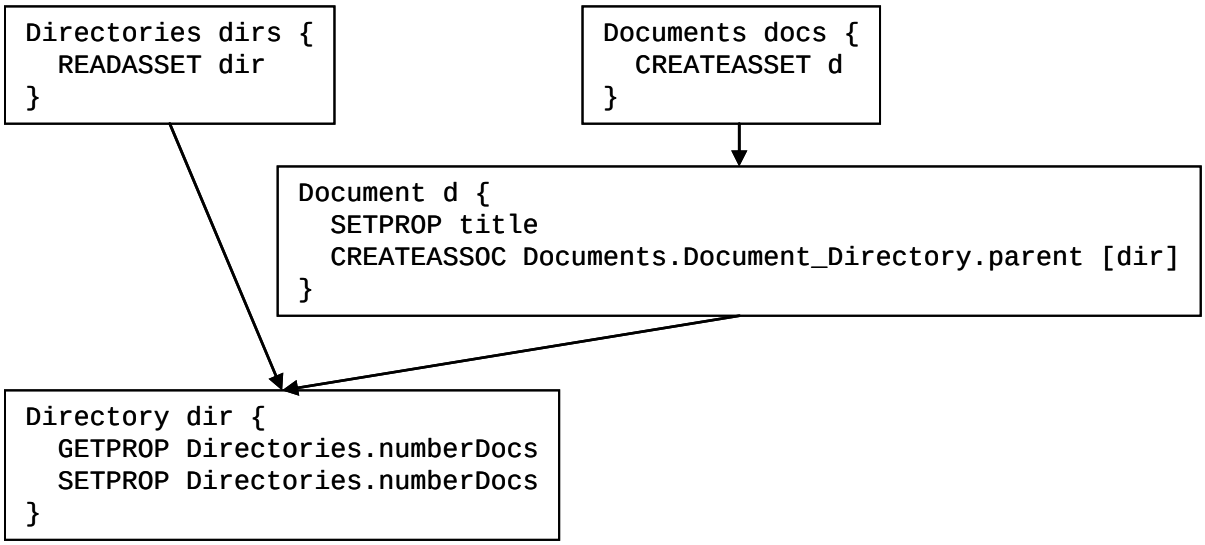


Abb. 6.52: AccessTypes-Netz der AccessTypes aus Abb. 6.42

Für die Quellcodegenerierung muss jetzt für jede Aktion einer Annotation im AccessTypes-Netz die entsprechende Policy, dargestellt durch eine PathExpression, identifiziert werden. Die PathExpressions sind als Sammlung abstrakter Klassen realisiert (vgl. Kap. 5.3.3). Um eine konkrete Instanz einer PathExpression zu bilden, müssen die jeweiligen abstrakten Methoden konkrete Assets zurückliefern, die den Anfang bzw. das Ende des Pfades im Datenmodell, den eine PathExpression beschreibt, darstellen. Diese Assets können meist aus dem AccessTypes-Netz oder aus der Umgebung hergeleitet werden. Für eine gruppenbezogene Berechtigungspolicy startet die dazugehörige PathExpression bei einer statisch bekannten Gruppe, es wird also kein konkretes Start-Asset benötigt. Der Pfad endet beim Sitzungsbenutzer. Jedem ServiceHandler steht eine lokale Variable *session_user* (vgl. einfache ServiceHandler-Klassen für jeden Diensttyp aus Kapitel 6.1) zur Verfügung, welche den aktuellen

Sitzungsbenutzer darstellt. Für eigentümerbezogene Policies ist klar, dass es sich beim Start-Asset um dasjenige Asset handeln muss, welches dem AccessType zugeordnet ist, aus dem die zur Policy gehörige Annotation stammt. Der eigentümerbezogene Zugriffspfad endet wieder beim Sitzungsbenutzer, welcher der Umgebung bekannt ist. Für das Start-Asset einer assetbezogenen Policy gilt dasselbe, wie für das Start-Asset einer eigentümerbezogenen Policy, falls die Typen des AccessTypes und des zu erwartenden Start-Assets übereinstimmen. Ansonsten kann das Start-Asset nicht vollautomatisch aus dem Kontext ermittelt werden. Eine Rückfrage an den Entwickler ist notwendig. Das unten gegebene Beispiel für die Generierung einer Policy für den *DocumentCreateService-Handler* behandelt diesen Fall. Das End-Asset einer assetbezogenen Policy ist wieder der Sitzungsbenutzer, wenn diese ein Objekt vom Typ *User* verlangt. Eine assetbezogene Policy, die eine Eigenschaft eines Assets beschreibt, also nur durch einen Filter implementiert ist, endet wieder beim Start-Asset. Für andere assetbezogene Policies, die Navigationen enthalten, gilt, dass entweder kein End-Asset angegeben werden muss oder dass die Policy nur semi-automatisch nach Rückfrage an den Entwickler erstellt werden kann. Bspw. gilt für die NotPolicy *unlockedDocument*, die beschreibt, dass ein Zugriff erlaubt ist, falls kein *AssetLock*-Objekt für das betreffende Dokument existiert (vgl. S. 78 / 79), dass diese unabhängig von einem spezifischen End-Asset ausgewertet wird.

Alle Asset-Objekte, welche für die Erstellung der Policies benötigt werden, müssen innerhalb der ServiceHandler-Klasse entweder als Felder zur Verfügung stehen und in der *initAssets( )*-Methode entsprechend initialisiert werden oder aus dem AccessTypes-Netz hergeleitet und in der *checkAccess( )*-Methode als lokale Variablen zur Verfügung gestellt werden. Betrachten wir hierzu ein Beispiel. Gegeben seien der *DocumentCreateServiceHandler* aus Abb. 6.36 (siehe S. 127) sowie das dazugehörige AccessTypes-Netz aus Abb. 6.52. Weiterhin seien die folgenden Zugriffspolicies für die Aktionen in den Annotationen der AccessTypes definiert (siehe Tabelle 6.1 und Abb. 6.53). Sind mehrere Policies (mehrere Tabellenzeilen) für eine Aktion reserviert, so sind diese mit einem ODER verknüpft. Es handelt sich also um OrPolicies.

Tab. 6.1: Aktionen mit dazugehörigen Policies für den DocumentCreateServiceHandler

Aktion	**PathExpression**	**Typ der PathExpression**	**Typ des Start-Assets**	**Typ des End-Assets**
Documents. CREATEASSET	DirectoryEditor_Policy	assetbezogen (Association)	Directory	User
Document. SETPROP	Owner_Policy	eigentümer-bezogen	Document	User
Document. SETPROP	DirectoryAuthor_Policy	assetbezogen (Association)	Directory	User
Document. SETPROP	DirectoryEditor_Policy	assetbezogen (Association)	Directory	User
Document. CREATEASSOC(parent)	DirectoryEditor_Policy	assetbezogen (Association)	Directory	User
Directories. READASSET	NotAnonymous_Policy	assetbezogen (Property)	User	User
Directory. GETPROP	NotAnonymous_Policy	assetbezogen (Property)	User	User
Directory. SETPROP	DirectoryAuthor_Policy	assetbezogen (Association)	Directory	User
Directory. SETPROP	DirectoryEditor_Policy	assetbezogen (Association)	Directory	User

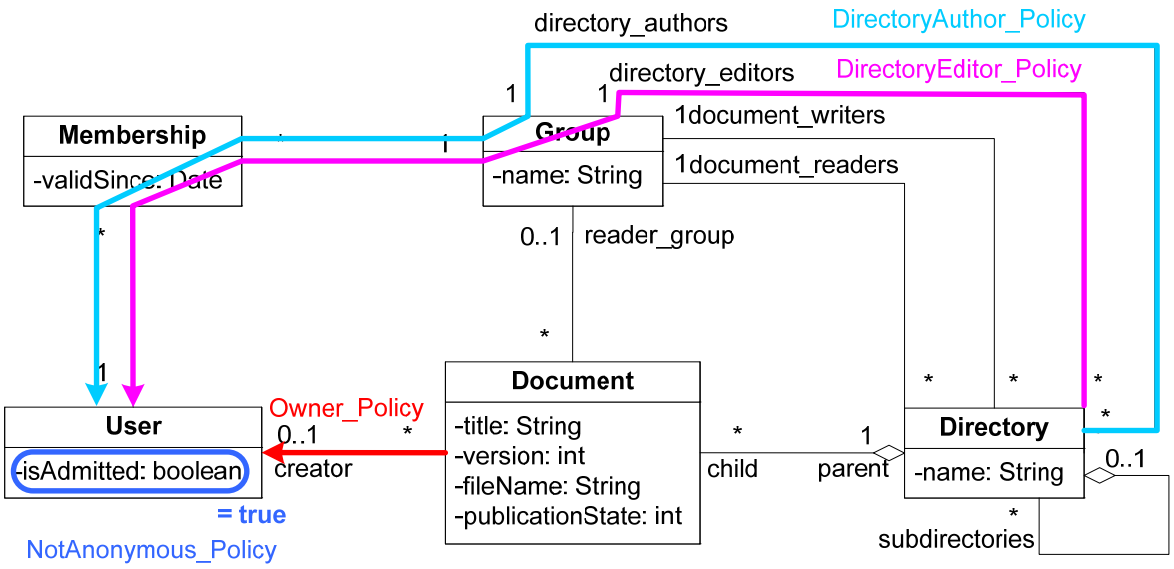


Abb. 6.53: Policies für die Aktionen in den Annotationen der AccessTypes des DocumentCreateServiceHandlers

Aus Konsistenzgründen müssen die Policies für das Initialisieren der Properties eines Dokuments, die nicht *null* sein dürfen, und die Policies für das Anlegen der Pflichtassoziation zwischen dem Dokument und seinem Verzeichnis so gewählt sein, dass es allen Benutzern, die die Berechtigung haben, ein Dokument anzulegen, also die Aktion *Documents.CREATEASSET* durchzuführen, möglich ist, auch diese Pflichtaktionen durchzuführen (vgl. S. 96). Bei der Generierung einer Berechtigungspolicy für den *DocumentCreateServiceHandler* können also die Policies für die Aktionen *Document.SETPROP* sowie *Document.CREATEASSOC(parent)* ignoriert werden.

Die Policy für *Documents.CREATEASSET* benötigt noch ein konkretes Objekt vom Typ *Directory*. Hiermit ist das Verzeichnis gemeint, welches dem Dienst als Parameter übergeben worden ist. Diese semantische Beziehung ist aber nicht aus der PathExpression ablesbar, so dass hier eine Rückfrage an den Entwickler stattfinden muss, welches Verzeichnisobjekt einzusetzen ist. Da die *Directory-Editor_Policy* bei einem Benutzer endet, wird hierfür das in der Umgebung bekannte Objekt *session_user*, das den aktuellen Sitzungsbenutzer darstellt, genommen. Die *NotAnonymous_Policy*, welche prüft, ob ein Benutzer authentifiziert ist, ist ein Filter, welcher sowohl als Anfangs- als auch als End-Asset ein Objekt vom Typ *User* benötigt. Die abstrakte Klasse *NotAnonymous_Policy* kann so implementiert werden, dass nur eine Methode zu überschreiben ist, die ein *User*-Objekt fordert. Für das *User*-Objekt wird wieder der Sitzungsbenutzer angenommen, da innerhalb der Implementierung kein anderes Objekt vom Typ *User* vorkommt. Es macht wenig Sinn, eine Policy zu definieren, bei der dem Sitzungsbenutzer eine Aktion erlaubt ist, falls irgendein anderer Benutzer authentifiziert ist. Es wird also eine Semantik der definierten Policies angenommen. Die Policies für die Aktion *Directory.-SETPROP* sind wieder assetbezogen und starten bei einem Objekt vom Typ *Directory*. Bei diesem Verzeichnis handelt es sich um das Objekt *dir*, welches zu dem AccessType gehört, welches die *SETPROP*-Annotation beinhaltet. Der Pfad endet wieder beim Sitzungsbenutzer. Mit diesen Informationen kann jetzt die Policy in die *checkAccess( )*-Methode generiert werden. Einige der Policies benötigen das Objekt *dir*, welches nur als lokale Variable in der *doBusinessLogic( )*-Methode zur Verfügung steht. Der Algorithmus, welcher die Policies erstellt, muss den Entwickler darauf hinweisen, dass er diese Variable als Feld deklarieren muss und seine Initialisierung in die Methode *initAssets( )* verschieben muss. Ebenso sollten keine Policies doppelt generiert werden, wenn sie für zwei verschiedene Annotationen benötigt werden. Das Resultat der Policygenerierung und der manuellen Quellcodeverschiebung ist in Abb. 6.54 gezeigt.

```
public class DocumentCreateServiceHandler extends ServiceHandler {
  String directoryId;
  String title;
  … //other values for other properties
  void getParameters(String[] parameters) {
    directoryId = …;
    title = …;
    …
  }
  AssetSchemaManager asm;
  Directory dir;
  initAsset() {
    asm = AssetSchemaManager.INSTANCE;
    Directories dirs = asm.getDirectories();
    dir = dirs.getDirectory(directoryId);
  }
  int doBusinessLogic(Session session) {
    Documents docs = asm.getDocuments();
    Document d = docs.createDocument(dir);
    …
  }
  boolean checkAccess() {
    //policy for Documents.CREATEASSET
    DirectoryEditor_Policy policy0 = new DirectoryEditor_Policy() {
      public Directory getDirectoryAsset() { return dir;}
      public User getUserAsset() { return session_user;}
    }
    boolean path0 = policy0.existsPath();
    //policy for Directories.READASSET
    NotAnonymous_Policy policy1 = new NotAnonymous_Policy() {
      public User getUserAsset() { return session_user;}
    }
    boolean path1 = policy1.existsPath();
    //policy for Directory.GETPROP exists
    //policy for Directory.SETPROP
    DirectoryAuthor_Policy policy2 = new DirectoryAuthor_Policy() {
      public Directory getDirectoryAsset() { return dir;}
      public User getUserAsset() { return session_user;}
    }
    boolean path2 = policy2.existsPath();
    //DirectoryEditor_Policy exists
    return (path0) && (path1) && (path1) && (path2 || path0);
  }
}
```

Abb. 6.54: Ergebnis der Policygenerierung für den DocumentCreateServiceHandler

Es sei hier angemerkt, dass für ein Netz mit ID-Äquivalenzen auch aus dem Netz ausgelesen werden kann, auf welche Weise die Assets entstehen, deren ID über einen Anfrageparameter dem Service-Handler übergeben werden. Anstatt die für die Berechtigungsüberprüfung benötigten Asset also in der

initAssets( )-Methode zu definieren, können diese auch als lokale Variablen in der *checkAccess( )*-Methode definiert werden. Aus den AccessTypes (vgl. Abb. 6.50),

```
String directoryId {
  id_equiv: dir
}
Directories $dirs1 {
  READASSET dir
}
```

wobei die Variable *directorId* in der Methode *getParameters( )* eines ServiceHandlers gefüllt wird, kann die Anweisung

```
Directory dir =
  AssetSchemaManager.INSTANCE.getDirectories().getAsset(directoryId);
```

automatisch erzeugt werden, da bekannt ist, dass id-äquivalente Strings aus der Anweisung *AssetSchema.getAsset(String id)* hervorgehen.

Als zweites Beispiel soll die Policygenerierung für den *DirectoryDeleteDocumentsServiceHandler* betrachtet werden. In Abb. 6.55 ist das AccessTypes-Netz gegeben, welches sich aus den AccessTypes für diesen ServiceHandler ergibt (vgl. Abb. 6.44, S. 134). Die Policygenerierung kann mit der Generierung der Policy für das gelesene Verzeichnis beginnen. Die Policy benötigt nur den bekannten Sitzungsbenutzer. Das gelesene Verzeichnis *dir* erzeugt nun einen Iterator durch Aufruf der Basisfunktionalität *AssetSchema.getAssociatedAssets( )* über die Assoziation *Documents_Directory* mit der Rolle *child*. Dies geht aus der Annotation des zu *dir* gehörigen AccessTypes eindeutig hervor. Die Erzeugung des Iterators kann also nachgebaut werden. Hierfür muss das Objekt *dir* in der *checkAccess( )*-Methode verfügbar sein, also muss die Policygenerierung dem Entwickler einen Hinweis geben, dass dieses Objekt als Feld zu deklarieren und in der *initAssets( )*-Methode zu definieren ist. Der Iterator-AccessType enthält eine NEXT-Annotation. Diese bildet den Aufruf von *Iterator.next( )* ab. Die Policygenerierung kann eine entsprechende while-Schleife generieren. Der Aufruf von NEXT kann auch gleichzeitig mit einer Leseberechtigung auf dem gelesenen Asset, also auf einem Asset vom Typ *Document* assoziiert werden. Die Policies zum Lesen eines Dokuments sind im gegebenen Beispiel (siehe Tabelle 6.2 und Abb. 6.56) alle asset- oder eigentümerbezogen und benötigen ein Start-Asset vom Typ *Document*. Hierfür wird das Objekt *doc* genommen, welches innerhalb der NEXT-Annotation zuzuordnen und in der while-Schleife definiert wird. Der AccessType, der die lokale Variable *doc* darstellt, taucht auch als Connection in einer REMOVEASSET-Annotation auf dem AssetSchema *Documents* auf. Also muss innerhalb der while-Schleife die Löschberechtigung überprüft werden. Das Ergebnis der Policygenerierung ist in Abb. 6.57 zu sehen.

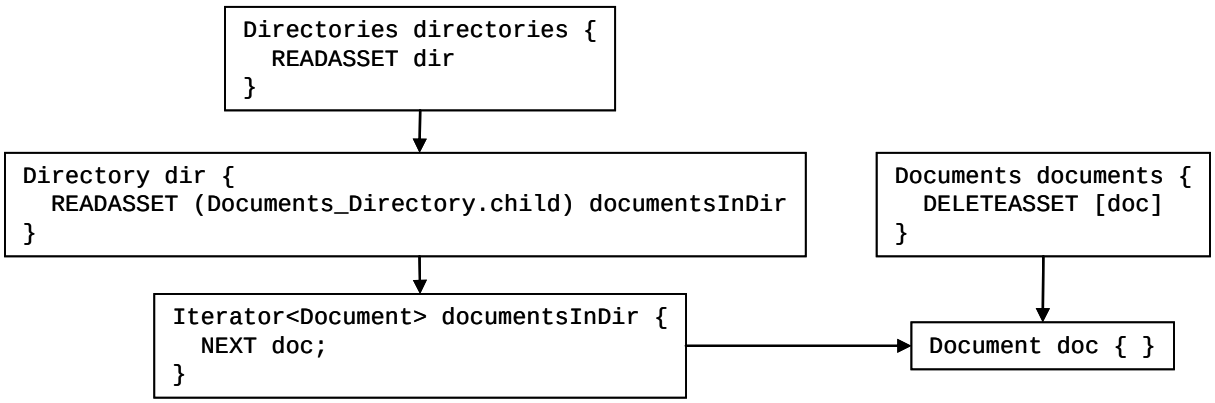


Abb. 6.55: AccessTypes-Netz für den DirectoryDeleteDocumentsServiceHandler

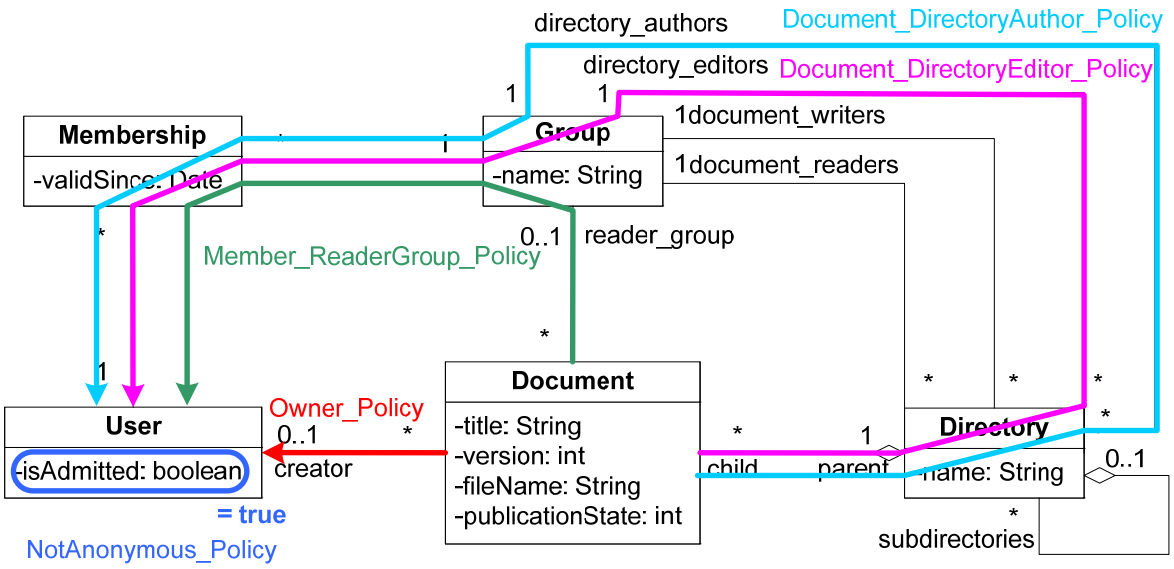


Abb. 6.56: Policies für die Aktionen in den Annotationen der AccessTypes des DirectoryDeleteDocumentsServiceHandlers

Tab. 6.2: Aktionen mit dazugehörigen Policies für den DirectoryDeleteDocumentsServiceHandlers

Aktion	**PathExpression**	**Typ der PathExpression**	**Typ des Start-Assets**	**Typ des End-Assets**
Directories. READASSET	NotAnonymous_Policy	assetbezogen (Property)	User	User
Documents. READASSET	Document_DirectoryEditor_Policy	assetbezogen (Association)	Document	User
Documents. READASSET	Document_DirectoryAuthor_Policy	assetbezogen (Association)	Document	User
Documents. READASSET	Member_ReaderGroup_Policy	assetbezogen (Association)	Document	User
Documents. READASSET	Owner_Policy	eigentümer-bezogen	Document	User
Documents. DELETEASSET	Document_DirectoryEditor_Policy	assetbezogen (Association)	Document	User
Documents. DELETEASSET	Document_DirectoryAuthor_Policy	assetbezogen (Association)	Document	User

```
public class DirectoryDeleteDocumentsServiceHandler extends
       ServiceHandler {
  String dirId;
  void getParameters(String[] parameters) {
    dirId = //read id-parameter
  }
  Directory dir;
  void initAsset() {
    Directories directories =
        AssetSchemaManager.INSTANCE.getDirectories();
    dir = directories.getDirectory(dirId);
  }
  boolean checkAccess() {
    //policy for Directories.READASSET
    NotAnonymous_Policy policy0 = new NotAnonymous_Policy() {
      public User getUserAsset() { return session_user;}
    }
    boolean path0 = policy0.existsPath();
    Iterator<Document> documentsInDir =
      dir.getAssociatedAssets(Documents.Documents_Directory.child);
    while(documentsInDir.hasNext()) {
       Document doc = (Document) documentsInDir.next();
       //policy for Documents.READASSET
       Document_DirectoryEditor_Policy policy1 =
           new Document_DirectoryEditor_Policy() {
         public Document getDocumentAsset() { return doc;}
         public User getUserAsset() { return session_user;}
       }
       if(! policy1.existsPath()) return false;
       … other read policies generated here …
       … delete policies generated here …
    }
    return (path0);
  }
  int doBusinessLogic(Session session) {
    Documents documents = AssetSchemaManager.INSTANCE.getDocuments();
    Iterator<Asset> documentsInDir =
        documents.getDocumentsOfDirectory(dir);
    while(documentsInDir.hasNext()) {
      Document doc = (Document) documentsInDir.next();
      documents.remove(doc);
    }
  }
}
```

Abb. 6.57: Generierte Policies für den DirectoryDeleteDocumentsServiceHandler

6.3 Auswertung

Die Realisierung der AEF ergibt sich aus dem Zusammenspiel eines Zugriffskontroll-Frameworks und einer statischen Quellcodeanalyse.

Das Framework sieht Eingriffspunkte vor, an denen die Berechtigungsdurchsetzung in den Quellcode der Anwendung eingebettet werden kann. Für jeden Dienstbearbeiter ist hierbei ein Eingriffspunkt zu Beginn der Dienstausführung vorgesehen, die *checkAccess( )*-Methode, so dass nur genau eine Berechtigungsüberprüfung pro Dienstausführung durchgeführt wird. Durch die Platzierung des Eingriffspunkts ist es möglich, zeitlich vor und entkoppelt von der Dienstausführung die Berechtigung für einen Dienst festzustellen. Auf diese Weise können automatisch Elemente auf der Benutzeroberfläche, wie etwa Links und Buttons, ausgeblendet oder deaktiviert werden, falls ein konkreter Benutzer diesen Dienst nicht ausführen darf. Nachteil einer festen Platzierung für die Durchführung der Zugriffskontrollüberprüfung ist sicherlich, dass die Effizienz der Berechtigungsüberprüfung in Frage zu stellen ist, wenn die generierte Berechtigungsüberprüfung mehrmaliges Lesen derselben Assets erfordert, wie in Kapitel 6.1.3 für den D-Dienst diskutiert.

Zusätzlich sind für die seiteneffektfreien Dienste R und Q weitere Frameworkerweiterungen für die Erstellung der Response vorgesehen. Dadurch ist es möglich, einzelne Attribute von Geschäftsobjekten sowie einzelne Suchergebniszeilen auszublenden oder durch eine Zugriffsverweigerungsmeldung zu ersetzen. Da ein R-Dienst häufig viele Attribute eines Geschäftsobjekts anzeigt, für die Attribute aber häufig identische Berechtigungen gelten, ist hier eine Caching-Strategie zur Erhöhung der Effizienz von Vorteil.

Die Lückenlosigkeit der Zugriffskontrolldurchsetzung für die Geschäftslogik eines Dienstbearbeiters wird durch eine statische Quellcodeanalyse der Dienstbearbeiter-Klassen der betrieblichen Anwendung gewährleistet. Die Analyse extrahiert aus dem Quellcode die Datenstruktur *AccessTypes*. Diese bildet ein zusammenhängendes Netz aus innerhalb der Geschäftslogik eines Dienstbearbeiters verwendeten Assets und den berechtigungsrelevanten Aktionen, die auf diesen ausgeführt werden. Aus diesen Informationen ist es je nach Struktur der zu den Aktionen gehörenden Zugriffsprinzipien möglich, Quellcode für die Zugriffsdurchsetzung automatisch oder semi-automatisch zu erstellen. Im vorgestellten Algorithmus werden alle Zugriffskontrollüberprüfungen in die vorgesehene *checkAccess( )*-Methode generiert, auch wenn hierdurch die Assets redundant gelesen werden müssen.

Hier wird eine konservative Berechtigungsüberprüfung realisiert, welche vom Kontrollfluss der Anwendung abstrahiert, d. h., auch wenn es zur Laufzeit der Anwendung nur einen Kontrollfluss geben kann, so wird die Berechtigungsdurchsetzungsfunktion so berechnet, als würden alle möglichen Kontrollflüsse ausgeführt. Für die meisten einfachen Dienste ist diese konservative Abschätzung ausreichend, da davon ausgegangen werden kann, dass für die einzelnen Zweige der Kontrollflüsse ähnliche Berechtigungen benötigt werden. Für die hier behandelten Dienste treten außer den Iteratoren, die Suchergebnisse durchlaufen und die durch die Analyse erkannt werden, keine Kontrollflüsse auf.

Die Generierung einer exakten Berechtigungsdurchsetzung für einen Dienst erfordert entweder, dass außer den Kontrollflüssen, die durch Iteratoren entstehen (siehe zweites Beispiel für die Policygenerierung aus Kapitel 6.2.4), auch andere Kontrollflüsse, wie etwa bedingte Anweisungsfolgen, in der *checkAccess()*-Methode nachgebildet werden müssen, oder dass mehrere bzw. beliebige Punkte im Kontrollfluss der Anwendung für die Platzierung einer Berechtigungsdurchsetzung zugelassen werden müssen. Der Algorithmus für die statische Codeanalyse und die AccessTypes müssen dann entsprechend erweitert werden. Die AccessTypes könnten z. B. für jede Annotation weitere Zustandsinformationen, wie etwa die Zeilennummer, in der eine Aktion aufgerufen wird, enthalten. Der erweiterte Algorithmus muss den Kontrollflussgraphen analysieren.

Die erste Variante für die Generierung einer exakten Berechtigungsdurchsetzung, dies ist die Variante bei der weiterhin nur ein Eingriffspunkt pro Dienst vorgesehen ist, ist natürlich nur dann möglich, wenn Auswertungsergebnisse von Bedingungen in bedingten Anweisungsfolgen und Schleifen nicht von vorher in der Geschäftslogik vorkommenden Zustandsänderungen abhängen. Hierfür müssen die AccessTypes so erweitert werden, dass auch Schleifen und Verzweigungen in den AccessTypes abgebildet werden, so dass diese für die automatische Quellcodegenerierung nachgebildet werden können. Hierbei müssen nur Schleifen nachgebildet werden, in deren Rumpf Aktionen auf Assets ausgeführt werden, die mit einer assetbezogenen Berechtigung assoziiert sind. Assetunab-

hängige Berechtigungen wie etwa rollenbasierte Berechtigungen können als Schleifeninvariante betrachtet werden und müssen nur einmal unabhängig vom Anwendungscode der Schleife überprüft werden. Für Verzweigungen gilt analoges. Sie müssen für assetunabhängige Berechtigungen nicht nachgebildet werden, sondern nur, falls es für verschiedene Ausführungspfade verschiedene assetabhängige Berechtigungen gibt.

In der zweiten Variante kann jeder bedingte Ausführungspfad zu Beginn separat mit einer Berechtigungsüberprüfung versehen werden. Der Algorithmus muss hier entsprechende Teilüberprüfungen an den Verzweigungspunkten generieren. Bei dieser Variante müssen zu jedem Zeitpunkt der Dienstausführung alle Kontextinformationen, wie z. B. der Sitzungsnutzer, bereitgestellt werden, die für die jeweilige Berechtigungsüberprüfung notwendig sind. Es ist zu beachten, dass dann die in Kapitel 4.2.4 aufgestellte und in Kapitel 6.1.4 in einem konzeptuellen Frameworkdesign umgesetzte Anforderung an die Konfigurierbarkeit der Response bei Nutzung einer Benutzeroberfläche nur eingeschränkt realisierbar ist, denn es steht nicht mehr vor Dienstausführung fest, ob der gesamte Dienst durch einen Benutzer wird ausgeführt werden können.

7 Fazit und Ausblick

7.1 Fazit

In der vorliegenden Arbeit wurde ein Modell für betriebliche Anwendungen aufgestellt. Dieses Modell wurde als Basis verwendet, um eine Sicherheitsarchitektur für die Zugriffskontrolle betrieblicher Anwendungen zu erstellen. Weiterhin wurden Anforderungen an die Zugriffskontrolle betrieblicher Anwendungen identifiziert. Die später entwickelte Spezifikationssprache für Berechtigungen sowie die entwickelten Frameworkelemente für die Integration der Zugriffskontrolle in eine Anwendung erfüllen diese Anforderungen. Desweiteren wurde die Implementierung einfacher Dienste untersucht, und auf Basis dieser ein Algorithmus aufgestellt, mit Hilfe dessen die Aktionen für alle im Dienst zugegriffenen Assets extrahiert werden können und eine Zugriffsdurchsetzung generiert werden kann.

Die Lückenlosigkeit der Zugriffskontrollüberprüfung wird durch eine statisch aus dem Quellcode generierte AEF für die einzelnen Dienste der betrieblichen Anwendung gewährleistet. Die Effizienz der Zugriffskontrollüberprüfung ist hierbei nur für einige Diensttypen, insbesondere für den U-Dienst und einfache R- und C-Dienste, gegeben.

Es wurde ein Vorgehensmodell beschrieben, mit dem es möglich ist, komplexe Berechtigungen zu definieren und ohne zusätzlichen Entwickleraufwand in die betriebliche Anwendung zu integrieren. Durch die Gestaltung eines geeigneten Frameworks ist die Integration der Zugriffskontrolldurchsetzung vom funktionalen Code entkoppelt, so dass ein- und dieselbe betriebliche Anwendung mit verschiedenen Zugriffskontrollpolicies ausgestattet werden kann. Hierdurch ist es möglich, die Installation einer betrieblichen Anwendung an unterschiedliche Zugriffskontrollanforderungen anzupassen.

Die Spezifikation der Berechtigungen erfolgt auf Basis des objektorientierten Datenmodells der verwalteten Geschäftsobjekte. Hierfür wird eine Berechtigung in zwei Bestandteile zerlegt, dieses sind das Zugriffsprinzip und die Aktion. Ein Zugriffsprinzip, oder eine *PathExpression*, spiegelt hierbei den Subjekt-Objekt-Teil einer Berechtigung wider. Ein Zugriffsprinzip wird als Pfad im Datenmodell der Anwendung spezifiziert, wobei es einen Assettyp im Datenmodell gibt, welcher das Subjekt einer Berechtigung repräsentiert. Dieser liegt der Anwendung als Quellcodeklasse vor. Auf diese Weise können die implementierten Zugriffsprinzipien wieder verwendet werden. Eine Zugriffsentscheidung prüft, ob ein Pfad des zur Berechtigung gehörenden Zugriffsprinzips in der Datenbank der betrieblichen Anwendung existiert. Die Zugriffsentscheidung stellt also eine Mini-API für Datenbankabfragen bereit, die es ermöglicht, eine Existenz-Anfrage auf Semi-Joins an die Datenbank zu stellen. Diese API kann um mengenwertige Anfragen erweitert werden, so dass die der Anwendung bereitgestellten Zugriffsprinzipien auch für Suchanfragen der Q-Dienste der Anwendung verwendet werden können. Für die Zugriffsprinzipien wurden eine Grammatik sowie eine graphische Notation aufgezeigt, die die in der Anwendung vorhandenen Zugriffsprinzipien auf intuitive und anschauliche Weise darstellt.

Für die Durchsetzung der Zugriffskontrolle wird zunächst ein Zugriffskontrollframework entwickelt. Dieses sieht sowohl für die Geschäftslogik eines Dienstbearbeiters einen dedizierten Eingriffspunkt für die Einstellung von Quellcode zur Berechtigungsüberprüfung vor als auch Eingriffspunkte, die während der Erstellung der Antwortnachricht genutzt werden können. Für Q-Dienste wurde eine

spezielle Erweiterung der PathExpressions für die Realisierung einer Limited View entwickelt, die es ermöglicht, Suchanfragen zu stellen, bei denen für einen Sitzungsbenutzer nicht sichtbare Ergebnisobjekte gar nicht erst im Suchergebnis enthalten sind.

Weiterhin werden für die Durchsetzung der Zugriffskontrolle die in den Berechtigungen spezifizierten Aktionen mit konkreten Quellcode-Anweisungen assoziiert. Eine statische Codeanalyse erstellt aus dem Quellcode eines jeden Dienstbearbeiters der Anwendung eine Datenstruktur, die *AccessTypes*. Die statische Codeanalyse bedient sich dabei der abstrakten Interpretation, so dass nur die für die Zugriffskontrolle wichtigen Quellcode-Artefakte von der statischen Codeanalyse analysiert werden müssen. Die Datenstruktur *AccessTypes* enthält Zugriffskontrollinformationen, die notwendig sind, um Quellcode für die Zugriffsdurchsetzung automatisch in die im Framework bereitgestellten Eingriffspunkte zu generieren. Sie stellt quasi eine Zusammenfassung über die relevanten im Quellcode vorkommenden Anweisungen dar, kann also auch die für Q- und D-Dienste benötigten Iteratorkonstrukte, welche über Assets iterieren, abbilden. Aus den *AccessTypes* können in einem zweiten Schritt die für einen Dienst notwendigen Berechtigungsüberprüfungen (semi-)automatisch generiert werden. Hierzu werden die innerhalb eines Dienstes vorkommenden Aktionen mit ihren Zugriffsprinzipien assoziiert. Sind für die konkreten Ausprägungen der Zugriffsprinzipien die Variablen im Quellcode bekannt, die als Parameter für die mit Hilfe der Zugriffsprinzipien generierten Datenbankanfragen fungieren müssen, so kann aus den Zugriffsprinzipien eine konkrete Policy für die Datenbankabfrage als Zugriffskontrolldurchsetzung generiert werden. Ansonsten ist die Generierung noch semi-automatisch nach Rückfrage mit dem Entwickler möglich. Durch die Struktur der *AccessTypes* können auch die im Quellcode vorkommenden Iteratorkonstrukte für die Generierung von Quellcode nachgebildet werden, sofern sie für die Berechtigungsüberprüfung relevant sind.

Durch die statische Codeanalyse ist sichergestellt, dass es keine Ausführungshistorien der Dienste der Anwendung gibt, die nicht durch die Zugriffskontrolle geschleust werden. Die Zugriffskontrolldurchsetzung ist also lückenlos. Die Effizienz der Berechtigungsüberprüfung soll dadurch gewährleistet werden, dass die Zugriffskontrolldurchsetzung nur zu Beginn eines jeden Dienstes ausgeführt wird. Bei der Generierung des Quellcodes zur Zugriffskontrolldurchsetzung reicht es aus, wenn innerhalb eines Dienstes mehrfach benötigte Berechtigungen nur einmal abgefragt werden. Doppelte Berechtigungsüberprüfungen werden also eliminiert. Durch die Generierung der Berechtigungsüberprüfung an genau einer vorgesehenen Stelle im Quellcode kann es allerdings zur Erstellung von ineffizientem Quellcode kommen. Dieses ist der Fall, wenn für die Berechtigungsüberprüfung Assets ausgelesen werden müssen, welche innerhalb der Diensterbringung ein zweites Mal ausgelesen werden.

Sowohl für die in dieser Arbeit entwickelte Framework-Erweiterung als auch für die Berechtigungsspezifikationssprache und für die statische Codeanalyse existiert eine prototypische Implementierung, die wesentliche Teile realisiert und so zeigt, dass deren Umsetzung möglich ist.

7.2 Ausblick

7.2.1 Erweiterte Spezifikation von Berechtigungen

Die hier gewählte Spezifikation von Berechtigungen beinhaltet Zugriffsprinzipien, welche auf dem objektorientierten Datenmodell spezifiziert und in eine Datenbankanfrage übersetzt werden. In der vorliegenden Arbeit wurden nur solche Pfade im Datenmodell zugelassen, deren Länge statisch bekannt ist. Für einige Zugriffspolicies ist es jedoch notwendig, rekursive Pfade, deren Länge nicht statisch bekannt ist, zu definieren. Diese können nützlich sein, wenn hierarchisch strukturierte Ressourcen die Vererbung von Berechtigungen erlauben. In einer Bankanwendung sind z. B. den Vorgesetzten mindestens genauso viele Rechte eingeräumt wie ihren Mitarbeitern. Der Zugriff auf das Konto eines Kunden sei also nicht nur dem direkten Betreuer, sondern auch den direkten und indirekten Vorgesetzten des Betreuers erlaubt (siehe Abb. 7.1). Ein indirekter Vorgesetzter ist hierbei ein Vorgesetzter über mehrere Hierarchiestufen hinweg.

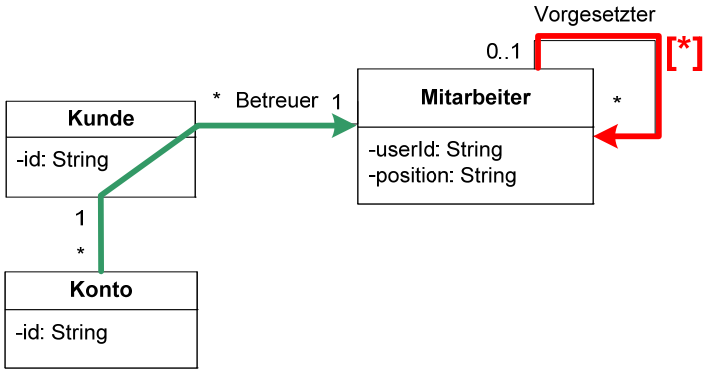


Abb. 7.1: Minimalmodell einer Bank

Das Modell der Zugriffsprinzipien kann auf Schleifen in den Zugriffspfaden erweitert werden. Dann allerdings ist es nicht mehr möglich, eine einzige, simple Datenbankanfrage aus einer definierten Berechtigung zu erstellen. Die Datenbankanfrage muss dann in drei Abschnitte unterteilt werden: den Pfadbeginn mit dem ersten Schleifendurchlauf, die Schleife selbst für alle weiteren Schleifendurchläufe und das Pfadende. Jeder Schleifendurchlauf wird durch eine einzelne Teilabfrage abgebildet. Eine Anfragesprache, die das Durchsuchen solcher Schleifenstrukturen erlaubt ist Datalog (siehe z. B. [KE06]). Im obigen Beispiel besteht der Pfadbeginn aus dem Pfad von Konto über Kunde über Mitarbeiter über die Vorgesetztenrolle zum Mitarbeiter, die Schleife ist mit [*] gekennzeichnet und bezeichnet den Pfad über die Vorgesetztenrolle. Das Pfadende ist leer. Um herauszufinden, ob einem Subjekt, welches eine Vorgesetztenfunktion hat, der Zugriff auf ein spezielles Konto erlaubt ist, müssten zunächst startend beim Konto, alle Mitarbeiter gefunden werden, die über genau eine Vorgesetzten-Beziehung direkten Zugriff auf das Konto haben. Falls das anfragende Subjekt in dieser Menge ist, kann die Zugriffsüberprüfung hier mit einer positiven Rückantwort beendet werden. Falls das anfragende Subjekt nicht in dieser Menge ist, wird die Vorgesetztenschleife noch einmal durchlaufen. Dieses wird so oft wiederholt, bis das anfragende Subjekt in der gesuchten Menge ist, oder bis alle Hierarchiestufen durchlaufen wurden.

Ebenso wurden Verpflichtungen und Delegation in dem hier vorliegenden Modell der Berechtigungen nicht berücksichtigt. Verpflichtungen können einfach integriert werden, indem eine neue Klasse in das Berechtigungsmodell aufgenommen wird, welches die Verpflichtung repräsentiert. Diese enthält eine Methode, die Quellcode enthält, der ausgeführt wird, wenn eine Berechtigung abgefragt wird, die mit der gegebenen Aktion assoziiert ist (siehe Abb. 7.2).

Die Delegation kann in das Modell der PathExpressions aufgenommen werden, indem eine zusätzliche Assoziation, die bei der Klasse endet und aufhört, die das Subjekt darstellt, eingeführt wird. Diese Assoziation beschreibt, dass ein Subjekt ein Recht an ein anderes Subjekt delegiert hat.

Ein wichtiger Bestandteil einer jeden Berechtigungsspezifikation ist die Überprüfung der Spezifikation auf Konsistenz. Die einzelnen Berechtigungen einer Gesamtspezifikation müssen konfliktfrei miteinander interagieren können. Die Überprüfung der Spezifikation auf Konsistenzbedingungen ist ein weiteres Forschungsgebiet. Abhandlungen zur Konsistenz und die Behandlung von Konflikten innerhalb einer Policy finden sich in [Spi85, LS99, Mar03].

Weiterhin können für die Ausführung eines Dienstes viele Berechtigungsabfragen notwendig sein. Die Auswertungsergebnisse von Berechtigungsabfragen können einander z. T. implizieren. So kann es z. B. in der Anwendung vorhandene Beziehungen zwischen Gruppenmitgliedschaften wie folgt geben: ist ein Benutzer Mitglied in Gruppe A, so ist er auch Mitglied in Gruppe B. Falls eine Berechtigungsabfrage ergibt, dass ein Benutzer Mitglied in Gruppe A ist, so braucht keine Berechtigungsabfrage stattfinden, die überprüft, ob derselbe Benutzer Mitglied in Gruppe B ist. Das Ergebnis ist im Vorhinein bereits bekannt. Sind diese Implikationsbeziehungen zwischen Policies bekannt, so kann die Effizienz der Berechtigungsüberprüfung für eine Anwendung erhöht werden, indem die sich implizierenden Policies nicht mehr abgefragt werden müssen. Eine Berechtigungsbeschreibungssprache, die die Spezifikation ähnlicher Ableitungsregeln vorsieht, ist die in Kapitel 5.2.2 und 5.2.6 diskutierte Sprache ASL [JSS97].

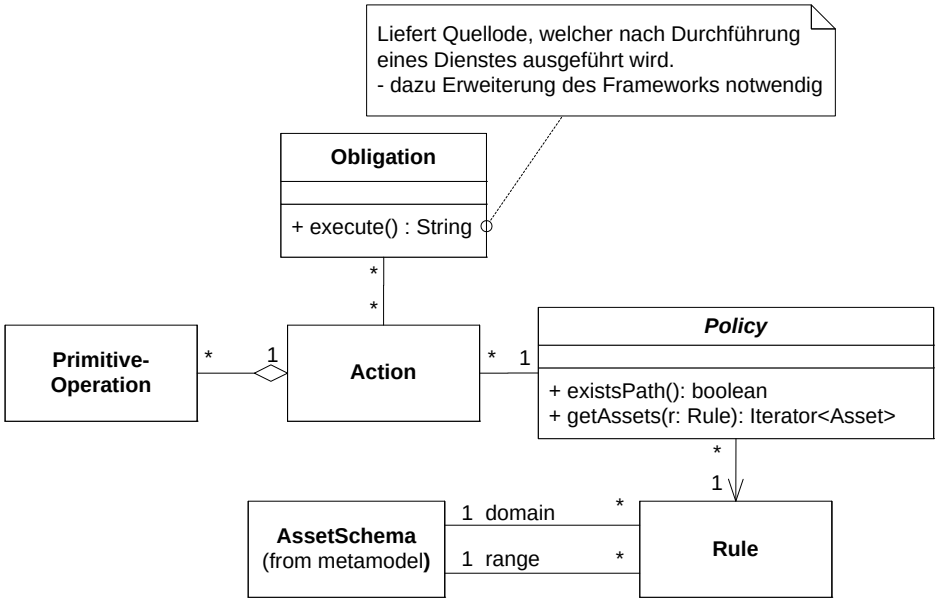


Abb. 7.2: Erweiterung der Berechtigungsspezifikation um Verpflichtungen

7.2.2 Informationsflusskontrolle

Ein interessantes Gebiet, welches im Zusammenhang mit der Zugriffskontrolle steht, ist die Informationsflusskontrolle [Mye99, SM03]. Während die Zugriffskontrolle nur den Zugriff auf eine Information oder ein Datum kontrolliert, beobachtet die Informationsflusskontrolle, wie sich einzelne Informationen oder Daten innerhalb einer Anwendung ausbreiten. Z. B. könnte eine geschützte Information innerhalb eines Programms in eine Variable kopiert werden, die nicht durch die Zugriffskontrolle abgedeckt ist. Diese Variable könnte z. B. öffentlich sein und von jedem Benutzer ausgelesen werden können. Hierdurch wird die Zugriffskontrolle ausgehebelt, bzw. diese Szenarien werden durch Mittel der Zugriffskontrolle nicht abgedeckt. Die Informationsflusskontrolle sucht nach solchen versteckten Kanälen und versucht Mechanismen bereitzustellen, die versteckten, ungewollten Informationsfluss verhindern. Es existiert eine Erweiterung von Java, die jedem Java-Typ ein Sicherheitslabel zuordnet. Dadurch ist es einfach möglich, den Informationsfluss zu analysieren [JiF06].

7.2.3 Administration und Wartung von Berechtigungen

Berechtigungen werden nicht einmalig spezifiziert, sondern können sich im Laufe des Betriebs der Anwendung ändern, so dass eine Wartung der spezifizierten Rechte erforderlich ist. Zu diesem Zweck sollte ein für den Administrator angenehmes Interface, vorzugsweise eine maskenorientierte GUI, zur Verfügung stehen. Die Verwaltung und Wartung von Berechtigungen ist eine große Herausforderung. Häufig kommt es nach einer gewissen Zeit zu einem Berechtigungswildwuchs, welcher sich aus der Unübersichtlichkeit eines Berechtigungskonzepts und fehlender Administrationsmöglichkeiten ergibt. Bis jetzt existiert kaum Literatur zu diesem Thema. In [Alt04] wird ein Vorschlag für die Administration von rollenbasierten Berechtigungen für Geschäftsprozesse einer betrieblichen Anwendung gemacht.

Eine Administrationsoberfläche, welche Berechtigungen auf Basis der PathExpressions verwaltet, müsste die Pflege von Berechtigungen erlauben sowie eine Reihe von Suchfunktionen zur Verfügung stellen. Für die Pflege der Berechtigungen müssen die einzelnen Zugriffsprinzipien verwaltet werden, es müssen neue Pfade und Berechtigungen angelegt, gelesen, editiert und gelöscht werden können. Spezifizierte Pfade müssen auf Integrität überprüft werden, d. h. es muss überprüft werden, ob der

Pfad spezifikationskonform ist. Weiterhin müssen die Aktionen verwaltet werden. Hier gibt es zu jedem Geschäftsobjekttyp vordefinierte Aktionen: *create*, *read*, *update* und *delete*, sowie feingranulare Lese- und Schreibaktionen auf den Attributen der Geschäftsobjekttypen. Die Berechtigungsspezifikation muss vollständig sein, so dass zu allen Aktionen aller Geschäftsobjekttypen Rechte definiert sind. Der Administrationsaufwand kann durch die Definition einer Defaultberechtigung verringert werden. Weiterhin müssen definierte Berechtigungen auf Widerspruchsfreiheit überprüft werden, z. B. sollten Update-Berechtigungen, die Read-Berechtigungen auf einem Geschäftsobjekt höchstens einschränken, um die Workflowabhängigkeiten der betrieblichen Anwendung zu berücksichtigen. Denn ansonsten gäbe es Situationen, in denen blindes Schreiben erlaubt ist. Selbiges gilt für hierarchische Berechtigungen: Read- oder Update-Berechtigungen auf Attributen von Geschäftsobjekten sollten die dazugehörige Read- oder Update-Berechtigung auf einem Geschäftsobjekt ebenfalls höchstens einschränken. Weiterhin ist eine ganze Reihe an Fehlern abzufangen. Namen für Zugriffsprinzipien und Berechtigungen sollten eindeutig vergeben werden, und es sollte keine zwei identischen Zugriffsprinzipien oder Berechtigungen mit unterschiedlichen Namen geben, da sonst die Übersichtlichkeit beeinträchtigt und der Wildwuchs von Berechtigungen gefördert wird.

Um die Pflege der Berechtigungen zu erleichtern, sollte eine große Bandbreite an Suchfunktionalitäten angeboten werden, so dass Aktionen und Zugriffsprinzipien nach verschiedenen Kriterien gesucht werden können, z. B. sollte nach Aktionen gesucht werden können, denen dieselben Zugriffsprinzipien zugeordnet sind.

In der in dieser Arbeit entwickelten Berechtigungsspezifikationssprache liegen die Zugriffsprinzipien der Anwendung als Quellcodeklassen vor. Der Quellcode soll als einzige Dokumentationsquelle dienen. Von daher sollte ein Werkzeug zur Berechtigungsverwaltung eine zweiseitige Administration bieten. Zum einen sollten über die Administrationsoberfläche neue Pfade, Berechtigungen und Aktionen definiert werden können, die dann automatisch in Quellcode-Artefakte umgewandelt werden, zum anderen sollten im Quellcode vorhandene Pfade, Berechtigungen und Aktionen erkannt und dargestellt werden können sowie über die Oberfläche editierbar sein können. Des Weiteren müssen einige Wartungsfunktionen zur Verfügung gestellt werden. Fehlerhafte Pfade und Berechtigungen sollten identifiziert werden können. Vorher korrekte Pfade können fehlerhaft werden, wenn sich das objektorientierte Datenmodell der Anwendung ändert. Weiterhin sollte es eine Anzeige geben, falls neue Klassen im Datenmodell hinzugekommen sind, für die noch keine Berechtigungen existieren. Zur Erleichterung der Administration sollten Namenskonventionen für Pfade, Berechtigungen und Aktionen eingehalten werden.

Die Abbildungen 7.3 und 7.4 umreißen, wie eine Administrationsoberfläche auf Basis der PathExpressions aussehen könnte. Sie zeigen die Dialoge zum Anlegen eines neuen Pfades im Datenmodell. Links in Abbildung 7.4 befindet sich eine Navigationsleiste, welche alle in der Anwendung vorkommenden Geschäftsobjekttypen, alle die zu den Geschäftsobjekttypen spezifizierten Zugriffsprinzipien und Aktionen enthält. In der Mitte ist das objektorientierte Datenmodell der Anwendung visualisiert. Rechts befinden sich, je nach Kontext, verschiedene Menus zur Definition eines Pfades oder einer Aktion.

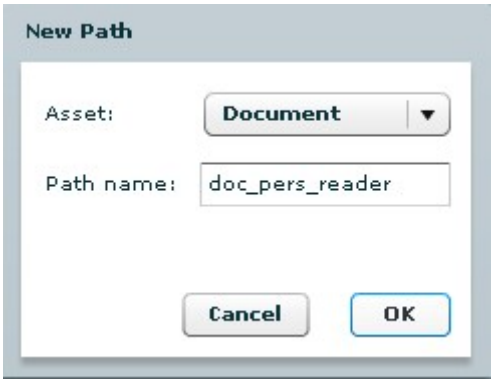


Abb. 7.3: Dialog zum Anlegen eines leeren Pfades

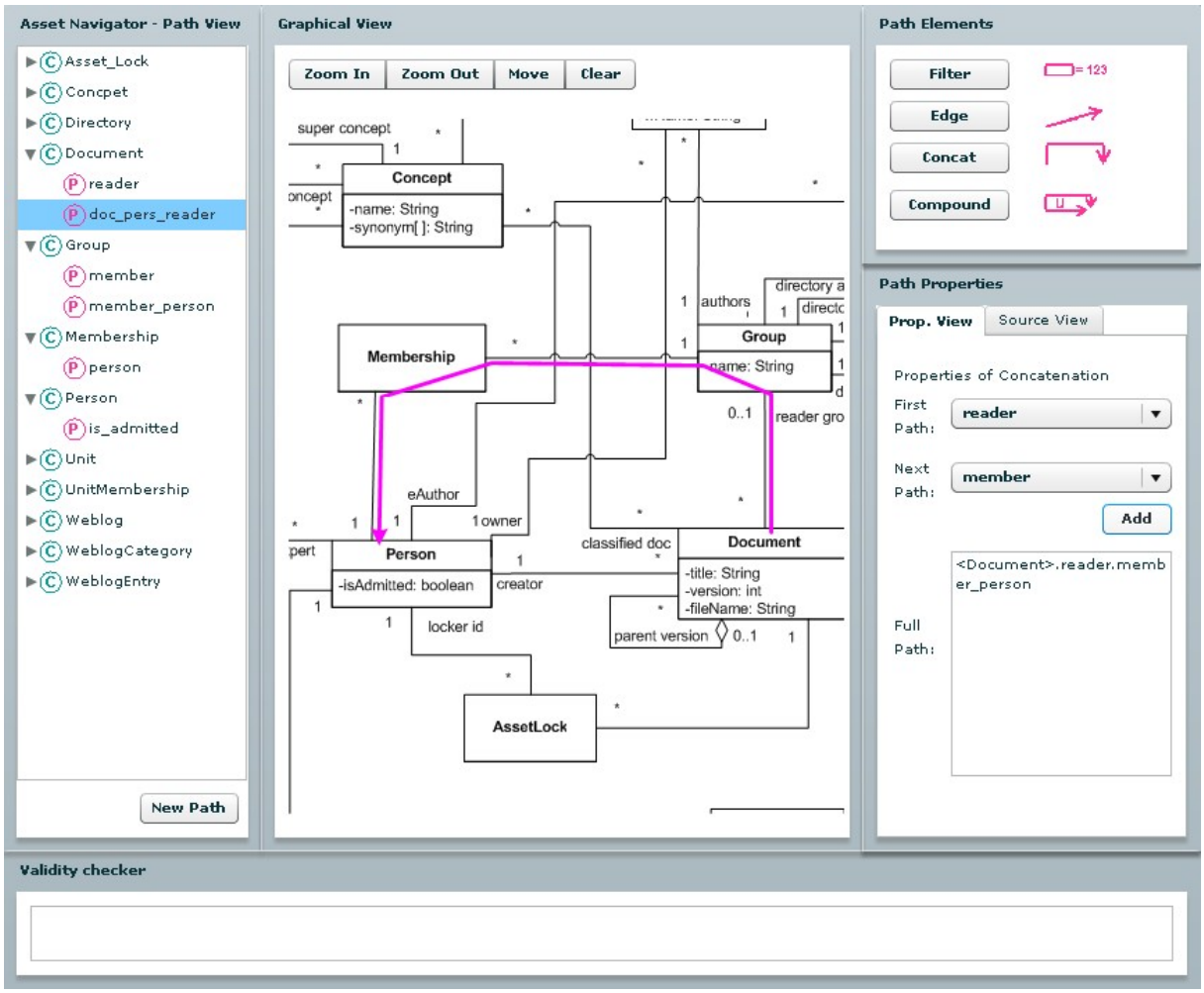


Abb. 7.4: Dialog zur Spezifikation eines Zugriffsprinzips

7.2.4 Effiziente Berechtigungsdurchsetzung mittels Caching

In der vorliegenden Arbeit wurde ein Ansatz vorgestellt, mit dem es möglich ist, Berechtigungen für die Ausführung der Geschäftslogik eines Dienstes zu bündeln und vor Dienstausführung abzufragen. Diese Bündelungsstrategie war aber nicht während der Erstellung der Antwortnachricht einsetzbar, sondern nur für die eigentliche Geschäftslogik eines Dienstbearbeiters. Während der Erstellung der Response war in der entwickelten Sicherheitsarchitektur direkt vor jeder Ausgabe eines Assets, welches in einem Suchergebnis auftaucht, sowie direkt vor jeder Ausgabe eines jeden Attributwerts eines Geschäftsobjekts eine Berechtigungsüberprüfung vorgesehen. Insbesondere Berechtigungen zum Lesen eines Attributwerts sind häufig für viele Attribute eines Geschäftsobjekts identisch. Statt jedes Mal eine Berechtigung auszuwerten, könnte hier ein Caching des Ergebnisses einer Berechtigungsabfrage gespeichert werden, so dass bei einer erneuten Abfrage der Wert aus dem Cache genommen werden kann. Dieses Vorgehen ist insbesondere für gruppenbezogene Policies sinnvoll, welche unabhängig von einem bestimmten Geschäftsobjekt sind.

Eine Berechtigungspolicy besteht häufig aus mehreren Regeln, von denen einige gruppenbezogen und andere eigentümer- oder assetbezogen sein können. Hier kann ein Caching von Auswertungsergebnissen einzelner Regeln, insbesondere der gruppenbezogenen Regeln, sinnvoll sein, so dass bei einer geforderten Berechtigungsüberprüfung nicht mehr alle Regeln, sondern nur noch Teile der gesamten Policy überprüft werden müssen.

Auswertungsergebnisse für gruppenbezogene Policies können für jeden Benutzer in seiner Sitzung gespeichert werden, da sie sich voraussichtlich während einer Sitzung nicht ändern. Auswertungsergebnisse eigentümer- und assetbezogener Policies gelten nur für ein bestimmtes Geschäftsobjekt. Eine Speicherung aller dieser Policyauswertungen innerhalb einer Sitzung wäre sehr aufwendig, da während einer Sitzung sehr viele verschiedene Geschäftsobjekte gelesen, aktualisiert und gelöscht werden können. Es würde sich aber ein Caching dieser Berechtigungsergebnisse innerhalb der Erstellung einer Antwortnachricht lohnen.

7.2.5 Berechtigungen für verteilte Anwendungen

In dieser Arbeit wurde die Zugriffskontrolle einer einzelnen betrieblichen Anwendung behandelt. Innerhalb eines Unternehmens agieren jedoch viele, unter Umständen hunderte von Anwendungen miteinander, so dass die Zugriffspolicies der einzelnen Anwendungen untereinander konsolidiert werden müssen. Ein Ansatz für eine gemeinsame Spezifikationssprache bietet hier XACML [GM+03]. Eine weitere Herausforderung ergibt sich bei der Interaktion von Diensten in einer dienstorientierten Architektur. Die einzelnen Dienste setzen ihre eigenen Zugriffskontrollpolicies durch. Dienste können aber wiederum Bestandteil anderer Dienste bzw. Bestandteil eines intra-organisatorischen Workflows sein. Während die vorliegende Arbeit die Konsolidierung einer Zugriffspolicy innerhalb eines einzelnen Dienstes einer betrieblichen Anwendung anstrebt, werden in [Wim07] Konzepte entwickelt, mit Hilfe derer die Zugriffspolicies für zusammengesetzte Anwendung konsolidiert werden können.

Literatur

[Ace06] Acegi Technology Pty Limited: *Acegi Security System for Spring* / SourceForce™ .net. http://www.acegisecurity.org (besucht am 4. Januar 2007)

[AI04] ANSI INCITS 359-2004: *Role-Based Access Contro*l.

[Alt04] Alter, Ewgeny: *Werkzeugunterstützte Analyse von Berechtigungen gegenüber Geschäftsprozessen*. Technische Universität München, Diplomarbeit, 2004.

[And01] Anderson, Ross: *Security Engineering*. New York : John Wiley & Sons, 2001. ISBN 0-471-38922-6

[ASL89] Alashqur, A. M.; Su, Stanley Y. W.; Lam, Herman: „OQL: A Query Language for Manipulating Object-oriented Databases". In: 15th International Conference on Very Large Data Bases (VLDB) (1989), Amsterdam, The Netherlands, Morgan Kaufmann (Hrsg.), S. 433--442.

[Bal00] Balzert, Helmut: *Lehrbuch der Software-Technik : Software-Entwicklung*. Heidelberg : Spektrum, Akademischer Verlag, 2. Auflage, 2000. ISBN 3-8274-0480-0

[BB+06] Berglund, Anders; Boag, Scott et al.: *XML Path Language (XPath) 2.0 : W3C Proposed Recommendation 21 November 2006*. http://www.w3.org/TR/xpath20/ (besucht am 4. Januar 2007)

[BCE+06] Ball, Jennifer; Carson, Debbie; Evans, Ian; Fordin, Scott; Haase, Kim; Jendrock, Eric: (Februar 2006) *The Java™ EE 5 Tutorial : For Sun Java System Application Server Platform Edition 9*. http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf (besucht am 4. Januar 2007)

[BDL03] Basin, David; Doser, Jürgen; Lodderstedt, Torsten: „Model Driven Security : from UML Models to Access Control Infrastructures", Technischer Bericht 414 2003.

[Bib77] Biba, Ken J.: *Integrity considerations for secure computer system*s. Technischer Bericht TR-3153. MITRE Corporation, Bedford, Massachusetts, April 1977.

[BL76] Bell, D. E.; La Padula, L. J.: *Secure Computer System : Unified Exposition and MULTICS Interpretation*. Technischer Bericht MTR-2997, MITRE Corporation, Bedford, Massachusetts, März 1976. http://csrc.nist.gov/publications/history/bell76.pdf (besucht am 4. Januar 2007)

[BN89] Brewer, D. F. C.; Nash, M. J.: „The Chinese Wall Security Policy“. In: IEEE Symposium on Security and Privacy (1989), Oakland, California, S. 206--214. http://www.gammassl.co.uk/topics/chwall.pdf (besucht am 4. Januar 2007)

[BPS+06] Bray, Tim; Paoli, Jean; Sperberg-McQueen, C.M. et al.: *Extensible Markup Language (XML) 1.1 (Second Edition)*, W3C Recommendation, 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/ (besucht am 21. Februar 2007)

[Buc03] Bucksteeg, Andreas: *Methoden und Techniken zur Sicherstellung des vertrauenswürdigen Managements von Benutzerprofildaten in vernetzten Anwendungen*. Technische Universität München, Diplomarbeit, 2003.

[Bue07] Büchner, Thomas: *Introspektive modellgetriebene Softwareentwicklung*. Technische Universität München, Dissertation, 2007.

[Cha03] Chalfant, Thomas M.: *Role Based Access Control and Secure Shell — A Closer Look At Two Solaris™ Operating Environment Security Features*. Sun Microsystems, Inc., Sun BluePrints™ OnLine, June 2003. http://www.sun.com/blueprints/0603/817-3062.pdf (besucht am 5. Januar 2007)

[Cou01] Cousot, Patrick: „Abstract Interpretation Based Formal Methods and Future Challenges”. In: Informatics --- 10 Years Back, 10 Years Ahead / Wilhelm, R. (Hrsg.). Lecture Notes in Computer Science (2001), Springer-Verlag, Vol. 2000, S. 138--156.

[CW87] Clark, David D.; Wilson, David R.: „A Comparison of Commercial and Military Computer Security Policies”. In: IEEE Symposium on Security and Privacy (1987), IEEE Computer Society Press, Los Alamitos, California, S. 184--194.

[DDL+00] Damianou, Nicodemos; Dulay, Naranker; Lupu, Emil; Sloman, Morris: „Ponder: A Language for Specifying Security and Management Policies for Distributed Systems : The Language Specification”. Imperial College Research Report DoC 2000/1, 2000. http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PonderSpec.pdf (besucht am 7. März 2007)

[DDL+01] Damianou, Nicodemos; Dulay, Naranker; Lupu, Emil; Sloman, Morris: „The Ponder Specification Language”. Lecture Notes of Computer Science (2001), Springer-Verlag, Vol. 1995, S. 18--38.

[DeT02] DeTreville, John: *Binder, a logic-based security language*. Microsoft Research, Technischer Bericht MSR-TR-2002-21, 2002.

[Die04] Dierstein, Rüdiger: „Sicherheit in der Informationstechnik – der Begriff IT-Sicherheit“. Informatik Spektrum (2004), Springer-Verlag, Vol. 27, Nr. 4, S. 343--353.

[Die05] Dierstein, Rüdiger: *IT-Sicherheit und ihre Besonderheiten – Duale Sicherheit*. Skript zur Vorlesung „IT-Sicherheit“, Wintersemester 2005 / 2006, Technische Universität München.

[Dij72] Dijkstra, Edsger W.: „Notes on Structured Programming : On The Reliability of Mechanisms”. In: Structured Programming (1972), O. J. Dahl et al. (Hrsg.), Academic Press, London, S. 1--82, Zitat S. 6.

[DIN88] DIN 44300 – Deutsche Norm – Informationsverarbeitung, Begriffe Teile 1–9. Deutsches Institut für Normung, Beuth Verlag GmbH, Berlin, November 1988.

[DPS03] De Capitani di Vimercati, Sabrina; Paraboschi, Stefano; Samarati, Pierangela: „Access control: principles and solutions“ In: Software – Practice and Experience (2003), John Wiley & Sons, Inc. : New York , Vol. 33, Nr. 5, S. 397--421.

[EA06] Eschweiler, Jörg; Atencio Psille, Daniel E.: *Security@Work*. Berlin : Springer Verlag, 2006. ISBN 3-540-22028-3

[Eck04] Eckert, Claudia: *IT-Sicherheit: Konzepte, Verfahren, Protokolle*. R. Oldenbourg Verlag, 2004. ISBN 3-486-20000-3

[EN02] Elmasri, Ramez; Navathe, Shamkant B.: *Grundlagen von Datenbanksystemen*. Pearson Studium, 2002. ISBN 3-8273-7021-3

[FKC03] Ferraiolo, David F.; Kuhn, D. Richard; Chandramouli, Ramaswamy: *Role-Based Access Control*. Boston : Artech House Inc., 2003. ISBN 1 – 58053 – 370 -1

[FS03] Foundstone Professional Services.
http://www.foundstone.com/index.htm (besucht am 5. Januar 2007)

[FSG+01] Ferraiolo, David F.; Sandhu, R.; Gavrila, S.; Kuhn, D. Richard; Chandramouli, Ramaswamy.: „Proposed NIST Standard for Role-Based Access Control". In: ACM Transactions on Information and Systems Security (2001), Vol. 4, Nr. 3, S. 224--274.

[FSV01] Fink, Andreas; Schneidereit, Gabriele; Voß, Stefan: *Grundlagen der Wirtschaftsinformatik*. Heidelberg : Physica-Verlag, 2001. ISBN 3-7908-1375-3

[GED03] Gong, Li; Ellison, Gary; Dagefore, Mary: *Inside Java 2 Platform Security : Architecture, API Design, and Implementation (2nd Edition)*. The Java Series, Addison-Wesley, 2003. ISBN 0-201-78791-1

[GHJ+95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2

[GKM+03] Großpietsch, K.-E.; Keller, H.; Münch, I.; Saglietti, F.: *Technische Sicherheit und Informationssicherheit* : Unterschiede und Gemeinsamkeiten. Technischer Bericht, 2003.

[GM+03] Godik, S.; Moses, T. et al.: *eXtensible Access Control Markup Language (XACML)*. OASIS Specification, 2003.
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
(besucht am 8. Januar 2007)

[Gol02] Gollmann, Dieter: *Computer Security*. Wiley, 2002. ISBN 0-471-97844-2

[GUW02] Garcia-Molina, Hector; Ullman, Jeffrey D.; Widom, Jennifer: *Database Systems : The Complete Book*. New Jersey : Prentice Hall, 2002. ISBN 0-13-031995-3

[HHV06] Hess, Andreas ; Humm, Bernhard; Voß, Markus: „Regeln für Serviceorientierte Architekturen hoher Qualität". Informatik Spektrum (2006), Springer-Verlag, Vol. 29, Nr. 6, S. 395--411.

[HK00] Hada, Satoshi; Kudo, Michiharu: *XML Access Control Language : Provisional Authorization for XML Documents*. Tokyo Research Laboratory, IBM Research, 2000.
http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html (besucht im Juli 2006)

[Hos92] Hosmer, Hilary H.: „Metapolicies I". In: ACM Special Interest Group on Security, Audit and Control (SIGSAC) (1992), Vol. 10, Nr. 2 – 3, S. 18--43.

[HRU76] Harrison, Michael A.; Ruzzo, Walter L.; Ullman, Jeffrey D.: „Protection in Operating Systems". In: Communications of the ACM (1976), Vol. 19, Nr. 8, S. 461--471.

[Hup05] Hupfauf, Franz: *Pflichtenheft ZugNeu*. HVB Systems GmbH, 2005.

[ISO93] ISO/IEC 2382-1: *Information technology -- Vocabulary -- Part 1: Fundamental terms*. 1993.

[ISO96] ISO/IEC-Norm 10181-3: *Information technology -- Open Systems Interconnection -- Security frameworks for open systems: Access control framework*. 1996.

[JBD98] Jahresbereicht 1998 des Berliner Datenschutzbeauftragten: „Teil 4.8 Organisation und Technik“. 1998.
http://www.datenschutz-berlin.de/jahresbe/98/teil4-8.htm (besucht am 1. Februar 2007)

[JiF06] JiF: Java + information Flow, http://www.cs.cornell.edu/jif/. (besucht im Juli 2006)

[JSS97] Jajodia, Sushil; Samarati, Pierangela; Subrahmanian, V. S.: „A Logical Language for Expressing Authorizations“. In: IEEE Symposium in Security and Privacy (1997), Oakland, California, S. 31--42.

[KBS05] Krafzig, Dirk; Banke, Karl; Slama, Dirk: Enterprise SOA : *Service-Oriented Architecture Best Practices*. Prentice Hall, 2005. ISBN 0-13-146575-9

[KE06] Kemper, Alfons; Eickler, André: *Datenbanksysteme : Eine Einführung*. Oldenbourg Verlag, 6. Auflage, 2006. ISBN 3-486-2690-9

[Kic96] Kiczales, Gregor: *Aspect-Oriented Programming*. ACM Computer Survey, Vol. 28, Nr. 4, 1996.

[KIL+97] Kiczales, Gregor; Irwin, John; Lamping; Loingtier, Jean-Marc; Videira-Lopes, Cristina; Maeda, Chris; Mendhekar, Anurag: „Aspect-Oriented Programming”. In: European Conference on Object-Oriented Programming (1997), Jyväskylä, Finland.
Lecture Notes in Computer Science, Springer-Verlag, Vol. 1241, S. 220--242.

[KLM+01] Kiczales, Gregor; Hilsdale, Erik; Hugunin, Jim; Kersten, Mik; Palm, Jeffrey; Griswold, William G.: „An Overview of AspectJ“. In: European Conference on Object-Oriented Programming (2001), Budapast, Hungary. Lecture Notes in Computer Science, Springer-Verlag, Vol. 2072, S. 327--355.

[Lam71] Lampson, Butler W.: „Protection“. In: Fifth Princeton Symposium on Information Sciences and Systems (1971), Princeton University, S. 437--443.

[LBD02] Lodderstedt, Torsten; Basin, David; Doser, Jürgen: „SecureUML : A UML-Based Modeling Language for Model-Driven Security”. In: UML 2002 – The Unified Modeling Language / Jézéquel, J.-M.; Hussmann, H.; Cook, S. (Hrsg.). Model Engineering, Languages, Concepts, and Tool. 5th International Conference, Dresden, Germany, September/October 2002. Proceedings Vol. 2460, Springer-Verlag, 2002, S. 426--441.

[LfD02] Landesbeauftragte für den Datenschutz (LfD), Bremen: *Aspekte des Datenschutzes bei SAP*. ca. 2002.
http://www.datenschutz-bremen.de/technik/datenschutz_sap.php
(besucht am 8. Januar 2007)

[LGK+99] Lai, Charlie; Gong, Li; Koved, Larry; Nadalin, Anthony; Schemers, Roland: „User Authentication and Authorization in the Java™ Platform”. In: 15th Annual Computer Security Applications Conference (1999), Phoenix, Arizona.

[LM04] Lehel, Vanda; Matthes, Florian: „Integration von Weblog-Funktionen in eine betriebliche Standardsoftware zum Wissensmanagement“. Beitrag zur Konferenz KnowTech (2004), München.

[LM05] Lehmann, Kathrin; Matthes, Florian: „Meta Model Based Integration of Role-Based and Discretionary Access Control Using Path Expressions". In: 7th International Conference on E-Commerce Technology (2005), München, IEEE Computer Society, S. 443--446.

[LS99] Lupu, Emil C.; Sloman, Morris: „Conflicts in Policy-Based Distributed Systems Managament". In: IEEE Transactions on Software Engineering (1999), Vol. 25, Nr. 6, S. 852--869.

[LT06] Lehmann, Kathrin; Thiemann, Peter: „Field Access Analysis for Enforcing Access Control Policies". In: International Conference on Emerging Trends in Information and Communication Security (2006), Freiburg, Germany. Lecture Notes in Computer Science, Springer-Verlag, Vol. 3995, S. 337--351.

[Man01] Mantel, Heiko: „Information Flow Control and Applications – Bridging a Gap". In: Formal Methods Europe on Formal Methods for Increasing Software Productivity (2001), International Symposium of Formal Methods Europe. Lecture Notes in Computer Science, Springer-Verlag, Vol. 2021, S. 153--172.

[Mar03] Marek, Detlef: *Sprachbasierte Konstruktion sicherer Systeme*. Technische Universität München, Dissertation, 2003.

[Mit02] Mitnick, Kevin: *The Art of Deception : Controlling the Human Element of Security*. Wiley Publishing, Inc., 2002. ISBN 0-471-23712-4

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4

[Mye99] Myers, A. C.: „JFlow: practical mostly-static information flow control". In: Symposium on Principles of Programming Languages (1999) / A. Aiken (Hrsg.), San Antonio, Texas, S. 228--241.

[Nie03] Nietsch, Thomas: *Verwaltung von Benutzer- und Berechtigungsdaten für Autorisierungszwecke - Administrationsdialoge für Quasar Authorization*. Technische Universität München, Systementwicklungsprojekt, November 2003.

[Nie04] Nietsch, Thomas: *Spezifikation der Semantik einer generischen Autorisierungskomponente als Basis für eine verbesserte Implementierung*. Technische Universität München, Diplomarbeit, September 2004.

[NNH99] Nielson, Flemming; Nielson, Hanne Riis; Hankin, Chris: *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN 3-540-65410-0

[Oak01] Oaks, Scott: *Java Security*. O'Reilly, 2001. ISBN 0-596-00157-6

[ODMG98] Object Database Management Group (ODMG): *ODMG OQL : User Manual. Release 5.0*. 1998.

[OMG02] Object Management Group (OMG): *Meta Object Facility (MOF) Specification*. 2002. http://www.omg.org/docs/formal/02-04-03.pdf (besucht am 8. Januar 2007)

[OMG03] Object Management Group (OMG): *MDA Guide Version 1.0.1*. 2003. http://www.omg.org/docs/omg/03-06-01.pdf (besucht im August 2006)

[OMG05] Object Management Group (OMG): *Unified Modeling Language : Superstructure. Version 2.0*, 2005. http://www.omg.org/docs/formal/05-07-04.pdf (besucht am 8. Januar 2007)

[OMG06] Object Management Group (OMG): *UML 2.0 OCL Specification*. 2006. http://www.omg.org/docs/formal/06-05-01.pdf (besucht am 5. Januar 2007)

[OQ05] OpenQuasar: http://www.openquasar.de (besucht am 8. Januar 2007)

[OSM00] Osborn, Sylvia; Sandhu; Ravi, Munawer, Qamar: „Configuring role-based access control to enforce mandatory and discretionary access control policies". In: ACM Transactions on Information and System Security (2002), Vol. 3, Nr. 2, S. 85--106.

[PAM95] Open Software Foundation: RFC 86.0 : *Unified Login with Pluggable Authentication Modules (PAM)*. 1995. http://www.opengroup.org/tech/rfc/mirror-rfc/rfc86.0.txt (besucht am 22. Januar 2007)

[PRG06] Policy Research Group: Ponder Toolkit : Ponder Policy Editor (v1.0.1). 2006. http://www-dse.doc.ic.ac.uk/Research/policies/pondereditor.shtml. (besucht am 30. März 2007)

[Qua07] Quasar: http://www.sdm.de/de/unternehmen/fe/quasar/index.html (besucht am 8. Januar 2007)

[RBK+91] Rabatti, Fausto; Bertino, Elisa; Kim, Won; Woelk, Darrell: „A Model of Authorization for Next-Generation Database Systems". In: ACM Transactions on Database Systems (1991). Vol. 16, Nr. 1, S. 88--131.

[RWL96] Reenskaug, Trygve; Wold, Per; Lehne, O.A.: *Working with Objects : The OOram Software Engineering Method*. Manning : Prentice Hall, 1996.

[Roe04] Rölle, Christopher: *Integration of authorization checks into business applications using AspectJ*. Technische Universität München, Bachelor Thesis, September 2004.

[Rud99] Rudloff, Andreas: *Nutzungsrechte in offenen Dienstinfrastrukturen*. Technische Universität Hamburg-Harburg, Dissertation, 1999.

[Sac07] Sackmann, Stefan: „ „Privacy" im World Wide Web". In: Wirtschaftsinformatik, Vol. 49, Nr. 1, S. 49—54.

[SC00] Samarati, Pierangela; De Capitani di Vimercati, Sabrina: „Access Control : Policies, Models, and Mechanisms". In: Foundations of Security Analysis and Design : Tutorial Lectures (FOSAD 2001), Berlin : Springer-Verlag, 2001, S. 137--196. ISSN 0302-9743

[Sch05] Schneider, Sergius: *Entwurf eines fachlichen Lösungskonzepts und eines Architekturmodells für das unternehmensweite Berechtigungsmanagement in einer Großbank*. Technische Universität München, Bachelor Thesis, Dezember 2005.

[Sch99] Schneier, Bruce: „Attack trees: Modeling security threats". In: Dr. Dobb's Journal, December 1999.

[Sie04] Siedersleben, Johannes: *Moderne Softwarearchitektur*. dpunkt Verlag, 2004. ISBN 3-89864-292-5

[SH02] Stahlknecht, Peter; Hasenkamp, Ulrich: *Einführung in die Wirtschaftsinformatik*. Springer-Verlag, 2002. ISBN 3-540-41986-1

[SL02] Sloman, Morris; Lupu, Emil: „Security and Management Policy Specification". In: IEEE Network (2002), Vol. 16, Nr. 2, S. 10--19.

[SM03] Sabelfeld, Andrei; Myers, Andrew C.: „Language-Based Information-Flow Security". In: IEEE Journal on Selected Areas in Communications (2003), Vol. 21, Nr. 1.

[SM98] Sandhu, Ravi; Munawer, Qamar: „How to do Discretionary Access Control Using Roles". In: Third ACM Workshop on Role Based Access Control (1998), Fairfax, Virginia, ACM Press, S. 47--54.

[SNL06] Steel, Christopher; Nagappan, Ramesh; Lai, Ray: *Core Security Patterns – Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2006. ISBN 0-13-146307-1

[Som05] Sommerville, Ian: *Software Engineering*. Pearson Studium, 2005. ISBN 3-8273-7001-9

[Spi85] Spies, P. P.: „Datenschutz und Datensicherung im Wandel der Informationstechnologien". In: 1. GI-Fachtagung, Datenschutz und Datensicherung im Wandel der Informationstechnologien (1985), Springer-Verlag, S. 1--25

[Spr06] Spring Framework: http://www.springframework.org/ (besucht am 8. Januar 2007)

[SS75] Saltzer, Jerome H.; Schroeder, Michael D.: „The Protection of Information in Computer Systems". In: Proceedings of the IEEE, Vol. 63, Nr. 9, 1975, S. 1278--1308

[SS94] Sandhu, Ravi; Samarati, Pierangela: „Access control: Principles and practice". In: IEEE Communications (1994), Vol. 32, Nr. 9, S. 40--48.

[Sun01] Sun Microsystems: *Java Authentication and Authorization Service (JAAS)*. Reference Guide, 2001.
http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html
(besucht am 8. Januar 2007)

[Sun05] Sun Microsystems: *Java Security Overview*. Technischer Bericht, 2005.
http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf
(besucht am 8. Januar 2007)

[Tan02] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. Pearson Studium, 2002.
ISBN 3-8273-7019-1

[The97] Theis, Horst E.: *Computerrecht - Für alle EDV- Anwender*. Luchterhand, 1997.
ISBN 3-4720-2515-8

[Val00] Vallée-Rai, Raja: *SOOT : A Java Bytecode Optimization Framework*. Master Thesis , School of Computer Science, McGill Universität, Montreal, Kanada, 2000.

[VBC01] Viega, John; Block, J. T.; Chandra, Pravir: „Applying Aspect-Oriented Programming to Security". In: Cutter IT Journal, Cutter Consortium, Vol. 14, Nr. 2, 2001.

[VHS+99] Vallée-Rai, Raja; Hendren, L.; Sundaresan, V.; Lam, P.; Gagnon, E.; Co, P.: „SOOT - A Java Optimization Framework". In: Proceedings of Centre of Advanced Studies Conference (CASCON) (1999), S. 125--135.

[Weg02] Wegner, Holm: *Analyse und objektorientierter Entwurf eines integrierten Portalsystems für das Wissensmanagement*. Technische Universität Hamburg-Harburg, Dissertation, 2002.

[Wei02] Weiler, Nathalie: *The Need for Privacy*. 2002

[Wil02] Wildensee, Chistoph: „SAP R/3-Besonderheiten des Systems bei der Prüfung des Berechtigungskonzeptes“. In: ZIR 4/2002, S. 163--167.

[Wim07] Wimmer, Martin: *Efficient Access Control for Service-oriented IT Infrastructures*. Technische Universität München, Dissertation, 2007.

[Win32] MSDN Library: *Win32 and COM Development*. http://msdn.microsoft.com/library/default.asp (besucht am 8. Januar 2007)

[WM97] Wilhelm, Reinhard; Maurer, Dieter: *Übersetzerbau : Theorie, Konstruktion, Generierung*. Springer-Verlag, 1997. ISBN 3-540-61692-6

[W3C02a] World Wide Web Consortium: *A P3P Preferences Exchange Language 1.0 (APPEL 1.0) Working Draft*, 2002. http://www.w3.org/TR/P3P-preferences (besucht am 12. Februar 2007)

[W3C02b] World Wide Web Consortium: *The Platform for Privacy Preferences 1.0 (P3P) Specification*, 2002. http://www.w3.org/TR/P3P/ (besucht am 12. Februar 2007).

[ZRG05] Zhang, Nan; Ryan, Mark; Guelev, Dimitar P.: „Evaluating Access Control Policies Through Model Checking“. In: Eighth Information Security Conference (2005). Lecture Notes in Computer Science, Springer-Verlag, Vol. 3650, S 446-460.

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ABAP	Advanced Business Application Programming
ACL	Access Control List
ADF	Access Decision Function
AEF	Access Enforcement Function
API	Application Programming Interface
APPEL	A P3P Preference Exchange Language
ASL	Authorization Specification Language
BDSG	Bundesdatenschutzgesetz
C	Create
D	Delete
DAC	Discretionary Access Control
EJB	Enterprise Java Bean
GUI	Graphical User Interface
HTML	HyperText Markup Language
http	Hypertext Transfer Protocol
ID	Identifier
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IT	Information Technology
J2EE	Java 2 Platform, Enterprise Edition (Sun)

JAAS	Java Authentication and Authorization Services
Java EE	Java Platform, Enterprise Edition (Sun)
jimple	Intermediäre Repräsentation von Java Quellcode im Framework SOOT
JSA	Java Security Architecture
MDA	Model Driven Architecture
MOF	Meta-Object Facility
NIST	National Institute of Standards and Technologies
OCL	Object Constraint Language
OQL	Object Query Language
P3P	Platform for Privacy Preferences
Q	Query
Quasar	Qualitätssoftwarearchitektur
R	Read
RBAC	Role-Based Access Control
shimple	SSA-Form von jimple
SOOT	Framework für die statische Quellcodeanalyse von Java der McGill-Universität, Montreal, Kanada
SSA	Static Single Assignment
U	Update
UML	Unified Modeling Language
URL	Uniform Resource Locator
XACML	eXtended Access Control Markup Language
XML	eXtended Markup Language