Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

DIPLOMARBEIT

Anwendungsspezifische Autorisierungsstrategien in polymorphen persistenten Programmierumgebungen: Ein Bibliotheksansatz

Dezember 1995

Thomas Kaß
Edwin-Scharff-Ring 4
22309 Hamburg

Tel.: +49 -40 -631 70 43

Betreuer:
Prof. Dr. Joachim W. Schmidt

Dr. Simone Fischer-Hübner

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

> Nihil tam certum est,
> cui periculum non sit,
> etiam ab invalido.
>
> *–Quintus Curtius*

Die Anforderungen an die Entwicklung von Computeranwendungen unterliegen einem stetigen Wandel. Während Softwaresysteme früher zentralisiert, einbenutzerorientiert und speziell auf eine Anwendung zugeschnitten waren, stehen im Zeitalter der Client-Server-Architekturen Aspekte wie Offenheit, Service-Orientierung, Verteilung, Mehrbenutzerbetrieb und Heterogenität im Vordergrund [RMS95].

Auf der einen Seite eröffnet dieser technologische Fortschritt die Erschließung neuer Informationsquellen, die dem gesteigerten Informationsbedarf, insbesondere von Wirtschaftsorganisationen und Verwaltungen, gerecht wird. Die Sammlung und Auswertung dieser Informationen führt jedoch auch zu stetig wachsenden Datenbeständen. Dadurch geraten die betroffenen Unternehmen zusehends in eine existentielle Abhängigkeit von der Informationstechnik [Mei93]. Selbst Computerausfälle von kürzester Zeitdauer können für Unternehmen weitreichende Konsequenzen, bis hin zum Konkurs, nach sich ziehen.
Auf der anderen Seite ist als Folge fortschreitender Informationstechnologie auch ein erheblicher Anstieg von Computerkriminalitätsdelikten zu verzeichnen. Durch diese Entwicklung ist die Fähigkeit, den Zugriff auf Informationen zu kontrollieren und zu begrenzen, ebenso wichtig geworden wie die Verfügbarkeit der Daten selbst [WL81].
Sicherheitsmaßnahmen, die früher hinter Aspekten wie Funktionalität, Performanz, Kompatibilität, Zuverlässigkeit und Kosten zurücktreten mußten, gewinnen dadurch zunehmend an Bedeutung. Dennoch erweisen sich die in der Praxis eingesetzten Methoden und Maßnahmen häufig als unzureichend, einen wirksamen und unumgänglichen Schutz gegen unbefugten und mißbräuchlichen Datenzugriff zu gewährleisten.

Der Bereich der Rechnersicherheit umfaßt ein großes Gebiet von Anforderungen und Methoden. Es erfolgt daher zunächst eine überblicksartige Einführung in grundlegende Aspekte der

Rechnersicherheit, aus denen im weiteren Verlauf die resultierenden Anforderungen an eine Entwicklungsumgebung zur Implementierung problemadäquater Zugriffskontrollmodelle abgeleitet werden. Eine Definition des Ziels dieser Arbeit bildet den Abschluß des ersten Kapitels.

1.1 Elementare Begriffe der Rechnersicherheit

Unter dem Begriff der Rechnersicherheit wird ein umfangreicher Katalog unterschiedlicher Anforderungen und Maßnahmen verstanden, die dem Schutz von Hardware und Software vor böswilligem oder zufälligem Zugriff, Gebrauch, Veränderung, Zerstörung oder Enthüllung dienen. Sicherheit bezieht sich dabei auf die Bereiche Mitarbeiter, Daten, Kommunikation, Medien und den physikalischen Schutz von Computerinstallationen [Tha93].

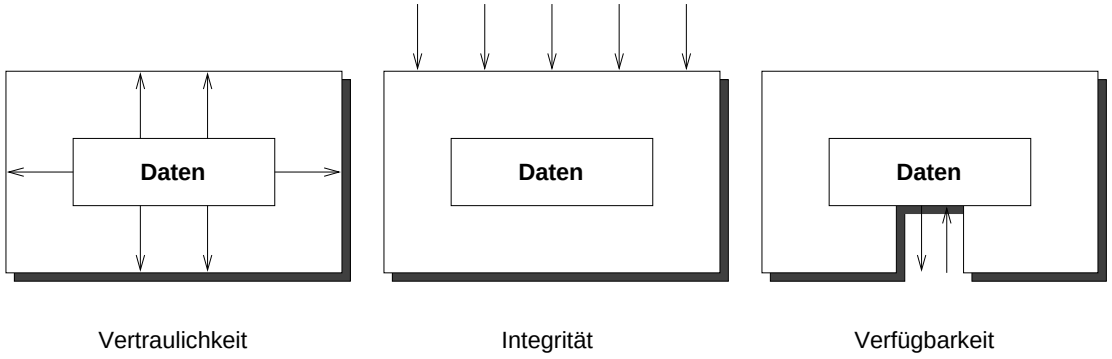


Abbildung 1.1: Elementare Sicherheitsanforderungen

Drei zentrale Anforderungen an ein sicheres System illustriert Abbildung 1.1 [FH92, Opp92]:

Vertraulichkeit (*confidentiality***):** Ihr Ziel ist es, unbefugten Informationsgewinn (Spionage, Verletzung des Datenschutzes) zu verhindern. Die Zusicherung von Vertraulichkeit stellt also sicher, daß Informationen nur legitimierten Personen zur Kenntnis gelangen [BEG95].

Integrität (*integrity***):** Sie gewährleistet, daß nur autorisierte Modifikationen an Daten vorgenommen werden, um die Konsistenz der Datenbasis zu erhalten. Die Integrität gilt als gewahrt, wenn Daten zum einen Informationen physikalisch korrekt repräsentieren und wenn zum anderen eine konkrete Anwendung aus diesen Daten die semantisch korrekte Information rekonstruieren kann. Teilziele zur Gewährleistung von Integrität sind der Schutz vor Modifikationen durch nicht autorisierte Benutzer, der Schutz vor inkorrekten Modifikationen durch autorisierte Benutzer, die Einhaltung der Konsistenz voneinander abhängiger Daten (innere Konsistenz) sowie die Korrektheit der Abbildung des Ausschnittes der realen Welt, der durch die Daten repräsentiert wird (äußere Konsistenz) [RWBM90]. Es muß also zugesichert werden, daß alle beabsichtigten und unbeabsichtigten Veränderungen von Daten ausgeschlossen werden, welche Daten verfälschen oder Anwendungen falsch, unvollständig oder nur vorgetäuscht ablaufen lassen [BEG95, Wis90].

Verfügbarkeit (*availability***):** Sie sichert zu, daß geschützte Funktionen oder Daten weder unbefugt gelöscht noch beeinträchtigt werden.

Die Sicherstellung dieser Anforderungen umfaßt eine Vielzahl von Methoden und Maßnahmen, die von seiten der Software und Hardware eingesetzt werden können. Die folgende Aufstellung abstrahiert von physikalischen Schutzmaßnahmen und stellt gebräuchliche Sicherheitskonzepte vor, die durch Programme realisiert werden:

Identifikation und Authentisierung: Die Beschränkung des Zugangs zu Rechenanlagen setzt Maßnahmen zur Identifikation und Authentisierung von Systembenutzern voraus. Unter einer Identifikation versteht man das Zuordnen einer Benutzeridentität zu Rechneraktivitäten. Der Vorgang der Authentisierung beschreibt die Verifikation einer behaupteten Identität durch den Benutzer. Im Regelfall besteht die Durchführung dieser Maßnahmen in der Eingabe einer Benutzerkennung und eines Paßwortes (Authentisierung durch Wissen). Identifikation und Authentisierung können jedoch zum Beispiel auch durch Chipkarten (Authentisierung durch Besitz) oder - wie es vornehmlich in militärischen Hochsicherheitstrakten üblich ist - durch den Einsatz biometrischer Verfahren (Fingerabdruckanalyse, Abtastung der Retina, Analyse des Stimmfrequenzmusters, Erkennung der Unterschriftsdynamik ...) erreicht werden (Authentisierung durch persönliche Merkmale) [DNP87, FH92].

Diese Maßnahmen reichen jedoch in einer offenen Systemumgebung, die sich durch eine rege Interaktion zwischen lokalen und entfernten Diensterbringern auszeichnet, nicht mehr aus. Da bei dieser Kommunikation Systemgrenzen überschritten und potentiell unsichere Netzwerkverbindungen in Anspruch genommen werden, kann das Betriebssystem die systemweite Eindeutigkeit der lokal verifizierten Benutzeridentität nicht mehr sicherstellen [RMS95]. Daraus resultiert die Notwendigkeit, sich auch beim entfernten Diensterbringer identifizieren und authentisieren zu müssen und die Datenübertragung dieses Kommunikationsvorgangs in einer fälschungssicheren und bei Bedarf auch abhörsicheren Form durchzuführen. Mit diesem Aufgabenkomplex beschäftigen sich Authentisierungsprotokolle, wie Kerberos [KN93, SNS88] und PGP[1] [Gar95, Sta95]. Die Identifikation beim externen Kommunikationspartner basiert dabei auf kryptographischen Verfahren (Verschlüsselung), zum Beispiel RSA[2] [RSA78] oder DES[3] [DES77].

Autorisierung: Die Zugangsberechtigung zu einem lokalen oder entfernten Rechner soll den authentisierten Benutzer im allgemeinen nicht dazu befähigen, auf alle Systemkomponenten uneingeschränkt zugreifen zu dürfen. Angestrebt wird vielmehr, nur solche Zugriffe zu gestatten, die für die Erfüllung der Aufgaben eines Benutzers unumgänglich sind (*least privilege principle*). Die Modellierung und Einhaltung solcher Zugriffsbeschränkungen obliegt der Autorisierung oder Zugriffskontrolle.

Protokollierung: Auch die Aufzeichnung sicherheitsrelevanter Ereignisse (*auditing*) leistet einen Beitrag zum Schutz von Computersystemen. Anhand dieser Aufzeichnung kann zum Beispiel überprüft werden, ob ein nicht autorisierter Benutzer innerhalb kürzester Zeit mehrfach versucht hat, auf sicherheitssensitive Daten zuzugreifen, oder ob das Sicherheitssystem Schwachstellen oder Lücken (*loopholes*) enthält, die unbefugte Zugriffe ermöglichen. Durch die Auswertung protokollierter Informationen erhöht sich die Wahrscheinlichkeit "Einbrüche" in Rechnersysteme zu erkennen (*intrusion detection*), beziehungsweise bereits im Vorfeld zu verhindern (*intrusion avoidance*). Eine Möglichkeit

[1] **P**retty **G**ood **P**rivacy

[2] Dieses asymmetrische *Public Key*-Verfahren ist nach den drei Entwicklern **R**ivest, **S**hamir und **A**dleman benannt.

[3] Der **D**ata **E**ncryption **S**tandard ist ein symmetrisches Verschlüsselungsverfahren.

dies zu erreichen bietet die Integration einer Expertensystemkomponente, die auf heuristischen Maßnahmen basiert (vergleiche zum Beispiel [FH92, FHB90]).

Schutz von Massenspeichern: Alle bisher beschriebenen Maßnahmen bleiben wirkungslos, wenn eine Bedrohung der Sicherheit durch andere als durch rechnergestützte Zugriffe entsteht. Auch durch die physikalische Entfernung eines Massenspeichers, zum Beispiel einer Festplatte, können unbefugte Benutzer in den Besitz sicherheitssensitiver Informationen gelangen. Einen wirksamen Schutz gegen solche Attacken bietet die Verschlüsselung (siehe oben) der gespeicherten Daten auf dem Datenträger.

Verbot der Wiederaufbereitung: Daten, die im Haupt- oder Sekundärspeicher stehen und aus einer Anwendung heraus entfernt werden sollen, dürfen nicht nur logisch gelöscht werden, sondern sind auch physikalisch zu entfernen. Diese Maßnahme verhindert, daß Daten unberechtigt wiederverwendet werden können (*object reuse*).

Die weiteren Ausführungen dieser Diplomarbeit konzentrieren sich darauf, softwaretechnische Maßnahmen zur Zugriffskontrolle und Zugriffsbeschränkung auf Daten und Informationen vorzustellen und zu analysieren.
Als grundlegende Domänen für Fragen der Zugriffskontrolle ergeben sich daraus die Menge der aktiv zugreifenden Entitäten, **Subjekte** genannt, die Menge passiver Entitäten, auf die zugegriffen wird, im folgenden als (**Schutz-**)**Objekte**[4] bezeichnet, und die Menge der **Zugriffsmodi** oder **Zugriffstypen** (*access types*), die die Art des Zugriffs beschreiben. Beispiele für Subjekte sind Systembenutzer und Prozesse; zur Menge der Objekte können Dateien und persistente Daten gehören. Bei den Zugriffstypen kann beispielsweise zwischen einem lesenden oder einem schreibenden Zugriff unterschieden werden. Grundsätzlich werden Entitäten nicht statisch der Klasse der Subjekte oder Objekte zugeordnet, da sie je nach Anwendungskontext sowohl als Objekt als auch als Subjekt fungieren können. Ein **Zugriffsrecht** (*access right*) oder eine **Autorisierung** beschreibt die Erlaubnis eines Subjektes, auf ein bestimmtes Objekt mit einem bestimmten Zugriffstyp zugreifen zu dürfen. Eine **Sicherheitspolitik** [Poh89] (*security policy*) schließlich ist “eine Menge exakt formulierter Grundsätze, Regeln, Anforderungen oder auch Verfahren in einem System zum Schutz und zur Sicherung sensitiver oder geheimhaltungsbedürftiger Informationen” [Poh89, Seite 107]. Ihr Ziel ist es, dafür zu sorgen, daß potentielle Bedrohungen gegen mögliche Schwachstellen des Systems nicht wirksam werden.

Aus der Sicht der Modellierung von Zugriffskontrollrechten ergeben sich zwei zentrale Anforderungen:

1. Der Zugriffsschutz soll grundsätzlich alle durch Subjekte referenzierbaren und modifizierbaren Systemkomponenten umfassen können und stabil gegen sich verändernde Zugriffskontrollanforderungen sein.

2. Die hohe Zahl von Objekten in großen schutzbedürftigen Applikationen macht eine explizite Autorisierung jedes zugriffsberechtigten Subjektes sehr umständlich. Das Autorisierungssystem sollte daher Methoden bereitstellen, mit denen Subjekte zu Gruppen zusammengefaßt und gewisse Rechte aus anderen Rechten abgeleitet werden können.

[4] Sofern eine Verwechslung mit dem Objektbegriff der objektorientierten Programmierung ausgeschlossen ist, wird im folgenden nur der Begriff Objekt verwendet.

Diese Anforderungen, die eine benutzerfreundliche Modellierung von Zugriffsrechten unterstützen sollen, werden von vielen kommerziellen Systemen nicht erfüllt: Die Vergabe von Zugriffsrechten ist in solchen Systemen oft durch fest vorgegebene Granularitäten für Subjekte und Objekte sowie eine feste Anzahl von Zugriffstypen beschränkt. Subjekte können zwar meist gruppiert werden; die Möglichkeit, Zugriffsrechte implizit aus anderen Rechten ableiten zu können, ist jedoch häufig nicht vorhanden.
Bei genauer Betrachtung beschreibt die zweite Anforderung in erster Linie eine Modellierungsaufgabe, da jede Rechteableitung einer genauen Spezifikation bedarf, um zuverlässig eingesetzt werden zu können. Im Bereich objektorientierter Zugriffskontrollmodelle existieren bereits erste Forschungsansätze, die sich mit dieser Aufgabe beschäftigen (siehe Abschnitt 2.4). Im Unterschied dazu wirft die erste Anforderung ein Problem der Architektur von Systemen auf, die schutzwürdige Anwendungen realisieren. Diese Systeme bieten auch unabhängig von Sicherheitsanforderungen nur feste Granularitäten an und sind zudem aufgrund ihrer geschlossenen Architektur nicht flexibel anpaßbar. Daher scheitern Versuche, die Einschränkungen bestehender Systeme durch Änderungen an den Sicherheitskomponenten zu realisieren.
Zur Erfüllung der genannten Anforderungen muß als Grundlage also vor allem eine geeignete Entwicklungsumgebung für Sicherheitsdienste und Anwendungen zur Verfügung stehen.

1.2 Anforderungen an eine Entwicklungsumgebung für Sicherheitsdienste

Der Grund für die oben genannten Einschränkungen besteht darin, daß Mechanismen der Rechnersicherheit oft eine systeminhärente Komponente von Applikationen darstellen. Dies bietet zwar den Vorteil, daß das Sicherheitssystem speziell auf die zu überwachende Anwendung zugeschnitten ist, weist aber die Nachteile mangelnder Flexibilität, Erweiterbarkeit, Austauschbarkeit und Wiederverwendbarkeit auf. Als Konsequenz entsteht ein hoher Entwicklungsaufwand, da für neue Anwendungen stets ein neuer Sicherheitsdienst erforderlich ist. Daher werden Sicherheitssysteme fast ausschließlich in große Systemdienste wie Datenbanken und Betriebssysteme integriert [RMS95]. Zum einen erschweren jedoch die oft unterschiedlichen Sicherheitssysteme dieser Dienste deren Kooperation untereinander, zum anderen zwingt die fest vorgegebene Funktionalität dieser Sicherheitsdienste die Benutzer zu einer (oft nicht gewollten) Anpassung ihrer Sicherheitspolitiken an das System, da Möglichkeiten zur bedarfsgerechten Ausgestaltung der vom ihm verfolgten Sicherheitspolitik in der Regel fehlen [Ste89, Hos92].
Aus dieser Erkenntnis heraus ergibt sich die Forderung, Funktionen zur Zugriffsbeschränkung nicht mehr als integralen Bestandteil einer Anwendung (*built-in*) zu implementieren, da eine monolithische Implementierung eines Zugriffskontrollsystems stets eine hoch restriktive Vergabe von Zugriffsrechten (*Alles oder Nichts-Prinzip*) zur Folge hat [Sie94]. Ziel muß es vielmehr sein, zum einen Applikationen in abgrenzbare Teilsysteme zu zerlegen, für die eigene Zugriffsbeschränkungen in einem Sicherheitsmodell spezifiziert werden können, und zum anderen Programmbibliotheken mit Sicherheitsprimitiven bereitzustellen, durch deren Kombinationsmöglichkeiten sich ein breites Spektrum von Applikationen unterstützen läßt (*add-on Ansatz*). Diese Programmbibliotheken befähigen den Anwendungsprogrammierer, seine Sicherheitskomponente unabhängig von der zu schützenden Applikation zu entwickeln und mit dieser zu einem sicheren Gesamtsystem zusammenzufügen. Im Kontext datenintensiver Anwendungen ist hierbei jedoch zu beachten, daß das Autorisierungsmodell konsistent mit dem Datenmodell ist, das dem zu schützenden Datenbanksystem zugrundeliegt [RBKW91, Dit94].

Zusammenfassend leitet sich ab, daß eine Entwicklungsumgebung für die Modellierung und Implementierung eines problemadäquaten Zugriffskontrollsystems im wesentlichen drei Voraussetzungen erfüllen muß:

1. Sie muß über ein Modularisierungskonzept verfügen, das die Zerlegung komplexer Applikationen in einfachere Teilkomponenten erlaubt. Der Vorteil dieses Konzeptes liegt in der Austauschbarkeit und freien Kombinierbarkeit der einzelnen Komponenten. Diese Komponenten sollten einerseits für die Entwicklung anderer Applikationen in Form einer Programmbibliothek zur Verfügung stehen, andererseits sollte in analoger Weise bei der Implementierung einer Applikation auf andere Programmbibliotheken zugegriffen werden können (Wiederverwendbarkeit). Prinzipiell können diese Programmbibliotheken forlaufend um weitere Module erweitert werden.

2. Der Zugriffsschutz komplexer Anwendungen bedingt die Entstehung einer Vielzahl von Daten und Zugriffskontrollinformationen. Die Verwaltung dieser Informationen erfordert daher Datenbankfunktionalität wie Persistenz, Massendatentypen, Integritätsbedingungen und Anfragen und unterliegt ihrerseits entsprechenden Sicherheitsanforderungen.

3. Ein Autorisierungssystem sollte so universell einsetzbar sein, daß es den Benutzer konzeptuell dazu befähigt, den Zugriff auf alle erzeugbaren Datenstrukturen kontrollieren zu können. Dies bedingt eine Entwicklungsumgebung, die sowohl über ein mächtiges Typsystem zur Datenmodellierung verfügt als auch hinreichende Generizität aufweist, um Anpassungen an geänderte Typanforderungen zu ermöglichen.

Die Implementierung eines Sicherheitssystems geschieht jedoch nicht zum Selbstzweck, da der sinnvolle Einsatz von Sicherheitsmaßnahmen stets an die Existenz schutzbedürftiger Applikationen oder Daten gekoppelt ist. Daher muß die Programmierumgebung neben diesen Forderungen, die sich konkret auf die Implementierung eines Sicherheitsdienstes beziehen, selbst auch in der Lage sein, die Anwendung und damit die zu schützenden Daten zur Verfuegung zu stellen.

1.3 Entwicklungsumgebungen für Datenbankanwendungen

Zur Bereitstellung von schutzbedürftigen, meist datenintensiven Anwendungen existieren zwei Alternativen. Zum einen kann ihre Implementierung in der Entwicklungsumgebung selbst vorgenommen werden, zum anderen können extern implementierte Dienste genutzt werden. Hierbei gelten für die Implementierung datenintensiver Anwendungen die gleichen Anforderungen hinsichtlich der Modellierbarkeit und Flexibilität, die auch an die Implementierung des Zugriffskontrolldienstes gestellt werden.
Kommerzielle Datenbanksysteme vermögen diese Anforderungen in der Regel jedoch nicht zu erfüllen. Ihre Aufgabe besteht zumeist in der Bereitstellung persistenter Daten eines bestimmten Datenmodells, die über eine allgemeine Programmiersprache genutzt und manipuliert werden. Die mangelnde Unterstützung eines konkreten Datenmodells durch eine allgemein konzipierte Programmiersprache führt jedoch meist zu einer schweren Handhabbarkeit und dem Verlust statischer Typprüfbarkeit.
Im Bereich der Datenbankforschung wird daher versucht, diese Einschränkungen durch die Anwendung neuer Konzepte, sowohl auf der Sprach- als auch auf der Systemebene, zu überwinden.

Erste Ansätze haben zu einer Verallgemeinerung des Persistenzkonzeptes und einer Integration der Persistenzdefinition sowie eines konkreten Datenmodells in eine allgemeine Programmiersprache geführt. Diese sogenannten Datenbankprogrammiersprachen, wie DBPL [MS92], Galileo [ACO85] und PS-Algol [ACC81], erhöhen den "Programmierkomfort" und erzwingen die Typsicherheit programmierter Anwendungen. Diese tiefe Integration des Datenmodells trägt jedoch nicht dazu bei, den Mangel an Flexibilität, Adaptierbarkeit, Erweiterbarkeit und Skalierbarkeit zu überwinden. Insbesondere eine Änderung der Systemfunktionalität, wie das Hinzufügen weiterer Datenmodelle, erweist sich als unmöglich. Dazu bedarf es eines Austauschs der Sprachschnittstelle durch Programmierschnittstellen und einer Bereitstellung der Funktionalität durch Bibliotheken. Dieser Bibliotheksansatz, der in Systemen wie Modula-3 [Nel91], Eiffel [Mey88] und Napier88 [DCBM89] verfolgt wird, erlaubt dem Anwendungsprogrammierer den selektiven Einsatz jener Bibliothekskomponenten, die für die Abbildung seiner gewünschten Funktionalität erforderlich sind.
Der Einsatz von Programmbibliotheken ist allerdings nur sinnvoll möglich, wenn die verwendeten Abstraktionen der Anwendungsprogrammierung mit den Abstraktionen in den Bibliotheken übereinstimmen. Dies erfordert die Benutzung derselben Programmiersprache, wobei jedoch gefordert werden muß, daß sowohl die Definition als auch die Benutzung der Bibliotheken in einer typsicheren Weise erfolgt. Da Systeme zugleich generische Funktionalität anbieten sollen, die vom Anwendungsprogrammierer in spezialisierter Form eingesetzt wird, erweisen sich monomorphe Programmiersprachen als unzureichend, den gestellten Anforderungen gerecht zu werden.
Hierzu bedarf es Programmiersprachen mit einem polymorphen Typsystem, wie ML [Mil84, MTH90], Miranda [Tur85] und $F_{\leq}$ [CMMS91]. Diese ermöglichen es, generische Funktionalität zu definieren und sowohl die Definition als auch deren konkrete Instantiierung durch den Anwendungsprogrammierer zum Übersetzungszeitpunkt statisch zu überprüfen.
Aufbauend auf einer solchen polymorphen Programmiersprache realisiert das Tycoon[5]-System [Mat93] des Arbeitsbereiches DBIS[6] der Universität Hamburg eine persistente Systementwicklungsumgebung. In dieser wird die wesentliche Funktionalität des auf einem persistenten Kern aufbauenden Systems durch Programmbibliotheken angeboten. Neben dem Ziel, den Anwendungsentwickler zur Modellierung beliebiger (persistenter) Daten zu befähigen, erlaubt das Tycoon-System, bedingt durch seine Gesamtarchitektur, extern realisierte Dienste anzubinden und über definierte Schnittstellen anzubieten.
Tycoon eignet sich somit als Entwicklungsplattform, da es sowohl die erforderlichen Voraussetzungen für die Implementierung und Nutzung eines Sicherheitsdienstes als auch der durch ihn zu unterstützenden (datenintensiven) Anwendung erfüllt.

1.4 Zielsetzung

Die Einschränkungen bestehender Zugriffskontrollsysteme bilden den Ausgangspunkt für diese Arbeit. Ziel ist es, ausdrucksstarke Modellierungsmittel und Programmierwerkzeuge für die Zugriffskontrolle innerhalb des Tycoon-Systems zu entwickeln und in einer Programmbibliothek zur Verfügung zu stellen. Diese Zugriffskontrollbibliothek soll den Anwendungsprogrammierer befähigen, mit Hilfe der bereitgestellten Module einen problemadäquaten Zugriffskontrolldienst zu entwickeln, mit dem er seine Applikationen schützen kann. Besondere Schwerpunkte

[5] **Ty**ped **c**ommunicating **o**bjects in **o**pen e**n**vironments
[6] **D**aten**b**anken und **I**nformations**s**ysteme

bilden dabei die Ausnutzung objektorientierter Zugriffskontrollabstraktionen und die Bereitstellung generischer Benutzerschnittstellen, mit denen Subjekte, Objekte und Zugriffsmodi beliebigen Typs und beliebiger Granularität verwaltet werden können. Die Konsistenzerhaltung modellierter Rechte und eine effektive Unterstützung des Benutzers bei der Modellierung seiner Applikation stellen weitere wichtige Gesichtspunkte dar. So müssen Potentiale zur Modellierungsvereinfachung erkannt, umgesetzt und als Hilfsmittel zur Verfügung gestellt werden. Schließlich ist auch zu analysieren, inwiefern bereits bestehende Applikationen noch nachträglich um Zugriffskontrollmechanismen erweitert werden können.

Diese Diplomarbeit weist folgende Struktur auf: Kapitel 2 analysiert bestehende Zugriffskontrollmodelle und wägt deren Vor- und Nachteile gegeneinander ab. Ziel ist es hierbei, jenes Modell zu eruieren, das ein Höchstmaß an Modellierungsfreiheit aufweist. Das dritte Kapitel stellt die Systementwicklungsumgebung Tycoon vor. Dabei wird, ausgehend von den Einschränkungen, denen kommerzielle Systeme unterworfen sind, gezeigt, wie diese Restriktionen in Tycoon behoben werden. Gegenstand des vierten Kapitels ist die Verallgemeinerung eines modellgebundenen Zugriffskontrollsystems für die modellfreie Tycoon-Umgebung. Hier wird detailliert die Wirkungsweise der Komponenten der Autorisierungsbibliothek gezeigt. Das anschließende Kapitel diskutiert Probleme und Besonderheiten bei der Implementierung der Konzepte. Den Abschluß des Hauptteils bildet Kapitel 6, das die Arbeit zusammenfaßt und Systemerweiterungen für kommende Forschungsarbeiten anregt.
Anhang A beschreibt in einer kurzen Zusammenfassung die wesentlichen Aspekte des objektorientierten Paradigmas, da dieses für das Verständnis der in der Arbeit verwendeten objektorientierten Zugriffskontrollmechanismen relevant ist. Diese werden durch eine spezielle graphische Notation, die in Anhang B erläutert wird, ergänzt. Die Visualisierung objektorientierter Zugriffskontrollstrukturen ist Gegenstand des Anhangs C, und in Anhang D schließen ausgewählte Programmschnittstellen der vorgenommenen Implementierung die Arbeit ab.

Die Medizin stellt ein typisches Anwendungsfeld dar, in dem Datensicherheit und Datenschutz eine wichtige Bedeutung zukommt [PP92, Ble94]. Die textbegleitenden Modellierungs- und Implementationsbeispiele dieser Arbeit entstammen daher dem medizinischen Umfeld.

Kapitel 2

Zugriffskontrollmodelle

Der Einsatz rechnergestützter Anwendungen führt zur Entstehung einer Vielzahl von Informationen, die es zu schützen gilt. Diese sicherheitssensitiven Informationen werden rechnerintern verwaltet und als Daten repräsentiert. Daten an sich haben keinen intrinsischen Wert mehr. Erst wenn sie eine konkrete Interpretation durch eine zugreifende Anwendung erhalten, werden sie zu Information. Wenn Information nicht mehr verfügbar ist, modifiziert wurde, unvollständig ist oder ihre Vertraulichkeit verloren hat, sind die Auswirkungen für die betreffende Organisation viel größer, als wenn dieser Mißbrauch nur an Daten erfolgen würde [For94]. Da jedoch nicht sichergestellt werden kann, daß nicht autorisierte Subjekte aus Daten Information gewinnen können, ist ein Schutz von Informationen nur dann gewährleistet, wenn auch die Daten, die diese Informationen repräsentieren, adäquat geschützt werden. Die Aufgabe von Zugriffskontrollmechanismen besteht somit darin, nur solche direkten Zugriffe auf Daten zuzulassen, die autorisiert worden sind. Zugriffskontrolle regelt den lesenden, ändernden und löschenden Zugriff auf Daten und Programme und schützt somit deren Geheimhaltung und Authentizität vor ungewollten (*accidental*) oder beabsichtigten (*malicious*) Bedrohungen [Den82].
Dieses Kapitel stellt Modelle und Mechanismen vor, mit denen die Vertraulichkeit und/oder Integrität von Daten beziehungsweise Informationen sichergestellt werden kann; Verfügbarkeit kann nicht durch Methoden der Zugriffskontrolle erzwungen werden [Abr93] und ist daher nicht Bestandteil der folgenden Betrachtungen.

Ausgehend von einem Anforderungskatalog, der an den Leistungsumfang eines Zugriffskontrollsystems zu stellen ist, werden zunächst bekannte Modellklassen und deren Ausprägungen in der Praxis sowie neuere Ansätze aus der Forschung vorgestellt. Aufbauend auf diesen Modellklassen erfolgt die Einführung von Methoden zur vereinfachten Modellierung von Schutzrechten durch Kollektionsbildung von Subjekten. Eine vergleichende Zusammenfassung der Zugriffskontrollmodelle schließt dieses Kapitel ab.

2.1 Anforderungen an Zugriffskontrollsysteme

Im folgenden wird das Anforderungsprofil an ein "mächtiges" Zugriffskontrollsystem herausgearbeitet. Der folgende Katalog umfaßt Aspekte aus [Dit83, Har81] und [LS87] sowie eigene Ergänzungen:

1. Das Schutzkonzept muß universell einsetzbar sein, das heißt es soll nicht auf ein bestimmtes Zielsystem zugeschnitten sein, sondern prinzipiell in verschiedenartigen Rechensystemen oder Teilen derselben anwendbar sein.

2. Das Sicherheitssystem hat ein größtmöglichstes Maß an Flexibilität zu gewährleisten. Hierunter wird insbesondere die Modellierungsfreiheit zur Ausgestaltung von Schutzrechten verstanden. Dabei sind die folgenden Eigenschaften von Interesse:

   - Objekte müssen skalierbar sein; Rechte müssen für verschiedene Objektgranularitäten spezifiziert werden können.
   - Die Rechtevergabe auf Subjekten muß variierbar sein: Es müssen sowohl einzelne Subjekte als auch Gruppen von Subjekten autorisiert werden können.
   - Aus Sicht der Autorisierungstypen sollen nicht nur elementare Operationen kontrolliert werden, sondern auch komplexere Zugriffsarten, die aus einer Reihe einfacherer Operationen bestehen.
   - Auch das Gesamtsystem muß skalierbar sein und Wahlmöglichkeiten bereitstellen. Diese müssen so zugeschnitten sein, daß je nach Sensitivität der Daten einer Anwendung das richtige Maß an Sicherheitsfunktionalität modelliert werden kann [Fai94].

3. Das System sollte einheitlich sein, was in einer geringen Anzahl unterschiedlicher Teilkonzepte zum Ausdruck kommt. Die Vorteile liegen in einem besseren Verständnis und einem geringeren Entwicklungsaufwand für neue Sicherheitssysteme. Darüber hinaus reduziert sich bei der Entwicklung komplexerer Systeme die Abstimmung der verschiedenen Teilkonzepte.

4. Das Kosten-Nutzen Verhältnis der Schutzmaßnahmen sollte in einem ausgewogenen Verhältnis stehen. Zwar ist der Nutzen von Sicherheitsmaßnahmen nicht quantifizierbar, er läßt sich aber zumindest mit Hilfe einer Ordinalskala analysieren. Auf der Kostenseite hingegen besteht die zentrale Forderung, daß Zugriffskontrollmechanismen einen möglichst geringen Einfluß auf das Laufzeitverhalten und den Speicherplatzbedarf der unterstützten Applikation haben sollen. Hier besteht ein Zielkonflikt, denn je feiner die autorisierbaren Granularitäten für die Subjekt-, Objekt- und Zugriffstyp-Domänen sind, desto höhere Kosten fallen für die Evaluierung von Sicherheitsanfragen und die Administration von Rechten an.

5. Die Objekte einer Applikation sind häufig langlebig. Zugriffsbeschränkungen auf diese Objekte können sich daher im Laufe der Zeit ändern [Bry94]. Die Durchführung dieser Änderungen oder die Vergabe neuer Zugriffsrechte muß somit dynamisch, das heißt im laufenden System, erfolgen können. Keineswegs sollte die Spezifikation neuer Autorisierungen nur am "stehenden" System vorgenommen werden können.

6. Ein Autorisierungssystem muß zu jedem Zeitpunkt logisch konsistent sein. Es hat die verfolgten Sicherheitspolitiken korrekt abzubilden.

7. Das System muß einfach in der Handhabung sein. Der Modellierer ist bei der Programmierung seiner Sicherheitspolitik bestmöglich zu unterstützen.

8. Die Sprache, in der das Zugriffskontrollsystem entwickelt wird, sollte ein Modulkonzept unterstützen. Auf diese Weise ist es möglich, Sicherheitsaspekte des Systems stets von der Anwendungssemantik zu trennen. Die Vorteile liegen in der isolierten Testbarkeit der Zugriffskontrollmodule hinsichtlich ihrer Korrektheit.

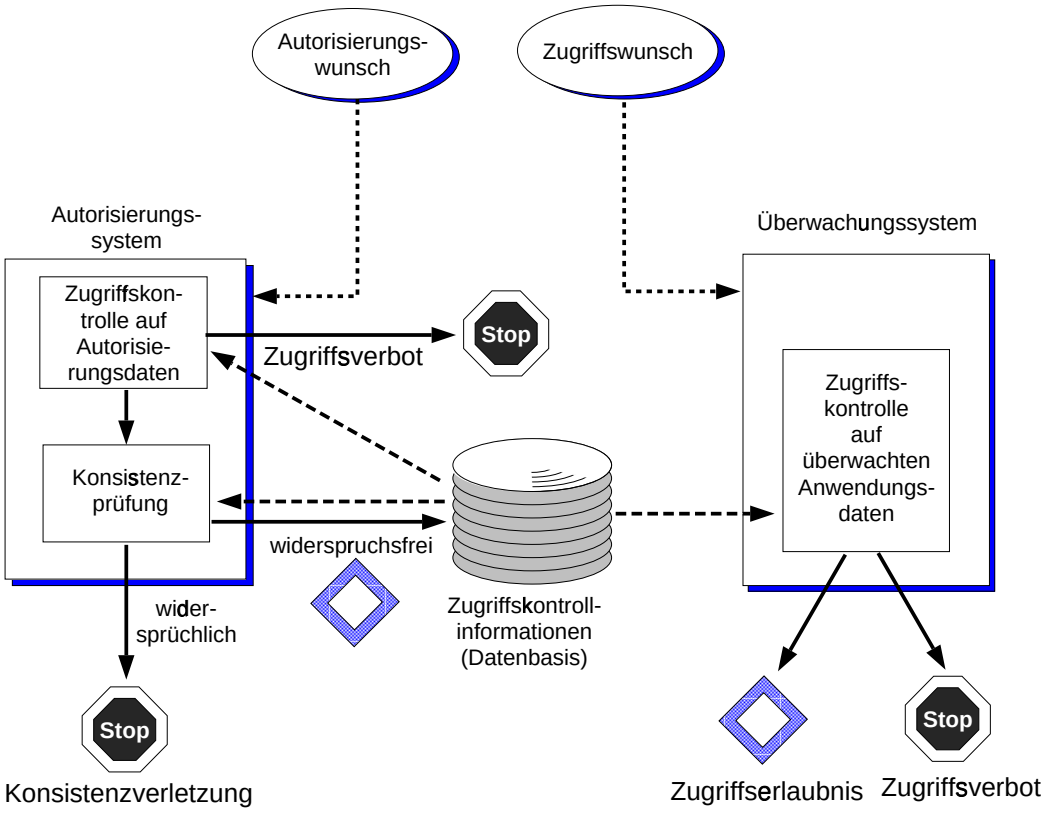


Abbildung 2.1: Aufbau eines Zugriffskontrollsystems

Abbildung 2.1 veranschaulicht den Aufbau eines Zugriffskontrollsystems. Es läßt sich logisch in drei Komponenten, die Datenbasis, das Überwachungssystem und das Autorisierungssystem, unterteilen. In der Datenbasis sind alle Zugriffsrechte gespeichert, die modelliert worden sind. Da ihre Anzahl sehr groß ist und ihr Informationsgehalt auch über Programmgrenzen hinaus (langlebig) erhalten bleiben soll, ist für die Bereitstellung dieser Komponente Datenbankfunktionalität erforderlich. Das Autorisierungssystem kontrolliert alle Anfragen, die die Menge der Rechte in der Datenbasis oder deren Informationsgehalt verändern. Möchte ein Benutzer ein neues Recht in die Datenbasis einfügen oder aus dieser entfernen, so wird zunächst anhand der gespeicherten Zugriffskontrollinformationen geprüft, ob eine solche Modifikation von diesem Subjekt durchgeführt werden darf. Ist dies der Fall, so muß geprüft werden, ob die gewünschte Änderung konsistent mit der Datenbasis ist, das heißt ob gewährleistet werden kann, daß durch diese Modifikation weder widersprüchliche Zugriffsrechte modelliert werden noch daß die Modifikation die durch das Zugriffskontrollsystem verfolgte Sicherheitspolitik kompromittiert. Nur

wenn auch diese Bedingung erfüllt ist, wird die Änderung durchgeführt; anderenfalls wird der Wunsch nach Änderung der Datenbasis abgelehnt. Das Überwachungssystem schließlich überprüft, ob dem Zugriffswunsch eines Subjektes aufgrund einer existierenden Zugriffsberechtigung in der Datenbasis entsprochen werden kann. Hierbei erfolgt lediglich ein lesender (nicht modifizierender) Zugriff auf die Datenbasis.

2.2 Klassische benutzerbestimmbare Zugriffskontrollmodelle

Autorisierungsmechanismen für zivile Anwendungen basieren fast ausschließlich auf benutzerbestimmbaren (diskreten) Zugriffskontrollmodellen (*discretionary access control models*) [FK93]. Der Einsatz benutzerbestimmbarer Zugriffskontrolle ist "eine Maßnahme, den Zugriff auf Objekte zu beschränken, die auf der Identität von Subjekten und/oder der Gruppen, zu denen sie gehören, basiert. Die Kontrolle ist benutzerbestimmbar in dem Sinne, daß ein Subjekt mit einem gewissen Zugriffsrecht fähig ist, einem anderen Subjekt diese Erlaubnis (gegebenenfalls auch indirekt) zu übertragen" [DoD85]. Kennzeichnend für diese Systeme ist also, daß die Vergabe und der Entzug von Zugriffsberechtigungen nicht durch eine zentrale Instanz (zum Beispiel den System- oder Sicherheitsadministrator) erfolgen, sondern daß es im Ermessen des Besitzers eines Objektes liegt, ob und wem er Zugriffsrechte auf dieses Objekt gewährt [Gas88]. In vielen dieser Systeme existiert daher eine logische Trennung zwischen Benutzerrechten (für den Zugriff auf ein Objekt) und administrativen Rechten (für die Modifikation von Benutzerrechten).

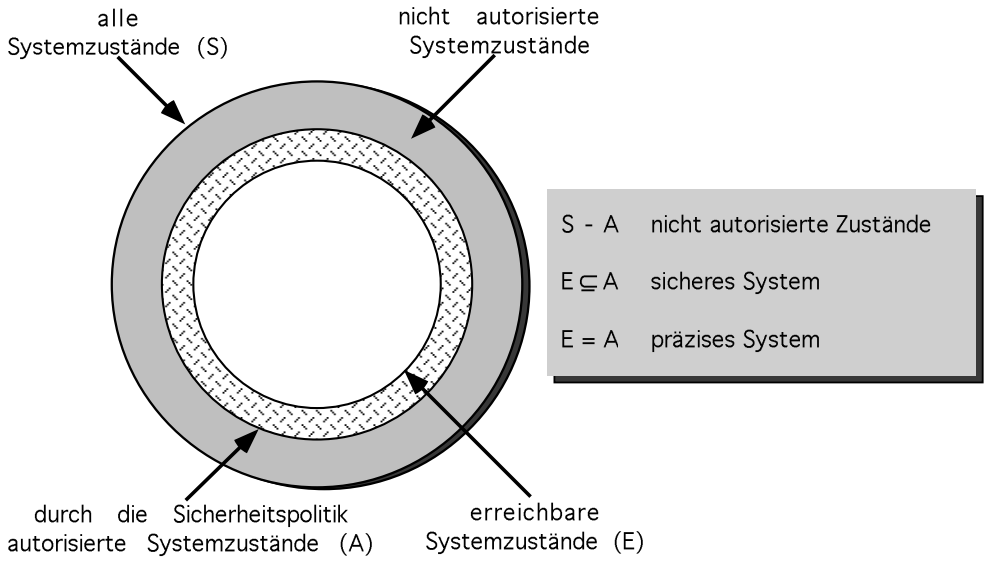


Abbildung 2.2: Präzision und Sicherheit in diskreten Zugriffskontrollsystemen

Abbildung 2.2 [Den82] zeigt, welche Eigenschaften ein präzises diskretes Zugriffskontrollsystem erfüllen muß. Ein solches Modell wird als **sicher** erachtet, wenn alle Systemzustände, für die das Überwachungssystem eine Zugriffserlaubnis ableitet, auch tatsächlich durch die Sicherheitspolitik autorisiert wurden. Ist jeder autorisierte Systemzustand darüber hinaus auch

erreichbar, so gilt das Zugriffskontrollsystem als **präzise**.

2.2.1 Zugriffskontrollmatrizen

Eine einfache Darstellungsform für die Domänen der Subjekte, Objekte und Zugriffsmodi, die die Rechte in einem System reflektiert, ist die Zugriffskontrollmatrix [HRU76]. Hierbei werden die Subjekte in den Zeilen und die Objekte in den Spalten abgetragen. Der Eintrag in Zeile i und Spalte j enthält die Menge von Zugriffsmodi, die das Subjekt i in bezug auf Objekt j ausführen darf [Lun91]. Ein fehlender Eintrag repräsentiert hierbei eine leere Menge von Zugriffstypen; das Subjekt besitzt somit keine Rechte für den Zugriff auf das Objekt.

		Objekte					
Subjekte		Obj_1	Obj_2	Obj_3	Obj_4	...	Obj_n
	Sub_1	r			w		
	Sub_2		w,r				w
	Sub_3			r		...	
	Sub_4			r			
	Sub_5				w,r		
	...		...			...	
	Sub_k	w					

Abbildung 2.3: Zugriffskontrollmatrix

Abbildung 2.3 veranschaulicht exemplarisch den Aufbau einer Zugriffskontrollmatrix. Die Buchstaben *r* und *w* repräsentieren den lesenden beziehungsweise schreibenden Zugriff. $Subjekt_1$ besitzt somit beispielsweise ein Leserecht auf $Objekt_1$ sowie ein Schreibrecht auf $Objekt_4$, auf allen anderen dargestellten Objekten dagegen keine Zugriffsrechte.
Zugriffskontrollmatrizen sind im allgemeinen schwach besetzt, das heißt sie enthalten meist nur wenige Einträge [Tal94]. Zugriffskontrollmatrizen eignen sich daher nicht als Struktur zur Verwaltung von Rechten, da alle leeren Einträge der Matrix mitgespeichert werden müssen. Daher fehlt ihnen die praktische Relevanz; als theoretisches Modell sind sie jedoch geeignet.

2.2.2 Fähigkeitslisten

Ein folgerichtiger Ansatz, mit dem das Abspeichern leerer Einträge in der Zugriffskontrollmatrix vermieden wird, besteht darin, diese zeilenweise zu zerlegen und alle leeren Einträge zu entfernen. Auf diese Weise entstehen Fähigkeitslisten (*capabilities*), die jedem Subjekt die erlaubten Zugriffsrechte bezüglich der einzelnen Objekte zuordnen und direkt beim Subjekt speichern. Eine Befähigung kann somit als eine Art “Eintrittskarte” betrachtet werden, die ein

Subjekt gegenüber einer Prüfinstanz vorweisen muß, wann immer ein Zugriff erwünscht ist. Der Besitz der Eintrittskarte gilt als hinreichender Beweis dafür, daß und wie der Benutzer zugreifen darf [Dit83, Pri90]. Vorteilhaft an dieser Art der Rechteverwaltung ist, daß nur noch relevante (nicht leere) Einträge gespeichert werden und daß die Frage, welche Rechte ein bestimmtes Subjekt hat, schnell entscheidbar ist. Nachteilhaft dagegen wirkt sich aus, daß es zu einer unkontrollierten Propagierung von Rechten (und daher auch zu Problemen beim Entzug (Revokation) dieser Rechte) kommen kann [DoD87a]: Das Subjekt kann seine Fähigkeiten kopieren und weiterreichen. Wird ihm ein Recht entzogen, so kann nicht sichergestellt werden, daß es noch weitere Kopien besitzt, mit denen es weiterhin Zugriff erlangen kann. Abhilfe schaffen hier nur kryptographische Verfahren (vergleiche zum Beispiel [FKK90]). Weitere Nachteile bestehen darin, daß die einzelnen Fähigkeitslisten sehr lang sind (da die Zahl der Objekte in einem System in der Regel die Zahl der Subjekte bei weitem übersteigt) und daß die Frage, wer auf ein bestimmtes Objekt zugreifen darf, aufgrund der gewählten Speicherungsstruktur für die administrierten Rechte nicht direkt beantwortet werden kann. Hierzu ist eine exhaustive Suche über die Fähigkeitslisten aller im System vorhandenen Subjekte erforderlich. Der Einsatz von Fähigkeitslisten gilt als unvorteilhaft, wenn mit einer hohen Fluktuation von Objekten im System zu rechnen ist [Pri90].

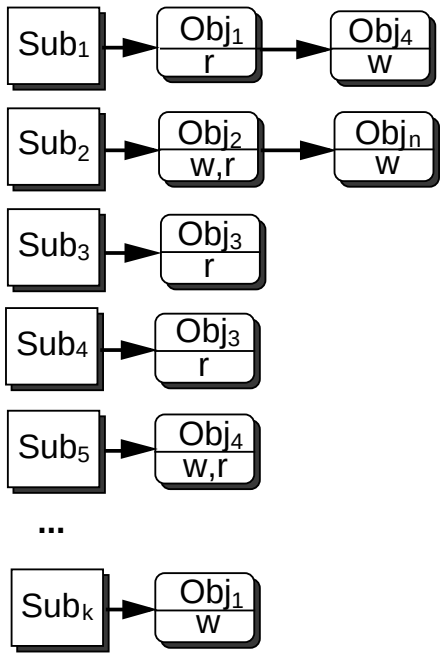


Abbildung 2.4: Fähigkeitslisten

Abbildung 2.4 zeigt eine Fähigkeitsliste, die sich durch zeilenweise Partitionierung aus der Zugriffskontrollmatrix in Abbildung 2.3 ableitet.

2.2.3 Zugriffskontrolle durch Paßworte

Eine wenig verbreite Methode zum Schutz von Objekten ist die paßwortbasierte Autorisierung [DoD87a]. Sie wird beispielsweise in dem Großrechnerbetriebssystem MVS[1] verwendet.
Besitzt jedes Subjekt ein eigenes Paßwort für die Objekte, auf die es zugreifen darf, so kann ein Paßwort ähnlich wie bei den Fähigkeitslisten als Eintrittskarte für den Zugriff auf ein Objekt angesehen werden. Die paßwortbasierte Zugriffskontrolle führt zu diversen Probleme mit sich [DoD87a]:

- Eine kontrollierte Propagierung von Rechten kann nicht durch das Autorisierungssystem erzwungen werden, da nicht verhindert werden kann, daß ein Subjekt das Paßwort für den Zugriff auf ein Objekt einem anderen (nicht autorisierten) Subjekt verrät.

- Da die Zahl der Objekte in einem System gewöhnlich die Zahl der Subjekte bei weitem übersteigt, muß ein Subjekt sehr viele Paßworte verwalten. Die Administration dieser Paßworte erfordert Datenstrukturen, die dann ihrerseits geschützt werden müssen.

- Programme, die während der Exekution auf bestimmte Daten zugreifen müssen, benötigen dafür die entsprechenden Paßworte. Dies führt dazu, daß diese dann in Form von Zeichenketten in das Programm eingebettet werden müssen. Jedes explizit niedergeschriebene Paßwort stellt jedoch eine potentielle Gefahr für die Sicherheit des Systems dar. Haben andere Benutzer ebenfalls Zugriff auf dieses Programm, sind aber nicht im Besitz aller Paßworte der durch die Ausführung tangierten Objekte, so können sie Informationen verarbeiten, für die sie nicht autorisiert wurden [Gas88].

- Um zwischen verschiedenen Zugriffsmodi differenzieren zu können, muß für jede Kombination von Zugriffsarten ein eigenes Paßwort bereitgestellt werden. Aufgrund des hohen administrativen Aufwandes beschränken sich die meisten real existierenden Systeme jedoch darauf, nur ein Paßwort pro Objekt zu verwalten, so daß entweder alle oder keine Funktionen auf diesem Objekt ausgeführt werden können.

- Der sichere Entzug eines erteilten Rechtes ist aufgrund der unkontrollierten Distribution von Paßworten nur wie folgt möglich: Das Paßwort wird ausgetauscht, und das neue Paßwort darf nur jenen Subjekten bekanntgegeben werden, die auch weiterhin auf das betrachtete Objekt zugreifen sollen. Eine Variante dieser Methode ist die periodische Substitution von Paßworten.

Während Paßworte ein probates Mittel für den Vorgang der Identifikation und der Authentisierung darstellen, ist ihr Einsatz für die Autorisierung ungeeignet [Gas88].

2.2.4 Zugriffskontrollisten

Ein weiterer Ansatz zur Reduktion des Speicheraufwandes für die Einträge in der Zugriffskontrollmatrix sind die Zugriffskontrollisten (*access control lists, ACL*). Hierbei erfolgt die Partitionierung der Zugriffskontrollmatrix spaltenweise. Wiederum werden dabei alle leeren

[1] **M**ultiple **V**irtual Memory **S**pace, MVS ist ein eingetragenes Warenzeichen der Firma **I**nternational **B**usiness **M**achines Inc.

Einträge entfernt. Somit werden jedem Objekt alle Subjekte und deren Menge von Zugriffsberechtigungen auf diesem Objekt zugewiesen, sofern diese Menge nicht leer ist. Die Rechte werden also direkt beim Objekt gespeichert.

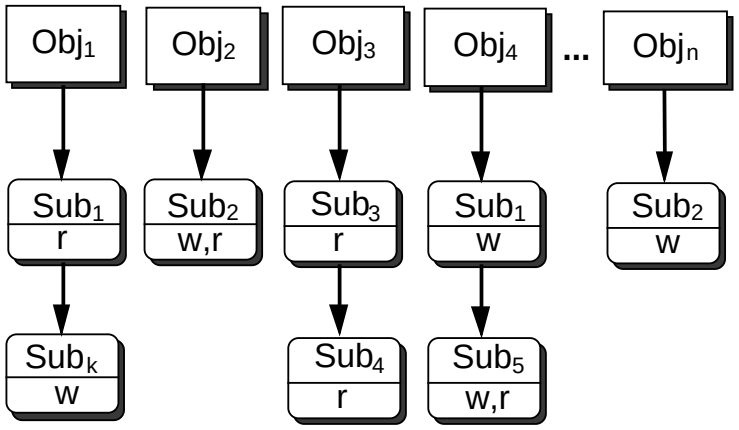


Abbildung 2.5: Zugriffskontrolliste

Eine Zugriffskontrolliste, die sich durch die spaltenweise Partitionierung der Zugriffskontrollmatrix aus Abbildung 2.3 ableiten läßt, zeigt Abbildung 2.5.
Ein Vorteil von Zugriffskontrollisten besteht darin, daß die zu verwaltenden Listen recht kurz sind (da die Zahl der Subjekte in einem System gewöhnlich kleiner als die Zahl der Objekte ist). Der Verwaltungsaufwand läßt sich sogar noch weiter reduzieren, wenn Subjekte in Gruppen angeordnet werden (vergleiche Abschnitt 2.5). Darüber hinaus lassen sich die für ein bestimmtes Objekt zugriffsberechtigten Subjekte schnell bestimmen, und es werden, wie bereits erwähnt, nur relevante Einträge verwaltet. Als Nachteil ergibt sich aufgrund der gewählten Speicherungsstruktur, daß die Fähigkeiten eines Subjektes nur dadurch bestimmt werden können, daß die Zugriffskontrollisten aller Objekte traversiert werden. Der Einsatz von Zugriffskontrollisten gilt als inadäquat, wenn mit einer hohen Fluktuation von Subjekten im System zu rechnen ist [Pri90].
Werden Subjekte, die in eine Zugriffskontrolliste eingefügt oder aus dieser entfernt werden sollen, als Zeichenketten (*String*) repräsentiert, ist es auch möglich, mit **Zeichenersatzdarstellungen** (*wildcards*) zu arbeiten [DRKJ85, DoD87a], wie sie die meisten Betriebssysteme bereitstellen. Hierbei ist ein Stern (*) die Ersatzdarstellung für eine Zeichenkette beliebiger Länge (inklusive 0) und beliebigen Inhalts, während ein Fragezeichen (?) genau ein beliebiges Zeichen repräsentiert. Mit einer Zeichenfolge “?w*” können dann beispielsweise alle Subjekte referenziert werden, deren zweiter Buchstabe ein “w” ist. Zeichenersatzdarstellungen tragen zur Reduktion des Modellierungsaufwandes bei, können aber Konflikte auslösen, wenn sich für bestimmte Subjekte mehrere sich widersprechende Einträge innerhalb einer Zugriffskontrolliste ergeben. Subsumierend läßt sich sagen, daß dieser Mechanismus die Komplexität diskreter Zugriffskontrolle erhöht, jedoch zu keiner Steigerung des Nutzens führt [DoD87a].
Eine weitere Vereinfachung kann durch **vorgegebene Zugriffskontrollisten** (*default ACL's*) erreicht werden. Dabei erhält jedes Objekt bei seiner Erzeugung eine Zugriffskontrolliste mit einer festgelegten Menge von Subjekten. Für benutzerbestimmbare Zugriffskontrolle ist es essentiell, daß bei der Erzeugung eines Objektes mindestens ein Subjekt in die Zugriffskontrol-

liste eingetragen wird (meist das erzeugende Subjekt). Abweichend von oder ergänzend zu dieser generellen Vorgabe können Zugriffskontrollisten systemweit oder für bestimmte Teilsysteme vorgegeben werden. Hierbei ist zwischen einer Kopiersemantik und einer Referenzsemantik zu differenzieren: Wird eine Zugriffskontrolliste mit Vorgaben zentral gespeichert, auf die die Zugriffskontrollisten aller anderen Objekte des (Teil-)Systems referenzieren (Referenzsemantik), so gelten Änderungen an der vorgegebenen Zugriffskontrolliste unmittelbar für alle anderen Zugriffskontrollisten. Werden die Vorgaben dagegen in jede einzelne Zugriffskontrolliste hineinkopiert (Kopiersemantik), haben Änderungen an der vorgegebenen Zugriffskontrolliste keinen Einfluß mehr auf die Zugriffskontrollisten, die bereits mit diesen Vorgaben arbeiten [DRKJ85, DoD87a].
Abschließend sei noch auf die Möglichkeit verwiesen, **benannte Zugriffskontrollisten** zu verwenden. Benannte Zugriffskontrollisten sind Vorlagen, die immer dann eingesetzt werden können, wenn viele Zugriffskontrollisten stets dieselben Subjekte enthalten sollen. Um dieses Ziel erreichen zu können, ist stets eine Referenzsemantik erforderlich. Benannte Zugriffskontrollisten müssen in der gleichen Weise geschützt werden wie "wirkliche" Zugriffskontrollisten. Benannte Zugriffskontrollisten sind entbehrlich, da sich ihre Eigenschaften auch mit Hilfe von Gruppen (siehe unten) oder Zugriffskontrollisten mit Vorgaben modellieren lassen [DRKJ85, DoD87a].

2.2.5 Zugriffskontrolle durch Schutzbits

Dieser Ansatz ist in vielen Betriebssystemen, wie zum Beispiel UNIX[2] und VMS[3], verwirklicht. Die Verwaltung der Zugriffskontrollinformationen erfolgt in Form einer reduzierten Zugriffskontrolliste beim Objekt. Hierbei wird die Administration der Liste der zugriffsberechtigten Subjekte durch die Verwaltung dreier Schutzbits substituiert. Im Falle von UNIX sind diese Schutzbits ein Indikator dafür, ob nur der Besitzer, seine Arbeitsgruppe oder jedes Subjekt des Systems Zugriffsrechte auf das Objekt besitzt. Differenziert wird zwischen drei Zugriffsmöglichkeiten (oder einer Kombination derselben): lesender, schreibender und ausführender Zugriff. Eine Änderung der Schutzbits kann nur der Besitzer des Objektes selbst (oder der Systemadministrator (*super user*)) vornehmen. Der Nachteil dieser Autorisierungsmethode liegt darin, daß der Objektbesitzer keine Möglichkeit hat, Autorisierungen auf Basis einzelner Benutzer zu erteilen. Auch die erforderliche Funktionalität zum Aufbau verschachtelter Subjekthierarchien (vergleiche Abschnitt 2.5) fehlt. Ebenso können keine gruppenübergreifenden Zugriffsrechte spezifiziert werden. Darüber hinaus erfolgt die Verwaltung der Subjekte, die zur Gruppe des Objektbesitzers gehören, durch den Systemadministrator. Aufgrund dieser mangelnden Flexibilität muß eine Autorisierung durch Schutzbits als unvollständige Methode des Zugriffsschutzes angesehen werden.

2.2.6 Delegierung von Rechten

Eine charakteristische Eigenschaft benutzerbestimmbarer Zugriffskontrolle in Abgrenzung zu regelbasierter Zugriffskontrolle (vergleiche Abschnitt 2.3) ist die Möglichkeit der Subjekte, Rechte für den Zugriff auf eigene Objekte weiterzupropagieren oder erteilte Rechte zu entziehen. Diese Befähigung der Rechtedelegierung wird aufgrund ihrer Mächtigkeit nicht als

[2] Unix ist ein eingetragenes Warenzeichen der Firma AT& T
[3] **V**irtual **M**emory **S**ystem, VMS ist ein eingetragenes Warenzeichen der Firma Digital Research

"gewöhnliches" Recht interpretiert, sondern stellt ein Metarecht (das Recht, Rechte erteilen zu dürfen) dar. Grundsätzlich sind zwei Arten von Rechtepropagierungen zu unterscheiden [CGG+77]:

- ▷ Wird ein Recht mit **Kopiererlaubnis** (*grant option, copy flag*) an ein Subjekt weiterpropagiert, so wird der Empfänger ermächtigt, dieses Recht seinerseits anderen Subjekten zu erteilen.
- ▷ Erfolgt die Delegierung ohne Kopiererlaubnis, so kann der Empfänger zwar die dadurch autorisierten Operationen auf Objekten durchführen, die Weitergabe dieser Rechte ist ihm jedoch untersagt.

Ziel eines wohldefinierten Autorisierungssystems muß es sein, Metarechte nicht unkontrolliert propagieren zu lassen, sondern die Historie der Delegierung genau zu verfolgen. Auf diese Weise soll erreicht werden, daß der Systemzustand nach dem Entzug eines Rechtes so ist, als ob dieses Recht nie erteilt worden wäre [Fag77, Den82]. Hierbei gilt es zwei Probleme zu berücksichtigen [Dit83]:

1. Ein Subjekt (Empfänger) kann dasselbe Recht von verschiedenen anderen Subjekten (Absendern) erhalten haben. Ruft einer der Absender dieses Recht zurück, so darf dies nicht zu einer Entziehung der Rechte führen, die durch andere Subjekte gewährt wurden.
2. Da das Empfängersubjekt nach dem Erhalt eines Rechtes mit Kopiererlaubnis berechtigt ist, dieses Recht seinerseits weiterzupropagieren, muß diese Möglichkeit der Weitergabe beim Rückruf eines Rechtes ebenfalls Berücksichtigung finden. Wurde dasselbe Recht auch von anderen Absendern erhalten (siehe Punkt 1), so können, je nach zeitlicher Reihenfolge, die Weitergaben von Rechten an Dritte trotz des Rückrufs eines Absenders unbeeinflußt bleiben.

Ein Propagierungs- und Revokationsalgorithmus, der diese beiden Probleme löst und sich bis heute durchgesetzt hat, wurde erstmals in System R [GW76] vorgestellt. Dort wird jedes propagierte Recht durch ein Quintupel mit folgenden Konstituenten repräsentiert:

1. dem Subjekt, das das Recht erteilt hat (Absender, *grantor*),
2. dem Subjekt, das das Recht empfangen hat (Empfänger, *grantee*),
3. dem Objekt, für das das Recht gelten soll,
4. den Zugriffsarten, die auf diesem Objekt autorisiert werden sollen,
5. einer Kopiermarke, die anzeigt, ob das empfangene Recht weiterpropagiert werden darf.

Um die Propagierungshistorie aufzuzeichnen, wird jedes erteilte Recht mit einem Zeitstempel (*time flag*) versehen. Die Geschichte eines propagierten Rechtes kann dann für ein gegebenes Objekt mit Hilfe eines gerichteten Graphen visualisiert werden. Die Knoten symbolisieren dabei absendende und empfangende Subjekte, während die Kanten die Propagierung eines Rechtes widerspiegeln und mit Zeitstempel, Zugriffsrecht und Kopiermarke versehen sind.

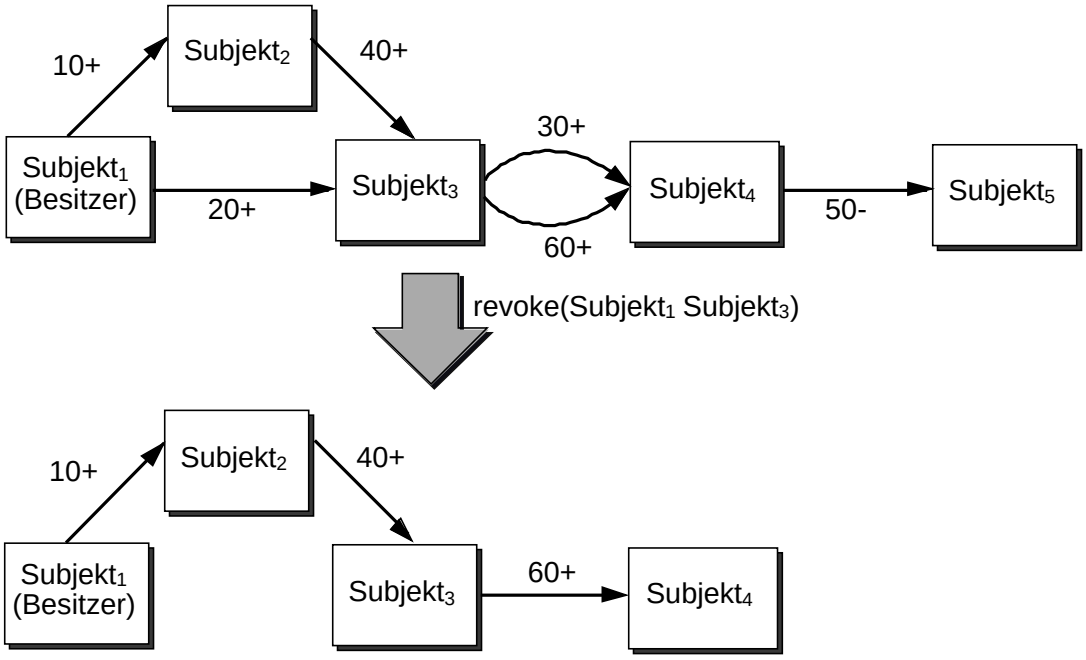


Abbildung 2.6: Propagierung und Revokation eines Rechtes

Abbildung 2.6 zeigt die Historie eines propagierten Rechtes (Objekt und Zugriffsmodus sind konstant). Die Zahlen repräsentieren Zeitstempel; ein Pluszeichen entspricht einer gesetzten Kopiererlaubnis. Entzieht $Subjekt_1$ dem $Subjekt_3$ das zum Zeitpunkt 20 erteilte Zugriffsrecht, so ist rekursiv zu prüfen, ob weitere Rechte entzogen werden müssen. Dazu wird der kleinste Zeitstempel aller eingehenden Kanten mit den Zeitstempeln der ausgehenden Kanten verglichen. Ausgehende Kanten, die einen kleineren Zeitstempel besitzen, werden entfernt. $Subjekt_3$ hat das Recht zum Zeitpunkt 40 empfangen, kann es also nicht zum Zeitpunkt 30 weitergegeben haben. Diese Propagierung kann daher nur aufgrund des gelöschten zum Zeitpunkt 20 erteilten Rechtes erfolgt sein. Daher wird auch diese Kante entfernt. Analog gilt, daß $Subjekt_4$ ein zum Zeitpunkt 60 erhaltenes Recht nicht zum Zeitpunkt 50 propagiert haben kann, daher wird auch diese Autorisierung aufgehoben. Die noch gültigen delegierten Rechte sind im unteren Teil der Abbildung dargestellt.

2.3 Regelbasierte Zugriffskontrollmodelle

Die Zugriffskontrolle in benutzerbestimmbaren Zugriffskontrollsystemen entscheidet auf der Basis von Benutzeridentitäten über den Zugriff auf Daten. Da jeder Besitzer von Daten prinzipiell in der Lage ist, jedem Subjekt Zugriff auf diese Daten zu gewähren, zeichnet sich benutzerbestimmbare Zugriffskontrolle durch ein hohes Maß an Flexibilität aus. Häufig erfordert die Anwendungssemantik jedoch eine restriktivere Einschränkung des Zugriffs durch fest definierte Anwendungsregeln. Die sachgerechte Benutzung diskreter Zugriffskontrolle allein reicht nicht aus, um diese Regeln einzuhalten, da nicht erzwungen werden kann, daß jedes Subjekt sie befolgt. Insbesondere kann benutzerbestimmbare Zugriffskontrolle durch Anomalien wie “Trojanische Pferde” kompromittiert werden.

Ein **Trojanisches Pferd** ist ein Block von Programmieranweisungen, die absichtlich und ohne Kenntnis des ausführenden Benutzers in ein anderes Programm "implantiert" wurden und deren Exekution für den Benutzer unbeabsichtigte und unerwünschte Folgen nach sich ziehen. Beispiele für Trojanische Pferde sind "logische Bomben" und "Zeitbomben". **Logische Bomben** sind bösartige Programme, die aktiviert werden, wenn der Systemzustand gewisse Bedingungen erfüllt. **Zeitbomben** sind logische Bomben, die an einem bestimmten Datum oder zu einer bestimmten Zeit aktiviert werden [Hof90]. Von vielen Computerviren ist bekannt, daß sie ein Trojanisches Pferd in sich bergen. In Abgrenzung zu Computerviren können sich Trojanische Pferde jedoch nicht selbst reproduzieren [For94]. Die Bedrohung durch Trojanische Pferde in benutzerbestimmbaren Zugriffskontrollsystemen kann nicht gänzlich verhindert werden. Sie läßt sich jedoch signifikant reduzieren, wenn die Weitergabe von Rechten derart begrenzt werden kann, daß eine strikte Trennung von Subjekten und Objekten in kleine Kompartimente ermöglicht wird [Den82, DoD87a].

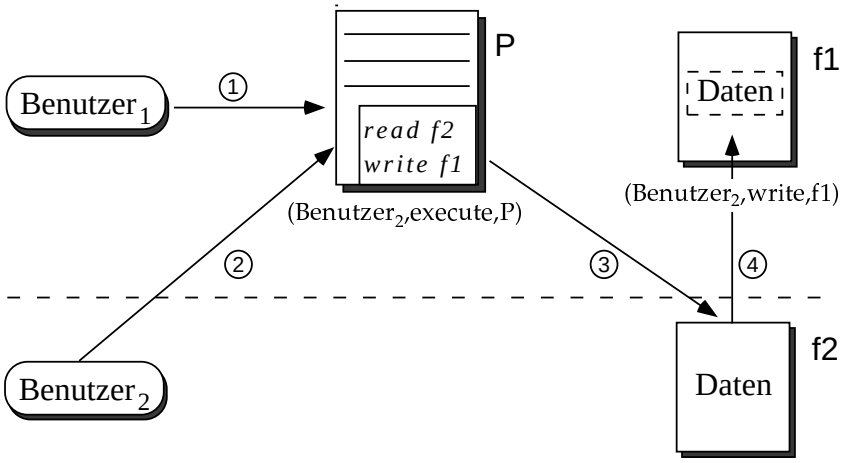


Abbildung 2.7: Wirkungsweise eines Trojanischen Pferdes

Die Umgehung diskreter Zugriffskontrolle durch ein Trojanisches Pferd veranschaulicht Abbildung 2.7. $Benutzer_2$ verwaltet sensitive Daten, die er in einer Datei f_2 abgelegt hat und auf die er als Besitzer die exklusiven Zugriffsrechte besitzt. $Benutzer_1$, der Besitzer einer leeren Datei f_1 ist, möchte in den Besitz dieser Daten gelangen; dies ist auf direktem Wege jedoch nicht möglich, da ihm die Zugriffsrechte fehlen. Daher schreibt er ein für $Benutzer_2$ nützliches Programm P, das ein verstecktes Stück Programmkode, bestehend aus einem lesenden Zugriff auf f_2 und einem schreibenden Zugriff auf f_1, enthält. $Benutzer_1$ verändert die Zugriffsrechte für P derart, daß $Benutzer_2$ eine Ausführungserlaubnis für P besitzt und autorisiert ihn ebenfalls für den schreibenden Zugriff auf f_1 (Schritt ①). Ruft $Benutzer_2$ P auf (Schritt ②), so wird dieses Programm unter seiner Benutzeridentität ausgeführt, das heißt alle Zugriffsanfragen werden gegen die Autorisierungen von $Benutzer_2$ geprüft. Mit dem Programm P werden insbesondere die versteckten Programmanweisungen ausgeführt, das heißt die Daten in f_2 werden gelesen (Schritt ③) und in die Datei f_1 geschrieben (Schritt ④). Da $Benutzer_2$ sowohl ein Leserecht auf f_2 als auch ein Schreibrecht auf f_1 besitzt, kommt es zu keiner Verletzung der Zugriffsrechte. Dennoch hat ein Transfer von geschützten Informationen

(aus f_2) in die Datei f_1 des nicht autorisierten Subjektes $Benutzer_1$ stattgefunden.
Der Wunsch, feste Regeln in einem Zugriffskontrollsystem definieren zu können, die den Kreis der möglichen zugreifenden Subjekte beschränkt, motiviert die Einführung regelbasierter oder mandatorischer Zugriffskontrollmodelle. Regelbasierte Zugriffskontrolle ist eine "Maßnahme, den Zugriff auf Objekte zu beschränken, die auf der Sensitivität der objektinhärenten Information (repräsentiert durch eine Sicherheitsmarke) und der formalen Autorisierung (Ermächtigung (*clearance*)) von Subjekten auf Informationen dieser Sensitivität zuzugreifen, basiert"[DoD85]. Charakteristisch für diese Systeme ist, daß Subjekten und Objekten feste Sicherheitsattribute zugeordnet werden [Gas88]. Die Vergabe dieser Sicherheitsattribute erfolgt entweder durch eine zentrale Instanz (zum Beispiel den Sicherheitsadministrator) oder wird aus den Regeln des darunterliegenden Betriebssystems abgeleitet. Das Zugriffskontrollsystem entscheidet dann aufgrund von Regeln, die bei ihrer Evaluierung auf diese Sicherheitsattribute zugreifen, über den Zugriff. Die Subjekte selbst sind nicht in der Lage, Sicherheitsattribute von Objekten oder anderen Subjekten beziehungsweise ihre eigenen Sicherheitsattribute zu modifizieren. Desweiteren gilt, daß Subjekte die mit ihren Sicherheitsattributen verbundenen Rechte nicht auf andere Subjekte übertragen können. Hieraus folgt insbesondere (im Gegensatz zu benutzerbestimmbarer Zugriffskontrolle), daß der Besitzer eines Objektes unter Umständen überhaupt keinen Zugriff mehr auf seine eigenen Objekte hat.
Grundsätzlich können die Sicherheitsregeln in einem solchen System frei definiert werden. Es ist jedoch zu analysieren, inwiefern die definierten Regeln angemessen sind, die verfolgte Sicherheitspolitik zu realisieren. Eine Vielzahl regelbasierter Zugriffskontrolle basiert auf **mehrstufiger Zugriffskontrolle** (*multilevel access control*), daher werden sich die weiteren Betrachtungen dieses Abschnitts auf diese Modellklasse konzentrieren. Regelbasierte Autorisierungsmodelle, die den Zugriff nicht durch mehrstufige Zugriffskontrolle beschränken, sind zum Beispiel das *Chinese-Wall*-Modell [BN89], das Verbreitungskontrollmodell (*dissemination control model*) [JMMN90] und regelbasierte Datenschutzmodelle (*privacy model*) [FH95].

Mehrstufige Zugriffskontrollmodelle, deren Anwendungsbereich in erster Linie militärische Anwendungen umfaßt [FK93], kontrollieren nicht nur den Zugriff auf Objekte, sondern auch den Fluß der darin enthaltenen Information durch das System. Hierzu wird eine Menge disjunkter, hierarchisch geordneter Sicherheitsklassen definiert. Jedes Objekt des Systems wird genau einer dieser Sicherheitsklassen zugeordnet und mit einer Sicherheitsmarke versehen, die diese Sicherheitsklasse repräsentiert. Die Klassifikation der zu schützenden Information reflektiert dabei den potentiellen Schaden, der durch eine nicht autorisierte Enthüllung (*disclosure*) der Information entstehen könnte [Lun91]. Analog zur Klassifikation von Objekten wird jedem Subjekt eine Ermächtigung zugeordnet, die einer der Sicherheitsklassen entspricht und somit regelt, auf welche Objekte es zugreifen darf.

In Anlehnung an Abbildung 2.2 veranschaulicht Abbildung 2.8 [Den82], welche Eigenschaften ein präzises sicheres Informationsflußkontrollsystem erfüllen muß. Ein solches System ist **sicher**, wenn jeder erreichbare Systemzustand auch durch die Sicherheitspolitik autorisiert wurde. Es ist **präzise**, wenn darüber hinaus jeder autorisierte Zustand auch erreichbar ist.
Im folgenden werden zwei der wichtigsten mehrstufigen Zugriffskontrollmodelle vorgestellt.

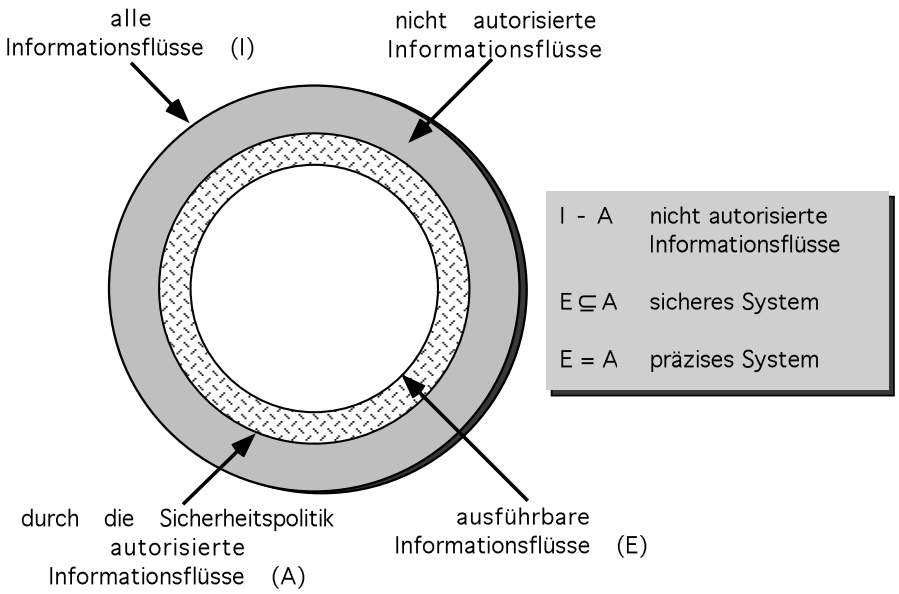


Abbildung 2.8: Präzision und Sicherheit von Informationsflußkontrollsystemen

2.3.1 Das Bell-LaPadula Modell

Im Modell von Bell und LaPadula [BL73, BL76] erfolgt eine Einteilung in Sicherheitsstufen, die partiell geordnet sind. Eine **Sicherheitsstufe** besteht aus einer Klassifikation (zum Beispiel: unklassifiziert, vertraulich, geheim und streng geheim) und einer Menge von Kategorien. Die Kategorie bezeichnet eine Menge von Kompartimenten, auf die ein bestimmtes Subjekt zur Erfüllung seiner Aufgaben zugreifen können muß (*need-to-know designation*). Beispiele für solche Kompartimente in einem Unternehmen sind die Buchhaltung, das Personalwesen oder der Einkauf.
Es werden vier Zugriffsmodi unterschieden: lesender Zugriff, schreibender Zugriff, ausführender Zugriff und anfügender Zugriff (Schreibrecht ohne Leseberechtigung). Die Fähigkeit, einem Subjekt neue Autorisierungen hinzuzufügen oder diese zu entfernen, wird über ein Recht für kontrollierenden Zugriff geregelt. Der Zustand eines Bell-LaPadula Modells wird durch ein Quadrupel (b,M,f,H) formalisiert:

- Die erste Komponente b bezeichnet die Menge der gegenwärtigen Zugriffsrechte (*current access set*). Diese Menge besteht aus Tripeln (S,O,Z) mit der Semantik, daß Subjekt S gegenwärtig auf Objekt O mit Zugriffsmodus Z zugreifen darf.

- Die Komponente M steht für eine Zugriffskontrollmatrix (*access permission matrix*), die bereits in Abschnitt 2.2.1 beschrieben wurde. Diese Matrix enthält alle erlaubten Zugriffsrechte und ist eine Obermenge der gegenwärtigen Zugriffsrechte.

- Die Komponente f designiert eine Stufenfunktion (*level function*). Diese Funktion ordnet jedem Subjekt eine Ermächtigung und jedem Objekt eine Sicherheitsstufe (siehe oben) zu. Zu beachten ist dabei, daß bei Subjekten zwischen einer maximalen Ermächtigung und einer gegenwärtigen Ermächtigung (wie sie zur Ausübung der gegenwärtigen Rechte

dieses Subjektes erforderlich ist) differenziert wird. Die maximale Ermächtigung dominiert[4] stets die gegenwärtige Ermächtigung. Eine Sicherheitsstufe A dominiert eine Sicherheitsstufe B, wenn die Klassifikation von A die Klassifikation von B dominiert und die Kategorien von B eine Teilmenge der Kategorien von A sind.

- ▷ Das Akronym H steht für eine Hierarchie von Objekten. Eine Hierarchie ist ein graphentheoretischer Wald (Menge von Bäumen) [BL76]. Objekte, die sich nicht in einem Baum befinden, gelten als inaktiv und nicht zugreifbar. Die Knoten des Objektgraphen sind so angeordnet, daß die Sicherheitsstufe eines Objektes stets die Sicherheitsstufe seines Vaterobjektes dominiert.

Zur Modifikation einzelner Komponenten dieses Quadrupels stehen insgesamt elf Transformationsregeln zur Verfügung. Bell und LaPadula haben formal bewiesen, daß unter ausschließlicher Verwendung dieser Transformationsregeln auf den Zustand eines Sicherheitssystems die nachfolgenden beiden Sicherheitseigenschaften erhalten bleiben [BL73, BL76]:

1. **Einfache Sicherheitseigenschaft** (*simple security property*). Die Zugriffsberechtigung für den lesenden oder exekutierenden Zugriff auf ein Objekt wird nur dann erteilt, wenn die Sicherheitsstufe des Subjektes die des Objektes dominiert (*no read up*).

2. **Rangeigenschaft** (**-property, confinement-property*). Ein Subjekt hat nur dann gleichzeitig einen Lesezugriff auf ein Objekt O_1 und Schreibzugriff auf ein Objekt O_2, wenn die Sicherheitsstufe von O_2 die Sicherheitsstufe von O_1 dominiert [Pfl89]. Daraus folgt insbesondere, daß ein schreibender Zugriff auf ein Objekt nur dann erlaubt ist, wenn die gegenwärtige Sicherheitsstufe des Subjektes von der des Objektes dominiert wird (*no write down*).

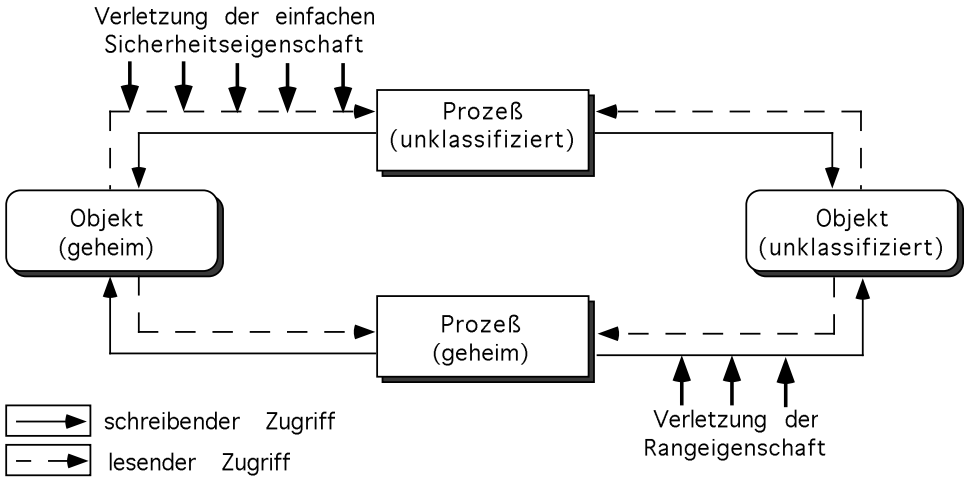


Abbildung 2.9: Autorisierte Zugriffe im Bell-LaPadula Modell

Darüber hinaus gilt das **Tranquilitätsprinzip**. Es besagt, daß kein Subjekt die Sicherheitsklasse eines Subjektes oder Objektes verändern darf [Poh89, MC84]. Abbildung 2.9 veranschaulicht

[4] Ex definitione dominiert jede Sicherheitsstufe sich selbst [MC84].

exemplarisch für zwei Klassifikationen, welche Informationsflüsse zwischen Prozessen und Objekten beim Einhalten der oben genannten Sicherheitseigenschaften unterbunden werden.
Das Bell-LaPadula Modell wird als Referenzmodell zur Evaluation und Zertifizierung konkreter Sicherheitssysteme nach dem sogenannten "Orange Book"[DoD85] eingesetzt. Das "Orange Book" enthält Sicherheitskriterien, mit denen die Sicherheit eines Systems eingestuft wird. Ein Sicherheitssystem, das die Anforderungen für regelbasierte Zugriffskontrolle erfüllen soll, muß die Eigenschaften des Bell-LaPadula Modells besitzen.
Trotz seiner hohen Signifikanz ist in der wissenschaftlichen Diskussion Kritik an diesem Modell geübt worden [LH94]: Das Bell-LaPadula Modell läßt Probleme der Integrität und Verfügbarkeit unberücksichtigt [Lip91, Par92], und der Bereich, in dem es Anwendung finden kann, ist zu eng gesteckt; er beschränkt sich quasi nur auf ein militärisches Umfeld [Lip91]. Eine Differenzierung von Objekten in verschiedene Granularitäten fehlt, und auch eine Trennung zwischen Objektcontainern und ihrem eigentlichen Inhalt findet nicht statt. Würde eine Anwendung beispielsweise mehrstufige Dokumente, das heißt Dokumente, die Daten verschiedener Klassifikationen enthalten, verwalten, so könnte sie nicht durch das Modell unterstützt werden [Lan81]. Operationen zum Anlegen neuer Subjekte sind nicht vorgesehen worden, und auch eine allgemeine Reklassifikation von Daten kann aufgrund der fehlenden Dynamik des Systems nicht durchgeführt werden. Hierzu bedarf es "besonders vertrauenswürdiger" Subjekte (*Trusted Subjects*), für die die Rangeigenschaft außer Kraft gesetzt ist [Lan81]. Auch gibt es keine konkreten Anhaltspunkte, nach welchen Kriterien die Ermächtigung eines Subjektes festgelegt werden kann. Im militärischen Bereich basiert diese Entscheidung unter anderem auf psychotechnischen und psychometrischen Analysen und Techniken, die als dubios einzustufen sind. Auch für die Klassifikation personenbezogener Daten, deren Sensitivität zweckabhängig einzustufen ist, erweist sich ein statisches System von Sicherheitsstufen als unangemessen. Darüber hinaus zeigt sich, daß das Bell-LaPadula Modell Vertraulichkeit in offenen verteilten Systemumgebungen nicht erzwingen kann, da es - historisch bedingt - für geschlossene Systemumgebungen entwickelt wurde [LH94].

2.3.2 Das Biba Modell

Im Gegensatz zu den bisher beschriebenen regelbasierten Zugriffsmodellen, deren Ziel die Gewährleistung von Vertraulichkeit war, ist es Ziel des Modells von Biba [Bib77], die Integrität von Systeminformationen zu wahren, das heißt nicht autorisierte Modifikation von Informationen zu unterbinden [MC84]. An Stelle von Sicherheitsstufen werden analog zu den Sicherheitsklassen *streng geheim, geheim* und *vertraulich* die Integritätsstufen *kritisch, sehr wichtig* und *wichtig* als Ermächtigungen beziehungsweise Objektmarkierungen bereitgestellt. Das Biba Modell unterstützt die folgenden vier Arten regelbasierter Integritätspolitiken [Bib77]:

Niedrigwassermarkenpolitik für Subjekte (*low water mark policy for subjects*): In dieser Politik ist die Integritätsstufe eines Subjektes abhängig von den Integritätsstufen der Objekte, auf die es vorher zugegriffen hat.

Nach jedem lesenden Zugriff auf ein Objekt entspricht die Integritätsstufe des Subjektes dem Minimum aus den Integritätsstufen des zugreifenden Subjekts und des Objekts, auf das zugegriffen wurde. Die Integritätsstufe eines Subjektes ist also dynamisch und monoton fallend. Die Wirkungsweise dieser Politik veranschaulicht Abbildung 2.10. $Subjekt_1$ hat in seiner bisherigen Historie nur Objekte der Integritätsstufe *kritisch* gelesen. Seine Ermächtigung wird daher auch als *kritisch* eingestuft (Teilbild I). Beim Zugriff auf das

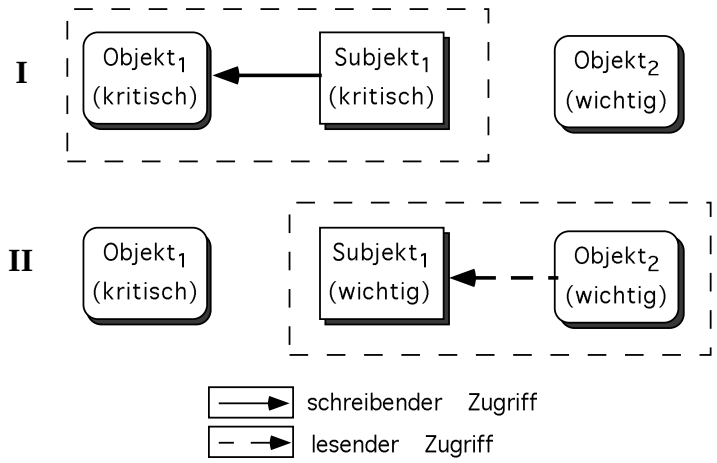


Abbildung 2.10: Niedrigwassermarkenpolitik für Subjekte

niedriger klassifizierte $Objekt_2$ (Integritätsstufe: *wichtig*) wird die Integritätsstufe des Subjektes irreversibel auf *wichtig* zurückgestuft. Ein Zugriff auf $Objekt_1$ ist fortan nicht mehr möglich. Zu beachten ist, daß bei dieser Politik das Kommutativgesetz nicht gilt: Hätte in dem Beispiel $Subjekt_1$ zuerst auf $Objekt_2$ zugegriffen, so wäre ein anschließender Zugriff auf $Objekt_1$ nicht mehr möglich.

Niedrigwassermarkenpolitik für Objekte (*low water mark policy for objects*): Analog zur vorherigen Politik wird durch diese Integritätspolitik die Integritätsstufe modifizierter Objekte verändert. Anstatt zu verhindern, daß Subjekte Objekte höherer Integritätsstufen verändern, wird die Integritätsstufe des Objektes auf die des Subjektes zurückgestuft.

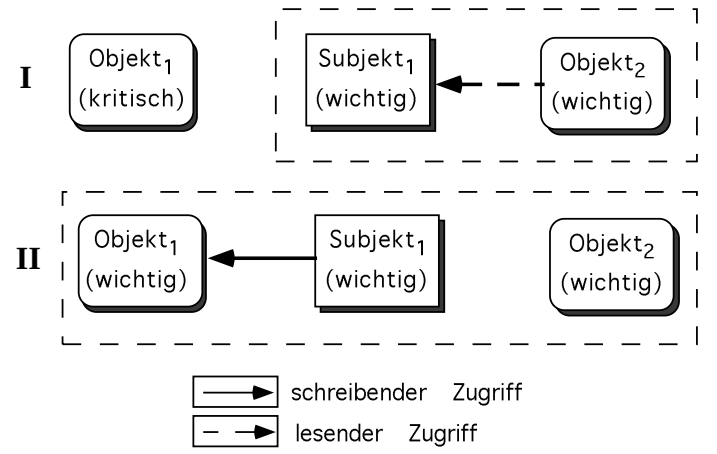


Abbildung 2.11: Niedrigwassermarkenpolitik für Objekte

Abbildung 2.11 zeigt, wie die Integritätsstufe (*kritisch*) des $Objektes_1$ durch den modifizierenden Einfluß des $Subjektes_1$ irreversibel auf dessen Integritätsstufe (*wichtig*) zurückgestuft wird.

Ringpolitik (*ring policy*): Die Integritätsstufen von Subjekten und Objekten sind unveränderlich. Für den schreibenden und aufrufenden Zugriff von Subjekten werden folgende Eigenschaften zugesichert:

- **Einfache Integritätseigenschaft** (*simple integrity property*): Es darf nur dann in ein Objekt geschrieben werden, wenn die Integritätsstufe des Subjektes höher als die des Objektes ist.
- **Aufrufeigenschaft** (*invocation property*): Sie trifft eine Regelung für Indirektionen im Informationsfluß zwischen Subjekten und Objekten. Ruft ein Subjekt beispielsweise einen Prozeß auf, der dann auf die geschützten Objekte zugreift, so fungiert der Prozeß selbst sowohl als Objekt (aus der Sicht des aufrufenden Subjektes) als auch als Subjekt (aus der Sicht der Objekte, auf die zugegriffen wird). Ein solcher Aufruf eines Subjektes (hier: des Prozesses) durch ein anderes Subjekt ist nur dann erlaubt, wenn die Integritätsstufe des aufrufenden Subjektes die des aufgerufenen Subjektes dominiert.

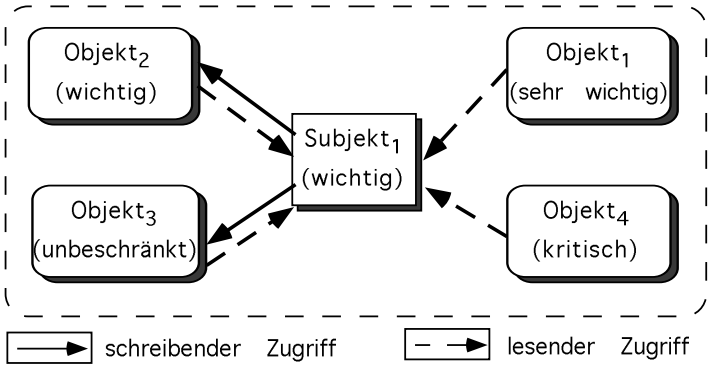


Abbildung 2.12: Erlaubte Informationsflüsse bei einer Ringpolitik

Alle Subjekte haben uneingeschränkten lesenden Zugriff auf alle Objekte. Das Subjekt muß selbst entscheiden, ob es niedrig klassifizierten Informationen traut. Alle im Rahmen der Ringpolitik erlaubten Zugriffe zeigt exemplarisch Abbildung 2.12.

Strikte Integritätspolitik (*strict integrity policy*): Diese Politik stellt das duale Gegenstück zum Sicherheitsmodell von Bell und LaPadula dar. Sie ist durch die folgenden drei Axiome charakterisiert:

1. **Einfache Integritätseigenschaft** (*siehe oben*).
2. **Integritätsabgrenzung** (*integrity confinement*). Die Zugriffsberechtigung für den lesenden Zugriff auf ein Objekt wird nur dann erteilt, wenn die Integritätsstufe des Subjektes kleiner oder gleich der des Objektes ist.
3. **Aufrufeigenschaft** (*siehe oben*).

Die Kontrolle autorisierter und prohibitiver Informationsflüsse im Biba Modell bei Anwendung einer strikten Integritätspolitik ist in Abbildung 2.13 graphisch dargestellt: Die Kontaminierung von Objekten hoher Integrität (hier: sehr wichtig) durch schreibende Prozesse niedriger Integrität (hier: unklassifiziert) verstößt gegen die einfache

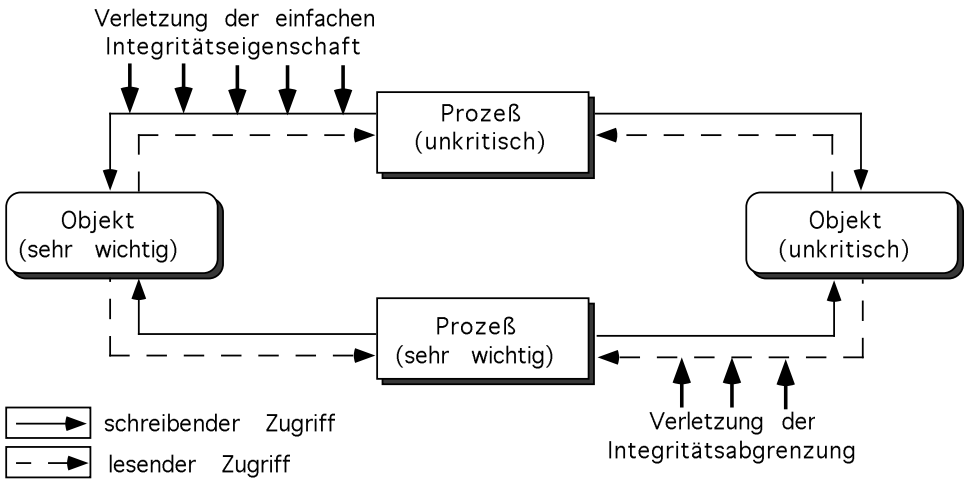


Abbildung 2.13: Autorisierte Informationsflüsse im Biba Modell unter Einhaltung einer strikten Integritätspolitik

Integritätseigenschaft und wird daher unterbunden. Ebenso werden Prozesse mit hoher Integrität davor geschützt, Informationen niedrigerer Integrität zu lesen, deren Weiterverarbeitung zu einer Verfälschung der Daten führen könnte.

Von praktischer Relevanz ist lediglich die strikte Integritätspolitik, da beim Gebrauch der anderen Sicherheitspolitiken Systemzustände entstehen können, in denen auf bestimmte Objekte überhaupt nicht mehr zugegriffen werden kann.
Da das Biba Modell unter Verwendung der strikten Integritätspolitik das duale Gegenstück zum Bell-LaPadula Modell darstellt, gelten einige der dort angeführten Kritikpunkte sinngemäß auch für dieses Modell. Das Biba Modell stellt in erster Linie den Schutz von Anwendungen vor Modifikationen durch nicht autorisierte Subjekte sicher [FH92]; die Erzwingung der inneren und äußeren Konsistenz sowie des Schutzes vor inkorrekten Modifikationen durch autorisierte Benutzer kann es nicht gewährleisten, da im Biba-Modell die Semantik der Daten nicht abgebildet werden kann.

2.3.3 Polyinstanziierung

Mit dem Konzept der Polyinstanziierung [DLS+87] ist der Versuch unternommen worden, die Eigenschaften mehrstufiger regelbasierter Zugriffskontrolle auf das relationale Datenmodell zu übertragen. Bei diesem Ansatz werden die Komponenten einer Datenbank mit einer Sicherheitsklasse versehen, die deren Sensitivität klassifizieren. Auf diese Weise werden Relationen zu mehrstufigen Relationen (*multilevel relations*) erweitert. Die Idee ist es dabei, Subjekten mit verschiedenen Ermächtigungen unterschiedliche Sichten (*views*, virtuelle Relationen) auf eine mehrstufige Relation zu gewähren. Auf diese Weise soll gewährleistet werden, daß Daten, die von hoch klassifizierten Benutzern eingefügt wurden, nicht von niedrig klassifizierten Benutzern gesehen werden können [Wis89]. Für das weitere Verständnis ist es wichtig, die folgenden beiden Begriffe einzuführen:

Ein **Primärschlüssel** (*apparent primary key*) ist eine minimale Menge von Attributen, mit der jedes Tupel einer nicht polyinstanziierten Relation eindeutig identifiziert werden kann.
Ein **Instanzidentifikator** ist ein Schlüssel, der aus einem Primärschlüssel und der Sicherheitsklasse für dessen Attribute gebildet wird.
Die Attributierung von Relationen mit Sicherheitsklassen kann auf einer der folgenden drei Ebenen erfolgen:

1. **Polyinstanziierte Relationen:** Hierbei werden jeweils ganze Relationen mit einer Sicherheitsklasse versehen. Dies führt dazu, daß prinzipiell mehrere Relationen mit dem gleichen Namen existieren können, die nur über ihre Sicherheitsklasse unterscheidbar sind.

2. **Polyinstanziierte Tupel:** Es existiert jeweils nur eine Relation, und jedes Tupel wird um ein Attribut für eine Sicherheitsklasse erweitert. Dadurch können mehrere gleichartige Tupel in der Relation koexistieren, das heißt zu einem Primärschlüssel kann es mehrere Tupel geben. Da diese Tupel durch ihre Sicherheitsklasse unterscheidbar sind, können sie über ihren Instanzidentifikator eindeutig referenziert werden.

3. **Polyinstanziierte Elemente:** Bei diesem Ansatz wird *jedes* Attribut der Relation um eine Sicherheitsklasse erweitert. Die Sicherheitsklassen für alle Attribute eines Primärschlüssels müssen dabei identisch sein. Zu einem Instanzidentifikator kann es somit mehrere Tupel geben, die sich nur durch die Sicherheitsklassen ihrer Nicht-Schlüsselattribute unterscheiden. Eine eindeutige Referenzierung von Tupeln ist dann nur noch möglich, wenn ein Schlüssel aus dem Instanzidentifikator und der geordneten Menge der Werte aller Sicherheitsklassen der Nicht-Schlüsselattribute gebildet wird (*full primary key*).

Die folgenden Betrachtungen beschränken sich auf den zweiten Ansatz.

Abbildung 2.14 zeigt ein Beispiel für eine Relation mit polyinstanziierten Tupeln, in der die Patienten eines Krankenhauses mit ihren Diagnosen verwaltet werden. Der Patient mit der Nummer 417 hat ein Bronchialkarzinom (bösartiger Tumor) mit geringen Heilungschancen. Diese richtige Diagnose sollen jedoch nur Subjekte mit der Ermächtigung *geheim* oder *streng geheim* (zum Beispiel Assistenz- und Oberärzte) kennen. Dem restlichen Krankenhauspersonal (Ermächtigung: *vertraulich*) sowie dem Patienten und seinen Angehörigen (Ermächtigung: *unbeschränkt*) soll nicht die ganze Wahrheit erzählt werden, um dem Patienten nicht die Hoffnung zu nehmen und damit zumindest seinen psychischen Zustand stabil zu halten. Dem Pflegepersonal soll daher mitgeteilt werden, der Patient habe ein Bronchialadenom (gutartiger Tumor); der Patient soll glauben, er habe eine gewöhnliche Bronchitis.
Das Tupel, das den Patient mit der Nummer 417 repräsentiert, wird daher polyinstanziiert. Das Attribut *Pat-Nr* fungiert hierbei als Primärschlüssel. Für jede der vier Sicherheitsklassen wird ein eigenes Tupel mit dem Primärschlüsselwert 417 angelegt. Diese vier Tupel können somit nur noch über ihren Instanzidentifikator, bestehend aus der Patientennummer und dem für alle Subjekte unsichtbaren Attribut Sicherheitsklasse, diskriminiert werden. Jedes Subjekt sieht dabei immer genau ein Tupel mit dem Primärschlüsselwert 417. Subjekten mit unterschiedlichen Ermächtigungen (hier: Patient und Oberarzt), die eine Anfrage gleichen Inhalts stellen, können daher unterschiedliche Antworten erhalten. Subjekte mit der maximalen Ermächtigung *streng geheim* sind in der Lage, alle Tupel des Primärschlüssels 417 zu sehen, da ihre gegenwärtige

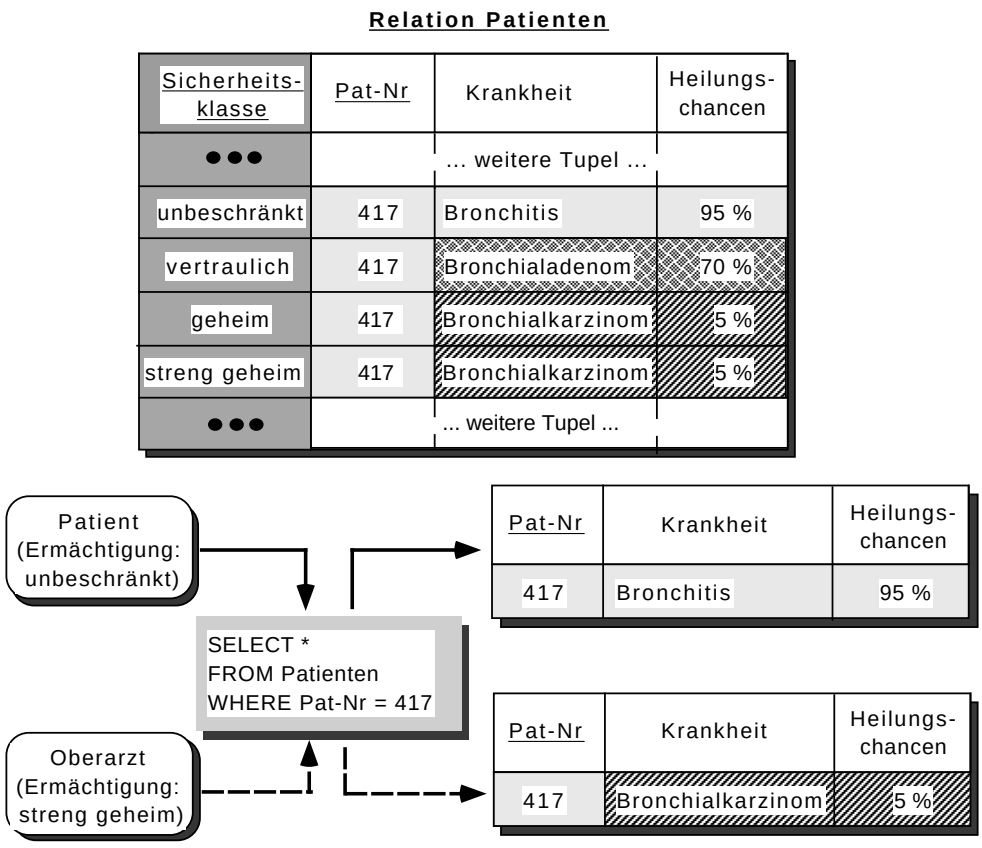


Abbildung 2.14: Polyinstanziierung in einer relationalen Datenbank

Ermächtigung alle Sicherheitsklassen annehmen kann (vergleiche Abschnitt 2.3.1).
Eine polyinstanziierte Datenbank erlaubt somit jeden Zugriff, gibt aber unter Umständen falsche Ergebnisse (*cover stories*) zurück.

2.4 Objektorientierte Zugriffskontrollmodelle

Im Zuge ständig wachsender Anforderungen an Informatikapplikationen hat sich in den vergangenen Jahren die eingeschränkte semantische Ausdrucksmächtigkeit traditioneller Systeme bemerkbar gemacht [DHP89]. Der gegenwärtige Trend tendiert daher zum Einsatz objektorientierter Systeme, da diese reichhaltigere Modellierungskonzepte unterstützen. Die Charakteristika objektorientierter Systeme, wie Vererbungshierarchien, Versionsmanagement und komplexe Objekte, erfordern jedoch neue Sicherheitsanforderungen, die von traditionellen Autorisierungssystemen nicht befriedigt werden können [BS94a]. Ebenso wie objektorientierte Datenbanken und Programmiersprachen die Modellierungsfähigkeiten gegenüber relationalen Datenbanksystemen und klassischen Programmiersprachen erweitern, bilden objektorientierte Zugriffskontrollmodelle in analoger Weise eine Obermenge zu herkömmlichen *Record*-basierten Zugriffskontrollsystemen [DHP89]. Der folgende Abschnitt arbeitet die Potentiale dieser neuen Systeme heraus. Dabei beschränkt sich die Betrachtung auf benutzerbe-

stimmbare objektorientierte Zugriffskontrolle. Für regelbasierte Ansätze sei zum Beispiel auf [Mor91, Thu91, BCCGY94, BJS94] verwiesen.

2.4.1 Konzepte objektorientierter Zugriffskontrolle

Objektorientierte Zugriffskontrollmodelle stellen eine signifikante Erweiterung zu den real existierenden diskreten Zugriffsmodellen dar, indem sie unter anderem die Modellierung komplexer Objekte und Versionskontrolle sowie den Schutz typspezifischer Funktionen ermöglichen. Viele dieser Modelle basieren auf den folgenden drei Konzepten, die im weiteren näher erläutert werden:

1. explizite und implizite Autorisierung
2. positive und negative Autorisierung
3. starke und schwache Autorisierung

Zur Veranschaulichung dienen dabei Beispiele, die auf der graphischen Modellierungsmethode OMT [RBP$^+$91] basieren. Diese wird im Anhang B kurz vorgestellt.

2.4.1.1 Explizite und implizite Autorisierung

Objekte und Vorgänge der realen Welt können multilaterale Beziehungen zueinander besitzen: Ein Objekt kann aus mehreren Teilobjekten bestehen; ein Bearbeitungsvorgang kann sich in mehrere sequentielle oder parallele Teilschritte untergliedern. Das Konzept der impliziten Autorisierung berücksichtigt diese Abhängigkeiten: Eine Autorisierung, die für ein Subjekt den Zugriff auf ein bestimmtes Objekt erlaubt, kann andere Autorisierungen zur Folge haben. Da eine Autorisierung aus drei Komponenten (Subjekt, Objekt, Zugriffsmodus) besteht, können aus einer explizit spezifizierten Autorisierung unter Umständen weitere Autorisierungen entlang jeder dieser drei Komponenten abgeleitet werden. Ein Recht, das eine Gruppe von Benutzern für den schreibenden Zugriff auf eine Gruppe von Objekten autorisiert, impliziert zum Beispiel folgende weitere Autorisierungen:

1. Der schreibende Zugriff auf eine Gruppe von Objekten ist neben der ganzen Gruppe von Benutzern auch jedem einzelnen Benutzer gestattet (Implikation entlang der Subjekt-Domäne).
2. Die Autorisierung für einen schreibenden Zugriff auf eine Gruppe von Objekten impliziert auch die Autorisierung für einen lesenden Zugriff auf diese Gruppe (Implikation entlang der Zugriffsmodus-Domäne).
3. Ist einer Gruppe von Benutzern der schreibende Zugriff auf eine Gruppe von Objekten erlaubt, so kann diese auch auf jedes einzelne Objekt schreibend zugreifen (Implikation entlang der Objekt-Domäne).

Eine **explizite Autorisierung** ist ein Recht, das tatsächlich erteilt wird; eine **implizite Autorisierung** dagegen ist ein Recht, das aus einer expliziten Autorisierung abgeleitet werden kann,

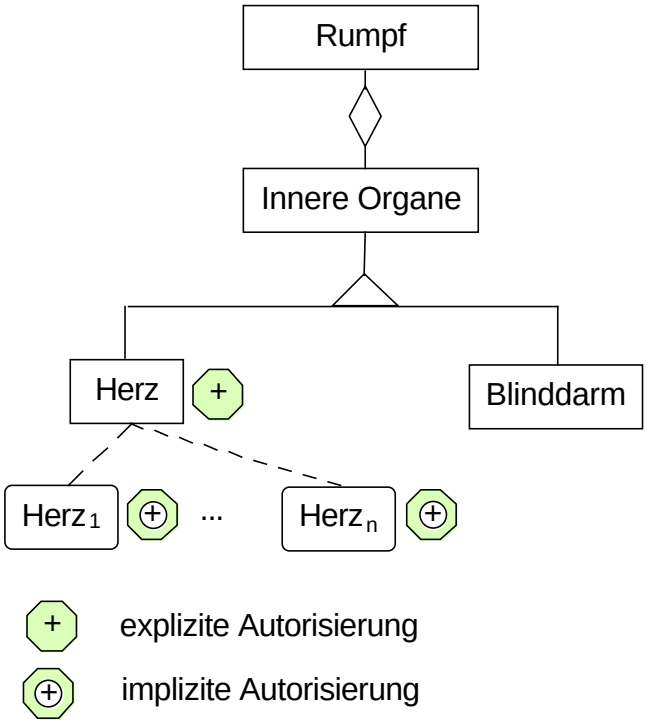


Abbildung 2.15: Explizite und implizite Autorisierung

beziehungsweise von dieser vererbt wird.

Zur Veranschaulichung dient Abbildung 2.15, in der die objektorientierte Modellierung der Anatomie des menschlichen Körpers (vergleiche Abbildung B.1) ausschnittsweise dargestellt ist. Die Klassen *Blinddarm* und *Herz* sind Spezialisierungen der Klasse *Innere Organe.* Die Klasse *Herz* umfaßt n Instanzen, die die Extension dieser Klasse bilden. Wird unterstellt, daß in der Abbildung das zu autorisierende Subjekt und der Zugriffsmodus fest vorgegeben sind, so führt eine explizite positive Autorisierung für die Klasse *Herz* dazu, daß nun auch alle Instanzen dieser Klasse implizit positiv autorisiert sind, da sie die Eigenschaften ihrer Klasse erben. Der Vorteil gegenüber anderen existierenden Modellen besteht darin, daß nun nicht mehr jede Autorisierung einzeln modelliert und gespeichert werden muß (im Beispiel: für jede Instanz eine Autorisierung), da sich jede implizite Autorisierung aus einer minimalen Menge explizit modellierter (und gespeicherter) Autorisierungen inferieren läßt. Diese Reduktion des Modellierungs- und Speicherungsaufwandes muß jedoch sorgfältig gegen den zusätzlichen zeitlichen Aufwand gewichtet werden, der durch die Auswertung einer konkreten Sicherheitsanfrage an das System entsteht. Überlegungen hierzu finden sich in Abschnitt 5.1.

2.4.1.2 Positive und negative Autorisierung

Zugriffskontrollsysteme, die auf traditionellen Autorisierungskonzepten beruhen, unterstellen, daß Subjekten der Zugriff auf ein Objekt solange verwehrt bleibt, bis diese explizit autorisiert werden. Ein Zugriffsverbot für ein bestimmtes Subjekt wird somit durch eine fehlende Autorisierung modelliert. Durch diesen Ansatz kann nicht verhindert werden, daß das Subjekt zu

einem späteren Zeitpunkt "versehentlich", zum Beispiel durch eine implizite Autorisierung, eine Zugriffsberechtigung erlangt [BS94a]. Dieses Problem wird durch die Differenzierung von Zugriffsrechten in positive und negative Autorisierungen gelöst: Eine **positive Autorisierung** ist die explizite Erteilung einer Zugriffserlaubnis. Diese Sichtweise entspricht der existierender Systeme. Eine **negative Autorisierung** ist dagegen ein explizites Zugriffsverbot. Diese Unterteilung in positive und negative Zugriffsrechte führt zu folgenden beiden Problemen:

- Für ein Subjekt, das durch eine Mehrfachvererbung eine implizite positive Autorisierung und eine implizite negative Autorisierung für den Zugriff auf ein Objekt besitzt, kann nicht entschieden werden, ob der Zugriff auf dieses Objekt erlaubt ist oder nicht. Folgende Möglichkeiten zur Lösung dieses Problems sind denkbar:
  - Die Herbeiführung inkonsistenter Zustände, bei der es zu widersprüchlichen Autorisierungen für den Zugriff eines Subjekts auf ein Objekt kommt, ist grundsätzlich verboten (*avoidance*).
  - Inkonsistente Zustände dürfen spezifiziert werden. Für die Evaluierung einer Sicherheitsanfrage, die an das System gestellt wird, existiert ein Konfliktmanager, der in Fällen widersprüchlicher Autorisierung eine eindeutige Entscheidung trifft (*detection*).

  Weitere Überlegungen zu diesem Aspekt finden sich in Abschnitt 4.1.11.

- Eine positive Autorisierung bedeutet, daß einem Subjekt der Zugriff auf ein Objekt erlaubt ist; eine negative Autorisierung verbietet diesen Zugriff. Ungeklärt ist, wie im Falle einer fehlenden Autorisierung zu entscheiden ist. Ein **geschlossenes System** erlaubt nur solche Zugriffe, die positiv autorisiert worden sind; ein **offenes System** erlaubt alle Zugriffe, für die keine negative Autorisierung existiert [CFMS95]. Objektorientierte Modelle behandeln diese Fragestellung meist wie "klassische" Systeme und sehen eine fehlende Autorisierung als Zugriffsverbot an (geschlossenes System). Eine differenziertere Betrachtungsweise dieses Problems wird in Abschnitt **??** unter dem Stichwort "Autorisierungsstrategie" diskutiert.

Die Einführung positiver und negativer Autorisierung hat isoliert betrachtet zunächst keinen positiven Einfluß auf die Modellierungsmächtigkeit des Zugriffskontrollsystems; im Gegenteil, sie ruft die soeben diskutierten Probleme hervor. Wesentlich für die Beurteilung, ob sich durch dieses Konzept die Modellierungsflexibilität erhöht, ist jedoch die synoptische Betrachtung aller Kontrollabstraktionen, da sich erst durch das Zusammenwirken der drei Basiskonzepte Synergieeffekte für die Modellierungsmöglichkeiten ergeben.

2.4.1.3 Starke und schwache Autorisierung

Das Konzept für die explizite und implizite Autorisierung stellt ein wirkungsvolles Modellierungsinstrument bereit, das die Spezifizierung genereller Autorisierungsabhängigkeiten erlaubt. Die explizite Autorisierung für den Zugriff auf ein komplexes Objekt impliziert beispielsweise die implizite Autorisierung für den Zugriff auf alle seine Teilkomponenten. Nicht immer ist jedoch eine solche Modellierung erwünscht. Es kann zum Beispiel -abweichend von der Regel- einige wenige Teilkomponenten geben, auf die nicht zugegriffen werden soll. Für die Modellierung solcher Ausnahmen auf impliziten Autorisierungen stehen bisher keine Werkzeuge zur

Verfügung. Dieser Mangel ist der Ausgangspunkt für die Einführung von starken und schwachen Autorisierungen.
Autorisierungen realer Systeme sind **stark**, da implizite Autorisierungen nicht überschrieben werden können. **Schwache Autorisierungen** dagegen ermöglichen die Überschreibung impliziter Autorisierungen und schaffen so die Voraussetzungen, um Ausnahmen zu modellieren. Eine implizite Autorisierung kann durch zwei Arten von Rechten überschrieben werden:

1. **Starke Rechte**. Das Überschreiben einer schwachen impliziten Autorisierung durch ein starkes explizites Recht stellt sicher, daß für alle weiteren, transitiv erreichbaren impliziten Autorisierungen kein Überschreiben mehr möglich ist (implizite starke Autorisierungen).

2. **Schwache Rechte**. Wird eine implizite schwache Autorisierung durch eine explizite schwache Autorisierung überschrieben, so können auch deren implizite Autorisierungen wiederum überschrieben werden. Auf diese Weise ist es zum Beispiel möglich "Ausnahmen von der Ausnahme" zu modellieren.

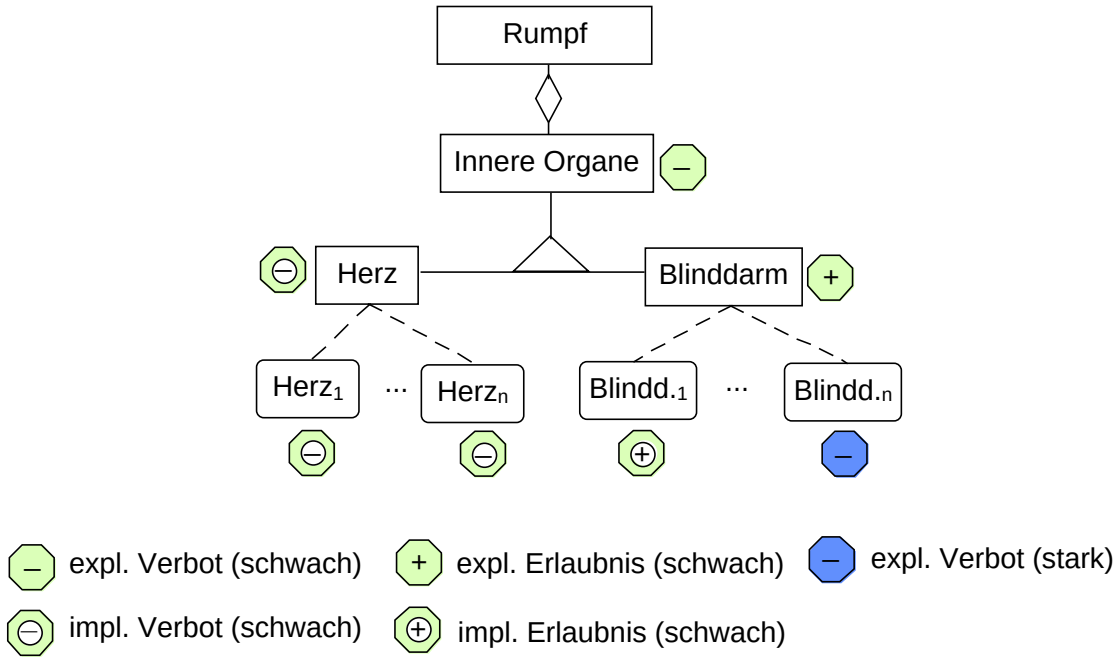


Abbildung 2.16: Starke und schwache Autorisierung

Diese Modellierungsmöglichkeiten sind Gegenstand der Abbildung 2.16. Aus Gründen der Übersichtlichkeit wird wiederum unterstellt, daß das Subjekt und der Zugriffsmodus fest vorgegeben sind und daß Rechte daher nur entlang der Objekthierarchie abgeleitet werden. Zum besseren Verständnis dieses Beispiels sei angenommen, das Subjekt repräsentiere einen Assistenzarzt und der Zugriffsmodus sei die Abbildung einer Funktion "operieren". Die Tatsache, daß ein Assistenzarzt keine generelle Erlaubnis haben soll, eine Operation auf einem beliebigen Organ durchzuführen, wird durch eine negative explizite Autorisierung auf der Klasse *Innere Organe* modelliert. Da es möglich sein soll, Ausnahmen von dieser generellen Regel

zu spezifizieren, wird nur ein schwaches Verbot erteilt. Die Klasse *Herz* ist eine Subklasse der Klasse *Innere Organe* und somit folgt insbesondere, daß ein implizites schwaches Verbot für die Durchführung einer Herzoperation gilt. Die Klasse *Blinddarm* ist ebenfalls eine Subklasse der Klasse Innere Organe und erbt deren Autorisierungen. Um die prinzipielle Erlaubnis zur Durchführung einer Blinddarmoperation zu modellieren, wird das implizite schwache Verbot, das von der Klasse *Innere Organe* ererbt wurde, durch eine positive explizite Autorisierung überschrieben. Um ebenfalls weitere Ausnahmen von dieser Ausnahme zulassen zu können, wird abermals ein schwaches Recht spezifiziert. Die explizite schwache Erlaubnis auf der Klasse *Blinddarm* impliziert diese Berechtigung ebenfalls für deren Instanzen. Sind bei der Durchführung einer konkreten Blinddarmoperation (hier: am $Blinddarm_n$) Komplikationen zu erwarten, so kann die schwache implizite Autorisierung für diese Instanz durch ein explizites Verbot überschrieben werden. Soll von diesem Verbot keine weitere Ausnahme möglich sein, so wird es durch eine starke Autorisierung abgebildet.

2.4.2 Prioritätsbasierte Zugriffskontrolle

Der Mechanismus von starker und schwacher Autorisierung ermöglicht die Modellierung genereller Rechte und davon abweichender Ausnahmen. Diese Ausnahmen können wiederum dadurch beliebig geschachtelt werden, daß implizite schwache Autorisierungen durch explizite schwache Autorisierungen überschrieben werden. Generalisiert man diesen speziellen Modellierungsansatz durch ein allgemeines Konzept, so kann der Mechanismus starker und schwacher Autorisierung durch ein beliebiges Prioritätssystem ersetzt werden, sofern auf diesen Prioritäten eine (partielle oder totale) Ordnungsrelation definiert ist [Brü92, Brü93]. Ein Beispiel für ein solches Prioritätssystem bilden die natürlichen Zahlen in einem vorgegebenen Intervall. Die Konzepte der expliziten und impliziten Autorisierung sowie der positiven und negativen Autorisierung bleiben hierbei erhalten. Auch können durch die Modellierung weiterhin Konflikte entstehen, wenn positive und negative Rechte gleicher Priorität für dasselbe Objekt spezifiziert werden.

2.4.3 Konditionale Zugriffskontrolle

Die in Abschnitt 2.4.1 eingeführten Modellierungskonzepte unterstützen die Implementierung allgemeingültiger (kontextunabhängiger) Autorisierungen, das heißt bei der Evaluierung einer Zugriffsanfrage braucht nicht auf das Objekt selbst zugegriffen zu werden. Eine entscheidende Erweiterung besteht nun darin, daß die Zugriffsbefugnis von der Gültigkeit bestimmter Bedingungen (*usage conditions*) abhängig gemacht werden kann. Diese Bedingungen können durch Prädikate ausgedrückt werden, deren Gültigkeit über die Zugriffsberechtigung entscheidet. Beispiele sind zeitliche, örtliche, wert- und größenabhängige Beschränkungen [SL93]. Die Einführung einer rein kontextabhängigen Zugriffskontrolle stellt keine Einschränkung hinsichtlich der Modellierungsfreiheiten im Referenzmodell dar, da mit Hilfe tautologischer Prädikate (Prädikate, die für alle Instanziierungen stets wahr sind) auch eine kontextunabhängige Zugriffskontrolle emuliert werden kann. Umgekehrt gilt jedoch, daß kontextabhängige Zugriffskontrolle eine Erhöhung der Modellierungsflexibilität bewirkt [BW94]. Konditionale Zugriffskontrolle führt jedoch gleichzeitig zu folgenden Nachteilen: Zum einen kann die Überprüfung der Gültigkeit von Prädikaten nur zur Laufzeit erfolgen [FGS91, Hof77], was neben der Evaluierung einer Zugriffsanfrage einen zusätzlichen Performanzverlust bedingt. Zum anderen können

sich explizit und implizit gültige Prädikate für ein Objekt ganz oder teilweise widersprechen. Zur Lösung dieses Problems bieten sich drei Sicherheitspolitiken an [FGS91]:

- **Vereinigung**. Alle explizit und implizit gültigen Prädikate werden disjunktiv verknüpft. Der Zugriff auf ein Objekt wird erlaubt, wenn mindestens eines dieser Prädikate erfüllt wird. Eine solche Politik wird auch in Systemen betrieben, in denen Möglichkeiten für die koexistente Modellierung kontextabhängiger und kontextunabhängiger Autorisierungen zur Verfügung stehen [BW94, BS94b].

- **Überschreibung**. Ein explizit gültiges Prädikat überschreibt stets ein implizites Prädikat. Dieser Mechanismus stellt ein Analogon zum Überschreiben einer schwachen Autorisierung durch eine (ebenfalls) schwache Autorisierung im Referenzmodell dar.

- **Durchschnitt**. Alle explizit und implizit gültigen Prädikate werden konjunktiv verknüpft. Der Zugriff auf ein Objekt wird nur dann garantiert, wenn alle Prädikate erfüllt sind.

Eine auf Konjunktion beruhende Politik ist somit am restriktivsten, während die Menge autorisierter Subjekte bei Anwendung einer Vereinigungspolitik am größten ist.

2.4.4 Gegenläufige Vererbung

Ein großes Potential objektorientierter Zugriffskontrollmodelle liegt darin, daß sich ein explizit erteiltes Recht entlang aller drei Domänen (Subjekte, Objekte, Zugriffsmodi) weitervererbt. Das klassische objektorientierte Paradigma geht dabei von **Abwärtsvererbung** (von der Wurzel der Hierarchie zu den Blättern) aus. Besonderes Augenmerk ist jedoch auf die Vergabe positiver und negativer Autorisierungen zu richten. Im folgenden soll gezeigt werden, daß die Vererbung positiver und negativer Rechte nicht immer gleichgerichtet und abwärts erfolgt, sondern daß sie durchaus auch gegengerichtet und aufwärts erfolgen kann.
Die isolierte Analyse der drei Domänen führt nach [RBKW91, Brü92] zu folgendem Ergebnis:

1. Eine positive Autorisierung, die für eine bestimmte Subjektklasse erteilt wurde, gilt auch für deren Subklassen; eine negative Autorisierung gilt dagegen auch für alle Superklassen. Es handelt sich somit um eine gegengerichtete Vererbung: Für positive Autorisierungen gilt Abwärtsvererbung, während sich negative Autorisierungen aufwärts vererben. Eine Erlaubnis, die den Krankenschwestern eines Krankenhauses erteilt wurde, soll somit auch für die Ärzte dieses Krankenhauses gelten; andererseits soll ein Verbot, das den Ärzten ausgesprochen wurde, auch für die Krankenschwestern gelten.

2. Ein positives Recht für eine bestimmte Autorisierungstypklasse vererbt sich an alle ihre Subklassen weiter; ein negatives Recht dagegen vererbt sich an alle Superklassen weiter. Hier gilt also ebenfalls eine gegengerichtete Vererbung, bei der positive Autorisierungen abwärts vererbt werden und sich negative Autorisierungen durch Aufwärtsvererbungen propagieren. Wer die Berechtigung hat, eine Operation durchzuführen, soll auch ermächtigt werden, eine Diagnose zu stellen; andererseits gilt, daß das explizite Verbot zum Stellen einer Diagnose auch das Verbot zur Durchführung einer Operation nach sich zieht.

3. Entlang der Objekthierarchie vererben sich positive und negative Autorisierungen gleichgerichtet abwärts. Die Erlaubnis (beziehungsweise das Verbot) zur Durchführung von Operationen am Thorax (Brustkorb) impliziert die Erlaubnis (beziehungsweise das Verbot), Operationen an den darin enthaltenen Organen durchzuführen. Akribischere Analysen [RBKW91] kommen jedoch zu der Erkenntnis, daß sich positive und negative Rechte zwar stets gleichgerichtet vererben, es aber durchaus vom konkret gewählten Autorisierungstyp abhängen kann, ob die Vererbung aufwärts, abwärts oder gegebenenfalls überhaupt nicht erfolgt: Eine Schreibberechtigung für den modifizierenden Zugriff auf eine Klasse beispielsweise, impliziert eine Schreibberechtigung für alle Instanzen dieser Klasse (Abwärtsvererbung). Die Berechtigung, die Definition einer Klasse lesen zu dürfen, impliziert das Recht, die Definition der sie erzeugenden Klasse lesen zu dürfen (Aufwärtsvererbung). Das Recht, eine beliebige Klasse anlegen zu dürfen, impliziert nicht automatisch das Recht, beliebige Instanzen erzeugen zu können. Vielmehr dürfen Instanzen nur für solche Klassen erzeugt werden, die auch von dem autorisierten Subjekt angelegt wurden, respektive für das es eine andere Autorisierung besitzt (kein allgemeiner Vererbungsmechanismus).

2.4.5 Synoptische Betrachtung konkreter Modelle

Die bisher vorgestellten Konzepte repräsentieren die Quintessenz aus mehreren objektorientierten Zugriffskontrollmodellen. Die nachfolgende Betrachtung untersucht konkrete Modelle und konzentriert sich dabei in einer vergleichenden Zusammenfassung auf vier der bekannteren Ansätze auf diesem Forschungsgebiet. Sie basieren auf Arbeiten von Bertino ([BW94, RBKW91, BOS94]), Brüggemann ([Brü93, Brü92]), Fernández ([FGS89, FWF94, FLPG94, FGS91]) und Kelter ([Kel90, Kel91]).

2.4.5.1 Das Modell von Bertino et al.

Das Modell von Bertino basiert auf den oben beschriebenen Konzepten der starken/schwachen, expliziten/impliziten und positiven/negativen Autorisierung. Diese bieten einen flexiblen Rahmen zur Modellierung genereller Rechte und beliebig tief geschachtelter Ausnahmen zu diesen Rechten. Da positive und negative Rechte gleichberechtigt auftreten, kann es zu Konflikten durch widersprüchliche Autorisierungen kommen, die es zu lösen gilt.
Es wird eine feste Anzahl von Zugriffstypen (lesen, schreiben, erzeugen, Definition lesen) mit festen Implikationen unterstellt. Positive Zugriffsrechte vererben sich entlang der Subjekt- und Zugriffstyphierarchie stets abwärts, negative Rechte dagegen stets aufwärts. Entlang der Objekthierarchie vererben sich positive und negative Rechte in die gleiche Richtung. Diese ist abhängig vom betrachteten Zugriffstyp: Bei einer Autorisierung für lesenden oder schreibenden Zugriff erfolgt die Vererbung abwärts, das Recht zum Lesen einer Definition erfolgt aufwärts, und die Autorisierung, ein neues Objekt anlegen zu dürfen, wird nicht vererbt.

2.4.5.2 Das Modell von Brüggemann

Das Modell von Brüggemann folgt fast uneingeschränkt dem Modell von Bertino. Die Möglichkeit zur Modellierung starker und schwacher Rechte ist jedoch durch einen allgemeineren **prioritätsorientierten** Ansatz ersetzt worden. Es können Prioritäten beliebiger Anzahl und

beliebiger Art definiert werden, sofern man eine lineare Ordnung für sie angeben kann. Werden mehrere Prioritätssysteme definiert, so muß für deren Gültigkeitsbereiche eine partielle Ordnung definiert werden. Ergeben sich bei der Evaluierung einer Zugriffsanfrage widersprüchliche Rechte, so hat das Recht mit der höchsten Priorität Gültigkeit. Das Problem auftretender Konflikte kann mit diesem Ansatz jedoch nicht behoben werden, da gegensätzliche Rechte mit einer gleichen Priorität ebenfalls einer Lösungsstrategie bedürfen.
Ein zweiter untergeordneter Unterschied besteht darin, daß sich in diesem Modell positive und negative Rechte unabhängig vom gewählten Zugriffstyp entlang der Objekthierarchie stets abwärts vererben.

2.4.5.3 Das Modell von Fernández et al.

Das Modell von Fernández bietet für das Auftreten von Konflikten eine feste Lösungsstrategie des Systems an. Diese sieht vor, daß eine negative Autorisierung stets eine positive Autorisierung dominiert (*denials take precedence*). Diese Sichtweise führt jedoch zu einer Einschränkung der Modellierungsfähigkeit, da es nun nicht mehr möglich ist, ein generelles Verbot durch eine spezielle Erlaubnis zu überschreiben. Diese Erkenntnis führt zu einer Modifikation der systeminternen Strategie [FGS91]. Unter Miteinbeziehung expliziter und impliziter Autorisierungen ergibt sich folgende Hierarchie für die Evaluation: Ein explizites Verbot überschreibt eine explizite Erlaubnis. Diese wiederum überschreibt ein implizites Verbot, und letzteres überschreibt eine implizite Erlaubnis. Die Festlegung einer festen Strategie erübrigt ein weiteres Konfliktmanagement, schränkt jedoch die Fähigkeiten des Modellierers, eigene Strategien zu spezifizieren, ein. Darüber hinaus ist es nicht möglich, das Überschreiben eines Rechtes in einer bestimmten Teilhierarchie zu verhindern, wie dies mithilfe eines starken Rechtes möglich ist.
Die Zugriffsbefugnis kann darüber hinaus von einem Prädikat abhängig gemacht werden (vergleiche Abschnitt 2.4.3). Verfügt ein Autorisierungssystem zusätzlich über prädikatbasierte Rechte, so wird eine Sicherheitspolitik, bei der ein implizites Prädikat durch ein explizites Prädikat überschrieben wird, als adäquat angesehen [FGS91].

2.4.5.4 Das Modell von Kelter

Die Forschungsarbeiten von Kelter haben zwei Schwerpunkte, zum einen die Analyse komplexer Objekte in objektorientierten Datenbanksystemen und zum anderen Überlegungen zur gruppenorientierten Bildung von Subjekthierarchien.
Wird ein explizites Zugriffsrecht auf ein komplexes Objekt erteilt, so muß dieses auch implizit für alle Komponenten dieses Objektes gelten. Ändert sich die Menge und Art der Komponenten, aus denen sich das komplexe Objekt zusammensetzt, so soll die Autorisierung für das Gesamtobjekt seine Gültigkeit behalten. Besondere Probleme treten dadurch auf, daß Objektkomponenten unter Umständen nicht exklusiv sind, sondern Bestandteil mehrerer komplexer Objekte sein können, und sich daher widersprüchliche implizite Autorisierungen ableiten lassen. Die Überprüfung einer Zugriffsberechtigung auf einem komplexen Objekt geschieht daher wie folgt: Ein Subjekt ist genau dann zugriffsberechtigt, wenn es für den Zugriff auf das komplexe Objekt explizit autorisiert wurde oder wenn keine Autorisierung für das komplexe Objekt definiert wurde und sich für keine der Komponenten des Objektes eine implizite negative Autorisierung ableiten läßt. Die Art der Zugriffe ist fest vorgegeben und beschränkt sich auf genau neun Zugriffstypen.

Zusätzlich zu der Beschreibung in Abschnitt 2.5 gelten folgende Neuerungen: Alle Subjekte (zu ihnen zählen explizit auch ausführbare Programme) sind direktes oder indirektes Mitglied einer vordefinierten Gruppe *WORLD*. Diese Gruppe bildet den Wurzelknoten einer Subjekthierarchie. Will ein Subjekt die Rechte einer Gruppe, in der es Mitglied ist, ausüben, so muß diese Gruppe aktiviert werden. Es können jedoch nicht willkürlich alle Gruppen aktiviert werden, in der ein Subjekt Mitglied ist, da sich deren Rechte unter Umständen wechselseitig ausschließen können. Kann andererseits nur eine Gruppe zur Zeit aktiviert werden, so kann unter Umständen nicht von anderen ("harmlosen") Rechten Gebrauch gemacht werden, die allen anderen Gruppen gemein sind. Es wird daher vorgeschlagen, eine technische Kontrollinstanz einzuführen, die die Aktivierung mehrerer Gruppen kontrolliert.

2.5 Gruppen, Rollen und Aufgaben

Während die letzten Abschnitte die Administration von Zugriffsrechten auf der Basis einzelner Subjekte in den Vordergrund gestellt haben, liegt die Betonung in diesem Abschnitt auf der Strukturierung von Subjekten. Hierbei haben sich in der Praxis zwei Ansätze durchgesetzt: Gruppen und Rollen.
Eine **Gruppe** ist eine benannte Kollektion von Subjekten und somit gleichzeitig ein Akronym zur Referenzierung dieser Subjektmenge. Gruppen können Rechte zugewiesen werden. Eine **Rolle** hingegen ist eine benannte Menge von Rechten, der Subjekte zugewiesen werden können. Da Rollen und Gruppen enge Substitute darstellen, gelten die folgenden Eigenschaften von Gruppen sinngemäß auch für Rollen:
Gruppen können erzeugt oder gelöscht werden, und autorisierte Benutzer können ihnen Mitglieder hinzufügen oder diese löschen [GLL89]. Sie können konzeptuell unendlich viele Mitglieder enthalten [Mül92]. **Mitglieder** einer Gruppe können Subjekte oder andere Gruppen sein. Ein Subjekt oder eine Gruppe kann Mitglied mehrerer Gruppen sein. Durch azyklische Verschachtelung von Gruppen können komplexe Subjekthierarchien konstruiert werden. Unter einem **direkten** Mitglied versteht man ein Subjekt, das explizit zum Mitglied einer Gruppe gemacht wird. Ein **indirektes** Mitglied einer Gruppe A ist ein direktes oder indirektes Mitglied einer Gruppe B, die direktes Mitglied der Gruppe A ist. Eine Gruppe enthält somit direkte und indirekte Mitglieder [GLL89].
Ist ein Subjekt Mitglied mehrerer Gruppen, so ist eine Gruppenpolitik zu spezifizieren, die festlegt, welche Rechte dieses Subjekt benutzen darf. Folgende Modellierungsmöglichkeiten sind denkbar [Lun89]:

- Das Subjekt darf von seinen eigenen Rechten **und** den Rechten aller seiner Gruppen Gebrauch machen.
- Das Subjekt darf seine eigenen Rechte **oder** die Rechte einer seiner Gruppen benutzen.
- Dem Subjekt stehen nur die Rechte einer seiner Gruppen zu.
- Es wird eine andere Politik spezifiziert.

Sind Gruppen hierarchisch verschachtelt, so ist bei der Modellierung folgendes zu beachten: Da Supergruppen allgemeiner als Subgruppen sind und damit weniger Rechte als diese besitzen, müssen bei der konkreten Abbildung einer Hierarchie die Subjekte mit den meisten Rechten

Mitglieder der untersten Subgruppe sein. Eine Unternehmenshierarchie von Vorgesetzten beispielsweise hat somit die Gruppe des gesamten Personals als Wurzelobjekt und die Gruppe der Manager als unterste Subgruppe [Bir94], wird also quasi "auf den Kopf" gestellt (*upside down modelling*). Eine solche Sichtweise, bei der eine Gruppe mit einer Menge von Rechten korrespondiert, wird als Rechteverwaltungsparadigma (*right package paradigm*) bezeichnet [Kel90].

Gruppen (und Rollen) können auch unter dem sogenannten Aufgabenparadigma (*task paradigm*) betrachtet werden. In diesem Fall korrespondiert eine Gruppe mit einer Aufgabe, die es zu lösen gilt. Subgruppen entsprechen dabei in analoger Weise Teilaufgaben. Jede Gruppe erhält dabei genau die Rechte, die zur Bearbeitung ihrer (Teil-)Aufgabe erforderlich sind (*least privilege principle*). Aus diesen Überlegungen heraus folgt, daß eine Supergruppe stets mehr Berechtigungen als eine Subgruppe besitzt, weil zum Beispiel für die Lösung einer ganzen Aufgabe mehr Rechte als für die Lösung einer Teilaufgabe erforderlich sind [Kel90].

2.6 Eigenschaften und Grenzen der Zugriffskontrollmodellklassen

Die folgende Abschlußbetrachtung vergleicht die vorgestellten Zugriffskontrollmodelle noch einmal in einer zusammengefaßten Darstellung.

Benutzerbestimmbare Zugriffskontrolle: Dieser Ansatz bietet einen flexiblen Rahmen für die Implementierung eines Zugriffskontrollsystems. Er ist jedoch nicht in der Lage, neuere, semantisch reichhaltigere Konzepte wie Objektorientierung adäquat zu reflektieren. Die Besitzer eines Objektes entscheiden nach ihrem eigenen Ermessen, welche Subjekte Zugriff auf ihre Objekte haben sollen. Eine solche bedarfsgerechte Vergabe von Zugriffsrechten schließt auch die Möglichkeit ein, das Recht zur Weitergabe eines empfangenen Rechtes zu erteilen. Diese Möglichkeit verursacht jedoch einen erhöhten administrativen Aufwand für die Verwaltung von Propagierungshistorien. Darüber hinaus besteht die Gefahr, daß Subjekte eine fremde Identität annehmen können und so an Informationen gelangen, die nicht für sie bestimmt sind. Diese Schwachstellen werden insbesondere von "Computeranomalien" wie Viren und Trojanischen Pferden ausgenutzt.

Regelbasierte Zugriffskontrolle: Soll eine spezielle Sicherheitspolitik durch das System erzwungen werden, so bietet sich ein regelbasiertes Zugriffskontrollsystem an. Alle Zugriffsrechte werden von einer zentralen Instanz administriert und bei Bedarf modifiziert. Die Verwaltung von Propagierungshistorien entfällt daher, weil an Benutzer erteilte Rechte nicht weiterdelegiert werden können. Durch Einsatz eines mehrstufigen Zugriffskontrollmodells wird dabei gleichzeitig der Informationsfluß zwischen verschiedenen Sicherheitsklassen unterbunden. Mehrschichtige Zugriffskontrollmodelle gelten als strikt und zuverlässig; ihnen fehlt jedoch die Flexibilität, da zur Gewährleistung der Sicherheit des Gesamtsystems große Teile der Sicherheitskomponente fest in das System eingebaut werden müssen. Ihr Anwendungsgebiet konzentriert sich daher im wesentlich auf geschlossene Systemumgebungen, nicht zuletzt weil mehrstufige Zugriffskontrollmodelle auch erhöhte Anforderungen an die zugrundeliegende Hard- und Software stellen.

Diskrete objektorientierte Zugriffskontrolle: Diese Modellklasse stellt eine Erweiterung zu herkömmlicher benutzerbestimmbarer Zugriffskontrolle dar. Die Stärken der objektori-

entierten Zugriffskontrolle liegen in erster Linie in der Art der Modellierung. Abstraktionen zur Modellierungsvereinfachung, wie die implizite Ableitung (Vererbung) oder das Überschreiben von Rechten zur Definition spezieller Ausnahmen von generellen Regeln, tragen zu einer deutlichen Reduktion des Modellierungsaufwandes bei. Der Einsatzbereich objektorientierter Zugriffskontrolle erstreckt sich daher auch auf neuere Konzepte wie den Schutz komplexer Objekte oder Versionskontrolle. Zusammenfassend bietet der objektorientierte Ansatz zur Modellierung von Zugriffsrechten nach [JKP93] folgende Vorteile:

- Natürliche Modellierbarkeit der Beziehungen und Rechte, die zwischen Subjekten und Objekten bestehen.
- Unterstützung für die Bildung von Objektklassen und die Modellierung generischer Regeln.
- Vererbung und Propagierung von Rechten.
- Leichte Abbildbarkeit auf objektorientierte Benutzeroberflächen.
- Unterstützung bei der Bildung von Rollen und Gruppen.
- Einfache Integration bestehender Sicherheitsmechanismen.

Zusammenfassend läßt sich feststellen, daß die Einsatzbereiche der einzelnen Modellklassen stark von den zu schützenden Anwendungen abhängig sind. Die Tycoon-Umgebung, die auch für den Gebrauch in offenen verteilten Umgebungen konzipiert ist, kann prinzipiell durch diskrete Zugriffskontrollabstraktionen besser unterstützt werden. Da hierbei insbesondere die flexibleren Modellierungsmöglichkeiten objektorientierter Zugriffskontrollmodelle einem generischen bibliotheksorientierten Ansatz eher entgegenkommen, ist diesen Modellen für das konkrete Implementierungsvorhaben der Vorzug gegenüber klassischer benutzerbestimmbarer Zugriffskontrolle zu geben.

Kapitel 3

Die Systementwicklungsumgebung Tycoon

Die Entwicklung im Umfeld der Informationstechnologie hat zu tiefgreifenden Veränderungen geführt: Standen früher zentralisierte, homogene geschlossene Systeme im Vordergrund, prägen heute verteilte heterogene Systemumgebungen das Szenario. Dieser Wandel bedingt die multilaterale Interaktion zwischen unterschiedlichen Systemen (Offenheit). Sie ermöglicht nicht nur die Erschließung neuer Informationsquellen, sondern führt auch zu einer Erweiterung der Anwendungsfelder [Kaß94], die neben der Verwaltung traditioneller (tupelbasierter) Daten auch multimediale Daten wie Tonsequenzen und Graphiken miteinschließt [Mat93]. Aus der wachsenden Informationsfülle heraus ergibt sich die Notwendigkeit, Datenstrukturen zur Verwaltung von Massendaten zur Verfügung zu stellen. Diese Datenstrukturen müssen aufgrund der Heterogenität der gespeicherten Daten generisch sein. Darüber hinaus besteht der Wunsch, die unterschiedlichen angebotenen Dienste bedarfsgerecht, handhabbar und zuverlässig zu kombinieren. Diese veränderten Anforderungen motivieren die Einführung persistenter Objektsysteme. "Der Begriff *persistente Objektsysteme* bezeichnet eine Klasse von Softwaresystemen, die ihren Benutzern einen flexiblen, problemadäquaten und sicheren Umgang mit großen Mengen langlebiger Objekte unterschiedlichster Art ermöglichen" [Mat93, Seite 1]. Neben einer Unterstützung der "Programmierung im Großen" (*programming in the large*) durch Konzepte wie Modularisierung, Generalisierung und Parametrisierung stehen im Kontext persistenter Objektsysteme insbesondere Aspekte wie Langlebigkeit, Generizität und Offenheit im Vordergrund [MS93a, MS93b].
Dieses veränderte Anforderungsprofil kann jedoch von einer Vielzahl der heutigen Systeme und Programmiersprachen noch nicht befriedigt werden. Die Mehrzahl der heutigen Systeme basiert auf Sprachen der dritten Generation (prozeduralen Sprachen), wie COBOL, Pascal, C und FORTRAN. Kennzeichnend für diese Sprachen ist ihr monomorphes Typsystem, ihre Typstrenge und ihre linguistische Ausdrucksmächtigkeit. Eine sehr eingeschränkte Form von

Generizität gewährleisten lediglich primitive Datenstrukturen, wie Zeiger, Bitvektoren oder Zeichenketten (*strings*). Bei der Transformation einer semantisch reichhaltigen Datenstruktur in diese primitiven Datentypen geht jedoch fast die gesamte Typinformation verloren. Sprachen der dritten Generation sind daher nicht in der Lage, die geforderte Generizität typsicher abzubilden. Da die oben genannten primitiven Datentypen jedoch in allen Programmiersprachen enthalten sind, eignen sie sich für eine Interaktion mit anderen Systemplattformen und Diensterbringern wie Datenbanken und Visualisierungswerkzeugen. Diese Kommunikation auf niedriger Systemebene veranschaulicht Abbildung 3.1[Mat93]. Datenbankobjekte sind hierbei durch ein D, Programmobjekte durch ein P und Bildschirmobjekte durch ein B repräsentiert.

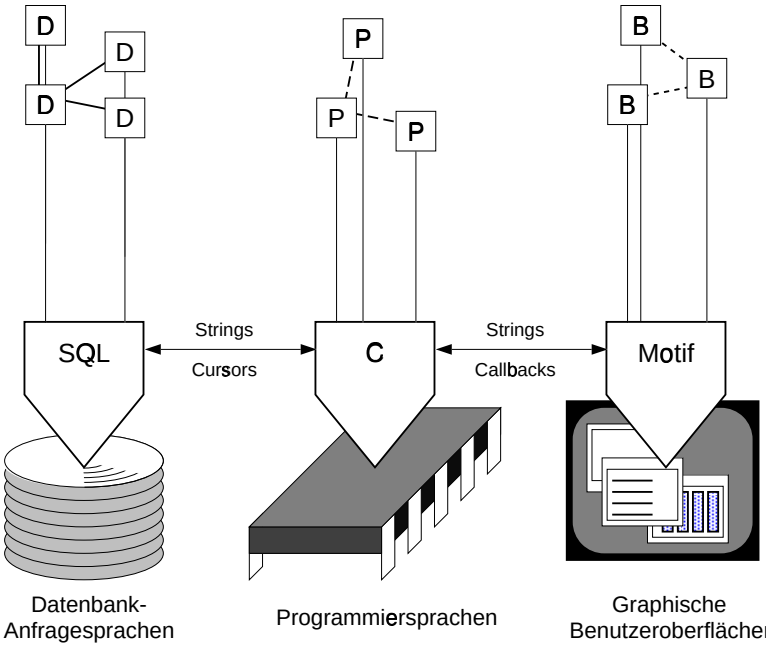


Abbildung 3.1: Schmale Dienstschnittstellen auf niedrigem Abstraktionsniveau

Die bestehenden Einschränkungen prozeduraler Sprachen haben zur Entwicklung von Sprachen der vierten Generation, wie PROLOG oder der Datenbankanfragesprache SQL, geführt. Sprachen der vierten Generation, auch deklarative oder deskriptive Programmiersprachen genannt, vermögen die Anforderungen datenintensiver Anforderungen besser zu erfüllen. Sie stellen dem Benutzer generische Benennungs-, Typisierungs- und Bindungsmechanismen zur Verfügung und ermöglichen so die Kommunikation auf einer logisch höheren (generischen) Abstraktionsebene [Mat93]. Diese generischen Abstraktionen sind jedoch fest in die Sprache integriert und können daher weder erweitert oder ausgetauscht noch selbst definiert werden.
Deklarative Sprachen sind daher fast ausschließlich Bestandteil proprietärer Systeme. Diese bieten dem Anwender neben klassischer Datenbankfunktionalität, wie einem persistenten Objektspeicher, einem eingebauten Datenmodell, Mehrbenutzerbetrieb und Fehlererholungsmechanismen, auch eine interaktive Benutzerschnittstelle an. Trotz dieser signifikanten Verbesserungen gegenüber Sprachen der dritten Generation bleiben die folgenden Probleme bestehen:

1. Die tiefe Integration deskriptiver Sprachen in proprietäre Systeme führt zu einem Portie-

rungsproblem. Es ist insbesondere nicht - oder nur mit sehr hohem Aufwand - möglich, eine Anwendung, die in einer Sprache der vierten Generation geschrieben ist, auf ein anderes System zu übertragen. Dies ist auch auf eine mangelnde Standardisierung deskriptiver Sprachen zurückzuführen.

2. Die fehlende Offenheit proprietärer Systeme bedingt die Integration aller erforderlichen Dienste, wie Visualisierungswerkzeuge und Benutzerschnittstelle, in das System (*all-in-one-system*). Insbesondere verfügt der Anwender über keine Möglichkeiten, einzelne Systemkomponenten auszutauschen oder zu erweitern. Diese monolithische Struktur, die aus einer fehlenden Modularisierbarkeit resultiert, führt zu einer schlechten Skalierbarkeit des Gesamtsystems. Unnötiger Verwaltungsaufwand entsteht insbesondere dadurch, daß nicht nur die jeweils anwendungsspezifische Funktionalität angeboten werden kann.

3. Da sowohl der Sprachumfang als auch die Systemarchitektur unveränderbar festgelegt sind, können moderne Paradigmen, wie Objektorientierung oder Konzepte funktionaler Programmierung, nicht berücksichtigt werden.

Das Ziel des Tycoon-Systems besteht darin, die Vorteile prozeduraler Sprachen, wie Offenheit und linguistische Ausdrucksmächtigkeit, und die Vorteile deskriptiver Sprachen, wie typsichere Generizität, miteinander zu kombinieren, um so zu einer flexiblen, skalierbaren und offenen Systemarchitektur zu gelangen. Das Erreichen dieses Zieles setzt die Möglichkeit, beliebige generische Programmabstraktionen definieren zu können, voraus. Diese Fähigkeiten stellen im allgemeinen nur polymorphe Programmiersprachen zur Verfügung. Im Gegensatz zu Spracherweiterungen, zum Beispiel DBPL [MS92, MS94a] oder Pascal/R [Sch77], mit einer fest eingebauten (*built-in*) Funktionalität bietet Tycoon dem Anwendungsprogrammierer darüber hinaus eine dynamisch erweiterbare Systemumgebung (*add-on*-Ansatz) an [MS93a]. Dieser Entscheidung liegt die Überzeugung zugrunde, daß ein “ideales” persistentes Objektsystem folgende Komponenten und Fähigkeiten besitzen sollte [MS93b]:

- Eine polymorphe Sprache höherer Ordnung, die nur eine gewisse Kernfunktionalität anbietet;

- Persistenz als orthogonale Eigenschaft beliebiger Programmabstraktionen;

- eine Systemarchitektur, die die Einbindung bereits existierender externer Dienste ermöglicht (*lean architecture*) und diese als Bibliotheksdienste bereitstellt (*externe Bibliotheken*);

- eine reichhaltige und erweiterbare Menge von Programmbibliotheken, die in der persistenten Sprache selbst geschrieben sind und dem Anwendungsentwickler vordefinierte und kombinierbare Dienste zur Verfügung stellen (*interne Bibliotheken*).

Zur Realisierung der oben genannten Fähigkeiten bedarf es sprachlicher und architektonischer Maßnahmen. Die konkret eingesetzten Maßnahmen und deren Zusammenwirken untereinander werden anhand der Architektur des Tycoon-Systems verdeutlicht.

3.1 Die Komponenten des Tycoon-Systems

Die Tycoon-Systemumgebung wird durch eine dreistufige Schichtenarchitektur realisiert. Hierbei erfolgt eine logische Trennung zwischen den Aufgaben der Datenmodellierung, der Datenmanipulation und der Datenspeicherung. Diese Schichten werden in den folgenden Abschnitten kurz vorgestellt. Einen Gesamtüberblick über die Architektur vermittelt Abbildung 3.2 [Gei95].

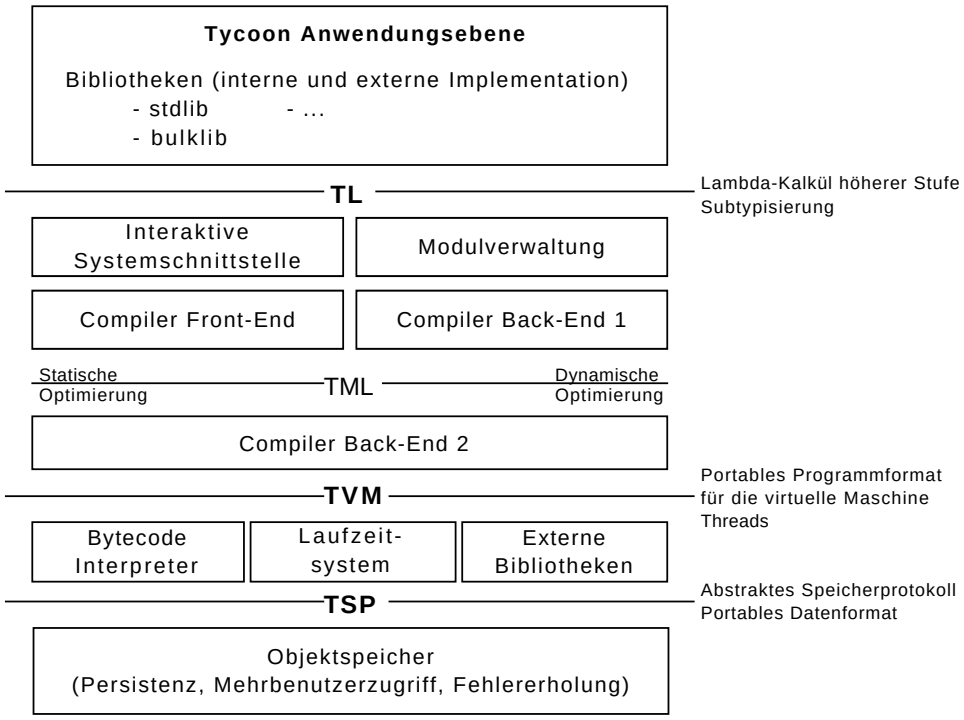


Abbildung 3.2: Architektur des Tycoon-Systems

3.1.1 Die Sprache TL[1]

Die Datenmodellierung für Tycoon-Anwendungen erfolgt in der Sprache TL. TL ist eine funktionale, polymorphe persistente Programmiersprache, die auch imperative Konstrukte beinhaltet. Sie ist strikt typisiert und strikt evaluativ und gewährleistet daher, daß bei einer kompilierten Applikation zum Exekutionszeitpunkt keine Typfehler auftreten.

Kennzeichnend für die Sprache TL sind die folgenden Sprachkonzepte:

- ▷ algorithmische Vollständigkeit
- ▷ universeller Polymorphismus
- ▷ Typoperatoren
- ▷ Funktionen höherer Ordnung

[1] **T**ycoon **L**anguage

- ▷ abstrakte Datentypen
- ▷ Ausnahmebehandlungsmaßnahmen
- ▷ Modularisierung und Bibliotheken

Die nachfolgenden Abschnitte stellen jene Sprachkonzepte kurz vor, die bei der Realisierung dieser Arbeit besondere Berücksichtigung gefunden haben. Ziel dieser Abschnitte ist es dabei nicht, einen kompletten Sprach-Report bereitzustellen. Für interessierte Leser findet sich in [MMS94] eine vollständige sprachliche Spezifikation. Für die nachfolgenden Beispiele gilt die Konvention, daß Schlüsselworte der Programmiersprache fett gedruckt erscheinen.

3.1.1.1 Parametrischer Polymorphismus

TL unterstützt das Konzept des parametrischen Polymorphismus [CW85], der durch die Einführung von Typsignaturen in Funktions- und Typdefinitionen entsteht. Dadurch wird die Spezifikation von Typparametern ermöglicht. Eine wichtige Bedeutung erhält dabei der Typ **Ok**, der Supertyp aller nicht parametrisierten Typen ist. Die Verwendung einer Signatur der Form A <:**Ok** ("A ist Subtyp von **Ok**") drückt aus, daß der Typparameter A zur Laufzeit mit einem beliebigen (nicht parametrisierten) Typ instanziiert werden darf. Darüber hinaus besteht ebenfalls die Möglichkeit, Typparameter zu spezifizieren, die nur von einer Teilmenge der möglichen Typausdrücke erfüllt werden kann. Dieser **eingeschränkte parametrische Polymorphismus** (*bounded parametric polymorphism*) wird durch Typparameter realisiert, deren Instanziierungen Subtyp eines von **Ok** verschiedenen Typs sein müssen. Das Konzept des parametrischen Polymorphismus wird in den folgenden beiden Anwendungen eingesetzt:

Polymorphe Funktionen: Polymorphe (generische) Funktionen enthalten Typvariablen als Parameter, die bei der Funktionsanwendung durch konkrete Typausdrücke instanziiert werden. Polymorphe Funktionen erlauben es somit, ein typunabhängiges Verhalten nur einmal zu modellieren. Zur Veranschaulichung dient die folgende polymorphe Funktion *printDoctor*. Sie besitzt einen Typparameter *Doctor*, der mit einem beliebigen Typ instanziiert werden kann. Des weiteren benötigt sie einen Wert *d* von diesem Typ und eine Funktion *convert*, die Werte vom Typ *Doctor* in eine Zeichenkette konvertiert. Der Funktionsaufruf von *printDoctor* bewirkt das Ausdrucken der Zeichenkette, die durch Anwendung der Funktion *convert* auf *d* zurückgegeben wird.

```
let printDoctor(Doctor <: Ok  d :Doctor  convert(:Doctor) :String) :Ok =
  print.string(convert(d))
```

Die nachfolgende Funktion *printSurgeon* stellt eine polymorphe Funktion mit eingeschränktem Typparameter *Surgeon* dar. Die Typausdrücke für die Instanziierung dieses Typparameters können somit nicht mehr frei gewählt werden, sondern müssen Subtyp des Typs *Doctor* sein. Ansonsten werden Funktionsaufrufe von *printSurgeon* analog zu Funktionsaufrufen von *printDoctor* ausgewertet.

```
Let Doctor = ...
let printSurgeon(Surgeon <:Doctor  s :Surgeon  convert(:Surgeon) :String) :Ok =
  print.string(convert(s))
```

Typoperatoren höherer Ordnung: Ebenso wie Funktionen sind auch Typen in TL Werte erster Klasse. Daher ist es möglich, auch Funktionen zu spezifizieren, die Typen auf Typen abbilden. Solche Funktionen werden als Typoperatoren bezeichnet und stellen eine weitere Möglichkeit dar, das Konzept des parametrischen Polymorphismus anzuwenden. Ein Beispiel liefert der Typoperator *PatientDB*. Das Attribut *db* unterstützt die Verwaltung von Patienten eines beliebigen Typs. Dieser Typ ist Eingabeparameter des Typoperators *PatientDB*.

```
Let PatientDB(Patient <:Ok) =
  Tuple
    hospital  :String
    db        :set.T(Patient)
  end
```

In TL können Typoperatoren darüber hinaus auch Typoperatoren auf Typoperatoren abbilden. Typoperatoren dieser Art werden als Typoperatoren höherer Ordnung bezeichnet [Car89, CL90]. Typoperatoren sind ein Sprachkonzept, das die typsichere und zuverlässige Definition eigener Abstraktionen unterstützt.

3.1.1.2 Funktionen höherer Ordnung

In TL werden Funktionen als gleichberechtigte Werte erster Klasse behandelt. Das heißt, Funktionstypen können einerseits in der Signatur einer Funktion als Eingabeparameter oder als Rückgabewert auftreten, andererseits aber auch Bestandteil benutzerdefinierter Typen sein. Im folgenden werden Beispiele für die Anwendung dieses mächtigen Programmierkonzeptes gegeben:

- **Aggregierte Funktionen:** Funktionen können Attribute in Typ- und Wertdefinitionen sein. Durch diese Eigenschaft kann beispielsweise das objektorientierte Paradigma in TL abgebildet werden, da hierfür sowohl die strukturelle als auch die verhaltensmäßige Beschreibung von Objekten in einer Klasse spezifiziert werden müssen. Das folgende Beispiel zeigt die Definition eines Typs *Patient*, der neben den strukturellen Attributen *name* und *disease* auch Funktionsattribute *getName* und *getDisease* enthält.

```
Let Patient =
  Tuple
    name       :String
    disease    :String
    getName    : Fun() :String
    getDisease : Fun() :String
  end
```

- **Funktionsparametrisierung:** Funktionen höherer Ordnung können dynamisch andere Funktionen als Parameter übergeben werden. Ein Beispiel liefert die Funktion *sortPatients*.

```
let sortPatients(max :Fun(a,b :Patient) :Patient
  patients :iter.T(Patient)) :iter.T(Patient) =
  begin
    let var result = iter.new(:Patient)
      ...
      if max(maximum  pat) == pat
      then maximum := pat
      else ok
      end
      result := iter.cons(:Patient  maximum  result)
      ...
    result
  end
```

Ausgehend von einer (ungeordneten) Iteration von Patienten (*patients*) und einer frei wählbaren Ordnungsfunktion (*max*) wird als Funktionswert eine geordnete Iteration von Patienten zurückgegeben.

- **Generatorfunktionen:** Generatoren sind Funktionen, deren Rückgabewert wiederum eine Funktion ist. Die folgende Generatorfunktion *prescribe* nimmt als Eingabe zwei Funktionen, bei denen der Ausgabeparameter der ersten Funktion (*createPatient*) zugleich Eingabeparameter der zweiten Funktion (*createReceipt*) ist. Der Rückgabewert ist eine Funktion, die beide übergebenen Funktionen hintereinander ausführt.

```
Let Receipt = ...

let prescribe(Receipt, Patient <:Ok  createPatient(name :String  disease :String) :Patient
  createReceipt(p :Patient) :Receipt) :Fun(:String :String) :Receipt =
    fun(name :String  disease :String) :Receipt
      createReceipt(createPatient(name disease))
```

3.1.1.3 Abstrakte Datentypen

Abstrakte Datentypen verbergen die strukturelle Komplexität definierter Datentypen vor dem Anwender. Ihre Schnittstelle beinhaltet einen opaken Datentyp, einen Konstruktor zur Erzeugung von Werten des abstrakten Datentyps sowie eine fest vorgegebene Menge von Methoden, mit denen der Zustand des Wertes eines abstrakten Datentyps verändert oder abgefragt werden kann (funktionales oder imperatives Kapselungskonzept, vergleiche [Wir85, Mün94]).
Der Typoperator *PatientDB* aus dem oberen Beispiel kann unter Miteinbeziehung von Methoden auch als abstrakter Datentyp definiert werden:

```
Let PatientDB(Patient <:Ok) =
  Tuple
    T <:Ok
    new() :T
    insert(p :Patient  db :T) :T
    delete(p :Patient  db :T) :T
```

```
    ...
end
```

Hierbei handelt es sich um eine funktionale Modellierung, da der Rückgabewert einer verändernden Operation der Wert des abstrakten Datentyps selbst ist. Das Konzept der Datenkapselung erlaubt es, einzelne Datenbereiche sicher voneinander zu trennen, da mit den Methoden eines abstrakten Datentyps nicht auf den Zustand eines anderen Datentyps zugegriffen werden kann.

3.1.1.4 Tupel mit Varianten

Es gibt Anwendungsszenarien, in denen eine Modellierung von Datentypen, die Fallunterscheidungen zulassen, sinnvoll erscheint. Dieses Konzept wird in TL durch Tupel mit Varianten bereitgestellt. Ihre Anwendung zeigt das folgende Programmbeispiel.

```
Let Arzt =
  Tuple
    name      :String
    praxisort :String
    case allgemeinmediziner
    case facharzt with
      spezialgebiet :String
  end
```

Der Typ *Arzt* differenziert zwischen zwei verschiedenen Varianten. Die Variante *allgemeinmediziner* umfaßt zwei Attribute (*name* und *praxisort*); die Variante *facharzt* ist ein Tupel mit drei Attributen (*name*, *praxisort* und *spezialgebiet*). Die Konstruktion eines Wertes vom Typ *Arzt* erfolgt unter Angabe seiner konkreten (aktuellen) Variante.

```
let drMeier :Arzt =
  tuple case facharzt of Arzt with "Meier" "AKH Barmbek" "Chirurgie" end
```

Zur Bestimmung der aktuellen Variante eines Wertes vom Typ *Arzt* wird dieser einem Variantentest unterzogen. Der Zugriff auf das variantenspezifische Attribut *spezialgebiet* erfolgt dabei durch Einführung einer lokalen Wertvariablen *l*.

```
case drMeier
  when allgemeinmediziner then
    print.string("Arzt für Allgemeinmedizin")
  when facharzt with l then
    print.string("Facharzt mit Spezialgebiet " <> l.spezialgebiet)
end
```

3.1.1.5 Ausnahmebehandlung

Das Konzept der Ausnahmebehandlung (*exception handling*) ermöglicht es, Fehlersituationen, die zur Laufzeit auftreten können, zu definieren und gegebenenfalls asynchron zu behandeln. Es stellt somit ein wichtiges Konzept moderner Programmiersprachen dar.

```
let insertPatient(p :Patient  db :set.T(Patient)) :Ok =
  try
    set.insert(:Patient  db  p)
  when set.error then
    print.string("### Patient already exists ###")
  end
```

Die Funktion *insertPatient* fügt der Menge *db* einen Patienten *p* hinzu. Existiert dieser Patient bereits in dieser Menge, so ist die Mengeneigenschaft verletzt, und es wird eine Ausnahme ausgelöst. Diese kann abgefangen werden, so daß der Anwender statt eines Laufzeitfehlers die Meldung "*Patient already exists*" erhält.

3.1.2 Die virtuelle Tycoon-Maschine

TL-Programmanweisungen, die entweder Bestandteile von Tycoon-Programmen sind oder deren Eingabe interaktiv über die Systemschnittstelle (*top level*) erfolgt, werden zunächst in eine sprachliche Zwischenrepräsentation (TML[2]) übersetzt. Die logische Trennung dieser Zwischenrepräsentation von der Sprache TL selbst ermöglicht die Portierung auf andere Hard- und Softwareplattformen [Mat93]. Darüber hinaus eignet sich diese Zwischenrepräsentation für die Durchführung statischer und dynamischer Optimierungsprozesse, durch die sich das Laufzeitverhalten von Programmkode, der in TL geschrieben ist, verbessert [Kir94, GM94]. Das Resultat eines zweiten Übersetzungsprozesses ist ausführbarer Bytekode. Dieser wird dann interpretiert und vom Laufzeitsystem der virtuellen Tycoon-Maschine (TVM[3]) ausgeführt. Für die Exekution des Bytekodes ist ein persistenter Objektspeicher als Adreßraum notwendig. Die Tycoon-Maschine verfügt außerdem über Schnittstellen zum Aufruf und damit zur Integration extern implementierter Dienste. Diese Schnittstellen gewährleisten die Offenheit des Tycoon-Systems.

3.1.3 Die Objektspeicherschnittstelle

Die Speicherung von Daten unterliegt der Objektspeicherschnittstelle TSP[4]. Ihre Aufgaben lassen sich wie folgt charakterisieren [Mat93]:

- **Automatische Freispeicherverwaltung:** TSP sichert die Allokation und die Freigabe von Speicherplatz zu. Im Gegensatz zu Programmiersprachen wie zum Beispiel C erfolgt diese Zusicherung automatisch, so daß der Benutzer von dieser Aufgabe befreit wird. Das Kriterium für die Freigabe von Speicherplatz, der durch ein Objekt belegt wird, ist dessen

[2] **T**ycoon **M**achine **L**anguage
[3] **T**ycoon **V**irtual **M**achine
[4] **T**ycoon **S**tore **P**rotocol

Erreichbarkeit. Solange (mindestens) eine Referenz auf ein Objekt existiert, darf sein Speicherplatz nicht freigegeben werden (*referentielle Integrität*), anderenfalls wird er im Rahmen einer *garbage collection* dealloziiert.

- **Persistenz:** TSP stellt sicher, daß alle von einem als langlebig deklarierten Objekt transitiv erreichbaren Objekte ebenfalls persistent gehalten werden.

- **Fehlererholung:** Datenbanksysteme müssen Mechanismen zur Fehlererholung (*recovery*) für den Fall eines Systemausfalls bereitstellen. Unter TSP ist es möglich, Sicherungspunkte zu definieren, die eine sukzessive Rekonstruktion des Systemzustandes erlauben.

- **Portabilität:** TSP zeichnet sich durch seine weitgehende Unabhängigkeit vom verwendeten Objektspeichersystem und damit auch von der zugrundeliegenden Hardwareplattform aus. Diese ermöglicht es, Daten und Programme auch auf anderen Hard- und Softwareplattformen ablaufen zu lassen.

- **Synchronisation:** Die Tycoon-Systemumgebung ist als Mehrbenutzersystem konzipiert worden. Auch die Objektspeicherschnittstelle muß daher Synchronisationsmechanismen für den Mehrbenutzerzugriff bereitstellen. Hierbei müssen Konzepte wie eine faire Bedienstrategie, Vermeidung von Systemverklemmungen (*deadlocks, lifelocks*), wechselseitiger Ausschluß (*mutual exclusion*) und die Serialisierbarkeit von Prozessen Berücksichtigung finden.

3.2 Bereitstellung von Sicherheitsmechanismen in Bibliotheken

Die vorangegangenen Ausführungen dieses Kapitels konstatierten, daß gegenwärtige persistente Objektsysteme mit Sprachen der vierten Generation überwiegend als proprietäre monolithische Programmpakete angeboten werden. Als Konsequenz der geschlossenen Architektur dieser Systeme folgte hieraus insbesondere, daß alle Dienste, die für den Systembetrieb erforderlich sind, als integraler Bestandteil des Systems vorhanden sein müssen. Dies gilt auch für Sicherheitsdienste; sie sind speziell auf das zu unterstützende System zugeschnitten und fest in dieses eingebaut (*built-in*-Ansatz). Ein solches Vorgehen führt zu einer strikten Erzwingung der in diesem System verfolgten Sicherheitspolitik. Nachteilig wirkt sich jedoch die fehlende Flexibilität des Sicherheitsdienstes aus. Aufgrund dieser Rigidität gibt es keine Möglichkeit, die Sicherheitspolitik an sich verändernde Systemanforderungen anzupassen.

In Einklang mit der Philosophie des Tycoon-Systems wird hier für die Implementierung von Sicherheitskonzepten ein bibliotheksorientierter (*add-on*) Ansatz verfolgt. Ziel ist die Bereitstellung eines generischen Sicherheitsdienstes, dessen Sicherheitspolitiken flexibel an die Erfordernisse der zu unterstützenden Applikation angepaßt werden können. Durch diesen adaptiven Charakter kann (statt einer einzigen Applikation) eine Vielzahl von sicherheitssensitiven Anwendungen unterstützt werden. Während die Flexibilität eines solchen Ansatzes sehr hoch ist, kann in einem rein bibliotheksorientierten Ansatz die Sicherheit nur selten erzwungen werden, da keine Mechanismen existieren, die verhindern, daß Subjekte die Benutzung der Sicherheitsbibliotheken umgehen.
Hieraus folgt, daß ein bibliotheksorientierter Sicherheitsdienst in der Regel immer zu einem

gewissen Grade durch einen systeminhärenten Sicherheitsdienst unterstützt werden muß. Dabei ist der Einbau von Sicherheitsmechanismen prinzipiell in allen drei Systemschichten der Tycoon-Umgebung (vergleiche Abbildung 3.2) vorstellbar. Die konkret vorhandenen eingebauten Sicherheitsmechanismen werden im folgenden anhand verschiedener Szenarien vorgestellt. Hierbei wird analysiert, welche unterschiedlichen Gewichtungen zwischen eingebauter und bibliotheksorientierter Zugriffskontrolle vorgenommen werden müssen, um die Sicherheit des Gesamtsystems in einer heterogenen verteilten Umgebung zu gewährleisten. Zudem wird gezeigt, wie Sicherheitspolitiken, die mit Hilfe von Programmbibliotheken modelliert werden, auch in tieferen Systemschichten durch einen eingebauten Mechanismus erzwungen werden.

3.3 Tycoon als sicherer Diensterbringer

Im Zuge einer zunehmenden Vernetzung von Informationssystemen einerseits und einer steigenden Zahl von Computeranwendern andererseits, kommt Netzwerkdiensten, denen eine Kunden-Diensterbringer-Architektur (*client-server architecture*) zugrundeliegt, eine wachsende Bedeutung zu. Diese Entwicklung drückt den Wunsch aus, die relativ teure Rechenleistung eines Zentralrechners (*mainframe*) durch ein flexibleres vernetztes System von (spezialisierten) Arbeitsplatzrechnern (*workstations*) zu ersetzen [Kaß94]. Eine solche Architektur stellt somit ein realistisches Anwendungsszenario für den Einsatz interagierender Tycoon-Umgebungen dar. Hierbei fungiert ein Tycoon-Prozeß als Diensterbringer (*server*), dessen Dienste von anderen Tycoon-Prozessen (*clients*) in Anspruch genommen werden. Entscheidend für den Einbau von Sicherheitsfunktionalität in den Diensterbringerprozeß ist die Frage, welche Systemschichten (TL, TVM, TSP) unter seiner alleinigen Kontrolle stehen und auf welche Systemschichten die Klientenprozesse einen Einfluß haben.

3.3.1 Kommunikation ohne Transfer von Kodeobjekten

In diesem Szenario verwaltet der Diensterbringer alle exekutierbaren Funktionen sowie alle Objekte, auf die im Rahmen der Diensterfüllung zugegriffen werden muß, selbst. Eine Konfiguration dieser Art ist zum Beispiel beim Auruf einer entfernten Funktion (*remote procedure call* [Cor91, OSF93]) gegeben. Der Klient kann hierbei nur aus einer durch den Diensterbringer vorgegebenen Menge von Diensten (zum Beispiel über ein vorgegebenes Menü) wählen und kann keinen darüber hinausgehenden Einfluß auf den Diensterbringer nehmen. Hieraus folgt, daß die Zugriffskontrolle für die Funktionsanwendungen uneingeschränkt beim Diensterbringer liegt und nicht von den Klienten umgangen werden kann. Daher kann für diesen besonderen Fall auf den Einbau von Sicherheitsmechanismen verzichtet werden. Die Zugriffskontrolle erfolgt dann ausschließlich über den Autorisierungsdienst der Programmbibliothek, also auf der Applikationsebene. Die Programmfragmente der Abbildung 3.3 illustrieren den zeitlichen Verlauf von der Auswahl eines Dienstes bis zur Ausführung der ihn erbringenden Funktion.

Empfängt der Diensterbringer die Auswahlentscheidung des Klienten, so wird abhängig von dessen Inhalt eine sichere diensterbringende Funktion aufgerufen (Auswahlroutine). Diese **sichere Funktion** überprüft die Zugriffsberechtigung des Klienten auf Basis der bibliotheksgestützten Funktion *authorized.* Ist der Klient zugriffsberechtigt, so wird die tatsächliche, ungeschützte diensterbringende Funktion aufgerufen und das Ergebnis an den Klienten

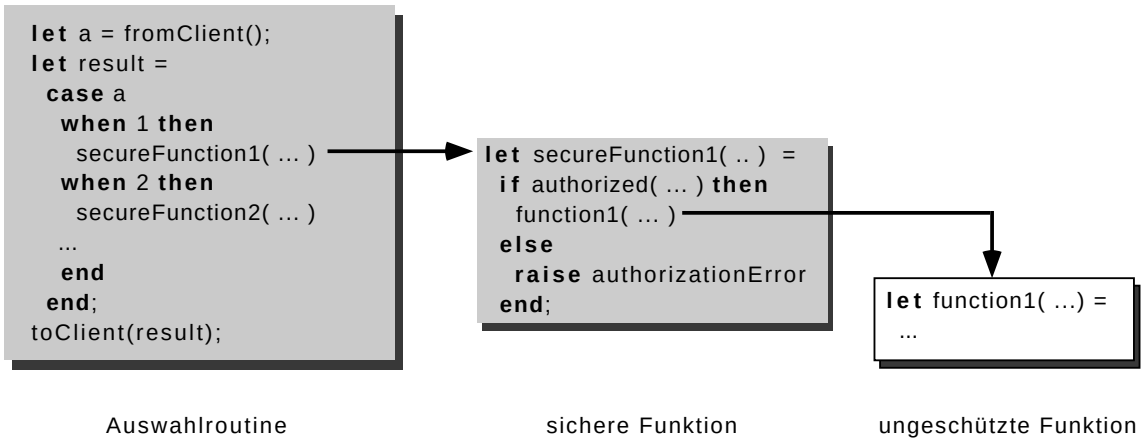


Abbildung 3.3: Funktionsausführung unter Kontrolle des Diensterbringers

zurückgegeben; anderenfalls wird eine Ausnahme ausgelöst. Mechanismen für die Implementierung der Funktion *authorized* werden in den nächsten beiden Kapiteln behandelt.

3.3.2 Kommunikation mit entfernter Exekution von Kodeobjekten

Statt den Klienten eine feste Menge von Funktionen über eine Exportschnittstelle zur Verfügung zu stellen, ist es Ziel neuerer Ansätze, dem Benutzer nur *eine* generische ausführbare Funktion höherer Ordnung anzubieten, die beliebig parametrisiert werden kann (*remote execution engines* [MMS95b, Car94]). Auf diese Weise ist es für die Klienten möglich, auch eigene Kodeobjekte an den Diensterbringer zu verschicken und diese bei ihm ausführen zu lassen. Eine wichtige Anwendung fur dieses Szenario sind Aktivitäten (*threads*). “Eine Aktivität beschreibt einen einzelnen sequentiellen Kontrollfluß in einem Programm” [MMS95a, Seite 2]. Es ist möglich, mehrere (parallele) Kontrollflüsse durch ein Programm zu haben (*multiple threads*), langandauernde Aktivitäten wie Geschäftsprozesse oder Ablaufsteuerungen (*workflow management*) persistent zu machen (*persistent threads*) [MS94b] oder Aktivitäten auf entfernten Rechnern ausführen zu lassen (*migrating threads*). Während multiple und persistente Aktivitäten ihren lokalen Sichtbarkeitsbereich nicht verlassen, können sich migrierende Aktivitäten dynamisch an entfernte Ressourcen binden. Bezogen auf die Tycoon-Systemumgebung bedeutet dies, daß Klienten ausführbare Programmeinheiten an den Diensterbringer senden, der diese interpretiert, ausführt und das Ergebnis an den Klienten zurückschickt. Dieser Vorgang wird durch folgendes TL-Fragment skizziert:

*clientFun :***Fun***() :String = fromClient();*
let *result = clientFun ();*
toClient(result);

In diesem Szenario ist eine reine bibliotheksgestützte Zugriffskontrolle auf der Seite des Diensterbringers unzureichend, da die vom Klienten versandten Kodeobjekte bei ihrer Ausführung unter Umständen auf Ressourcen des Diensterbringers zugreifen. Solche Zugriffe

sind insbesondere dann sehr wahrscheinlich, wenn gewisse Ressourcen, wie zum Beispiel Druckdienste oder Betriebssystemfunktionen, an allen Knoten des Netzwerks vorhanden sind (*ubiquitous ressources*). Es ist daher neben bibliotheksgestützter Zugriffskontrolle auch erforderlich, eingebaute Maßnahmen zur Authentisierung und Autorisierung zu verwenden. Diese müssen in die virtuelle Tycoon-Maschine (TVM) eingebaut werden, da hier die konkrete Interpretation und Ausführung der gesandten Kodeobjekte erfolgt.
Ein einfacher besitzerorientierter Ansatz für den Einbau dieser Zugriffskontrollmaßnahmen, bei dem alle Klienten nur auf die von ihnen erzeugten Objekte zugreifen können, stellt keine adäquate Lösung für diese Problemstellung dar. Zwar ist die Sicherheit in einem solchen System gegeben, die Klienten können aber nicht mehr auf die vom Diensterbringer bereitgestellten Kodeobjekte zugreifen, da sie nicht Besitzer dieser Objekte sind.

Der besitzerorientierte Ansatz wird daher um die Möglichkeit erweitert, kurzfristig die Identität eines anderen Subjektes anzunehmen (*switch-user-mechanism*): Während der Klient Besitzer der von ihm erzeugten Objekte ist und der Diensterbringer Besitzer aller von ihm verwalteten *ungeschützten* Objekte ist, kann ein exekutierender Zugriff auf die sicheren Funktionen des Systems von allen Subjekten, das heißt dem Diensterbringer und allen Klienten, durchgeführt werden. Kann die Zugriffsberechtigung eines Klienten für die Ausführung der ungeschützten Funktion erfolgreich verifiziert werden, nimmt er kurzfristig die Identität des Besitzers dieser Funktion an. Wird der Sichtbarkeitsbereich der geschützten Funktion verlassen, so nimmt das zugreifende Subjekt wieder seine ursprüngliche Identität an.

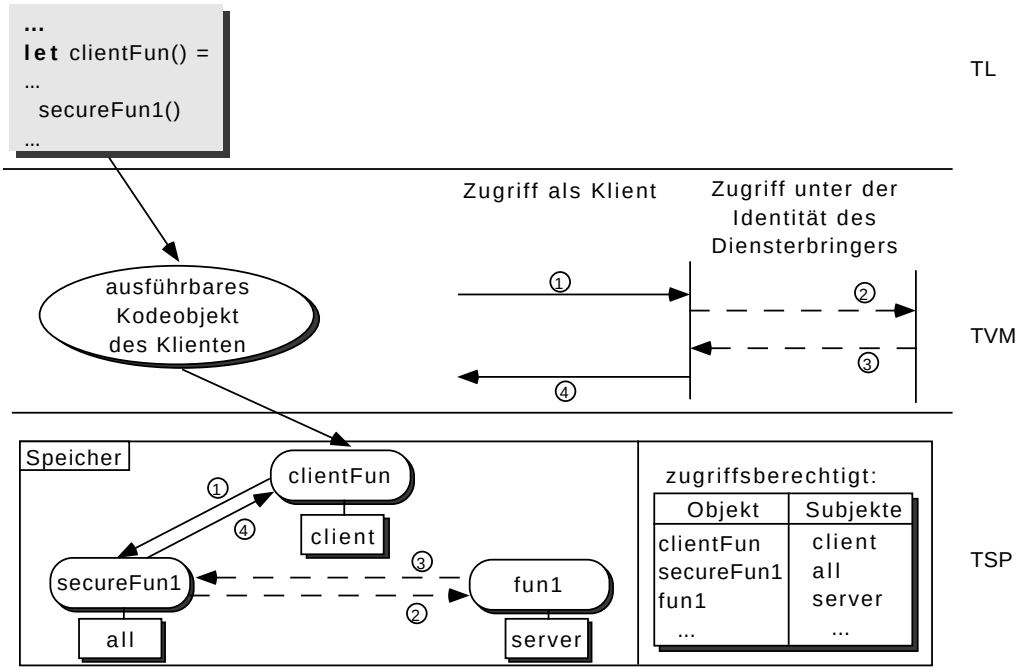


Abbildung 3.4: Eingebaute Zugriffskontrolle mit wechselnder Subjektidentität

Dieses Konzept wird durch Abbildung 3.4 exemplarisch veranschaulicht: Der Klient schickt ein ausführbares Kodeobjekt (*execution unit*) an den Diensterbringer, das die Funktion *clientFun* enthält. Diese Funktion greift auf die vom Diensterbringer bereitgestellte sichere Funktion

secureFun1 zu (Schritt ①). Der Zugriff auf diese Funktion ist erlaubt, da alle Subjekte Zugriff auf dieses Objekt haben. Die Funktion *secureFun1* überprüft anhand der über die Bibliotheksfunktionen definierten Sicherheitspolitik, ob der Klient berechtigt ist, den durch die ungeschützte Funktion *fun1* bereitgestellten Dienst in Anspruch zu nehmen. Ist dies der Fall, so nimmt der Klient kurzfristig die Identität des Diensterbringers an, da nur dieser Zugriff auf die Funktion *fun1* hat (Schritt ②). Wird nach deren Ausführung ihr Sichtbarkeitsbereich verlassen und die Kontrolle an die Funktion *secureFun1* zurückgegeben, so arbeitet der Klient wieder unter seiner eigenen Identität (Schritt ③). Abschließend wird das Ergebnis der aufgerufenen Dienstfunktion an die Funktion *clientFun* zurückgegeben (Schritt ④).

Dieses Szenario zeigt, wie relativ einfache eingebaute Sicherheitsmaßnahmen auf der Systemebene mit Sicherheitspolitiken beliebiger Komplexität, die auf der Anwendungsebene modelliert werden, kombiniert werden können. Diese eingebauten Sicherheitsprimitive sind somit ein Basismechanismus zur Erzwingung bibliotheksgestützer Autorisierungskontrolle auf der Anwendungsebene. Während jeder Diensterbringer systemweit konzeptuell die gleichen eingebauten Sicherheitsmechanismen hat, können die Sicherheitspolitiken so definiert werden, daß sie für jede Anwendung maßgeschneidert sind. Hierbei kann das gesamte Spektrum von paarweise disjunkten lokalen Sicherheitspolitiken bis hin zu einer globalen Sicherheitspolitik für alle Diensterbringer eines Systems abgedeckt werden.

Kapitel 4

Konzeption der Zugriffskontrollbibliothek

Die vorangegangenen beiden Kapitel haben die Systementwicklungsumgebung Tycoon als Entwicklungswerkzeug für verteilte datenintensive Anwendungen sowie objektorientierte Zugriffskontrollmodelle als flexibles Modellierungsinstrument vorgestellt. Die Systementwicklungsumgebung Tycoon erlaubt die orthogonale Modellierung beliebiger polymorpher Datenstrukturen und deren typsichere Abbildung in Programmbibliotheken, die zur Entwicklung persistenter Anwendungen eingesetzt werden können.

Sie bietet somit die Voraussetzung für die Implementierung komplexer Anwendungen. Der Schutz dieser Anwendungen erfordert Zugriffskontrollmaßnahmen, für deren Realisierung sich objektorientierte Zugriffskontrollabstraktionen als adäquates Modellierungsinstrument herauskristallisiert haben. Das Potential objektorientierter Zugriffskontrollabstraktionen liegt zudem in einem erhöhten Maß an Modellierungsfreiheit, da nicht jede Autorisierung explizit modelliert werden muß.

Als Ausgangspunkt für die weiteren Überlegungen (vergleiche Abbildung 4.1) ergibt sich das Problem, geeignete Abstraktionen zu finden, mit denen objektorientierte Zugriffskontrollmechanismen unter Beibehaltung dieser Modellierungsfreiheit in generische Tycoon-Bibliotheken abgebildet werden können. Ziel ist es dabei nicht, das objektorientierte Paradigma in Tycoon zu integrieren. Vielmehr sollen die Kontrollabstraktionen dieses Zugriffsmodells mit Hilfe des Tycoon-Systems als generischer Zugriffskontrolldienst in einer Programmbibliothek zur Verfügung gestellt werden. Diese Bibliothek umfaßt Basisdienste, die der Anwender zur Implementierung seiner konkreten Sicherheitspolitik bedarfsgerecht kombinieren kann. Ebenso unterstützt sie die Entwicklung neuer *modellgebundener* Autorisierungssysteme, die dann als "höherer" Bibliotheksdienst zur Verfügung gestellt werden können. Die folgenden Abschnitte erklären diesen Abbildungsprozeß schrittweise. Hierzu wird zunächst ein Basismodell zum Schutz von Werten eingeführt, dessen Einsatzmöglichkeiten anhand von Anwendungsbeispielen erläutert werden. Aufbauend auf diesem Modell werden Modellierungsvereinfachungen

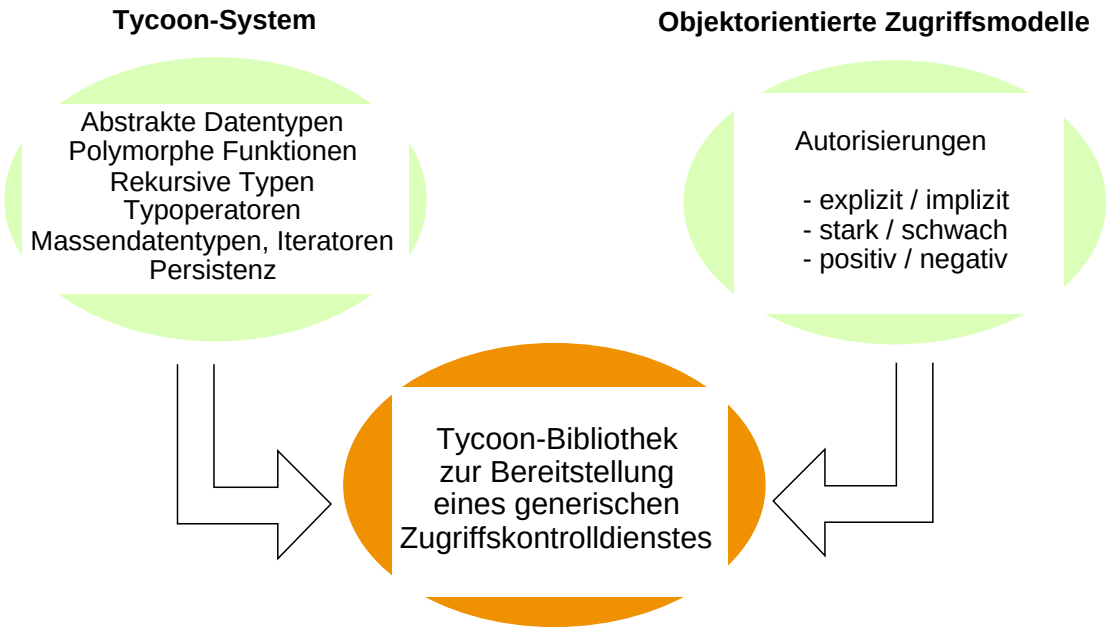


Abbildung 4.1: Integration objektorientierter Kontrollabstraktionen in Tycoon

vorgestellt, die den Prozeß der Rechtevergabe unterstützen. Eine Diskussion spezieller Konzepte, wie der Modellierung unterschiedlicher Vererbungsrichtungen und der Auflösung von Konfliktsituationen, schließt sich an und vervollständigt den Abbildungsprozeß objektorientierter Zugriffskontrollmechanismen. Die formale Spezifikation eines erweiterten Basismodells sowie die konzeptuelle Abbildung von Subjekthierarchien nach Tycoon bringen dieses Kapitel zum Abschluß.

4.1 Kontrollstrukturen für Werte

Die Domänen der Subjekte, Objekte und Zugriffsmodi sind die relevanten Determinanten für die Definition einer Menge von Rechten (vergleiche Abschnitt 1.1). Die Übertragung eines Zugriffskontrollmodells in die Tycoon-Umgebung setzt daher zunächst die Identifikation dieser Domänen voraus.
Die Bestimmung der Subjekte ist im wesentlichen eine Fragestellung der Authentisierung. Da im Kontext dieser Arbeit davon ausgegangen wird, daß eine Authentisierung bereits erfolgt ist, wird im folgenden die Existenz von Repräsentationen für Subjekte als gegeben angenommen. Strukturierungsmöglichkeiten für Subjekte werden zunächst außer acht gelassen und in Abschnitt 4.2 behandelt.
Objekte sind passive Entitäten, also Einheiten, auf die zugegriffen wird. In Tycoon kann der Zugriff sowohl auf Werte (zum Beispiel durch Funktionen) als auch auf Typen (durch Typoperatoren) erfolgen. Da Typen jedoch lediglich strukturelle Information beinhalten, wird im folgenden die Domäne der Objekte als Menge aller in TL erzeugbaren Werte definiert, da nur Werte als konkrete Instanziierung von Typen sicherheitsrelevante Information repräsentieren können.
Die Menge der Zugriffsmodi beschreibt die Möglichkeiten, mit denen auf ein Objekt zugegriffen werden kann. Diese läßt sich differenzieren in Konstruktoren (zur Erzeugung von Ob-

jekten), Mutatoren (zur Veränderung von Objekten), Selektoren (für den lesenden Zugriff auf Komponenten eines Objektes) und Destruktoren (für die Invalidierung von Objekten). Diese Zugriffsarten werden in Tycoon durch Funktionen repräsentiert. Diese wiederum sind, aus der Sicht des Tycoon-Systems betrachtet, Werte eines beliebigen Funktionstyps. Zusammenfassend läßt sich feststellen, daß eine sinnvolle Zugriffskontrolle in Tycoon auf der Basis von Werten erfolgen muß. Dies können Werte eines beliebigen Typs sein; Werte unterliegen somit keinen Beschränkungen hinsichtlich ihrer Komplexität.
Der folgende Abschnitt beschäftigt sich mit dem Aufbau von Wertstrukturen. Hierfür wird zunächst ein Basismodell entwickelt, mit dem geschützte Werte erzeugt und über gerichtete Konnektoren miteinander verbunden werden können. Aufbauend auf diesem Basismodell werden dann Modellierungsvereinfachungen vorgestellt, mit denen der Modellierungsaufwand für die Spezifikation einer sicherheitssensitiven Applikation erheblich reduziert werden kann.

4.1.1 Ein Basismodell zum Schutz von Werten

Der Schutz eines Wertes kann prinzipiell mit zwei verschiedenen Methoden erreicht werden:

1. Die Zugriffskontrollinformation wird systeminhärent verwaltet, beziehungsweise für jeden Typ wird bereits zum Zeitpunkt der Datenmodellierung ein Attribut für seine Zugriffskontrollinformation bereitgestellt (*built-in*-Ansatz). Dieses Vorgehen ist nur dann möglich, wenn Sicherheitsaspekte bereits beim Entwurf in eine Anwendung eingeflossen sind. Auf diese Weise werden Werte durch einen systeminternen Mechanismus geschützt. Bestehende ungeschützte Applikationen können jedoch nicht nachträglich in sichere Applikationen transformiert werden.

2. Die zu schützenden Werte werden nicht modifiziert. Zugriffskontrolle wird dadurch gewährleistet, daß neben der Verwaltung des ungeschützten Wertes auch eine Administration von wertspezifischer Zugriffskontrollinformation vorgenommen wird (*add-on*-Ansatz). Durch diese Strategie können auch bestehende ungeschützte (Teil-)Systeme noch nachträglich um Zugriffskontrollmechanismen erweitert werden. Bei diesem Vorgehen muß jedoch sichergestellt werden, daß nicht auf die unsicheren Komponenten allein zugegriffen werden kann. (vergleiche Abschnitt 3.2).

Gemäß der gestellten Zielsetzung, ein flexibles Zugriffskontrollsystem zu entwickeln, konzentrieren sich die weiteren Betrachtungen auf den zweiten Ansatz. Dazu bedarf es der konzeptuellen Verschmelzung eines ungeschützten Wertes und seiner Zugriffskontrollinformation zu einer neuen Abstraktion, die einen **geschützten Wert** repräsentiert. Ein solches Konglomerat aus ungeschütztem Wert und Zugriffskontrollinformation wird auch als **Wertwächter** (*Valueguard*) bezeichnet. Die Bestandteile der Zugriffskontrollinformation sind dabei prinzipiell frei wählbar. Eine Konkretisierung erfolgt in Abschnitt 4.1.10. Ein Wertwächter schützt immer genau einen Wert beliebiger Komplexität. Für genaue Implementierungsdetails sei auf das folgende Kapitel verwiesen.

Abbildung 4.2 zeigt die Erzeugung eines Wertwächters anhand eines konkreten Beispiels: Der Wert *W* wird mit Zugriffskontrollinformation versehen. Diese enthält eine Menge zugriffsberechtigter Subjekte (Sub_1, Sub_2 und Sub_3). Durch das Verschmelzen von Wert und Zugriffskontrollinformation kann bei Verwendung des Wertwächters nun nicht mehr direkt auf den ungeschützten Wert zugegriffen werden. Ein solcher Zugriff findet nur dann statt, wenn sich aus

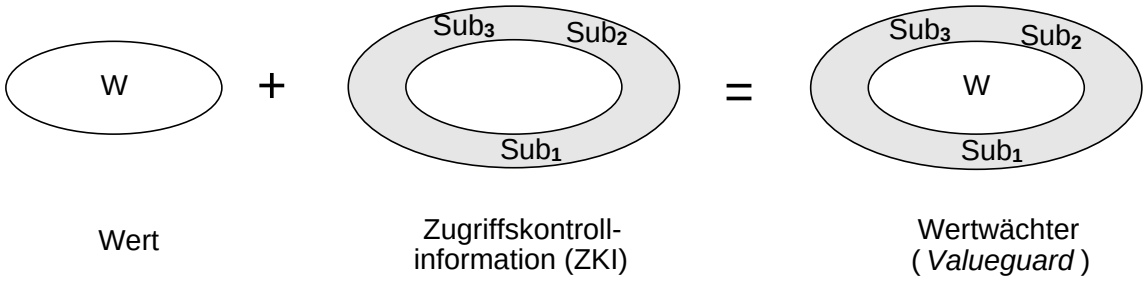


Abbildung 4.2: Erzeugung eines Wertwächters

der Auswertung der den Wert umgebenden Zugriffskontrollinformation (hier: der Menge der Subjekte) eine Zugriffsberechtigung ableiten läßt. Die Auswertung der Zugriffskontrollinformation ergibt sich nicht automatisch aus der gewählten Modellierung. Werden jedoch geeignete Funktionen durch die Autorisierungsbibliotheken zur Verfügung gestellt, ist der Anwender in der Lage, diese gemäß seiner Anforderungen in die Applikation zu integrieren.
Die bisherigen Überlegungen haben sich ausschließlich auf den Schutz *einzelner* Werte beschränkt. Die Anwendungsbeispiele in Abschnitt 2.4.1 zeigen aber, daß die gesamte Bandbreite objektorientierter Zugriffskontrollabstraktionen erst dann zweckdienlich ausgenutzt werden kann, wenn die zu schützenden Werte in hierarchischen Beziehungen zueinander stehen. TL als “modellfreie Sprache” besitzt jedoch keine sprachlichen Möglichkeiten zur Modellierung von Vererbung. Für die applikationsspezifische Vernetzung von Wertwächtern werden daher gerichtete Konnektoren (Kanten) bereitgestellt, die die Semantik expliziter und impliziter Autorisierung in objektorientierten Zugriffskontrollsystemen widerspiegeln.

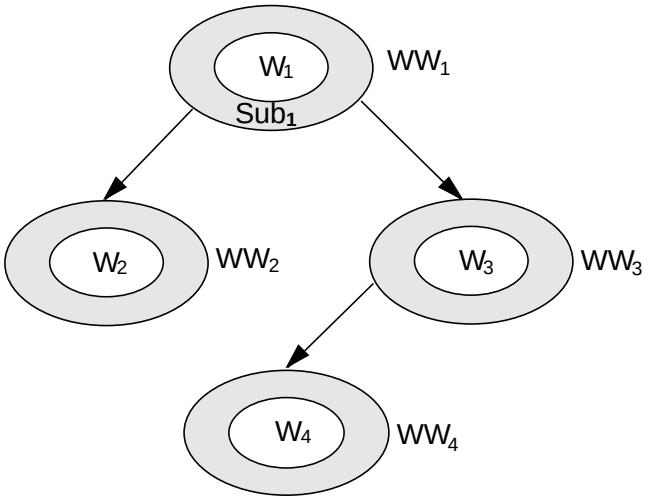


Abbildung 4.3: Konnektoren zwischen geschützten Werten

Abbildung 4.3 illustriert die Weitergabe von Rechten zwischen Wertwächtern über Konnektoren: Subjekt Sub_1 besitzt eine Zugriffsberechtigung für den Zugriff auf den Wert W_1, der durch den Wertwächter WW_1 geschützt ist. Eine solche explizite Autorisierung fehlt für den Zugriff auf die Werte W_2, W_3 und W_4. Verknüpft man die Wertwächter WW_1 mit WW_2, WW_1 mit

WW_3 und WW_3 mit WW_4 mit je einem Konnektor, so ist Sub_1 nun auch für den Zugriff auf die Werte W_2, W_3 und W_4 implizit autorisiert, da es jeweils gerichtete Kantenzüge von WW_1 zu allen anderen Wertwächtern gibt. Die kombinierte Modellierung geschützter Werte und ihrer Beziehungsstruktur zueinander erlaubt die Abbildung von Objekthierarchien beliebiger Komplexität.
Wertwächter und Konnektoren bilden die wesentlichen Bestandteile des Basismodells, auf dem im folgenden weitere Modellierungsvereinfachungen aufbauen.

4.1.2 Geschützte Repräsentationen für Objekte

Eine Einsatzmöglichkkeit des Basismodells besteht darin, Werte, die Daten repräsentieren, - im folgenden Datenwert genannt - zu schützen. Die Zugriffskontrolle einer solchen Modellierung ist jedoch recht unspezifisch. Insbesondere gilt, daß ein Subjekt, für das eine Zugriffserlaubnis auf einem bestimmten Datenwert besteht, mit jeder beliebigen Funktion auf diesen Wert zugreifen darf, da in der Zugriffskontrollinformation nicht nach einzelnen Zugriffstypen unterschieden wird. In einem speziellen Anwendungsszenario kann eine derartige einfache Modellierung nützlich erscheinen. Für die Mehrzahl von Anwendungen ist jedoch die Granularität einer solchen Modellierung, die entweder alle Funktionszugriffe auf einen Wert oder keinen Zugriff erlaubt, zu groß.

4.1.3 Geschützte Repräsentationen für Zugriffstypen

Die Zugriffskontrolle des Tycoon-Systems erstreckt sich auf Werte beliebigen Typs und beliebiger Komplexität. Hieraus folgt insbesondere, daß auch Funktionen, als Repräsentationen für Zugriffstypen, der Zugriffskontrolle unterstellt werden können, da Funktionen Werte eines Funktionstyps sind. Nachfolgend werden Funktionen, die einen Zugriffstyp repräsentieren, daher auch als Funktionswert bezeichnet. Es wird nunmehr die Funktion als ungeschützter Wert betrachtet, der durch Hinzufügen administrativer Zugriffskontrollinformation zu einem Wertwächter aggregiert wird.

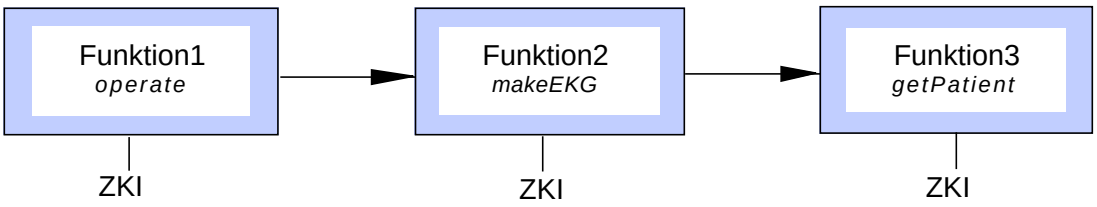


Abbildung 4.4: Konnektoren zwischen geschützten Funktionen

Auch zwischen Zugriffsmodi können in der realen Welt Beziehungen bestehen: Es gibt komplexe Vorgänge, die nur von qualifiziertem Fachpersonal durchgeführt werden dürfen, ebenso wie einfache Routineaufgaben, die wenig Qualifikation erfordern. Diese Vorgänge können durch Funktionen abgebildet werden. In der Regel wird ein komplexer Vorgang stärkeren Sicherheitsrestriktionen unterliegen als ein Routinevorgang. Um diese Abhängigkeiten im Zugriffskontrollsystem repräsentieren zu können, ist es erforderlich, Wertwächter über Konnektoren

miteinander zu verbinden. Dies veranschaulicht exemplarisch Abbildung 4.4. Dargestellt sind hier drei Funktionen, die in der realen Welt Zugriffsarten auf das Objekt “menschliches Herz” repräsentieren: *operate* für die Durchführung einer Herzoperation, *makeEKG* für die Erstellung eines Elektrokardiogramms und *getPatient* zur Ermittlung des zugehörigen Patienten. Die Zugriffstypen dieses Beispiels lassen in eine logische totale Ordnung bringen: Wer autorisiert ist, eine Herzoperation durchzuführen, dem soll auch das Recht zur Erstellung eines Elektrokardiogramms erteilt werden, welches wiederum die Berechtigung zur Ermittlung des involvierten Patienten impliziert.
Die Abbildung einer derartigen Zugriffstyphierarchie wird durch Wertwächter für Funktionen sowie Konnektoren, die diese Wertwächter miteinander verbinden, modelliert. Zu beachten ist jedoch, daß bei dieser Art der Modellierung die explizite Autorisierung eines Subjektes für die Benutzung einer bestimmten Funktion nicht nur den impliziten Zugriff auf alle durch Konnektoren transitiv erreichbaren geschützten Funktionen gewährleistet, sondern außerdem zur Folge hat, daß diese auf alle Werte des Typs Herz angewandt werden können.
Auch diese Modellierung kann im Kontext einer speziellen Anwendung von Interesse sein; häufig werden jedoch restriktivere Zugriffsbeschränkungen angestrebt. Notwendig ist es daher, den Zugriff einer bestimmten Funktion auf einen bestimmten Wert durch einen Wertwächter abbilden zu können.

4.1.4 Geschützte Repräsentationen für Zugriffe auf Objekte

In der Regel reichen die bisher beschriebenen Modellierungsmethoden nicht aus, den Schutz beliebiger Applikationen adäquat zu gewährleisten, da sowohl eine Zugriffsberechtigung für beliebige Funktionen auf einem bestimmten Wert als auch eine Autorisierung für den Zugriff einer Funktion auf beliebige Werte meist zu allgemein ist. Nachfolgend wird daher gezeigt, wie der Zugriff einer bestimmten Funktion auf einen bestimmten Wert modelliert wird.

Die Komplexität eines geschützten Wertes unterliegt keinen Beschränkungen. Daraus folgt unmittelbar, daß auch mehrere Werte durch einen Wertkonstruktor (zum Beispiel *tuple* oder *record*) zu einem komplexeren Wert aggregiert werden können. Dieses Modellierungsinstrument kann zum Schutz von Funktionszugriffen auf bestimmte Werte eingesetzt werden. Die prinzipielle Vorgehensweise wird durch Abbildung 4.5 veranschaulicht. Während ein Wertwächter bisher entweder einen Datenwert *oder* einen Funktionswert geschützt hat, besteht der geschützte Wert nun aus einem Tupel mit zwei Komponenten: dem Datenwert *und* dem Funktionswert. Es wird deutlich, daß nun selektiv jene Funktionen mit dem konkreten Wert aggregiert werden können, die der Zugriffskontrolle unterstellt werden sollen. Hieraus folgt insbesondere, daß Funktionen uneingeschränkt auf Werten ausgeführt werden können, für die kein geschütztes Tupel existiert. Somit wird erreicht, daß bestimmte Funktionen auf bestimmten Werten vom gesamten Benutzerkreis (alle Subjekte) ausgeführt werden können.
Das Konzept der Vernetzung von Wertwächtern durch Konnektoren bleibt weiterhin erhalten. An dieser Stelle sei betont, daß auch die Konnektoren zwischen Wertwächtern beliebig gesetzt werden können. Durch diese uneingeschränkte Konnektivität können nicht nur hierarchische (baumartige) Objekthierarchien modelliert werden, sondern auch Netzwerkstrukturen (*lattice*) oder zyklische Graphen. Somit ist die Modellierungs- und Ausdrucksmächtigkeit gegenüber bestehenden objektorientierten Zugriffsmodellen deutlich erweitert worden.
Die vorgestellten Anwendungsbeispiele zum Schutz von Werten beliebiger Komplexität durch Wertwächter und deren wahlfreie Vernetzung miteinander demonstrieren die Einsatzbereiche

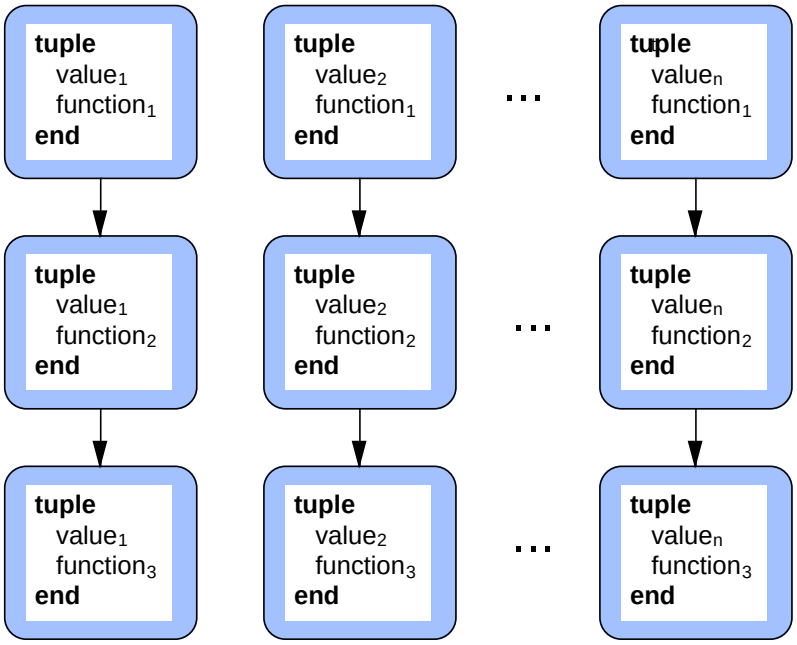


Abbildung 4.5: Schutz von Funktionszugriffen auf bestimmte Werte

des Basismodells. Wie gezeigt, läßt sich mit diesem Modell insbesondere jeder Funktionszugriff auf jeden Wert kontrollieren. Dieses Vorgehen gestaltet sich jedoch oftmals recht aufwendig, wie die Abbildung 4.5 erkennen läßt. Geschützt wird hier der Zugriff von drei konkreten Zugriffsfunktionen auf n konkrete Werte. Für den Schutz des ersten Wertes müssen jeweils drei Tupel, bestehend aus Wert und Zugriffsfunktion, gebildet werden, die dann von je einem Wertwächter geschützt werden. Die Wertwächter ihrerseits sind dann durch zwei Konnektoren miteinander verbunden. Für alle weiteren Werte ergibt sich das gleiche Szenario: Sämtliche Spezifikationsanweisungen sind zu wiederholen; lediglich der konkrete Wert als Komponente der drei Tupel ist unterschiedlich. Eine zentrale Forderung ist daher die Vereinfachung repetitiver Modellierungsvorgänge für den Benutzer.

4.1.5 Dynamische Schablonen und Implikationen

Im Kontext datenintensiver Anwendungen, in denen Container für Massendaten oft Tausende von Werten des gleichen Datentyps verwalten, ist die Modellierung geschützter Werte nach dem bisherigen Verfahren ineffektiv: Für jeden Wert, der der Zugriffskontrolle unterstellt werden soll, müssen sowohl sämtliche Funktionen in Wertwächter transformiert, als auch die Konnektoren zwischen diesen Wertwächtern erneut modelliert werden. Wird unterstellt, daß zum Schutz von Werten desselben Typs stets die gleichen Programmieranweisungen erforderlich sind, so muß bei der Konzipierung einer effektiven Zugriffskontrollbibliothek die Ausnutzung dieses Rationalisierungspotentials Berücksichtigung finden.
Die Bereitstellung von Schemata, mit denen geschützte Werte mit gleichen Eigenschaften erzeugt werden können, ist durch das Konzept **dynamischer Schablonen** realisiert worden. Eine dynamische Schablone ist ein Hilfsmittel, mit der eine beliebige Strukturdefinition, bestehend aus geschützten Werten eines beliebigen Typs und deren Beziehungen zueinander, vervielfältigt

werden kann. Alle Kopien, die aus einer dynamischen Schablone instanziiert werden, besitzen folglich eine identische Struktur von Wertwächtern und Konnektoren, die sie verbinden.

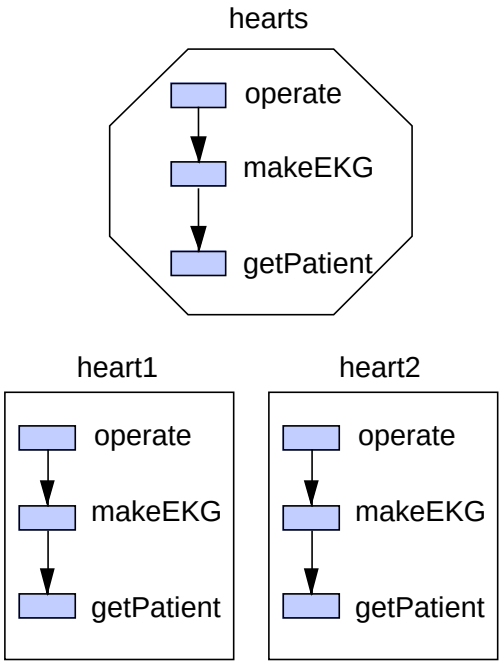


Abbildung 4.6: Dynamische Schablone mit zwei erzeugten Instanzen

Kopien einer dynamischen Schablone werden im folgenden als **Instanzen** bezeichnet; Vorlagen für geschützte Werte, die in einer Schablone verwaltet werden, werden fortan als **Quellwertwächter** (*Source-Valueguards*) bezeichnet.
Dynamische Schablonen dienen nicht nur der Erzeugung von Instanzen, sondern kontrollieren auch deren Struktur. Ändert sich in einer Schablone die Anzahl der Quellwertwächter oder deren Beziehungsgeflecht zueinander, so wirken sich diese Änderungen nicht nur auf alle neu erzeugten Instanzen aus; es kommt auch zu einer dynamischen Anpassung der Strukturdefinitionen aller bereits aus dieser Schablone erzeugten Instanzen. Die Struktur von Instanzen ist also nicht invariant, das heißt sie kann sich im Laufe der Zeit ändern. Auch kann nie eine Instanz ohne eine zugehörige Schablone existieren, da sie nur von dieser erzeugt und kontrolliert werden kann. Dynamische Schablone bieten somit folgende Funktionalität:

- Sie sind Vorlagen für die Erzeugung von Instanzen mit gleichen strukturellen Eigenschaften.
- Sie sind ein Kontrollorgan für die Durchführung konsistenter Strukturänderungen auf allen Instanzen, die von ihr erzeugt wurden.
- Sie tragen zu einer Strukturierung geschützter Werte bei: Während das Basismodell eine ungeordnete Vernetzung geschützter Werte beliebiger Typen gestattet, unterstehen nun Werte genau eines Typs der Kontrolle einer dynamischen Schablone.

Der Visualisierung des Konzeptes dynamischer Schablonen dient Abbildung 4.6. Für die Kontrolle von Werten eines Typs *Heart.T* ist eine dynamische Schablone (dargestellt durch ein Okta-

gon) erzeugt worden. Diese enthält drei Quellwertwächter (die Funktionen *operate, makeEKG* und *getPatient*), für die ein kontrollierter Zugriff auf alle Instanzen sichergestellt werden soll. Zwischen diesen Quellwertwächtern existieren zwei Konnektoren. Derartige Konnektoren, die genau zwei Quellwertwächter einer dynamischen Schablone miteinander verbinden, heißen **Implikationen**. Die Erzeugung einer neuen Instanz (*heart1* oder *heart2*) führt nun dazu, daß sich die Struktur der dynamischen Schablone automatisch auf sie überträgt. Wird der dynamischen Schablone *hearts* später ein weiterer Quellwertwächter hinzugefügt, so werden auch alle Instanzen, die bereits aus dieser Schablone erzeugt wurden, um einen Wertwächter erweitert.
Bezeichnet der Name einer Instanz gleichzeitig auch den Namen des Wertes, der vor nicht autorisierten Funktionszugriffen geschützt werden soll (was für den Rest dieses Kapitels angenommen wird) und wird weiterhin unterstellt, daß alle Funktionen in der Abbildung nur einen Eingabeparameter vom Typ (*Heart.T*) besitzen, so kontrolliert beispielsweise die Instanz *heart1* die Funktionsanwendungen *operate*(heart1), *makeEKG*(*heart1*) und *getPatient*(*heart1*).

4.1.6 Kreuzreferenzen zwischen dynamischen Schablonen

Die Möglichkeit, geschützte Werte mit Hilfe dynamischer Schablonen zu strukturieren, führt zunächst zu einer Beeinträchtigung der Modellierungsmächtigkeit. Die freie Konnektivität von Wertwächtern im Basismodell ist derart eingeschränkt worden, daß nur noch geschützte Werte eines bestimmten Typs miteinander verbunden werden können. In der realen Welt sind Objekte jedoch häufig in einer komplexeren Weise miteinander verknüpft. Ihr Beziehungsgeflecht umfaßt nicht nur Beziehungen zu gleichartigen, sondern auch zu artfremden Objekten. Folgerichtig ist die ursprüngliche Modellierungsmächtigkeit sukzessive wiederherzustellen. Für die Vernetzung von Wertwächtern verschiedener Schablonen wird daher das Konzept der **Kreuzreferenzen** eingeführt.
Eine Kreuzreferenz ist ein Konnektor, der je einen Quellwertwächter zweier verschiedener dynamischer Schablonen miteinander verbindet. Da dynamische Schablonen als Vorlagen für die Erzeugung und Änderung von Instanzen fungieren, bedeutet dies, daß auch Kreuzreferenzen auf die Ebene der Instanzen projiziert werden. Eine Kreuzreferenz bewirkt daher, daß je ein Wertwächter einer Instanz der ersten Schablone mit den korrespondierenden Wertwächtern aller Instanzen der zweiten Schablone verbunden wird.

Die Wirkungsweise veranschaulicht Abbildung 4.7. Zwischen den Quellwertwächtern *operate* der dynamischen Schablonen *hearts* und *appendices* existiert eine Kreuzreferenz. Daraus folgt auf der Ebene der Instanzen, daß zwischen den Wertwächtern *operate* der Instanzen *heart1* und *heart2* und den Wertwächtern *operate* der Instanzen *appendix1* und *appendix2* ebenfalls Beziehungen hergestellt werden.

4.1.7 Interinstanzreferenzen

Die Möglichkeit, Beziehungen zwischen Quellwertwächtern verschiedener dynamischer Schablonen durch Einfügen einer Kreuzreferenz herzustellen, ist ein mächtiges Modellierungsinstrument, um die Wertwächter der darunterliegenden Instanzen paarweise miteinander zu verknüpfen. Nicht immer läßt sich jedoch die Realität mit allgemeinen Regeln dieser Art beschreiben. Eine wichtige Ausnahme bildet die Modellierung komplexer Objekte (siehe Anhang A),

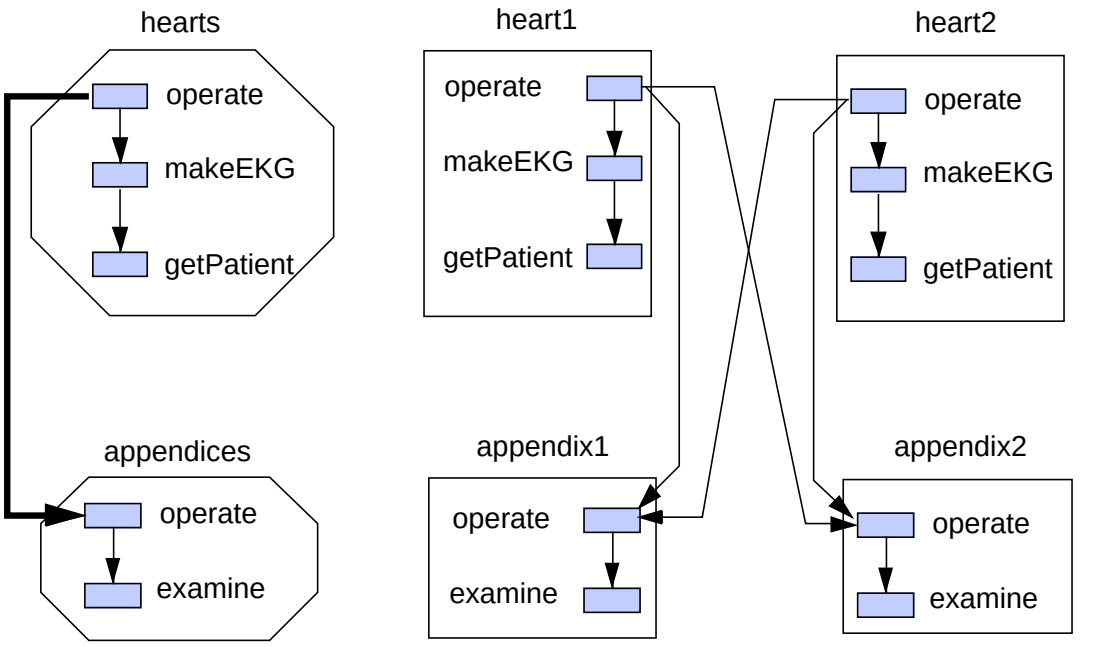


Abbildung 4.7: Kreuzreferenz zwischen Quellwertwächtern

wie Modulhierarchien, die neben einfachen Werten typische Zugriffseinheiten eines Programmentwicklungssystems bilden [Kel90].
Der Wunsch eines Benutzers ist es nun häufig, daß die explizite Autorisierung eines komplexen Objektes die implizite Autorisierung aller seiner transitiv erreichbaren Komponenten zur Folge hat. Die bisher vorgestellten Modellierungsvereinfachungen vermögen derartige Abhängigkeiten jedoch nicht, oder nur sehr umständlich, abzubilden. Aus Gründen der Orthogonalität muß daher auch die Vernetzung von geschützten Werten verschiedenen Typs auf der Ebene der Instanzen möglich sein.

Konnektoren dieser Art heißen **Interinstanzreferenzen**. Ein Beispiel für die Modellierung komplexer Objekte durch Interinstanzreferenzen zeigt Abbildung 4.8. Modelliert werden soll ein kombinierter Hör- und Sehtest, der in Form einer Checkliste implementiert ist. Ein solcher Test ist ein komplexes Objekt, das aus zwei Komponenten besteht: der Untersuchung der Augen und der Untersuchung der Ohren. Das Autorisierungssystem soll sicherstellen, daß die explizite Autorisierung für die Durchführung des Tests bei einer bestimmten Person die implizite Autorisierung für die Einzeluntersuchungen bei dieser Person nach sich zieht. Dieser Sachverhalt wird durch drei dynamische Schablonen (für Augen, Ohren und Sinnesorgane) mit je zwei Instanzen (*tom*, *pit*) abgebildet.

4.1.8 Intrainstanzreferenzen

Die Modellierung von Beziehungen zwischen Quellwertwächtern innerhalb einer dynamischen Schablone erfolgt durch Implikationen. Analog muß es möglich sein, auch Wertwächter innerhalb einer bestimmten Instanz miteinander zu verknüpfen. Auf diese Weise können, abweichend von der in einer dynamischen Schablone modellierten generellen Struktur, zusätzliche

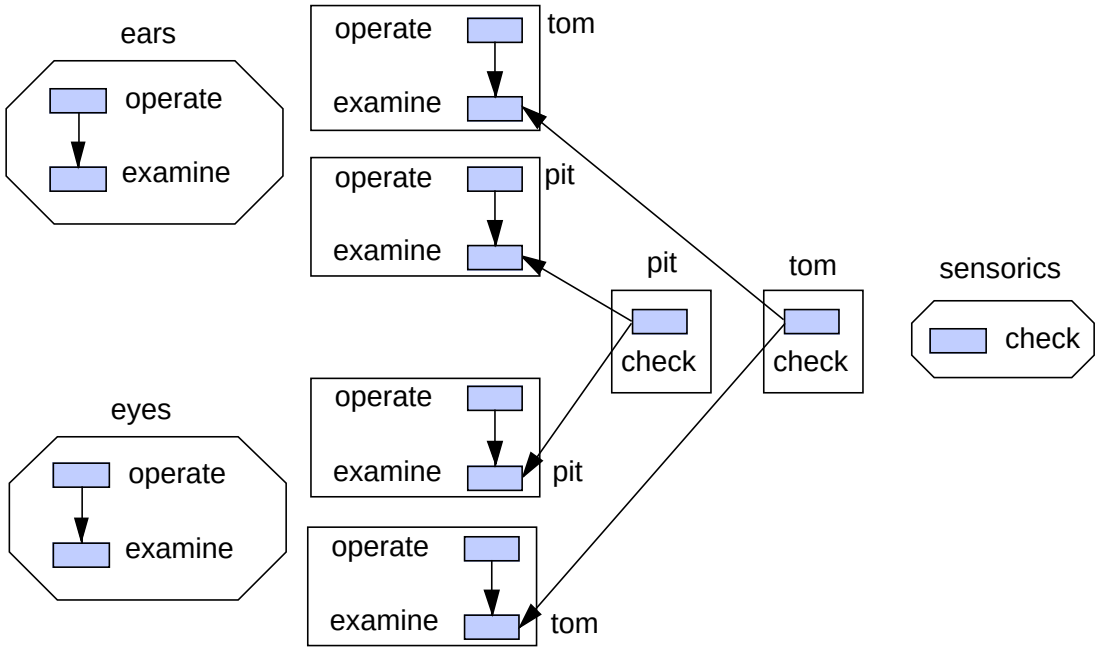


Abbildung 4.8: Modellierung komplexer Objekte

Konnektoren für einzelne Instanzen modelliert werden. Konnektoren dieser Art werden im folgenden als **Intrainstanzreferenzen** bezeichnet. Intrainstanzreferenzen zwischen Wertwächtern dürfen auch modelliert werden, wenn bereits eine Implikation zwischen ihren korrespondierenden Quellwertwächtern existiert. Diese Konvention erzwingt, daß eine solche Intrainstanzreferenz auch dann fortbestehen kann, wenn eine zu ihr entsprechende Implikation aus der dynamischen Schablone entfernt wird.

4.1.9 Isoinstanzreferenzen

Die Modellierungsmöglichkeiten für eine applikationsspezifische Vernetzung von Wertwächtern erstrecken sich bisher auf eine Verknüpfung von Wertwächtern innerhalb einer bestimmten Instanz (Intrainstanzreferenzen) und auf die Referenzierung des geschützten Wertes einer Instanz auf einen geschützten Wert unterschiedlichen Typs einer anderen Instanz (Interinstanzreferenzen).

Um die freie Konnektivität aller beliebigen Wertwächter untereinander wiederherzustellen, ist es erforderlich, auch noch eine Möglichkeit bereitzustellen, mit der Wertwächter verschiedener Instanzen der gleichen Schablone miteinander verbunden werden können. Konnektoren, die solche geschützten Werte des gleichen Typs verbinden, heißen **Isoinstanzreferenzen**.
Abbildung 4.9 zeigt ein Anwendungsbeispiel, in dem Isoinstanzreferenzen verwendet werden. Modelliert wird hier die Möglichkeit, Subjekten, denen der operierende Zugriff auf dem linken Auge eines konkreten Patienten erlaubt ist, auch den Zugriff für die *operate*-Funktion auf dem rechten Auge dieses Patienten zu gewähren (und umgekehrt). Eine analoge Beziehung besteht zwischen den Wertwächtern für die *examine*-Funktion. Gleichzeitig illustriert dieses Beispiel die Notwendigkeit, auch Zyklen modellieren zu können.

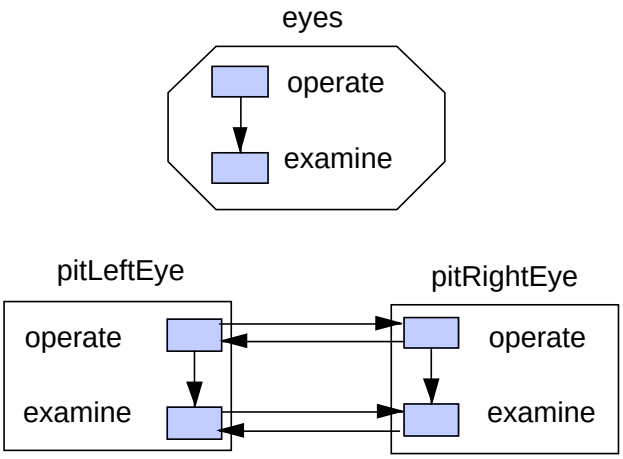


Abbildung 4.9: Isoinstanzreferenzen zwischen Wertwächtern

Die nachfolgende Tabelle faßt die verschiedenen Konnektoren, die für die Vernetzung von Wertwächtern beziehungsweise Quellwertwächtern eingesetzt werden können, überblicksartig zusammen und zeigt deren unterschiedliche Einsatzbereiche.

Implikation	verbindet Quellwertwächter einer Schablone
Kreuzreferenz	verbindet Quellwertwächter verschiedener Schablonen
Interinstanzreferenz	verbindet Wertwächter aus Instanzen verschiedener Schablonen
Isoinstanzreferenz	verbindet Wertwächter aus Instanzen derselben Schablone
Intrainstanzreferenz	verbindet Wertwächter innerhalb einer Instanz

4.1.10 Vererbungsrichtungen

Die bisherigen Ausführungen haben implizit unterstellt, daß die Zugriffskontrollinformation eines Wertwächters lediglich die zugriffsberechtigten Subjekte verwaltet. Analog zu den in Abschnitt 2.4.1 vorgestellten Basiskontrollabstraktionen wird im folgenden zu jedem Subjekt auch die Polarität (positiv oder negativ) und die Priorität (stark oder schwach) der erteilten Autorisierung verwaltet. Die Einführung positiver und negativer Autorisierung bedingt jedoch die Differenzierung in gleichgerichtete und gegengerichtete Vererbung (vergleiche Abschnitt 2.4.4). Während die bisherigen Betrachtungen eine gleichgerichtete Vererbung positiver und negativer Autorisierungen über Konnektoren unterstellt haben, erfolgt nun eine differenziertere Untersuchung. Zu diesem Zweck erhält jeder Konnektor eine Markierung (+ oder -), die Aufschluß über die Polarität der über ihn vererbbaren Rechte gibt.

Mit Hilfe dieser Erweiterung kann sowohl gleichgerichtete als auch gegengerichtete Vererbung abgebildet werden. Dies wird anhand von Abbildung 4.10 erläutert:
Diese Darstellung enthält vier Wertwächter (WW_1 bis WW_4), die durch Konnektoren miteinander verbunden sind. Zwei Wertwächter (WW_1 und WW_2) enthalten in ihrer Zugriffskontrollinformation positiv und negativ autorisierte Subjekte (Sub_1 bis Sub_4), deren Rechte über die Konnektoren weitervererbt werden. Hierbei wird davon ausgegangen, daß alle erteilten Autorisierungen die gleiche Priorität haben. Zwischen WW_1 und WW_2 existieren zwei Konnektoren,

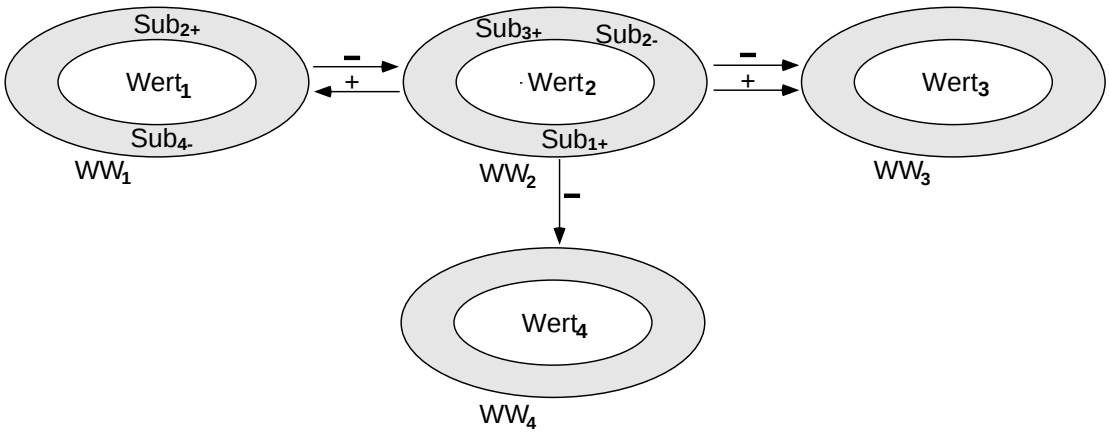


Abbildung 4.10: Modellierung gleich- und gegengerichteter Vererbung

die eine unterschiedliche Vererbungsrichtung aufweisen und mit unterschiedlichen Polaritäten markiert sind. Diese Struktur eignet sich zum Abbilden gegengerichteter Vererbung. Positive Autorisierungen werden nur von WW_2 nach WW_1 vererbt, negative Autorisierungen nur von WW_1 nach WW_2. Die Markierung der Konnektoren stellt insbesondere sicher, daß die konkurrierenden Autorisierungen für Sub_2 keinen Konflikt hervorrufen (siehe Abschnitt 4.1.11). Zwischen WW_2 und WW_3 besteht eine gleichgerichtete Vererbungsbeziehung, da die zwei Konnektoren zwischen diesen Wertwächtern mit unterschiedlichen Polaritäten markiert sind und die gleiche Vererbungsrichtung aufweisen.
Das Konzept, Konnektoren mit einer Polarität zu markieren, ist hinreichend mächtig und allgemein, um auch Vererbungsbeziehungen zu modellieren, die in objektorientierten Zugriffskontrollmodellen nicht spezifiziert werden können. Beispielsweise kann ein einzelner Konnektor modelliert werden, ohne daß gleichzeitig ein Konnektor entgegengesetzter Polarität spezifiziert werden muß. Dies verdeutlicht der Konnektor zwischen WW_2 und WW_4. Darüber hinaus besteht ebenfalls die Möglichkeit, zusätzlich zu einer gleichgerichteten oder gegengerichteten Vererbungsbeziehung einen weiteren Konnektor zu modellieren. Ein Beispiel wäre die Vererbung negativer Autorisierungen über einen (in der Abbildung nicht dargestellten) Konnektor, der von WW_3 nach WW_2 zeigt.
Bezugnehmend auf die Abbildung ergeben sich folgende Autorisierungen: Positiv autorisiert sind Sub_1 für WW_1, WW_2 und WW_3, Sub_2 für WW_1 und Sub_3 für WW_1, WW_2 und WW_3. Negative Autorisierungen besitzen Sub_2 für WW_2, WW_3 und WW_4 sowie Sub_4 für alle Wertwächter.

4.1.11 Konfliktmanagement

Die Fähigkeit objektorientierter Zugriffskontrolle, positive und negative Rechte modellieren zu können, birgt die Gefahr auftretender Konflikte in sich. Diese Gefahr entsteht dadurch, daß positive und negative Rechte gleichberechtigt sind und ihr simultanes Auftreten die Evaluation von Zugriffsanfragen zu einem konkreten eindeutigen Ergebnis unmöglich macht. Fehler dieser Art sind vom modellierenden Benutzer nicht gewollt und sollten daher so früh wie möglich erkannt werden. Zur Lösung des Problems ist eine Kontrolle in Form einer präventiven Konsistenzprüfung durchzuführen: Werden neue Rechte oder Konnektoren zwischen Wertwächtern in das

System eingetragen, so ist zunächst zu überprüfen, ob das Ergebnis der Modifikation in keinem Widerspruch zur bereits vorhandenen Struktur des Autorisierungsgraphen steht. Erst wenn diese Eigenschaft erfolgreich verifiziert ist, können alle durch die Veränderung ableitbaren Rechte im System abgespeichert werden. Mögliche Konflikte sind (in Anlehnung an [Brü93]):

- **Tatsächliche Konflikte**. Ein tatsächlicher Konflikt tritt auf, wenn sich für ein bestimmtes Subjekt zwei widersprüchliche starke oder zwei widersprüchliche implizite schwache Rechte ableiten lassen. Da im Rahmen dieser Arbeit positive und negative Rechte als gleichrangig angesehen werden, gibt es für diesen Antagonismus keine Lösungsstrategie. Die konfliktauslösende Aktion wird daher zurückgewiesen.

- **Grundsätzliche Konflikte**. Ein grundsätzlicher Konflikt entsteht, wenn sich für ein bestimmtes Subjekt ein starkes und ein widersprüchliches schwaches Recht ableiten lassen oder wenn ein explizites und ein implizites schwaches Recht verschiedener Polarität aufeinandertreffen. Die Frage der Zugriffsberechtigung kann in diesem Fall immer entschieden werden, da starke Rechte schwache Rechte dominieren und somit überschreiben und analog ein explizites schwaches Recht ein implizites schwaches Recht dominiert.

- **Latente Konflikte**. Ein latenter Konflikt entsteht, wenn ein tatsächlicher Konflikt durch einen grundsätzlichen Konflikt "maskiert" wird. Dies ist zum Beispiel der Fall, wenn sich für ein Subjekt zwei widersprüchliche schwache implizite Rechte ableiten lassen, die durch ein explizites starkes (oder schwaches) Recht überschrieben worden sind.

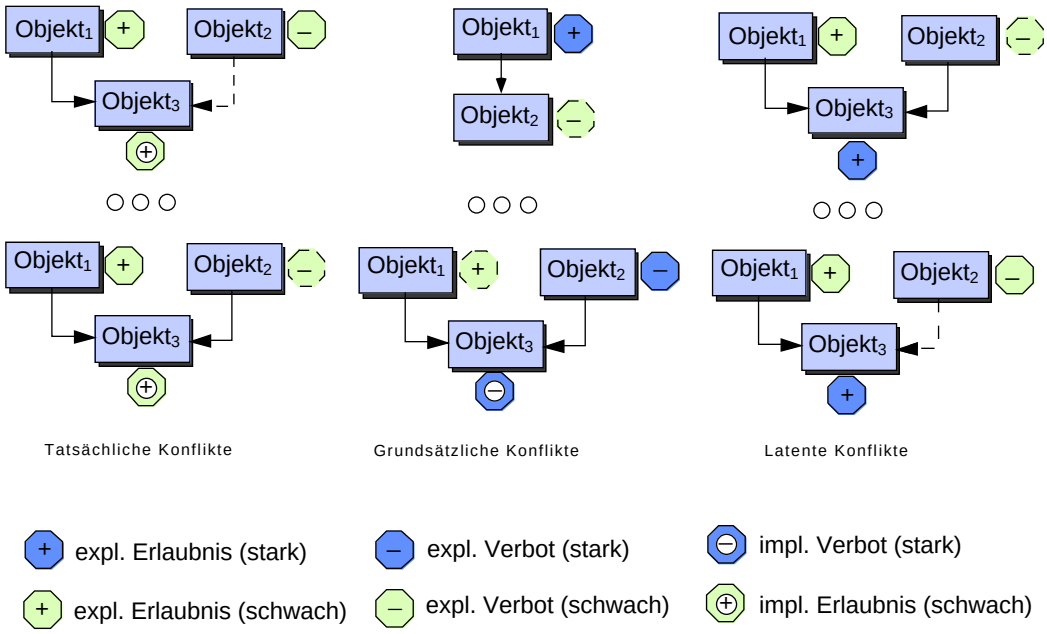


Abbildung 4.11: Konfliktarten

Beispiele für diese drei Konfliktarten zeigt Abbildung 4.11. Gestrichelte Rechte oder Vererbungspfeile repräsentieren dabei die konfliktauslösende Aktion; Subjekt und Zugriffsmodus werden als konstant angenommen. Im linken Teil der Abbildung führt die Vererbung des

schwachen expliziten Verbots von $Objekt_2$ zu $Objekt_3$ zu einem tatsächlichen Konflikt, da für $Objekt_3$ bereits eine schwache implizite Erlaubnis existiert. Konflikte dieser Art werden daher konterkariert.
Der mittlere Ausschnitt der Abbildung veranschaulicht grundsätzliche Konflikte. Das hinzugefügte schwache Recht hat jeweils keine Auswirkungen auf den Autorisierungsgraphen, da es vom Skopus des konkurrierenden starken Rechtes überdeckt wird. Konflikte dieser Art werden zugelassen, da sie aufgrund der höheren Priorität starker Rechte stets entscheidbar sind.
Der rechte Teil schließlich zeigt Beispiele für latente Konflikte. Für $Objekt_3$ existieren neben einer expliziten starken Erlaubnis sowohl eine positive als auch eine negative implizite schwache Autorisierung. Würde die starke explizite Erlaubnis für $Objekt_3$ widerrufen werden, so würden die konkurrierenden impliziten schwachen Autorisierungen einen tatsächlichen Konflikt erzeugen. Diese Operation ist daher zurückzuweisen.

4.1.12 Besitzerkonzept

Kennzeichnend für ein diskretes Zugriffskontrollmodell ist, daß die Vergabe und der Entzug von Zugriffsrechten auf Objekte im Ermessen des Objektbesitzers liegt (vergleiche Abschnitt 2.2). Eine benutzerbestimmbare Modellierung von Zugriffsrechten bedingt daher ein Besitzerkonzept. Im Kontext dieses Modells erfolgt die Vergabe des Besitzerstatusses auf Quellwertwächtern. Der Besitz eines Objektes ermächtigt ein Subjekt zur Modifikation der Zugriffskontrollinformationen des Quellwertwächters und aller aus diesem durch Instanziierung hervorgegangenen Wertwächter. Die Berechtigung zur Veränderung von Zugriffskontrollinformationen umfaßt insbesondere das Hinzufügen, Ändern und Löschen von zugriffsberechtigten Subjekten der Zugriffskontrolliste oder Propagierungshistorie, sowie das Verändern der Polarität oder Priorität erteilter Autorisierungen.
Auch das Recht zur Vernetzung von Quellwertwächtern oder Wertwächtern durch gerichtete Konnektoren ist eng mit dem Besitzerkonzept verknüpft. Es gilt die Vorgabe, daß ein Konnektor zwischen zwei (Quell-)Wertwächtern nur dann modelliert werden darf, wenn das modellierende Subjekt Besitzer beider Quellwertwächter ist. Es besteht jedoch die Möglichkeit, diese vorgegebene Regelung zu deaktivieren und durch eine andere (selbst implementierte) Strategie zu ersetzen.

4.2 Strukturierungshilfen für Subjekte

Neben Modellierungsvereinfachungen zur Verwaltung von geschützten Werten können gemäß der Überlegungen in Abschnitt 2.5 auch Subjekte strukturiert werden. Die folgenden Ausführungen stellen den gruppenorientierten Ansatz, der dem konzeptuellen Modell zugrundeliegt, vor.

4.2.1 Grupppenbildung

Für die Bildung von Gruppen erfolgt eine strikte Trennung hinsichtlich ihrer Komplexität. Es werden zwei Arten von Gruppen unterschieden:

1. **Basisgruppen**. Eine Basisgruppe ist eine Gruppe, die nur direkte Mitglieder enthält. Sie ist folglich nicht verschachtelt. Alle Mitglieder sind paarweise disjunkt.

2. **Komplexe Gruppen**. Komplexe Gruppen sind Gruppen, deren direkte Mitglieder wiederum Gruppen sind. Diese Mitglieder können Basisgruppen und/oder komplexe Gruppen, jedoch keine (unverschachtelten) Subjekte sein. Hieraus resultiert, daß komplexe Gruppen direkte und indirekte Mitglieder enthalten können.

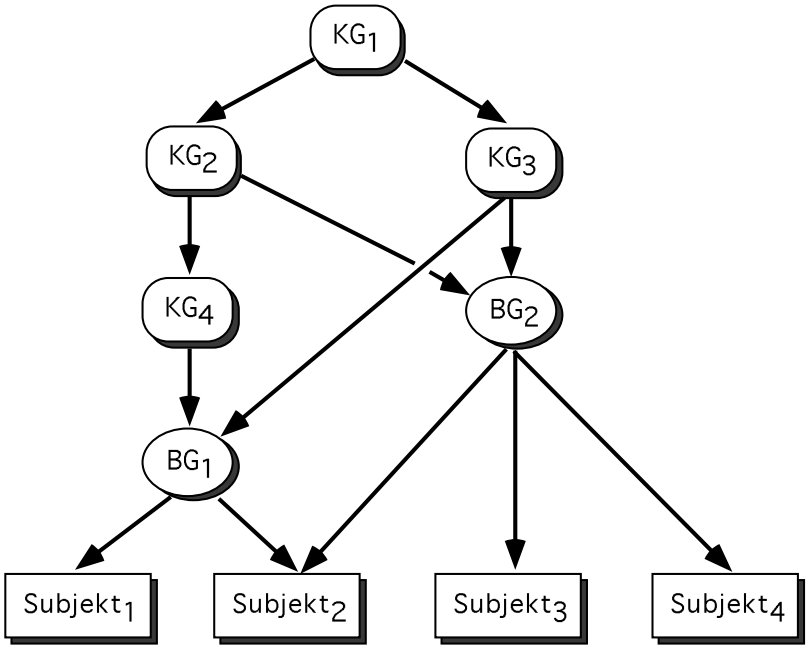


Abbildung 4.12: Beispiel für eine Subjekthierarchie

Dieses Vorgehen ermöglicht eine klare Strukturierung in Gruppen, die nur einzelne Subjekte enthalten, und solche, die nur Gruppen enthalten. Abbildung 4.12 zeigt ein Beispiel für eine Subjekthierarchie. Die Gruppen BG_1 und BG_2 repräsentieren Basisgruppen. Diese sind unverschachtelt und enthalten nur einfache Subjekte ($Subjekt_i$). Gruppen mit dem Präfix KG stellen komplexe Gruppen dar. Ihre direkten Mitglieder umfassen entweder nur Basisgruppen (KG_3, KG_4) oder nur komplexe Gruppen (KG_1) oder sowohl Basisgruppen als auch komplexe Gruppen (KG_2).

4.2.2 Gruppenstrukturpolitiken

Es können mehrere Klassifikationen für die Struktur von Gruppenhierarchien beziehungsweise einzelner Gruppen modelliert werden:

Disjunkt. Eine Gruppe heißt disjunkt, wenn ihre direkten Mitglieder paarweise verschieden sind. Diese Eigenschaft, von der gruppen- und rollenbasierte Zugriffskontrollmechanismen implizit ausgehen, wird bereits durch die Datenstruktur erzwungen, da die direkten Mitglieder einer Gruppe in einer Menge verwaltet werden.

Tief disjunkt. Um die Struktur disjunkter komplexer Gruppen weiter einzuschränken, besteht außerdem die Möglichkeit, diese Gruppen als tief disjunkt zu klassifizieren. Eine komplexe Gruppe ist tief disjunkt, wenn ihre direkten und indirekten Mitglieder paarweise verschieden sind. Diese Eigenschaft kann nicht durch die Datenstrukturen erzwungen werden. Sie wird daher durch Systemroutinen beim Einfügen einer Gruppe in eine andere Gruppe überprüft. Würde die tiefe Disjunktheit einer Gruppe durch die Einfügeoperation verlorengehen, so wird diese Operation zurückgewiesen. Jede tief disjunkte Gruppe ist gleichzeitig disjunkt.

Azyklisch. Gruppen- und Rollenhierarchien werden stets durch azyklische gerichtete Graphen beschrieben [Ste89]. Die Implementierung muß daher sicherstellen, daß durch Einfügen eines neuen Mitgliedes in eine Gruppe keine Zyklen entstehen. Diese Eigenschaft der Zyklenfreiheit wird über das Setzen eines entsprechenden Parameters erzwungen.

Zyklisch. Bei der Implementierung des Gruppenstrukturmoduls ist auch an eine mögliche Wiederverwendbarkeit gedacht worden. Die Funktionen und Datenstrukturen unterstützen daher auch den Aufbau und die Verwaltung zyklischer Strukturen. Im Kontext von Autorisierungsmaßnahmen findet diese Eigenschaft jedoch keine Anwendung, da Subjektstrukturen azyklisch sein müssen.

Kapitel 5

Implementierung in TL

Die Abbildung der vorgestellten Modellierungskonzepte in die Tycoon-Umgebung erfordert Überlegungen darüber, wie trotz der Einführung von Sicherheitsmechanismen eine möglichst hohe Performanz modellierter Anwendungen garantiert werden kann. Die folgenden Ausführungen werden daher zunächst darstellen, welche Überlegungen im Hinblick auf ein laufzeit- und zugriffseffizientes Verhalten in die Programmierung eingeflossen sind. Aufbauend auf den Konzepten des vorangegangenen Kapitels werden anschließend die wesentlichen Komponenten der Zugriffskontrollbibliothek vorgestellt. Diese lassen sich in die Kategorien Modellierung von Hierarchien geschützter Werte, Strukturierung von Subjekten, Verwaltung von Propagierungshistorien und Generatoren zur Abbildung sicherer Funktionen einteilen. Hierbei wird neben der Erläuterung der Modulstrukturen auf ausgewählte Implementierungsdetails, insbesondere die Datenmodellierung, eingegangen. Komplexe Beispiele, die den Einsatz und die Handhabung der einzelnen Bibliothekskomponenten schrittweise erläutern, bilden den Abschluß dieses Kapitels.

5.1 Effizienz

Die bisherigen Überlegungen haben sich fast ausschließlich mit der Frage der konsistenten Spezifikation von Benutzerrechten und dem Aufbau komplexer Subjekt- und Werthierarchien beschäftigt. Operationen dieser Art können sowohl statisch (vor Inbetriebnahme einer Anwendung) festgelegt werden, als auch dynamisch, das heißt im laufenden Betrieb einer Applikation, exekutiert werden. Hierbei handelt es sich um Funktionen, die einen modifizierenden Einfluß auf den Autorisierungsgraphen besitzen. Im Gegensatz dazu steht die Evaluierung von Anfragen an das Autorisierungssystem bezüglich der Berechtigung für einen konkreten Zugriff. Die Auswertung solcher Anfragen erfolgt stets zur Laufzeit, ruft jedoch keine Änderungen am Zustand des Autorisierungssystems hervor. Grundsätzlich kann konstatiert werden, daß im laufenden System die Zahl der Anfragen erheblich größer als die Zahl der Modifikationen sein wird. Daraus folgt, daß Anfragen an das Autorisierungssystem die Systemleistung signifikant beeinflussen, während die Modifikation von Zugriffskontrollinformation weniger zeitkritisch

ist [Kel91]. Da sich durch Einführung von Schutzvorkehrungen die Systemleistung notwendigerweise verringert, besteht eine zentrale Forderung darin, diesen Performanzverlust gering zu halten [LS87]. Auf der anderen Seite stellen objektorientierte Zugriffskontrollmodelle - und damit auch die daraus abgeleitete Tycoon Sicherheitsbibliothek - ein wirksames Instrument bereit, um den administrativen Aufwand zur Speicherung von Zugriffsrechten zu reduzieren, da alle Autorisierungen aus einer minimalen Menge von explizit gespeicherten Autorisierungen zur Laufzeit inferiert werden können [RBKW91].
Hierdurch entsteht offensichtlich ein Zielkonflikt zwischen minimalen räumlichen und minimalen zeitlichen Kosten. Da Speicherplatz und Zeit (in gewissen Grenzen) Substitute darstellen, ist es möglich, zwischen diesen beiden Zielen sorgfältig abzuwägen.
Steht die Minimierung der Zugriffszeit im Vordergrund, so sind beim Eintragen einer expliziten Autorisierung alle ableitbaren impliziten Autorisierungen zu erfassen und explizit mitzuspeichern. Die Anfrage nach einer Zugriffsberechtigung ist dann (bei wahlfreiem Zugriff) in konstanter Zeit (Zeitkomplexität: O(1)) zu beantworten. Bezeichnet W die Menge der geschützten Werte und S die Menge der Subjekte, so beträgt die Platzkomplexität zur Speicherung eines expliziten Rechtes und aller seiner impliziten Ableitungen im ungünstigsten Fall (*worst case*) - wenn die Erteilung des expliziten Rechtes Auswirkungen auf alle geschützten Werte und Subjekte hat - O($|W| \cdot |S|$).
Unter dem Gesichtspunkt der Reduzierung des Platzbedarfs ergeben sich entgegengesetzte Komplexitäten: Die Platzkomplexität beträgt O(1), da nur die explizite Autorisierung selbst gespeichert werden muß und sich alle anderen Autorisierungen zur Laufzeit daraus ableiten lassen. Die Evaluierung der Zugriffsberechtigung erfordert im ungünstigsten Fall O($|W| \cdot |S|$) Zugriffe, wenn alle Kombinationen von Subjekten und geschützten Werten zu überprüfen sind, um eine Zugriffsberechtigung oder ein Zugriffsverbot abzuleiten.
Im Zuge sinkender Hardware-Kosten für Massenspeicher einerseits und steigender Performanzanforderungen von Anwendungssoftware andererseits, steht für viele Anwendungen das Ziel, die zeitlichen Zugriffskosten gering zu halten, im Vordergrund. Hierbei ist es jedoch nicht sinnvoll, das zeitminimale Optimum zu realisieren: Subjekte, die in Gruppen organisiert sind, sollten bei der Rechtevergabe nicht aus diesen extrahiert werden, da eine Änderung der Gruppenstruktur extensive Änderungen der gespeicherten Zugriffsrechte hervorrufen würde. Dieses Vorgehen stellt keine wesentliche Einschränkung dar, da Gruppenhierarchien im allgemeinen recht kurz sind und das Ableiten von Zugriffsrechten entlang dieser Hierarchie somit nicht besonders zeitaufwendig ist [DHP89]. Es werden daher nur implizite Autorisierungen gespeichert, die sich entlang der Werthierarchie ableiten lassen. Die Speicherkomplexität beträgt somit O($|W|$) und die Zeitkomplexität beträgt (O($|S|$)).

Eine substantielle Forderung für das effiziente Laufzeitverhalten eines Autorisierungssystems ist nicht nur der geringe Zeitbedarf bei der Evaluierung einer Sicherheitsanfrage, sondern auch eine kurze Zugriffszeit auf die administrierten Rechte. Diese ist durch die Wahl der Speicherstruktur determiniert. Während die Speicherung oder das Auffinden von Daten in ungeordneten Listen im Mittel eine relativ teure Operation ist, ergeben sich für Streuspeicherverfahren (Schlüsseltransformation, *hashing*) nur geringe zeitliche Kosten. Abbildung 5.1 veranschaulicht die Zahl der mittleren Zugriffe für eine statische Streuspeichertabelle mit linearer Schlüsseltransformationsfunktion (*f(x)*), beziehungsweise einer Transformationsfunktion, die eine Gleichverteilung realisiert (*g(x)*).

Zwei Kritikpunkte, die der Benutzung einer statischen Streuspeicherstruktur entgegenstehen, sind zum einen die feste Größe der Tabelle, die den Benutzer zwingt, die Zahl seiner

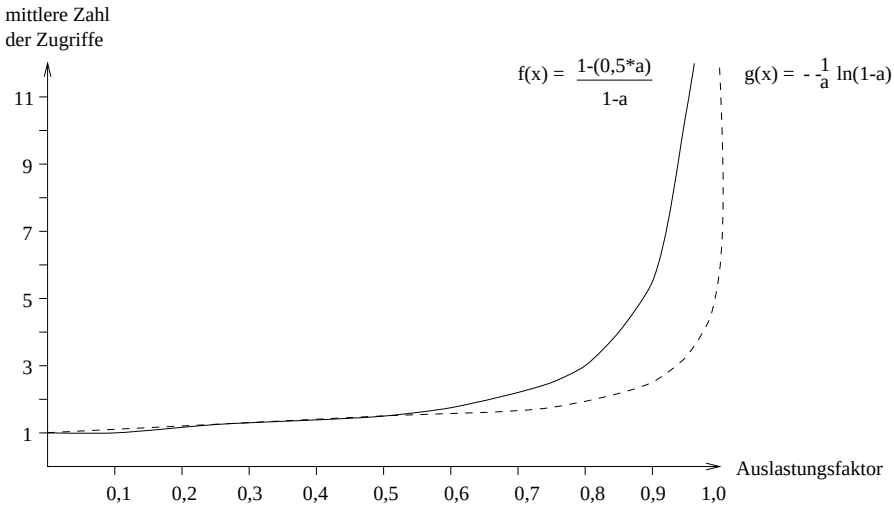


Abbildung 5.1: Mittlere Zugriffszahl bei statischer linearer und gleichverteilter Streuspeicherung für $n \to \infty$

zu speichernden Datenobjekte vorher abzuschätzen und die umständliche Durchführung von Löschoperationen [Wir86]. Zur Lösung dieser Probleme ist daher für die Realisierung der Autorisierungsbibliothek ein offenes dynamisches Streuspeicherverfahren gewählt worden. Ein offenes Verfahren ist dadurch gekennzeichnet, daß die durch Schlüsselkollision entstehenden Überlaufelemente (im Gegensatz zu einer externen Verkettung) in der Streuspeichertabelle selbst abgelegt werden. Der dynamische Aspekt berücksichtigt die Möglichkeit, Indexgrenzen verschieben zu können. Dies ist beispielsweise erforderlich, wenn aufgrund überproportional vieler Einfügeoperationen der Speicherplatz der Tabelle ausgeschöpft ist (vergleiche zum Beispiel [OW93]).
Aus Abbildung 5.1 geht jedoch hervor, daß sich das Zugriffsverhalten für hohe Auslastungsgrade der Streuspeichertabelle dramatisch verschlechtert. Lineare Streuspeicherung beispielsweise erfordert bei einem Auslastungsgrad von 99% im Mittel 50,5 Speicherzugriffe. Als Präventivmaßnahme werden die Indexgrenzen in der konkreten Implementierung daher bereits bei einem Füllungsgrad von 90% verschoben.

5.2 Modellierung von Wertkontrollstrukturen

Der prinzipielle Aufbau der Module für die Modellierung von Wertkontrollstrukturen ist in Abbildung 5.2 dargestellt. Der Aufbau dieser Module spiegelt das folgende Vorgehen bei der Implementierung wider: Abstraktionen, die für die Modellierung objektorientierter Zugriffskontrollmechanismen erforderlich sind, müssen erkannt und in adäquater Weise auf Typstrukturen abgebildet werden. Diese Abbildung erfolgt zum einen unter Ausnutzung bereits vorhandener Basisdienste wie Massendatentypen und Routinen zur Ein- und Ausgabe, die von der Tycoon-Umgebung zur Verfügung gestellt werden. Zum anderen wird sie durch selbst erstellte Hilfsfunktionen, wie eine Namensraumverwaltung, zugriffseffiziente Massenspeicherstrukturen und einen Dienst zur Verwaltung einfacher Zugriffskontrollisten, maßgeblich unterstützt. Als wesentliche Modellierungseinheiten lassen sich allgemeine Muster (dynamische Schablonen)

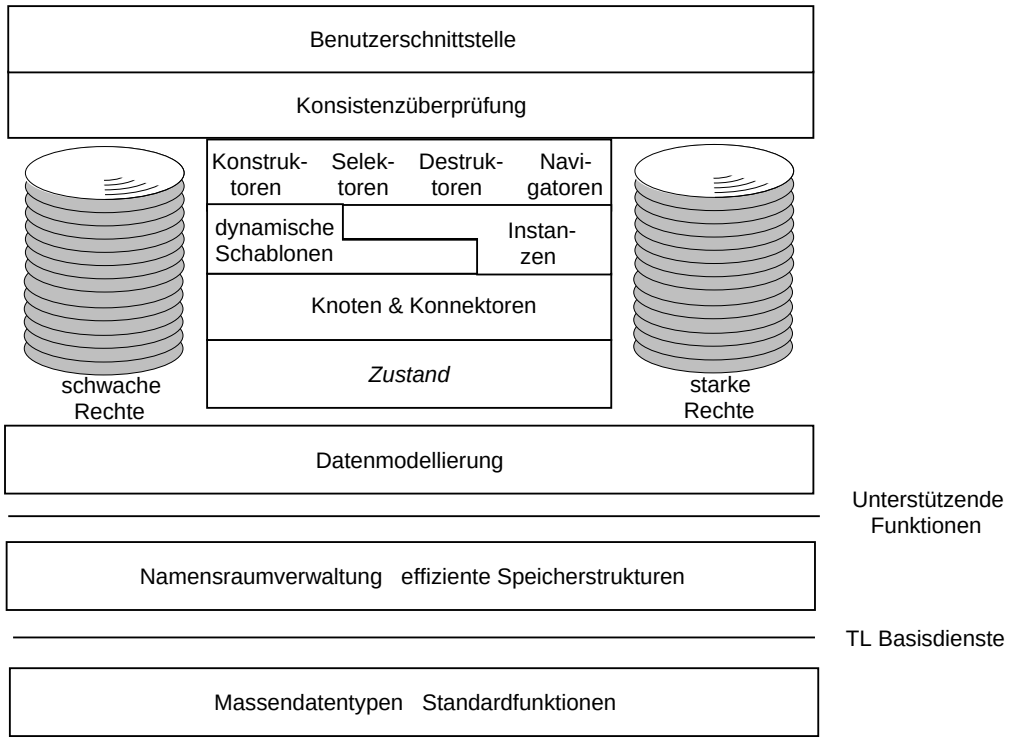


Abbildung 5.2: Grobarchitektur der Module zur Modellierung von Wertkontrollstrukturen

und spezielle Instanziierungen (Instanzen) extrahieren. Diese bestehen wiederum aus Knoten (Quellwertwächtern beziehungsweise Wertwächtern) und Konnektoren zwischen diesen Knoten (Implikationen und Kreuzreferenzen beziehungsweise Inter-, Iso- und Intrainstanzreferenzen). Zwischen dynamischen Schablonen und Instanzen bestehen starke Interdependenzen, da dynamische Schablonen zum einen für die Erzeugung und korrekte Verwaltung von Instanzen zuständig sind und zum anderen jede Instanz für die Anfrage nach ihrer aktuellen Struktur auf die sie erzeugende dynamische Schablone zugreifen muß.
Werden die aus der Datenmodellierung resultierenden Datenstrukturen mit Werten instanziiert, so reflektieren sie den Zustand des Autorisierungsgraphen. Änderungen an dessen Struktur werden über definierte Schnittstellen vorgenommen. Hierbei wird zwischen Konstruktoren, Destruktoren, Selektoren und Navigatoren unterschieden. Konstruktoren dienen der Erzeugung von Funktionseinheiten wie Knoten, Konnektoren, dynamischen Schablonen und Instanzen. Destruktoren stellen als Umkehroperation zu Konstruktoren die Invalidierung von Funktionseinheiten sicher. Weitere Mutatoren brauchen nicht bereitgestellt zu werden, da sich alle modifizierenden Operationen am Autorisierungsgraphen auf die Erzeugung oder Invalidierung reduzieren lassen. Selektoren sind Funktionen, mit denen Anfragen an die Graphenstruktur gestellt werden können (zum Beispiel zur Determinierung von Knotennamen oder ausgehenden Kanten). Sie haben somit keinen modifizierenden Einfluß auf den Zustand des Autorisierungssystems. Navigatoren dienen dazu unter Zuhilfenahme von Selektoren transitiv erreichbare Vorgänger- oder Nachfolgerknoten ausfindig zu machen. Sie sind wichtig, um zu bestimmen, welche Knoten vom Einfügen einer neuen Kante oder eines neuen Rechtes tangiert werden.
Die Rechteverwaltung erfolgt getrennt nach starken und schwachen Rechten. Die Ursache

hierfür liegt darin, daß die Administration schwacher Rechte höhere Anforderungen stellt als die Verwaltung starker Rechte (siehe weiter unten). Die so verwalteten Rechten bilden die Grundlage für die Entscheidung, ob ein Subjekt zugriffsberechtigt ist oder nicht. Im Vorgriff auf die nächsten Unterabschnitte sei verraten, daß **explizit** autorisierte Subjekte darüber hinaus auch in Zugriffskontrollisten am Knoten selbst eingetragen werden. Diese kontrollierte Redundanz wird bewußt verwendet, um die Navigationsfunktionen beim Auffinden der für einen bestimmten Knoten autorisierten Subjekte zu unterstützen. Auf die genauen Zusammenhänge wird weiter unten eingegangen.
Da jede Änderung eines widerspruchsfreien Autorisierungsgraphen diesen wiederum in einen widerspruchsfreien Zustand überführen soll, ist vor jedem Einfügen von neuen Rechten oder Konnektoren sowie einigen "Spezialfällen" (vergleiche Abschnitt 4.1.11) eine Konsistenzprüfung durchzuführen. Bei erfolgreicher Konsistenzprüfung werden alle Änderungen am Autorisierungsgraphen durchgeführt und die verwalteten Rechte um die aus dieser Änderung resultierenden, neu abgeleiteten oder entfernten Rechte aktualisiert. Die Benutzerschnittstelle schließlich stellt dem Anwender die gesamte Funktionalität der darunter liegenden Systemschichten zur Verfügung.

5.2.1 Datenmodellierung

Die Notwendigkeit, einerseits eine Menge von Instanzen als Attribut einer dynamischen Schablone zu modellieren und andererseits die erzeugende dynamische Schablone einer Instanz in dessen Datenstruktur zu verwalten, führt zu rekursiven Typabhängigkeiten. Da für die Wertwächter von Instanzen und die Quellwertwächter von dynamischen Schablonen darüber hinaus je eine Zugriffskontrolliste bereitgestellt werden muß, die Werte für Subjekte eines frei wählbaren Typs verwaltet, ergeben sich für die Modellierung Datenstrukturen mit rekursiven Typoperatoren. Das folgende Programmbeispiel stellt aus Gründen der Übersichtlichkeit nur die wichtigsten Datenstrukturen dar:

```
...
Let Rec Template(E, Subject <:Ok) <:TPublic(E Subject) =
 Tuple
    graph          :directedGraph.T(NValue  Implication) (* Abstrakter Datentyp *)
    vertices       :nameDB.T(graph.Node)
    edges          :nameDB.T(graph.Edge)
    ext            :NodeExtension(graph.Node  Subject)
    auth           :AuthStrategy
    linksFrom      :dictionary.T(ToType(Subject)  To(Subject))
    linksTo        :dictionary.T(FromType(Subject)  From(Subject))
    instances      :iter.T(Instance(E  Subject))
 end
...
and Valueguard (Subject <:Ok) <:Ok =
 Tuple
    vg                      :Element
    ctrlList                :extendedACL.T(Subject)
    weakCtrlList            :extendedACL.T(Subject)
    interConnectionsFrom :dictionary.T(InterInstCrossFrom(Subject)  InterTo(Subject))
```

```
    interConnectionsTo     :dictionary.T(InterInstCrossTo(Subject)  InterFrom(Subject))
    isoConnectionsFrom     :dictionary.T(IsoInstCrossFrom(Subject)  IsoTo(Subject))
    isoConnectionsTo       :dictionary.T(IsoInstCrossTo(Subject)  IsoFrom(Subject))
    intraConnectionsFrom :dictionary.T(IntraInstCrossFrom(Subject)  IntraTo(Subject))
    intraConnectionsTo     :dictionary.T(IntraInstCrossTo(Subject)  IntraFrom(Subject))
  end
...
and Instance(E, Subject <:Ok) <:Ok =
  Tuple
    parent          :Template(E  Subject)
    inst            :E
    valueguards     :dictionary.T(Valueguard(Subject)  Element)
    var valid       :Bool
    name            :String
  end
...
```

Grundsätzlich dient eine dynamische Schablone der Verwaltung von Werten eines beliebigen Typs. Dieser frei wählbare Typ wird dem Typoperator *Template* über den Typparameter *E* übergeben. Die Struktur einer dynamischen Schablone besteht aus einem Graphen, dessen Knoten Quellwertwächtern entsprechen und dessen Kanten Implikationen repräsentieren. Um die Graphenstrukturen einer dynamischen Schablone vor Manipulationen durch eine andere dynamische Schablone abzuschirmen, wird hierfür der als abstrakter Datentyp eines TL-Basisdienstes zur Verfügung gestellte Typ *directedGraph.T* benutzt. Die Eindeutigkeit von Knoten- und Kantennamen innerhalb einer dynamischen Schablone wird durch die Namensräume *vertices* und *edges* sichergestellt.
Für jeden Quellwertwächter einer dynamischen Schablone werden darüber hinaus zwei erweiterte Zugriffskontrollisten (für starke und schwache Rechte) bereitgestellt, sowie ein Besitzer dieses Knotens verwaltet, der berechtigt ist, Änderungen an diesen Zugriffskontrollisten vorzunehmen. Eine erweiterte Zugriffskontrolliste besteht aus einer Zugriffskontrolliste, in der positive Rechte verwaltet werden und einer, die die Verwaltung negativer Rechte gewährleistet. Die vom Benutzer frei wählbare Autorisierungsstrategie (*auth*) entscheidet darüber, ob bei einer fehlenden Zugriffserlaubnis und einem fehlenden Zugriffsverbot ein Zugriff erlaubt ist oder nicht. Durch diese Wahlmöglichkeiten können dynamische Schablonen offene oder geschlossene Autorisierungsteilsysteme abbilden (vergleiche Abschnitt 2.4.1).
Die Abbildung von Interdependenzen zwischen Quellwertwächtern verschiedener dynamischer Schablonen erfolgt getrennt nach eingehenden und ausgehenden Kreuzreferenzen in den Massendatentypen *linksFrom* und *linksTo* und schließlich verwaltet jede dynamische Schablone alle von ihr erzeugten und noch gültigen Instanzen, die im folgenden auch als **Extension** bezeichnet wird.
Viele der Informationen, die ein Wertwächter (*Valueguard*) benötigt, werden bereits durch Werte des Typs *Template* bereitgestellt. Um Redundanz zu vermeiden, konzentriert sich die Modellierung von Wertwächtern daher darauf, die noch fehlenden Informationen abzubilden. Der Typ *Element* des Attributs *vg* entspricht dem Typ **Ok** und damit der Objektidentität des geschützten Wertes. Die Diskriminierung verschiedener geschützter Werte ist somit gewährleistet. Mit Hilfe der Attribute *ctrlList* und *weakCtrlList* können explizite starke oder schwache Rechte modelliert werden, die nur für den Wertwächter selbst (und implizit für alle von ihm aus transitiv erreichbaren Wertwächter) gelten sollen. Inter-, Iso- und Intrainstanzreferenzen schließlich, werden

getrennt nach eingehenden und ausgehenden Konnektoren durch die letzten sechs Attribute des Typs *Valueguard* modelliert.
Die Modellierung einer Instanz (*Instance*) bedingt die Abbildung der folgenden Attribute: Da die generelle Struktur von Wertwächtern und deren Beziehungen zueinander für Instanzen in der sie erzeugenden Schablone abgelegt ist, ist ein Attribut zur Referenzierung dieser Schablone (*parent*) unumgänglich. Ebenso unverzichtbar ist ein Attribut für die Administration des zu schützenden Wertes (*Inst*). Darüber hinaus wird für jeden Quellwertwächter der erzeugenden Schablone eine Repräsentation des korrespondierenden Wertwächters in der Instanz benötigt. Dies ist zum einen erforderlich, um Wertwächter über Inter-, Iso- und Intrainstanzreferenzen miteinander vernetzen zu können und zum anderen, um den Wertwächtern Zugriffskontrollisten hinzuzufügen, die die expliziten Zugriffsrechte auf ihn verwalten (*valueguards, Valueguard*). Da referenzierbare Werte in Tycoon nicht gelöscht werden können (vergleiche Abschnitt 5.2.4), muß außerdem ein Attribut existieren, das Aufschluß über die Gültigkeit einer Instanz gibt. Eine als ungültig markierte Instanz wird automatisch aus der Extension der sie erzeugenden dynamischen Schablone entfernt. Schließlich können die verwalteten Werte noch mit einem Namen versehen werden.
Die Modellierung verdeutlicht, daß alle Konnektoren, die bereits in einer dynamischen Schablone werwaltet werden (Kreuzreferenzen, Implikationen), nicht mehr in den Instanzen der Extension dieser Schablone administriert werden müssen.

5.2.2 Rechteverwaltung

Die Verwaltung expliziter und daraus ableitbarer impliziter Rechte erfolgt in vier Streuspeichertabellen (*hash tables*). Zum einen erfolgt eine Diskriminierung hinsichtlich der Priorität der Rechte (stark oder schwach); zum anderen wird unterschieden, ob das verwaltete Recht aus der expliziten Autorisierung eines Quellwertwächters (Recht, das für die korrespondierenden Wertwächter aller Instanzen gilt) oder eines Wertwächters abgeleitet worden ist. Diese strikte logische Trennung führt zu einer klaren Strukturierung der administrierten Rechte und erhöht die Zugriffsgeschwindigkeit auf die gespeicherten Rechte, da nun nicht mehr in einer großen Streuspeichertabelle gesucht werden muß.
Der folgende Programmausschnitt illustriert, welche Datentypen der Verwaltung schwacher Rechte, die sich aus explizit autorisierten Wertwächtern ableiten lassen, zugrunde liegt:

```
Let FromInst(Subject <:Ok) =
  Tuple
    n     :NVal
    i     :Instance(Subject)
  end

Let ImplicitInst(Subject <:Ok) =
  Tuple
    explicit    :FromInst(Subject)
    implicit    :set.T(FromInst(Subject))
  end

Let WeakInstHashType(Subject <:Ok) =
  Tuple
```

```
    node      :NVal
    inst      :Instance(Subject)
    subject   :Subject
    var pos   :Bool
    from      :dictionary.T(ImplicitInst(Subject)  FromInst(Subject))
  end

Let WeakHashKey(Subject <:Ok) =
  Tuple
    node      :NVal
    inst      :Instance(Subject)
    subject   :Subject
  end
```

Ein Wert von Typ *WeakHashKey* reicht aus um zu prüfen, ob ein bestimmtes Subjekt Zugriff auf den Wert eines Wertwächters (repräsentiert durch die Attribute *node* und *inst*) hat. Gleichzeitig identifiziert ein solcher Schlüssel gegebenenfalls eindeutig einen Wert des Typs *WeakInstHash-Type*, der in der Streuspeichertabelle verwaltet wird. Die beiden zusätzlichen Attribute (*pos* und *from*) weisen dabei auf zwei besondere Notwendigkeiten bei der Modellierung hin:
Erstens kann sich durch eine modifizierende Operation die Polarität (positiv oder negativ) einer schwachen Autorisierung ändern. Dies ist beispielsweise der Fall, wenn eine explizite negative schwache Autorisierung gelöscht wird und eine durch diese Autorisierung überschriebene implizite positive schwache Autorisierung wieder "sichtbar" wird. Das Attribut *pos*, das die Polarität eines Rechtes repräsentiert, muß daher variabel sein. Zweitens muß die Herkunft eines abgeleiteten Rechtes mitverwaltet werden, wofür das Attribut *from* vorgesehen ist. Dabei ist zu beachten, daß sich die Autorisierung eines Subjektes für den Zugriff auf einen Wertwächter sowohl aus mehreren expliziten Autorisierungen ableiten lassen kann als auch die Möglichkeit, daß ein explizit vergebenes Recht einen Wertwächter aufgrund verschiedener Vererbungspfade mehrfach autorisiert.

5.2.3 Navigatoren

Wesentlich für die Durchführung einer Konsistenzprüfung ist das Auffinden und die Erfassung aller Wertwächter, die durch eine modifizierende Operation (beispielsweise dem Einfügen oder Löschen eines expliziten Rechtes oder eines Konnektors) tangiert werden. Diese ist Aufgabe der Navigatoren. Als vorteilhaft erweist es sich hierbei, neben den so gefundenen Wertwächtern auch deren direkte Vorgänger mitzuerfassen, da ein explizites Recht über mehrere Pfade zu einem anderen Wertwächter propagiert werden kann. Diese Information wird später für die Modifikation der administrierten Rechte benötigt (siehe Abschnitt 5.2.2). Daher sollen die Navigatoren jeweils eine Menge solcher Paare von Wertwächtern zurückgeben. Wenn in diesen Paaren der eine Wertwächter als Startwertwertwächter und der andere als Zielwertwächter aufgefaßt wird, kann ein solches Paar auch als Darstellung für einen Konnektor angesehen werden. Konzeptuell kann ein solcher Konnektor beliebige Wertwächter miteinander verbinden. Hieraus folgt insbesondere, daß ein Konnektor auch Wertwächter verbinden kann, die unter der Kontrolle verschiedener dynamischer Schablonen stehen (zum Beispiel Interinstanzreferenzen). Diese dynamischen Schablonen sind in der Regel mit unterschiedlichen Typen parametrisiert. Damit hängt ein Konnektor von zwei Typen ab. In einem ersten Ansatz könnte dies zur Verwendung

eines Typoperators mit zwei Typparametern (Konnektortypoperator) führen. Bei der Verwendung eines so definierten Typoperators zur Bestimmung des Elementtyps einer Menge würde dies jedoch nur dann funktionieren, wenn die Aktualtypparameter der dynamischen Schablone des Startwertwächters und des Zielwertwächters jeweils für alle Konnektoren gleich wären (homogene Menge). Da Konnektoren jedoch beliebige Wertwächter verbinden können, ist dies meist nicht der Fall. Daraus folgt also insbesondere, daß die Aktualtypparameter der dynamischen Schablone nicht direkt zur Parametrisierung des Konnektortypoperators verwendet werden können. Sie müssen daher in dem durch den Konnektortypoperator konstruierten Typen direkt verwaltet werden:

```
Let ... =
 Tuple
    E <:Ok
    ...
    EFrom <:Ok
    ...
 end
```

Werte der Typen *E* und *EFrom* selbst können dabei nur eingeschränkt verwendet werden, da von ihnen aus statischer Sicht (zum Übersetzungszeitpunkt) nur bekannt ist, daß sie ein Subtyp von **Ok** sind. Dies reicht jedoch aus, um sie zur Instanziierung von Typoperatoren zu benutzen.

```
Let InstWithImpl(Subject <:Ok) =
 Tuple
    E <:Ok
    t           :template.T(E  Subject)
    node        :t.graph.Node
    inst        :instance.T(E  Subject)
    EFrom <:Ok
    tfrom       :template.T(EFrom  Subject)
    nodefrom    :tfrom.graph.Node
    instfrom    :instance.T(EFrom  Subject)
 end
```

Die in den Typoperatoren (*template.T(..), instance.T(..)*) verwendeten Signaturen sind jetzt statisch bekannt. Damit ist es im folgenden möglich, daß die strukturelle Information von Werten mit so konstruiertem Typ auch genutzt werden kann. Der Typoperator *InstWithImpl* (Instanzknoten mit implizitem Vorgänger) ist daher geeignet, auch heterogene Mengen von Konnektoren zu verwalten.
Die nachfolgende Funktion illustriert, wie ausgehend von einem beliebigen gegebenen Wertwächter, alle seine transitiv erreichbaren nachfolgenden Wertwächter (und deren direkte Vorgänger) ausfindig gemacht werden können.

```
let getLower(Subject <:Ok  i :InstWithImpl(Subject)
 pos :Bool) :iter.T(InstWithImpl(Subject)) =
 begin
   let marked = set.new(:InstWithImpl(Subject)
```

```
  fun(a,b :InstWithImpl(Subject)) :Bool instImplEqual(:Subject a b))
 let rec work(j :InstWithImpl(Subject)) :Ok =
  if
   set.member(:InstWithImpl(Subject) j marked)
  then
   ok
  else
   set.insert(:InstWithImpl(Subject) marked j)
   let next = findNode.findAllNext(:Subject j pos)
   iter.forEach(:InstWithImpl(Subject) next
   fun(a :InstWithImpl(Subject)) :Ok work(a))
  end
 work(i)
 set.elements(:InstWithImpl(Subject) marked)
end
```

Die Fähigkeit auch zyklische Autorisierungsabhängigkeiten durch Module der Zugriffskontrollbibliothek abbilden zu können, bedingt die Markierung und Administration bereits erfaßter Konnektoren beim Traversieren eines Autorisierungsgraphen. Markierte Konnektoren, die in der Menge *marked* verwaltet werden, bilden die Vorraussetzung für die Durchführung einer rekursiven Tiefensuche, die der Funktion *getLower* zugrundeliegt: Zunächst erfolgt eine Überprüfung, ob der als Startpunkt übergebene Wertwächter bereits markiert wurde. Ist dies der Fall, so bricht der Algorithmus ab, anderenfalls wird der Wertwächter markiert. Dieser Markierung schließt sich eine Bestimmung seiner direkt (in einem Ableitungsschritt) erreichbaren Nachfolgewertwächter an (Funktion *findNode.findAllNext*). Diese sind durch Werte des Typs *InstWithImpl* repräsentiert, wobei als direkter Vorgänger der Startwert eingetragen wird. Mit jedem der auf diese Weise neu gefundenen Wertwächter wird die Funktion (*work*) erneut aufgerufen. Der Algorithmus bricht ab, wenn alle erreichbaren Kombinationen aus einem Wertwächter und dessen direkten Vorgänger markiert sind. Die Markierungsmenge entspricht gleichzeitig dem Rückgabewert.

5.2.4 Invalidierung von Funktionseinheiten

Im Tycoon-System unterliegt die gesamte Speicherverwaltung der Objektspeicherschnittstelle TSP. Diese Abstraktionsbarriere zum konkreten darunterliegenden Betriebssystem hat für den Benutzer den Vorteil, daß er sich weder um die Allokation noch um die Freigabe von Speicherplatz für die von ihm erzeugten oder gelöschten Objekte kümmern muß. Objekte werden im Speicher gehalten, solange noch mindestens eine Referenz auf sie verweist (Erreichbarkeitskriterium). Anderenfalls werden sie aus dem Speicher entfernt (*garbage collection*). Im Rahmen der Realisierung der Autorisierungsbibliothek ist es von Interesse, auch Funktionen anzubieten, mit denen Instanzen oder ganze Schablonen gelöscht werden können. Dies ist jedoch nicht möglich, da sich Benutzer nicht ihrer benannten Referenzen auf diese Funktionseinheiten entledigen können. Der Löschvorgang muß daher adäquat simuliert werden können.
Hierzu ist jede Instanz und jede Schablone bei der Datenmodellierung mit einem booleschen Attribut versehen worden, das Aufschluß über die Gültigkeit des Objektes gibt. Soll beispielsweise eine Instanz gelöscht werden, so wird sie zunächst aus der Extension der Schablone entfernt. Anschließend werden alle Referenzen, die auf Wertwächter anderer Instanzen bestehen

sowie alle nicht mehr gültigen Rechte gelöscht. Abschließend wird die gesamte Instanz durch Änderung des booleschen Attributes invalidiert. Modifikationen, die in kausalem Zusammenhang mit dieser Instanz stehen, können nun nicht mehr durchgeführt werden. Der Vorgang der Invalidierung ist nicht reversibel, das heißt eine als ungültig markierte Instanz kann nicht wieder Gültigkeit erlangen.

5.2.5 Konsistenzprüfung

Wie in Abschnitt 2.4.1.2 erwähnt, eignen sich grundsätzlich zwei Strategien für die Konsistenzprüfung: Eine Strategie besteht darin, die durch eine Operation hervorgerufenen Änderungen zu erfassen und hinsichtlich ihrer Widerspruchsfreiheit zu überprüfen, bevor diese Änderungen tatsächlich durchgeführt oder im Fehlerfall verworfen werden. Die andere Strategie sieht vor, zunächst alle Änderungen durchzuführen und danach eine Konsistenzprüfung vorzunehmen. Wird dabei ein Fehler erkannt, so müssen alle durchgeführten Änderungen zurückgesetzt werden, um den alten konsistenten Zustand wiederherzustellen.
Die gewählte Implementierung verfolgt den ersten Ansatz. Dieser zeigt gegenüber dem zweiten Ansatz ein besseres Laufzeitverhalten, da beim Erkennen eines Widerspruchs keine zusätzlichen zeitlichen Kosten entstehen, um bereits getätigte Änderungen rückgängig zu machen.
Wird ein Widerspruch erkannt, so wird die konfliktauslösende Operation stets zurückgewiesen. In den meisten Fällen erhält der Systembenutzer eine detaillierte Warnung, die den erkannten Konflikt anzeigt und plausibel erläutert. Grundsätzlich können Widersprüche (tatsächliche Konflikte) bei folgenden Operationen auftreten:

1. Einfügen eines starken oder schwachen Rechtes, sofern dieses in Widerspruch mit einem Recht an dem Wertwächter selbst oder an einem von diesem Wertwächter aus transitiv erreichbaren Nachfolgewertwächter steht.

2. Einfügen eines Konnektors, wenn die Rechte an den von dessen Startpunkt aus erreichbaren Vorgängerknoten in Widerspruch mit den Rechten an den von dessen Zielpunkt aus erreichbaren Nachfolgeknoten stehen. Beim Einfügen von Kreuzreferenzen gilt zusätzlich, daß auch die einzufügenden Rechte der transitiv erreichbaren Vorgängerknoten in keinem Widerspruch zueinander stehen dürfen.

3. Einfügen der ersten Instanz einer dynamischen Schablone, wenn bereits Kreuzreferenzen zu anderen dynamischen Schablonen bestehen und sich auf diese Weise ein Kantenzug zwischen widersprüchlichen Rechten bildet, oder wenn in der erzeugenden dynamischen Schablone widersprüchliche Rechte an Quellwertwächtern stehen, die durch eine Implikation miteinander verbunden sind. Diesen Sachverhalt veranschaulicht exemplarisch die Abbildung 5.3. Dargestellt sind drei dynamische Schablonen mit je einem Quellwertwächter (QWW_1, QWW_2 und QWW_3). Die erste und dritte dynamische Schablone haben je eine Instanz; die zweite dynamische Schablone besitzt keine Instanz. Die Quellwertwächter QWW_1 und QWW_3 sind über zwei Kreuzreferenzen miteinander verbunden und mit zwei starken Rechten unterschiedlicher Polarität versehen. Hierdurch ist eine Konfliktsituation entstanden. Da Kreuzreferenzen stets Wertwächter aus Instanzen verschiedener dynamischer Schablonen miteinander verbinden, Instanzen der zweiten Schablone jedoch nicht existieren, handelt es sich um einen latenten Konflikt; die Konsistenz der Graphstruktur ist (noch) gewahrt. Wird nun die erste Instanz der zweiten dynamischen Schablone erzeugt und damit auch die Wertwächter miteinander verbunden,

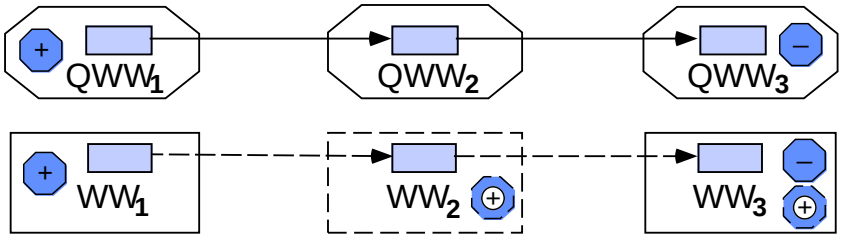


Abbildung 5.3: Konflikt beim Einfügen der ersten Instanz

so entsteht ein Kantenzug zwischen den Wertwächtern WW_1 und WW_3. Wertwächter WW_3 besitzt daraufhin eine implizite starke Zugriffserlaubnis und ein explizites starkes Zugriffsverbot; dies entspricht einem tatsächlichen Konflikt. Die Erzeugung der Instanz muß somit zurückgewiesen werden.

4. Löschen eines expliziten starken oder schwachen Rechtes, wenn am explizit autorisierten Knoten ein latenter Konflikt vorliegt, oder wenn die durch das Löschen weitervererbten impliziten Rechte an diesem Knoten in Widerspruch mit schwachen Rechten an einem der Nachfolgeknoten stehen. Zur näheren Erläuterung wird Abbildung 5.4 herangezo-

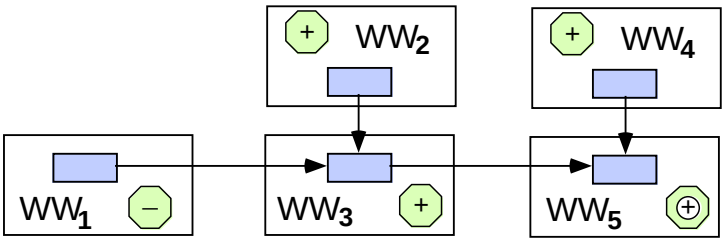


Abbildung 5.4: Tatsächlicher Konflikt beim Löschen von Autorisierungen

gen. Hier ist ein latenter Konflikt dargestellt, der durch die widersprüchlichen schwachen Autorisierungen an den Wertwächtern WW_1 und WW_2, die durch die explizite schwache Autorisierung am Wertwächter WW_3 überschrieben sind, entstanden ist. Die Autorisierung an Wertwächter WW_3 kann daher nicht gelöscht werden, weil sonst durch die resultierenden widersprüchlichen impliziten Autorisierungen am Wertwächter WW_3 ein tatsächlicher Konflikt entstünde.
Auch nachdem die positive Autorisierung am Wertwächter WW_2 gelöscht worden ist, kann die positive Autorisierung am Wertwächter WW_3 nicht gelöscht werden, da dann durch widersprüchliche implizite Autorisierungen ein tatsächlicher Konflikt am Wertwächter WW_5 entsteht.

5.2.6 Modellierung gegenläufiger Vererbung

Der Einsatz objektorientierter Zugriffskontrollmechanismen wirft das Problem auf, neben gleichgerichteter Vererbung für positive und negative Autorisierungen auch gegengerichtete Vererbungsrichtungen spezifizieren zu können (vergleiche Abschnitt 2.4.4). Eine Lösung wird

dadurch erreicht, daß jeder Konnektor um ein Attribut *pos* erweitert wird, das angibt, ob jeweils nur positive oder nur negative Rechte über diesen Konnektor vererbt werden sollen.

```
Let Implication =
 Tuple
   name        :String
   var valid   :Bool
   pos         :Bool
 end
```

Eine gleichgerichtete Vererbung zwischen zwei Wertwächtern A und B wird dadurch erreicht, daß **zwei** Konnektoren mit gleicher Richtung und unterschiedlicher Polarität modelliert werden ($A \xrightarrow{+} B$ und $A \xrightarrow{-} B$). Die Modellierung gegengerichteter Vererbung wird entsprechend erreicht, wenn zwei Konnektoren mit unterschiedlicher Richtung und unterschiedlicher Polarität spezifiziert werden ($A \xrightarrow{+} B$ und $A \xleftarrow{-} B$).
Darüber hinaus erweitert die gewählte Implementierung die Modellierungsmächtigkeit gegenüber objektorientierten Zugriffskontrollmodellen. Letztere erwarten stets die Angabe einer Vererbungsrichtung für Rechte beider Polaritäten. Bei dem hier gewählten Ansatz können außerdem Konnektoren für die Vererbung von Rechten einer Polarität modelliert werden, ohne daß gleichzeitig ein Konnektor für die Vererbung von Rechten der entgegengesetzten Polarität spezifiziert werden muß.

5.3 Aufbau von Subjekthierarchien

Abbildung 5.5 stellt den prinzipiellen Aufbau der Module zur Konstruktion von Subjekthierarchien dar: Die Abbildung von Subjekthierarchien in die Implementationssprache TL erfolgt unter Ausnutzung von TL-Basisdiensten, wie Massendatentypen und Ein- und Ausgaberoutinen, sowie selbst erstellten Hilfsfunktionen, wie einer Namensraumverwaltung für die modellierten Subjekte. Ausgangspunkt für die Implementierung ist die Modellierung adäquater Datenstrukturen. Aufbauend auf diesen Strukturen werden Primitive zur Erzeugung einfacher und verschachtelter Gruppen und zur Spezifikation ihrer hierarchischen Beziehungen zueinander bereitgestellt. Der lesende oder modifizierende Zugriff auf den Zustand dieser Gruppen erfolgt über fest definierte Operationen und Anfragen. Operationen, die den Zustand von Gruppen oder deren hierarchische Beziehungen zueinander verändern, unterliegen einer Konsistenzprüfung. Diese weist jene Operationen zurück, die zu einer Verletzung der spezifizierten Gruppenstrukturpolitik führen. Die Benutzerschnittstelle schließlich, die in Form eines abstrakten Datentyps realisiert ist, bietet die gesamte Funktionalität der darunterliegenden Systemschichten nach außen an. Die nachfolgenden Betrachtungen greifen die wichtigsten Modulschichten heraus. Eine komplette Beschreibung der Benutzerschnittstelle befindet sich in Anhang D.1.

5.3.1 Datenmodellierung

Der folgende Programmausschnitt präsentiert die grundlegenden Datenstrukturen für die Modellierung von Basisgruppen und komplexen Gruppen.

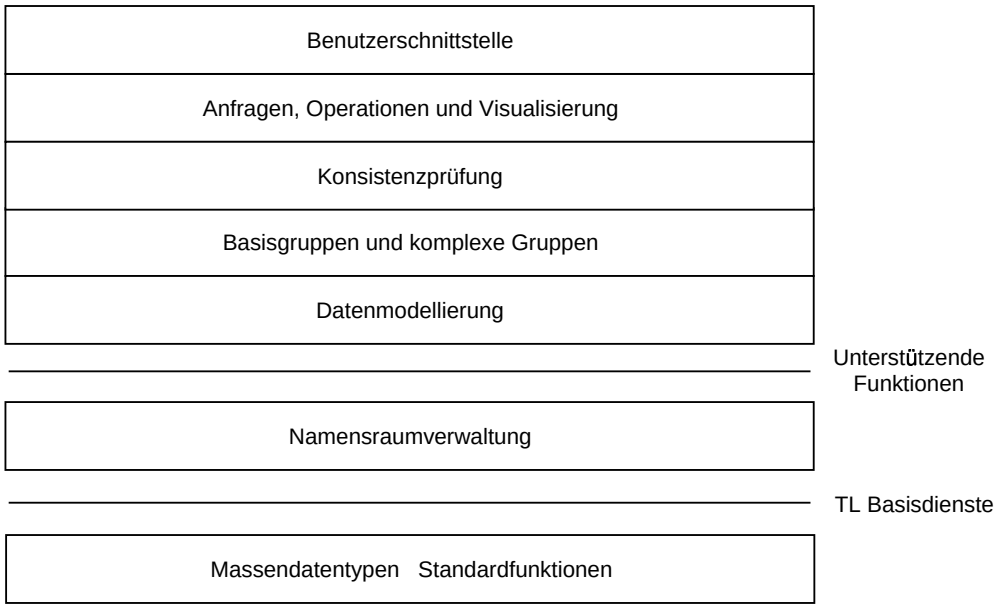


Abbildung 5.5: Grobarchitektur der Module zur Modellierung von Subjekthierarchien

```
Let Classification =
 Tuple
  case deepDisjunct
  case disjunct
 end

Let Rec Bottom(E, P <:Ok) <:Ok =
 Tuple
    var properties  :optional.T(P)
    superGroups     :set.T(Nested(E  P))
    members         :set.T(E)
 end
and Nested(E, P <:Ok) <:Ok =
 Tuple
    var properties  :optional.T(P)
    superGroups     :set.T(Nested(E  P))
    members         :set.T(Member(E  P))
    var kind        :Classification
 end
and Member(E, P <:Ok) <:Ok =
 Tuple
  case bottom with
    member :Bottom(E  P)
  case nested with
    member :Nested(E  P)
 end
```

Der Typoperator *Bottom* dient der Modellierung einer Basisgruppe. Er umfaßt zwei Eingabetypparameter und drei Attribute. Der Typparameter *E* repräsentiert den zu verwaltenden Subjekttyp, während der Typparameter *P* den Typ einer beliebigen Eigenschaft darstellt. Diese Eigenschaft modelliert das variable Attribut *properties.*

In dem Mengentyp des Attributes *superGroups* werden alle direkten Supergruppen dieser Basisgruppe verwaltet, das heißt alle Gruppen, in denen diese Basisgruppe direktes Mitglied ist. Das Attribut *members* schließlich verwaltet alle direkten Mitglieder, die zur Basisgruppe gehören. Diese dürfen keine Gruppen sein; der Elementtyp der verwalteten Menge entspricht daher dem übergebenen Subjekttyp *E*.
Die Modellierung komplexer Gruppen erfolgt analog mit dem Typoperator *Nested.* Die Funktionalität der Attribute *properties* und *superGroups* entspricht dabei jenen der Basisgruppe. Die direkten Mitglieder werden mit dem Mengentyp *members* administriert. Diese können konzeptuell sowohl Basisgruppen als auch komplexe Gruppen sein, daher muß ein weiterer Typ *Member* definiert werden, der genau diese beiden Varianten abbildet (siehe weiter unten). Verschachtelte Gruppen verfügen darüber hinaus über eine Klassifikation (siehe Abschnitt 4.2.2), mit der die Menge ihrer direkten und indirekten Mitglieder eingeschränkt wird. Diese Klassifikation wird dem Attribut *kind* zugewiesen. Da die Anwendungssemantik eine Änderung der Klassifikation erfordern kann (siehe Abschnitt 5.3.2), folgt, daß dieses Attribut variabel sein muß. Der Typ *Classification* beschreibt die beiden Varianten, die als Klassifikation spezifiziert werden können: disjunkt und tief disjunkt.
Der Typoperator *Member* repräsentiert eine Gruppe, abstrahiert jedoch nach außen von deren konkreter Art. Diese Eigenschaft wird zum Beispiel benutzt, um die Menge der direkten Mitglieder einer komplexen Gruppe zu beschreiben. Die Varianten *bottom* und *nested* spezifizieren genauer, um welche Art von Gruppe es sich handelt. Diese Information bleibt dem Benutzer jedoch verborgen und kann nur durch einen Variantenselektionstest in Erfahrung gebracht werden.
Die wechselseitigen Beziehungen zwischen den Typoperatoren *Bottom*, *Nested* und *Member* bedingen den Einsatz rekursiver Typoperatoren, da nur diese alle gegenseitigen Referenzierungen korrekt abbilden.

5.3.2 Konsistenzprüfung

Auch der Aufbau von Subjekthierarchien erfordert Prüfungen zur Gewährleistung der Widerspruchsfreiheit. Diese betreffen die Einhaltung der spezifizierten Gruppenstrukturpolitiken und müssen stets beim Einfügen eines Subjektes oder einer Gruppe in eine andere Gruppe durchgeführt werden. Für die einzelnen Gruppenstrukturpolitiken sehen sie wie folgt aus:

- Eine Überprüfung, ob die Zyklenfreiheit einer Subjekthierarchie gewahrt ist, muß lediglich beim Einfügen einer verschachtelten Gruppe A in eine verschachtelte Gruppe B vorgenommen werden, da Basisgruppen selbst nur aus einfachen Subjekten bestehen. Hierbei wird für alle transitiv erreichbaren Supergruppen von B geprüft, ob sie A enthalten. Gibt es eine solche Übereinstimmung, so kann die Einfügeoperation nicht durchgeführt werden, da sonst ein Zyklus entsteht.

- Die tiefe Disjunktheit einer Subjekthierarchie kann verletzt werden, wenn einer Gruppe ein neues Mitglied hinzugefügt wird. Mögliche Konsistenzverletzungen veranschaulicht

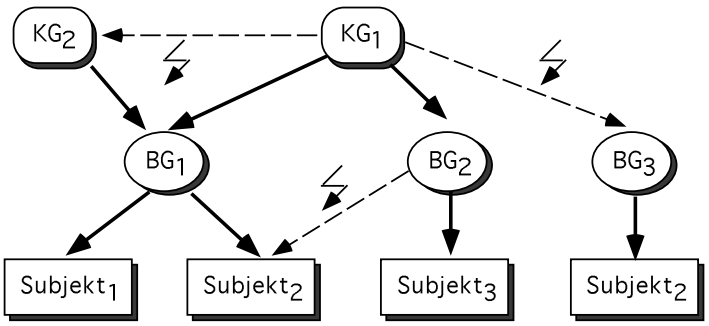


Abbildung 5.6: Verletzung von tiefer Disjunktheit

Abbildung 5.6. Die direkten Mitglieder der komplexen Gruppe KG_1 sind die Basisgruppen BG_1 und BG_2, die ihrerseits die Subjekte $Subjekt_1$ und $Subjekt_2$ beziehungsweise $Subjekt_3$ enthalten. Ist KG_1 als tief disjunkt klassifiziert, so sind folgende drei dargestellten Operationen konsistenzverletzend und damit verboten:

1. das Einfügen von $Subjekt_2$ in BG_2, da $Subjekt_2$ dann zweimal indirektes Mitglied von KG_1 ist.
2. das Einfügen von BG_3 in KG_1, da $Subjekt_2$ dann ebenfalls zweimal indirektes Mitglied von KG_1 ist.
3. das Einfügen der komplexen Gruppe KG_2 in KG_1, da dann sowohl $Subjekt_1$ als auch $Subjekt_2$ zweimal als indirekte Mitglieder von KG_1 vorkommen.

Darüber hinaus unterbindet die Konsistenzprüfung das Einfügen einer disjunkt klassifizierten komplexen Gruppe KG_i in eine tief disjunkt klassifizierte Gruppe KG_j, auch wenn die tiefe Disjunktheit von KG_j dadurch nicht unmittelbar verletzt wird. Ist KG_i auch tief disjunkt, so besteht die Möglichkeit, die Klassifikation mit der Funktion *changeClassification* (vergleiche Anhang D.1) zu verschärfen. Anderenfalls kann die Klassifikation von KG_j mit derselben Funktion nach 'disjunkt' geändert werden.

- ▷ Die Verwaltung der direkten Mitglieder einer Gruppe erfolgt durch eine Menge. Diese Implementationsentscheidung stellt sicher, daß die Disjunktheit der direkten Mitglieder einer Gruppe systeminhärent erzwungen wird. Der Versuch, ein bereits in dieser Menge vorhandenes Mitglied ein zweites Mal einzufügen, führt zur Auslösung einer Ausnahme. Die konsistenzverletzende Operation wird zurückgewiesen.

5.4 Verwaltung von Propagierungshistorien

Die Realisierung eines Moduls, mit dem die Historie eines delegierten Rechtes nachvollzogen werden kann, ist in enger Anlehnung an den in Abschnitt 2.2.6 beschriebenen Algorithmus vollzogen worden. Folgende drei Ergänzungen sind dabei hinzugefügt worden:

1. Aufgezeichnete Propagierungshistorien können mit dem Visualisierungswerkzeug *daVinci* graphisch repräsentiert werden. Hierzu wird der zugrundeliegende Graph in eine

abstrakte Syntax überführt und *daVinci* als Eingabedatei übergeben. Für nähere Einzelheiten sei auf Anhang C verwiesen.

2. Der Besitzer eines Objektes kann im laufenden Betrieb geändert werden. Alle propagierten Rechte und Zeitstempel behalten dabei ihre Gültigkeit.

3. Die Kopiererlaubnis für ein bestimmtes Recht kann im laufenden Betrieb geändert werden. Wird einem Subjekt eine erteilte Kopiererlaubnis unter Beibehaltung der Zugriffserlaubnis entzogen, so werden alle von diesem Subjekt weitergegebenen Rechte zurückgenommen.

Die komplette Benutzerschnittstelle befindet sich in Anhang D.3.

5.5 Generatoren für sichere Funktionen

Die Bereitstellung der vorliegenden Autorisierungsbibliotheken befähigt Programmentwickler, sichere Operationen für ihre Applikationen zu spezifizieren und zu implementieren. Dieser Implementationsvorgang erfolgt jedoch ohne programmtechnische Unterstützung für die korrekte Transformation des entworfenen Sicherheitsmodells in korrekten Programmcode. Ebenso fehlen für die Integration einer speziellen Sicherheitspolitik in die jeweilige Anwendungssemantik programmgestützte Hilfsfunktionen. Es sind daher zwei Defizite erkennbar: Die Programmierung und Integration eines konkreten Sicherheitsmodells sind fehlerträchtig, und Änderungen der Sicherheitsanforderungen, die zu einem Austausch sicherheitsrelevanter Programmabschnitte führen, sind verhältnismäßig aufwendig.
Da der Implementationsprozeß von Sicherheitsmodellen aus vielen repetitiven Teilaufgaben besteht, bietet sich eine Unterstützung durch **Generatoren** an. In der Tycoon-Programmierumgebung kann der Generatorprozeß durch folgende beiden Methoden realisiert werden [RMS95]:

1. **Generierung durch Funktionen höherer Ordnung:** In Tycoon sind Funktionen Werte erster Klasse. Funktionen höherer Ordnung, die Funktionenswerte als Ein- beziehungsweise Ausgabeparameter besitzen können, eignen sich daher als Generatoren. Die Transformation einer unsicheren in eine um Zugriffskontrolle erweiterte (sichere) Funktion kann dadurch erreicht werden, daß eine Funktion höherer Ordnung bereitgestellt wird, die aus einer unsicheren Funktion und einem ein Autorisierungssystem repräsentierenden Wert als Eingabeparameter eine sichere Funktion als Rückgabewert erzeugt. Die Generierung erfolgt also vollständig auf der Anwendungsebene.
   Nachteilig bei diesem Ansatz ist, daß keine Generatorfunktion spezifiziert werden kann, die alle beliebigen ungeschützten Funktionen als Eingabewerte nimmt. Der Grund für diese Einschränkung liegt in der Arität (Anzahl der Eingabeparameter) der Funktionen: Es können zum Beispiel keine generellen Subtypbeziehungen zwischen beliebigen Funktionen mit einem und beliebigen Funktionen mit zwei Eingabeparametern abgeleitet werden. Daher können Funktionen höherer Ordnung immer nur als Generator für eine Menge von Eingabefunktionstypen gleicher Arität eingesetzt werden.

2. **Generierung durch Code-Generatoren:** Die Entwicklung großer Applikationen nutzt in der Regel das Modulkonzept für die Realisierung von Teilsystemen aus. Diese Teilsysteme werden durch Schnittstellen repräsentiert. Ziel eines sicheren Systems muß es sein,

daß die Schnittstellen dieser Teilsysteme nur sichere Funktionen enthalten. Obwohl eine Schnittstelle in TL nichts anderes als ein Tupeltyp ist, kann sie nicht von Generatorfunktionen erzeugt werden, da die einzelnen Komponenten dieses Tupeltyps nicht generisch beschrieben werden können. Ist eine Tupelkomponente beispielsweise eine Funktion, so tritt das oben beschriebene Aritätsproblem auf. Dieses Problem lösen Code-Generatoren, da sie immer ganze Schnittstellen generieren. Sie erzeugen aus der ungeschützten Applikation und der Schnittstelle des Sicherheitssystems sowohl die Schnittstelle wie auch das Implementationsmodul der sicheren Applikation.

Die vorliegende Arbeit verfolgt den ersten Ansatz. Die Autorisierungsbibliothek enthält Generatorfunktionen zur Generierung sicherer Funktionen. Das nachfolgende Programmfragment zeigt die konkrete Implementierung einer Generatorfunktion, die sichere Funktionen mit einem Eingabeparameter generiert.

```
let error = exception "authGenerators"

let generate1(E, F, N, Subject <:Ok f(:E) :F auth :authorization.T(Subject)
  node :auth.Node(N) convert(:E) :N eq(:N :N) :Bool) :Fun(:E sub :Subject) :F =
  fun(e :E sub :Subject) :F
    begin
      let instances = auth.instances(auth.getTemplate(node))
      try
        let inst = iter.any(:auth.Instance(N) instances
          fun(a :auth.Instance(N)) :Bool
            eq(convert(e) auth.getInst(:N a)))
        if auth.allowed(:N node sub inst)
        then
          f(e)
        else
          raise error
        end
      when iter.error then
        f(e)
      end
    end
```

Die Funktion *generate1* benötigt folgende Eingabeparameter:

- einen beliebigen Subjekttyp *Subject*;
- die zu schützende Funktion *f* mit einem beliebigen Eingabeparameter vom Typ *E* und einem beliebigen Rückgabewert vom Typ *F*;
- einen Autorisierungsgraphen *auth*, in dem ein Repräsentant dieser Funktion verwaltet wird;
- einen Quellwertwächter *node* dieses Autorisierungsgraphen, der die Funktion *f* repräsentiert und deren Zugriffskontrollinformationen verwaltet;

- ▷ eine Funktion *convert*, die Werte vom Typ *E* in Werte eines beliebigen (nicht notwendigerweise verschiedenen) Typs *N* konvertiert. Diese Funktion ist erforderlich, da dynamische Schablonen nur einen polymorphen Typparameter besitzen. Insbesondere bei zu schützenden Funktionen mit mehreren oder keinem Eingabeparameter muß daher spezifiziert werden, welcher Typ maßgeblich ist, um die Zugriffskontrollinformationen für die Funktion *f* wiederzufinden;

- ▷ eine Gleichheitsfunktion *eq*, die zwei Werte vom Typ *N* vergleicht. Sie wird benötigt, um die richtige Instanz einer dynamischen Schablone zu finden.

Die zurückgegebene Funktion wird um einen Eingabeparameter sub vom Typ $Subject$ erweitert. Dieser repräsentiert das Subjekt, das die Funktion aufruft. Kann sichergestellt werden, daß die Systemgrenzen nie verlassen werden, so kann die Ableitung des Subjektparameters auch systemintern erfolgen. Auf diesen Parameter kann dann verzichtet werden.
Der Funktionsrumpf der zurückgegebenen Funktion arbeitet wie folgt: Mit Hilfe des übergebenen Quellwertwächters *node* wird die zugehörige dynamische Schablone ermittelt, zu der dieser gehört. Die Implementierung stellt sicher, daß diese Schablone mit dem Typ *N* instanziiert ist. Für alle Instanzen dieser Schablone, die somit Werten vom Typ *N* entsprechen, wird überprüft, ob es eine Entsprechung mit dem konvertierten Wert vom Typ *E* der Funktion *f* gibt. Ist dies der Fall, so wird anhand der Zugriffskontrollinformation des zu *node* entsprechenden Wertwächters der gefundenen Instanz entschieden, ob der Zugriff für das anfragende Subjekt erlaubt ist oder nicht. Im Falle einer Zugriffsberechtigung erfolgt die Anwendung der ungeschützten Funktion *f*; anderenfalls wird eine Ausnahme ausgelöst. Wird keine entsprechende Instanz gefunden (**when** *iter.error*), so untersteht der übergebene Wert vom Typ *E* nicht der Zugriffskontrolle, und die ungeschütze Funktion *f* darf uneingeschränkt auf diesen Wert angewendet werden.

5.6 Benutzung der Bibliotheken

Abschließend wird die konkrete Benutzung der einzelnen Komponenten der bereitgestellten Zugriffskontrollbibliothek anhand kleiner Beispiele vorgestellt und erläutert. Die Kombination dieser Basisdienste ermöglicht die Realisierung konkreter Zugriffskontrollsysteme. Darüber hinaus bilden sie die Basis für weitergehende Erweiterungen der Zugriffskontrollbibliothek.

5.6.1 Modellierung von Wertkontrollstrukturen

Das folgende Beispiel bildet die in Abbildung 4.6 dargestellten Wertkontrollstrukturen mithilfe des Moduls *authorization* (Modulschnittstelle, siehe Anhang D.2) in die Systemumgebung Tycoon ab. Hierbei wird zusätzlich eine Markierung der modellierten Konnektoren mit Polaritäten vorgenommen. Zunächst erfolgt die Modellierung der ungeschützten Werte und Funktionen.

```
Let Heart =
  Tuple
    patient :String
    ....
  end
```

```
let h1 :Heart = tuple "Meier" ... end
let h2 :Heart = tuple "Lehmann" ... end
let operate(h :Heart) :Ok = ...
let makeEKG(h :Heart) :Ok = ...
let getPatient(h :Heart) :String = h.patient
let peter = "Peter"
let mary = "Mary"
let sysop = "Systemoperator"
```

Der Typ *Heart* repräsentiert die rechnerinterne Darstellung für ein Herz; *h1* und h2 sind Werte dieses Typs und stellen konkrete Herzen dar. Darüber hinaus existieren drei Funktionen (*operate*, *makeEKG* und *getPatient*) die auf Werten des Typs *Heart* arbeiten. Der Subjekttyp, der grundsätzlich frei wählbar ist, wird in diesem Beispiel durch *Strings* dargestellt. Es existieren genau drei Subjekte: *peter*, *mary* und *sysop*. Das Subjekt *sysop* erhält im weiteren Verlauf den Besitzerstatus für alle zu spezifizierenden Quellwertwächter.
Die folgende Anweisung erzeugt ein neues Autorisierungssystem, das mit dem Bezeichner *heartDB* referenziert werden kann.

```
let heartDB = authorization.new(:String "" fun(a :String) :String a
  fun(b,c :String) :Bool string.equal(b c))
```

Innerhalb dieses Autorisierungssystems wird eine neue dynamische Schablone erzeugt, die Werte vom Typ *Heart* verwaltet. Dieser werden drei Quellwertwächter (*opSourceNode, ekgSourceNode* und *getSourceNode*) hinzugefügt, die die Funktionen *operate, makeEKG* und *getPatient* repräsentieren. Zwischen diesen Quellwertwächtern werden Implikationen eingefügt. Da es sich bei den geschützten Werten um Funktionen handelt, bietet sich gemäß der in Abschnitt 2.4.4 getroffenen Feststellungen die Modellierung gegengerichteter Vererbung an. Hierbei findet eine Vererbung positiver Autorisierungen von *opSourceNode* über *ekgSourceNode* nach *getSourceNode* statt, während negative Rechte in entgegengesetzter Richtung vererbt werden. Außerdem wird für den Zugriff auf einen geschützten Wert, für den keine Autorisierung modelliert wurde, eine pessimistische Autorisierungsstrategie spezifiziert, das heißt alle nicht positiv oder negativ autorisierten Zugriffe werden als verboten angesehen (geschlossenes System).

```
let hearts = heartDB.newTemplate(:Heart authorization.pessimistic)
let opSourceNode = heartDB.addElement(:Heart hearts operate "function operate" sysop)
let ekgSourceNode = heartDB.addElement(:Heart hearts
  makeEKG "function makeEKG" sysop)
let getSourceNode = heartDB.addElement(:Heart hearts
  getPatient "function getPatient" sysop)
let impl1 = heartDB.addImplication(:Heart opSourceNode ekgSourceNode
  "operate -> makeEKG" true)
let impl2 = heartDB.addImplication(:Heart ekgSourceNode opSourceNode
  "makeEKG -> operate" false)
let impl3 = heartDB.addImplication(:Heart ekgSourceNode getSourceNode
  "makeEKG -> getPatient" true)
let impl4 = heartDB.addImplication(:Heart getSourceNode ekgSourceNode
  "getPatient -> makeEKG" false)
```

Aus der dynamischen Schablone *hearts* werden Instanzen für die Werte h1 und h2 erzeugt. Dies führt dazu, daß zu jedem Quellwertwächter der dynamischen Schablone ein entsprechender Wertwächter in den beiden Instanzen erzeugt wird und daß alle Implikationen zwischen den Quellwertwächtern in die Instanzen übertragen werden.

let *heart1 = heartDB.addInstance(:Heart hearts h1 “heart1”)*
let *heart2 = heartDB.addInstance(:Heart hearts h2 “heart2”)*

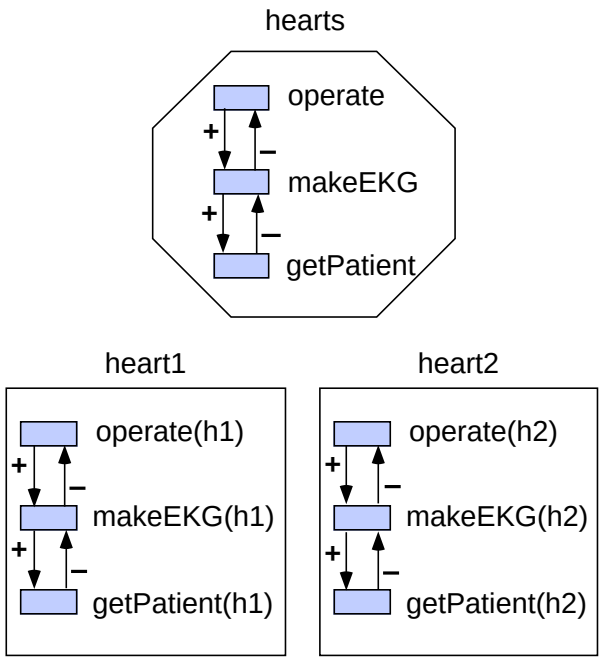


Abbildung 5.7: Modellierte Anwendung mit zwei Instanzen

Das Ergebnis der bisherigen Modellierung zeigt Abbildung 5.7. Die Systemanfrage, ob das Subjekt *peter* für die Anwendung der Funktion *getPatient* auf das Objekt *h1* berechtigt ist, wird verneint, da keine Autorisierung für *peter* existiert und daher die pessimistische Autorisierungsstrategie den Zugriff verbietet. Wird eine positive schwache Autorisierung für *peter* in den Wertwächter *opSourceNode* der Instanz *heart1* eingetragen, so resultieren hieraus aufgrund der Implikationen zwischen den Wertwächtern auch positive schwache Autorisierungen für die anderen beiden Wertwächter.

heartDB.allowed(:Heart getSourceNode peter heart1)
=> **false**
heartDB.insertWeakInst(:Heart opSourceNode peter heart1 true sysop)
heartDB.allowed(:Heart opSourceNode peter heart1)
=> **true**
heartDB.allowed(:Heart ekgSourceNode peter heart1)
=> **true**
heartDB.allowed(:Heart getSourceNode peter heart1)
=> **true**

Durch die folgenden beiden Programmanweisungen wird ein latenter Konflikt modelliert. Zunächst wird eine ebenfalls positive schwache Autorisierung für den Wertwächter *ekgSourceNode* modelliert. Hierdurch entsteht Redundanz, denn die Wertwächter *ekgSourceNode* und *getSourceNode* sind bereits durch die Autorisierung am Wertwächter *opSourceNode* implizit positiv autorisiert worden. Die neu eingefügte explizite Autorisierung überschreibt jedoch die impliziten Autorisierungen für *getSourceNode* und *opSourceNode*. Wird anschließend eine explizite schwache negative Autorisierung für *peter* auf dem Wertwächter *getSourceNode* modelliert, so ist eine Konfliktsituation entstanden, die in Abbildung 5.8 dargestellt ist. Es handelt sich hierbei

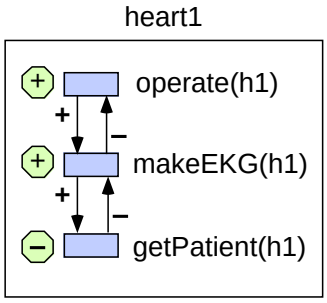


Abbildung 5.8: Anwendung mit einem latenten Konflikt

nicht um einen tatsächlichen Konflikt, denn die Gültigkeit der explizit erteilten Autorisierungen beschränkt sich aufgrund des gegenseitigen Überschreibens nur auf den jeweils explizit autorisierten Wertwächter selbst. Das Resultat der Modellierung ist jedoch ein latenter Konflikt, da die explizite Autorisierung am Wertwächter *ekgSourceNode* nun nicht mehr gelöscht werden kann, ohne daß ein tatsächlicher Konflikt entsteht. Dieser kommt dadurch zustande, daß im Falle eines Löschens sowohl eine positive implizite Autorisierung (vom Wertwächter *opSourceNode*) als auch eine negative implizite Autorisierung (vom Wertwächter *getSourceNode*) für den Wertwächter *ekgSourceNode* existiert. Dies entspricht jedoch einer Verletzung der Konsistenz und löst daher eine Ausnahme aus.

heartDB.insertWeakInst(:Heart ekgSourceNode peter heart1 true sysop)
heartDB.insertWeakInst(:Heart getSourceNode peter heart1 false sysop)
heartDB.deleteWeakInst(:Heart ekgSourceNode peter heart1 sysop)
==> ***** Exception:** {**“### delete refused, latent conflict for subject Peter ###”**}

Die letzten beiden Operationen werden rückgängig gemacht und für Subjekt *mary* wird eine schwache negative Autorisierung in den Wertwächter *getSourceNode* eingetragen. Hierbei entsteht kein Konflikt, weil die erteilten Autorisierungen für die Wertwächter *getSourceNode* und *opSourceNode* für verschiedene Subjekte gelten.

heartDB.deleteWeakInst(:Heart getSourceNode peter heart1 sysop)
heartDB.deleteWeakInst(:Heart ekgSourceNode peter heart1 sysop)
heartDB.insertWeakInst(:Heart getSourceNode mary heart1 false sysop)

Die Autorisierungen für die Instanz *heart2* werden analog zu den Autorisierungen der Instanz *heart1* jedoch mit vertauschten Subjekten erteilt.

heartDB.insertWeakInst(:Heart opSourceNode mary heart2 true sysop)
heartDB.insertWeakInst(:Heart getSourceNode peter heart2 false sysop)

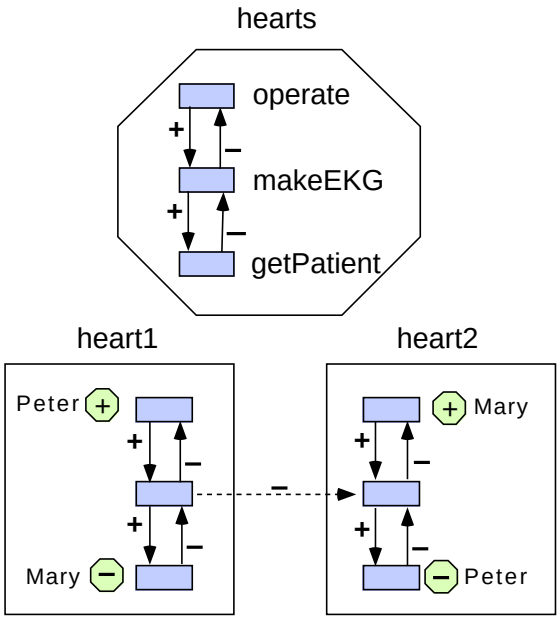


Abbildung 5.9: Modellierte Anwendung mit vier schwachen Autorisierungen

Um auszudrücken, daß Subjekte, die keine Autorisierung für die Anwendung der Funktion *makeEKG* auf den Wert *h1* besitzen, diese Funktion auch nicht auf dem Wert *h2* ausführen dürfen, wird versucht, eine Isoinstanzreferenz zwischen den Wertwächtern *ekgSourceNode* der Instanzen *heart1* und *heart2* einzufügen. Hierdurch würde jedoch ein tatsächlicher Konflikt durch widersprüchliche Autorisierungen am Wertwächter *ekgSourceNode* der Instanz *heart2* für Subjekt *mary* entstehen. Die konfliktauslösende Operation wird daher zurückgewiesen. Da keine Isoinstanzreferenz eingefügt wird und die Funktion *heartDB.addIntra* folglich auch keine Referenz auf einen solchen Konnektor zurückgeben kann, wird eine Ausnahme ausgelöst. Das Autorisierungssystem vor (und nach) dem Versuch, eine Isoinstanzreferenz einzufügen, zeigt Abbildung 5.9.

let *iso1 = heartDB.addIntra(:Heart ekgSourceNode ekgSourceNode heart1 heart2 false)*
==> ***** Exception:** {**"### insert refused due to a conflict for subject Mary ###"**}

Ausgehend von dem erreichten Zustand wird aus der ungeschützten Funktion *getPatient* und dem Autorisierungssystem *heartDB* eine sichere Funktion *secureGetPatient* generiert. Diese Generierung erfolgt durch die im Modul *authGenerators* bereitgestellten Generatorfunktionen (vergleiche Anhang D.4).

let *secureGetPatient = authGenerators.generate1(:Heart :String :Heart :String*
getPatient heartDB getSourceNode **fun***(h :Heart) :Heart h*
fun*(h1,h2 :Heart) :Bool h1 == h2)*

Entsprechend der im Autorisierungssystem modellierten Rechte zeigt die sichere Funktion *secureGetPatient* für verschiedene übergebene Subjekte und Werte vom Typ *Heart* das folgende Verhalten:

secureGetPatient(h1 peter)
=>"Meier" :String
secureGetPatient(h1 mary)
=> * Exception: {"No access permission"}**
secureGetPatient(h2 peter)
=> * Exception: {"No access permission"}**
secureGetPatient(h2 mary)
=>"Lehmann" :String

Der Vorgang der Generierung braucht nur einmal durchgeführt zu werden, vorausgesetzt das zugrundeliegende Autorisierungssystem soll beibehalten werden. Auch nach Änderung des Zustands des Autorisierungssystems ist keine neue Generierung erforderlich. Durch die nachfolgende Programmanweisung werden die negativen schwachen Autorisierungen für Subjekt *peter* in der Instanz *heart2* durch positive starke Autorisierungen überschrieben. Wird die sichere Funktion *secureGetPatient*, ohne erneutes Generieren, mit den Parametern *h2* und *peter* aufgerufen, wird sie, im Gegensatz zu oben, aufgrund der nun existierenden positiven Autorisierung ausgeführt.

heartDB.insertInstACL(:Heart opSourceNode peter heart2 true sysop)
secureGetPatient(h2 peter)
=>"Lehmann" :String

5.6.2 Modellierung von Subjekthierarchien

Für die Modellierung von Subjekthierarchien steht das Modul *groupStructure* zur Verfügung (siehe Anhang D.1). Zunächst werden vier einfache (nicht gruppierte) Subjekte *uwe*, *sylvia*, *ramona* und *drMeier* modelliert. Der Typ dieser Subjekte ist frei wählbar; aus Gründen der leichteren Verständlichkeit erfolgt die Modellierung in diesem Beispiel auf der Basis von Zeichenketten.

let *uwe = "Uwe"*
let *sylvia = "Sylvia"*
let *ramona = "Ramona"*
let *drMeier = "Dr.med.Meier"*

Die nächste Anweisung erzeugt eine neue Subjektstruktur, die mit dem Namen *khPersonal* referenziert wird. Der boolesche Wert *false* drückt dabei aus, daß diese Struktur keine Zyklen enthalten darf. Darüber hinaus werden vier zunächst leere Basisgruppen (*chirurgen, hilfskraefte, opSchwestern* und *stSchwestern*) definiert. Direkte Mitglieder dieser Gruppen gelten als gleich, wenn ihre Zeichenketten gleich sind.

let *khPersonal = groupStructure.new(:String* **fun***(a :String) :String a false)*
let *chirurgen = khPersonal.newBottom("Chirurgen"* **fun***(a,b :String) :Bool string.equal(a b))*

```
let hilfskraefte = khPersonal.newBottom("Hilfskraefte"
  fun(a,b :String) :Bool string.equal(a b))
let opSchwestern = khPersonal.newBottom("OP-Schwestern"
  fun(a,b :String) :Bool string.equal(a b))
let stSchwestern = khPersonal.newBottom("Stations-Schwestern"
  fun(a,b :String) :Bool string.equal(a b))
```

Im nächsten Schritt erfolgt die Modellierung von drei komplexen Gruppen (*khSchwestern*, *jedermann* und *fachaerzte*). Als Gruppenstrukturpolitik für diese Gruppen wird die Integritätsbedingung "tief disjunkt" spezifiziert.

```
let khSchwestern = khPersonal.newNested("Krankenschwestern"
  tuple case deepDisjunct of groupStructure.Classification end)
let jedermann = khPersonal.newNested("Jedermann"
  tuple case deepDisjunct of groupStructure.Classification end)
let fachaerzte = khPersonal.newNested("Fachaerzte"
  tuple case deepDisjunct of groupStructure.Classification end)
```

Mithilfe der folgenden Anweisungen erhalten die komplexen Gruppen ihre direkten Mitglieder. Die resultierende Subjekthierarchie veranschaulicht Abbildung 5.10.

```
khPersonal.addNested(khSchwestern
  tuple case bottom of khPersonal.Member with opSchwestern end)
khPersonal.addNested(khSchwestern
  tuple case bottom of khPersonal.Member with stSchwestern end)
khPersonal.addNested(khSchwestern
  tuple case nested of khPersonal.Member with fachaerzte end)
khPersonal.addNested(fachaerzte
  tuple case bottom of khPersonal.Member with chirurgen end)
khPersonal.addNested(jedermann
  tuple case bottom of khPersonal.Member with hilfskraefte end)
khPersonal.addNested(jedermann
  tuple case nested of khPersonal.Member with khSchwestern end)
```

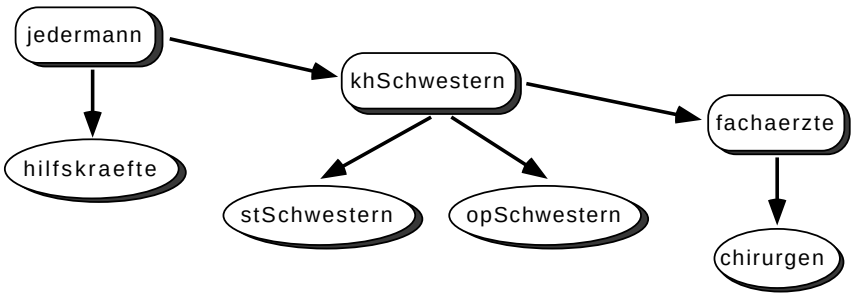


Abbildung 5.10: Modellierte Subjekthierarchie

Eine charakterisierende Eigenschaft von Subjekthierarchien ist ihre Azyklizität (vergleiche Abschnitt 4.2.2). Der folgende Versuch, die (komplexe) Gruppe *jedermann* in die (komplexe)

Gruppe *fachaerzte* einzufügen, wird daher vom System als eine konsistenzverletzende Operation erkannt und durch Auslösung einer Ausnahme zurückgewiesen.

khPersonal.addNested(fachaerzte
tuple case *nested* **of** *khPersonal.Member* **with** *jedermann* **end***)*
=> ***** Exception:** {**"### group not inserted (cycle detected) ###"**}

Die nicht gruppierten Subjekte werden in die Basisgruppen eingefügt. Aus Gründen der Einfachheit wird jeweils nur ein Subjekt hinzugefügt. Das Resultat der Modellierung ist in Abbildung 5.11 dargestellt. Maßgeblich sind hierbei die durchgezogenen Pfeile.

khPersonal.addBottomElements(opSchwestern iter.enum **of** *ramona* **end***)*
khPersonal.addBottomElements(stSchwestern iter.enum **of** *sylvia* **end***)*
khPersonal.addBottomElements(hilfskraefte iter.enum **of** *uwe* **end***)*
khPersonal.addBottomElements(chirurgen iter.enum **of** *drMeier* **end***)*

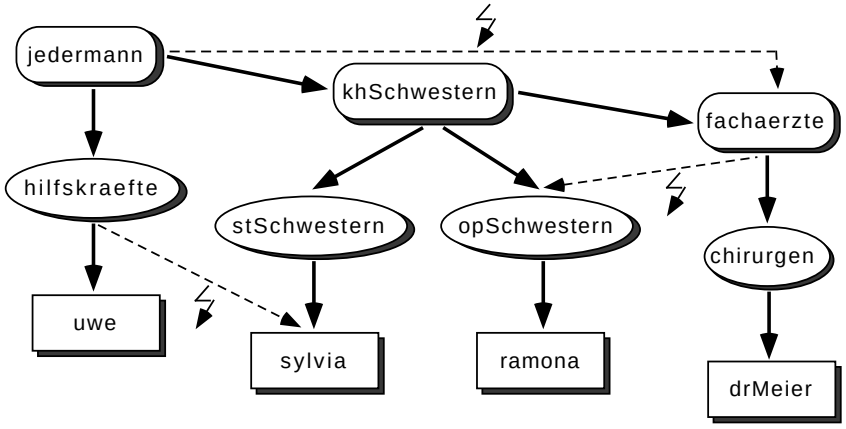


Abbildung 5.11: Endgültige Subjekthierarchie mit Überprüfung tiefer Disjunktheit

Alle komplexen Gruppen sind so modelliert worden, daß sie stets die Eigenschaft der tiefen Disjunktheit erfüllen. Die nachfolgenden Programmanweisungen, mit denen versucht wird, diese Eigenschaft auf unterschiedliche Weise zu gefährden, werden als konsistenzverletzend erkannt und lösen Ausnahmen aus.

khPersonal.addBottom(hilfskraefte sylvia)
=> ***** Exception:** {**"### Element not insert (deep disjunction violated) ###"**}
khPersonal.addNested(fachaerzte
tuple case *bottom* **of** *khPersonal.Member* **with** *opSchwestern* **end***)*
=> ***** Exception:** {**"### Element not insert (deep disjunction violated) ###"**}
khPersonal.addNested(jedermann
tuple case *nested* **of** *khPersonal.Member* **with** *fachaerzte* **end***)*
=> ***** Exception:** {**"### Element not insert (deep disjunction violated) ###"**}

Fügt man Subjekt *sylvia* in die Basisgruppe *hilfskraefte* ein, so wird die tiefe Disjunktheit der komplexen Gruppe *jedermann* verletzt, da *sylvia* nun zweimal indirektes Mitglied dieser Gruppe

ist. Ebenso führt das Einfügen der Basisgruppe *opSchwestern* in die komplexe Gruppe *fachaerzte* zu einer Konsistenzverletzung. Diese Basisgruppe wäre dann zweimal Mitglied der Gruppen *khSchwestern* und *jedermann*. Schließlich ist es auch nicht möglich, die komplexe Gruppe *fachaerzte* als direktes Mitglied der komplexen Gruppe *jedermann* fungieren zu lassen, da für *jedermann* dann ebenfalls die tiefe Disjunktheit verletzt wäre. Diese drei Operationen, die in Abbildung 5.11 durch gestrichelte Pfeile repräsentiert sind, werden daher zurückgewiesen.

5.6.3 Modellierung von Propagierungshistorien

Die Delegierung und Revokation von Rechten wird mithilfe des Moduls *granting* (siehe Anhang D.3) modelliert. Die nachfolgende Propagierungshistorie erfolgt dabei in enger Anlehnung an Abbildung 2.6. Bei dem geschützten Wert handelt es sich diesmal um ein Tupel mit drei Komponenten: einem Namen, einem Datenwert und einem Funktionswert.

```
Let Heart =
  Tuple
    patient :String
    ...
  end
let h1 = tuple "Meier" ... end
let getPatient(h :Heart) :String = h.patient

Let SaveValue =
  Tuple
     name  :String
     h     :Heart
     a     :Fun(:Heart) :String
  end
let saveValue =
  tuple
    "getPatient(h1)"
    h1
    getPatient
  end
```

Es folgt die Definition von fünf Subjekten, die durch Zeichenketten repräsentiert werden.

```
let sub1 = "Subjekt1"
let sub2 = "Subjekt2"
let sub3 = "Subjekt3"
let sub4 = "Subjekt4"
let sub5 = "Subjekt5"
```

Die Administration der Propagierungshistorie erfolgt in *admin*, einem Wert des abstrakten Typs *granting.T*.

```
let ex(a :String) :String = a
let exSave(s :SaveValue) :String = s.name
let admin = granting.new(:String  :SaveValue  ex  exSave)
```

$Subjekt_1$ erhält den Besitzerstatus für den Wert *saveValue*. Die darauf folgenden vier delegierten Rechte, die zu den Zeitstempeln 10-40 führen, werden jeweils mit Kopiererlaubnis erteilt. Das Ergebnis der Modellierung veranschaulicht Abbildung 5.12a.

admin.grantOwnership(sub1 val)
admin.grant(sub1 sub2 saveValue true)
admin.grant(sub1 sub3 saveValue true)
admin.grant(sub3 sub4 saveValue true)
admin.grant(sub2 sub3 saveValue true)

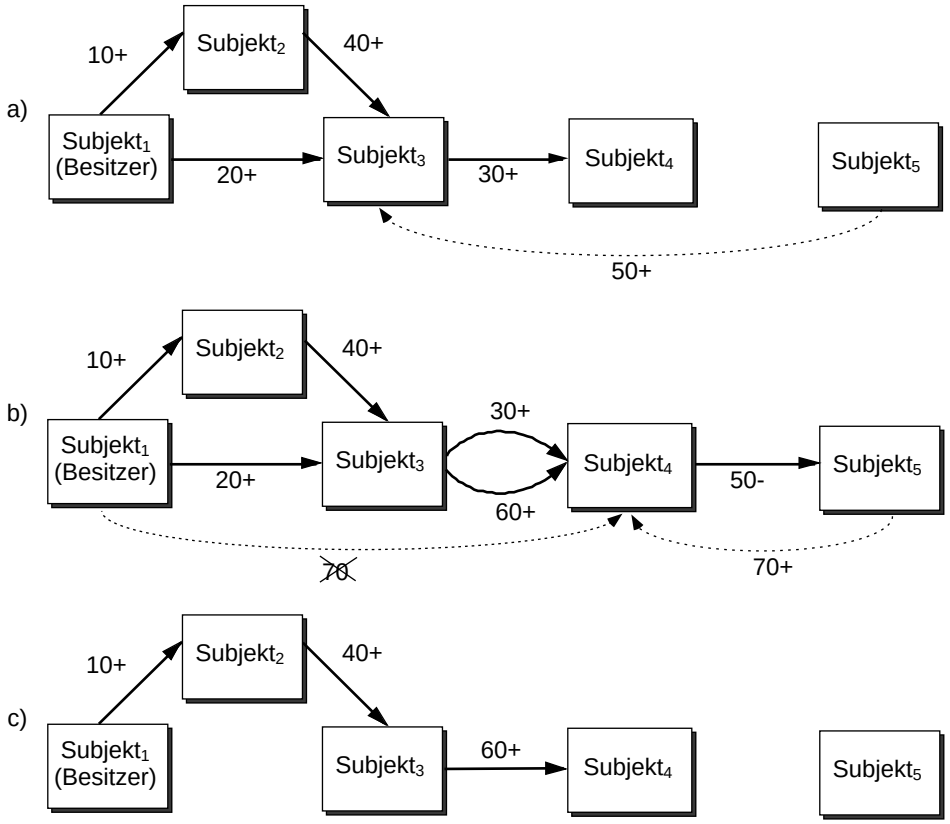


Abbildung 5.12: Modellierungsbeispiel: Propagierung und Revokation von Rechten

$Subjekt_5$, das kein Zugriffsrecht auf *saveValue* besitzt, versucht $Subjekt_3$ ein solches Zugriffsrecht zu erteilen. Dieser Vorgang ist in Abbildung 5.12a durch einen gestrichelten Pfeil gekennzeichnet. Die unerlaubte Operation wird erkannt und durch eine Ausnahme zurückgewiesen.

admin.grant(sub5 sub3 saveValue true)
=> ***** Exception:** {**"### granting refused (no access permission) ###"**}

Die folgenden beiden Operationen vervollständigen die Propagierungshistorie. $Subjekt_5$ wird zwar der Zugriff auf *saveValue* erteilt, es darf dieses Recht jedoch nicht weiterpropagieren. $Subjekt_4$ empfängt eine weitere Autorisierung von $Subjekt_3$. Die resultierende graphische Darstellung des Modellierungsprozesses zeigt Abbildung 5.12b.

admin.grant(sub4 sub5 saveValue false)
admin.grant(sub3 sub4 saveValue true)

Die Anfrage nach dem Besitzer von *saveValue* liefert $Subjekt_1$. Alle fünf Subjekte sind zugriffsberechtigt.

admin.getOwner(saveValue)
=> **"Subjekt1" :String**
admin.accessAllowed(sub1 saveValue)
...
admin.accessAllowed(sub5 saveValue)
=> **true :Bool**

$Subjekt_5$ unternimmt den Versuch, das erhaltene Zugriffsrecht weiterzupropagieren. Aufgrund der fehlenden Kopiererlaubnis löst diese Operation eine Ausnahme aus. Ebenso gelingt es $Subjekt_1$ nicht, $Subjekt_4$ die Autorisierung für den Zugriff auf *saveValue* zu entziehen, da $Subjekt_1$ dieses Recht nicht direkt an $Subjekt_4$ erteilt hat. Die entsprechenden Operationen sind in Abbildung 5.12b als gestrichelte Linien dargestellt.

admin.grant(sub5 sub4 saveValue true)
=> ***** Exception:** {**"### granting refused (no grant permission) ###"**}
admin.revoke(sub1 sub4 saveValue)
=> ***** Exception:** {**"### can not revoke privilege (not granted) ###"**}

Entzieht $Subjekt_1$ dem $Subjekt_3$ das zum Zeitpunkt 20 erteilte Recht, so ergibt sich der Zustand, der in Abbildung 5.12c dargestellt ist.

admin.revoke(sub1 sub3 saveValue)

$Subjekt_3$ bleibt zugriffsberechtigt, da es auch nach Entzug des Rechtes eine Autorisierung von $Subjekt_1$ über $Subjekt_2$ besitzt. $Subjekt_5$ hat sein Zugriffsrecht verloren.

admin.accessAllowed(sub3 saveValue)
=> **true :Bool**
admin.accessAllowed(sub5 saveValue)
=> **false :Bool**

Kapitel 6

Zusammenfassung und Ausblick

Als Resultat dieser Arbeit ist eine allgemeine Bibliothek zum Schutz von Werten beliebigen Typs durch Wertwächter entstanden. Diese Wertwächter verwalten Zugriffskontrollinformationen, die den Zugriff auf geschützte Werte regeln. Bestandteil dieser Zugriffskontrollinformation ist neben einer Liste zugriffsberechtigter Subjekte auch die Polarität und Priorität der erteilten Rechte. In Anlehnung an objektorientierte Zugriffskontrollmodelle können Wertwächter durch Konnektoren wahlfrei miteinander verbunden werden. Ein Konnektor zwischen zwei Wertwächtern repräsentiert hierbei die implizite Ableitung von Rechten einer bestimmten Polarität. Das Ergebnis einer derartigen Vernetzung ist ein allgemeiner Autorisierungsgraph, dessen Zustand und Zustandsübergänge mit einem Formalismus spezifiziert worden sind. Aufbauend auf diesen allgemeinen Autorisierungsgraphen sind Modellierungsvereinfachungen, wie dynamische Schablonen, Quellwertwächter, Implikationen und Kreuzreferenzen, vorgestellt worden, die den Modellierungsaufwand zur Abbildung eines konkreten Zugriffskontrollsystems erheblich reduzieren. Alle impliziten Rechte werden dabei sofort abgeleitet und zugriffseffizient gespeichert. Umfangreiche Konsistenzprüfungen stellen die Widerspruchsfreiheit des Autorisierungsgraphen sicher und vermeiden so Modellierungsfehler durch den Anwender.
Die Strukturierung von Subjekten erfolgt durch deren Zusammenfassung in einfache und verschachtelte benannte Gruppen. Für die resultierenden Subjekthierarchien können Gruppenstrukturpolitiken spezifiziert werden, deren Einhaltung durch Konsistenzprüfungen erzwungen wird. Darüber hinaus steht eine große Kollektion von Anfragen zur Verfügung, mit denen weitere Eigenschaften der Subjekthierarchie oder einzelner Gruppen abgefragt werden können. Die graphische Darstellung der strukturellen Komplexität einer solchen Hierarchie erfolgt durch die Aktivierung eines Visualisierungswerkzeugs, für das eine abstrakte Syntaxrepräsentation des Autorisierungsgraphen generiert wird.
Die Delegierung von Benutzerrechten zwischen Subjekten bedingt die Verwaltung von Propagierungshistorien. Die Visualisierung dieser Historien sowie die konsistente Erteilung und der konsistente Entzug eines Rechtes sind Gegenstand weiterer implementierter Module.
Zur einfachen Benutzung der Bibliotheken unterstützen Generatorfunktionen die automatische Erzeugung sicherer Funktionen. Ausgehend von einer ungeschützten Funktion und einem konkreten Autorisierungssystem werden beide Komponenten zu einer geschützen Funktion aggre-

giert. Änderungen am Zustand des Autorisierungssystems werden dabei für alle geschützten Funktionen sofort wirksam, ohne daß eine erneute Generierung vorgenommen werden muß.

6.1 Zusammenfassende Abschlußbetrachtung

Wird der in Abschnitt 2.1 definierte Anforderungskatalog an Zugriffskontrollsysteme herangezogen, um zu überprüfen, inwiefern die implementierten Zugriffskontrollbibliotheken den gestellten Anforderungen gerecht werden, so ergibt sich folgendes Bild: Die Autorisierungsbibliotheken ermöglichen einen universellen Einsatz, da sie nicht auf eine feste Sicherheitspolitik zugeschnitten sind, sondern die Modellierung beliebiger Politiken unterstützen. Der Einfluß auf das Laufzeitverhalten einer um Zugriffskontrolle erweiterten Anwendung erweist sich als gering, da implizite Autorisierungen, die sich aus expliziten Rechten ergeben, bereits zum Zeitpunkt der Modellierung abgeleitet und zugriffseffizient gespeichert werden. Zugriffsanfragen können daher schnell evaluiert werden. Zustandsänderungen am Zugriffskontrollsystem können dynamisch (am laufenden System) erfolgen. Sie gelten dann sofort für alle sicheren Funktionen, ohne daß eine erneute Generierung vorzunehmen ist (vergleiche Abschnitt 5.6.1). Konsistenzprüfungen stellen die Widerspruchsfreiheit des Systems sicher; sie erkennen und lösen Konflikte und weisen widersprüchliche Operationen zurück. Das zugrundeliegende Konzept, Wertwächter mit Konnektoren zu verbinden, ist aus Sicht des Anwendungsentwicklers leicht zu verstehen und basiert auf bekannten Methoden. Im Umgang mit dem System kann daher das gewünschte Resultat einer Modellierung mit relativ geringem Aufwand erzielt werden (vergleiche Modellierungsbeispiele in Abschnitt 5.6 und Anhang C). Die logische Aufteilung der Gesamtfunktionalität in einzelne Teilkomponenten hat nicht nur den Entwicklungsprozeß der Autorisierungsbibliothek selbst durch die Möglichkeit isolierbarer Testbarkeit beschleunigt, sondern unterstützt auch die bedarfsgerechte Kombination einzelner Module zu einem konkreten Autorisierungssystem. Damit erfüllt die Autorisierungsbibliothek die gestellten Anforderungen.
Durch die implementierten Zugriffskontrollbibliotheken kann darüber hinaus eine Vielzahl von Beschränkungen traditioneller Autorisierungskonzepte (und ihrer Realisierung in kommerziellen Zugriffskontrollsystemen) aufgehoben werden. Die nachfolgende Tabelle 6.1 diskutiert diese Verbesserungen überblicksartig.

Tabelle 6.1: Zusammenfassender Vergleich

Kriterium	Traditionelle Autorisierungskonzepte	Tycoon-Autorisierungsbibliothek
Flexibilität	keine oder geringe Adaptionsmöglichkeiten an neue Anforderungen	hohes Adaptionsvermögen durch Austauschbarkeit der Bibliothekskomponenten
Erweiterbarkeit	schwer und nur eingeschränkt	einfach durchführbar
Skalierbarkeit	monolithisch	bedarfsgerechte Kombination von Basisdiensten
Granularität der Zugriffsarten	fest vorgegeben, meist eingeschränkt auf Lesen, Schreiben, Ausführen ...	freie Definierbarkeit funktionsbasierter Zugriffsmodi
Granularität	fest vorgegeben, häufig beschränkt	Werte beliebiger Kom-

Fortsetzung auf der nächsten Seite

Fortsetzung der vorigen Seite

Kriterium	Traditionelle Autorisierungskonzepte	Tycoon-Autorisierungsbibliothek
der Objekte	auf System, Relation, Tupel ...	plexität und beliebigen Typs
Granularität der Subjekte	fest vorgegeben, nur Benutzer, Arbeitsgruppe, alle Subjekte ...	frei definierbare Subjekte, beliebig komplexe Gruppenhierarchien
Modellierung von Rechten	explizite Modellierung jedes einzelnen Rechtes	explizite Vergabe oder implizite Ableitung von Rechten

Die Implementierung eines objektorientierten Zugriffskontrolldienstes erhöht deutlich die Flexibilität hinsichtlich der Erweiterbarkeit, Skalierbarkeit und Adaptierbarkeit gegenüber traditionellen benutzerbestimmbaren Zugriffskontrollmethoden. In bezug auf die Modellierungsmächtigkeit wird eine echte Erweiterung erzielt, da sowohl die gesamte Funktionalität einfacher Zugriffskontrollisten als auch die speziellen Fähigkeiten objektorientierter Zugriffskontrolle zur Verfügung gestellt werden. Das zugrundeliegende objektorientierte Zugriffskontrollmodell ist dabei nicht nur in Form einer Bibliothek abgebildet worden, sondern existierende Einschränkungen sind erkannt und durch generelle Modellierungskonzepte ersetzt worden. Zu derartigen Verbesserungen zählen unter anderem die Ersetzung fest vorgegebener Subjekt-, Objekt- und Zugriffstypgranulate durch frei definierbare Werte generischer Typen und die Möglichkeit, Konnektoren zu definieren, die nur Rechte einer bestimmten Polarität (positiv oder negativ) vererben. Letztere eignen sich, um die durch das Modell geforderte gleichgerichtete und gegengerichtete Vererbung abzubilden; es können darüber hinaus jedoch auch Konnektoren für Rechte einer Polarität spezifiziert werden, ohne einen Konnektor für Rechte der entgegengesetzten Polarität spezifizieren zu müssen.

6.2 Ausnutzung von TL Sprachkonzepten

Typparameter leisten einen wichtigen Beitrag zur Überwindung bestehender Einschränkungen durch fest vorgegebene Granularitäten. Hierdurch können Funktionen und Typoperatoren beliebig parametrisiert werden, so daß beispielsweise eine Änderung des zu verwaltenden Subjekttyps einer Sicherheitsanwendung nicht grundsätzlich eine Änderung der Implementierung der Zugriffskontrollfunktionen nach sich zieht, sondern durch die bestehende Implementierung befriedigt werden kann. Ein solches Vorgehen vermeidet den Aufwand für Programmanpassungen, der entsteht, wenn zukünftige Anforderungen an Zugriffskontrollsysteme die Benutzung neuer Datentypen erfordern.
Die rekursiven Verflechtungen, die den verwendeten Datenstrukturen zugrundeliegen (siehe Abschnitt 5.2.1), bedingen darüber hinaus den Einsatz rekursiver Typen und Typoperatoren. Als weiteres Sprachkonzept finden abstrakte Datentypen Verwendung, deren Einsatzgebiet die Implementierung der Benutzerschnittstellen ist. Zum einen verstecken abstrakte Datentypen die Komplexität der Datentypen und Funktionen vor dem Anwender und erlauben nur fest definierte Zugriffsmöglichkeiten auf ihre Datenstruktur. Zum anderen sichert die typtechnisch erzwungene Trennung zwischen zwei Werten eines abstrakten Datentyps zu, daß nicht mit den Funktionen des einen Wertes auf die Struktur des anderen Wertes zugegriffen werden kann. Dies stellt eine wichtige Eigenschaft dar, weil auf diese Weise verschiedene Autorisierungsbereiche (Werte des abstrakten Typs *authorization.T*) sicher gegeneinander abgeschirmt werden. Schließlich hat sich auch die Orthogonalität der Sprache TL als vorteilhaft erwiesen, da die Konstruktion von Typen und Typoperatoren keinen Restriktionen durch die Sprache unterliegt.

6.3 Ausnutzung weiterer Tycoon-Konzepte

Zu den weiteren wichtigen Eigenschaften eines Zugriffskontrollsystems zählen die Verwaltung und persistente Speicherung modellierter und abgeleiteter Zugriffsrechte. Hierbei erweist sich das Tycoon-System als geeignet, da das Persistenzkonzept zu seinen grundlegenen Eigenschaften zählt. Aus dem Angebot an bereits existierenden Programmbibliotheken sind insbesondere Module zur Verwaltung von Massendaten, zur Iterationsabstraktion und zur Ein- und Ausgabe wiederverwendet worden. Verwendbare Grundbausteine für den Aufbau eines Zugriffskontrolldienstes sind jedoch nicht verfügbar gewesen und konnten erst durch die vorliegende Implementierung bereitgestellt werden. Das den Bibliotheken zugrundeliegende Konzept der Modularisierung leistet einen wichtigen Beitrag zur Beherrschung der implementierten Software, denn die Komplexität der Zugriffskontrollbibliothek (ca. 600 Kilobyte Quellprogramme) erfordert Möglichkeiten zur getrennten Entwicklung und Kompilierung einzelner Systemkomponenten. Auch der Offenheit der Tycoon-Umgebung kommt eine wichtige Bedeutung zu. Sie ist die notwendige Grundlage für die Einbindung eines Visualisierungsdienstes (hier: *daVinci*) gewesen.

6.4 Erweiterungsmöglichkeiten

Die Erstellung komplexer Programmbibliotheken ist stets einem dynamischen Prozeß unterworfen. Auch die vorliegende Arbeit kann daher nur vorläufig als fertiges Programmpaket angesehen werden. Nachfolgend sind daher eine Reihe möglicher Erweiterungen aufgeführt, die die Modellierungsflexibilität und Funktionalität weiter erhöhen können:

- Die Einführung von Zugriffskontrollmaßnahmen in real existierende Systeme führt zur Modellierung einer Vielzahl von Benutzerrechten. Dadurch ergibt sich meist ein Übersichtsproblem: Mit Hilfe der Funktionen der Zugriffskontrollbibliothek läßt sich zum Beispiel nicht entscheiden, für welche Kombinationen von Subjekten, Objekten und Zugriffstypen noch keine Autorisierungen (weder positiv noch negativ) existieren, ob und wo es latente Konflikte gibt und ob bestimmte Rechte “überflüssig” sind, das heißt ohne Einfluß auf die existierenden Zugriffsberechtigungen entfernt werden können. Auch die Darstellung der Kantenzüge, über die sich eine explizite Autorisierung zu einer bestimmten impliziten Autorisierung propagiert, ist bis dato nicht möglich. Da es in der Natur menschlicher Wahrnehmung liegt, komplexe Zusammenhänge eher in Bildern als in textuellen Repräsentationen zu erfassen, ist ein **Modul zur Visualisierung** des Autorisierungsgraphen mit der Möglichkeit, **Übersichtsanfragen** zu stellen, vorteilhaft. Ebenso erscheint es wünschenswert, auch aus einer graphischen Darstellung eines Autorisierungsgraphen heraus ein äquivalentes Autorisierungssystem in Tycoon zu erzeugen und alle Änderungen an diesem System durch Modifikationen an der graphischen Repräsentation auszudrücken. Eine Teilvisualisierung zur Darstellung von Subjekthierarchien und Propagierungshistorien delegierter Rechte ist - wie in Anhang C beziehungsweise Abschnitt 5.4 beschrieben - bereits Teil der existierenden Implementierung.

- Dieser Arbeit liegt ein objektorientiertes Zugriffskontrollmodell zugrunde. Prädestiniert für den Einsatz dieser Zugriffskontrollbibliothek sind daher auch objektorientierte Sprachen und Entwurfsumgebungen. Im Rahmen des europäischen FIDE-2[1]-Projektes sind im

[1] **F**ully **I**ntegrated **D**ata **E**nvironments

Forschungsumfeld dieser Arbeit die Entwurfsumgebung STYLE[2] [Wet94] und die Programmiersprache TooL[3] [GM95] entwickelt worden. Die Zugriffskontrollbibliotheken könnten daher auch in diesen Umgebungen eingesetzt werden.

- Die Autorisierungsbibliothek läßt sich zu einem Sicherheitskernsystem weiterentwickeln, wenn neben der Möglichkeit zur Vergabe, Prüfung und zum Entzug von Zugriffsrechten auch eine **Protokollierung**, Untersuchung und Auswertung **sicherheitsrelevanter Aktivitäten** (*auditing*) durchgeführt werden. Einen guten Überblick über dieses Teilgebiet der Computersicherheit bietet [DoD87b].

- Für die Generierung sicherer Funktionen stehen bisher nur Generatorfunktionen zur Verfügung. Diese bilden jeweils genau eine unsichere Funktion auf eine sichere Funktion ab. Ein Code-Generator (vergleiche Abschnitt 5.5), der aus einer kompletten ungeschützten Schnittstelle eine sichere Schnittstelle generiert, sollte den bestehenden Generatoransatz ergänzen.

- Die Autorisierungsbibliotheken sind auf der Basis eines diskreten Zugriffskontrollmodells implementiert worden, da dieses ein hohes Maß an Flexibilität gewährleistet. Erweitert man diese Autorisierungsbibliotheken um einen Dienst für die Modellierung regelbasierter mehrstufiger Autorisierungen, kann neben dem Zugriff auf Informationen auch der Informationsfluß kontrolliert werden. Während der Implementierungsaufwand auf der Applikationsebene dabei deutlich geringer ausfallen dürfte, stellt ein solcher Dienst erhöhte Anforderungen an die übrige Hard- und Software wie zum Beispiel den Einsatz eines mehrstufigen Betriebssystems (*multilevel operating system*) oder eines erweiterten Sicherheitskonzeptes für die Systemschicht des Tycoon-Systems.
  Existieren diese beiden Dienste parallel, so können sie für die Modellierung eines **hybriden Gesamtsystems** kombiniert werden. Während das mehrstufige Autorisierungssystem Subjekte und Objekte disjunkten Kompartimenten zuordnet und den Informationsfluß von Applikationen kontrolliert, kann benutzerbestimmbare Zugriffskontrolle innerhalb dieser Kompartimente sicherstellen, daß die Flexibilität der Sicherheitsmodellierung erhalten bleibt.

[2] **S**ystematics of **Ty**ped **L**anguage **E**nvironments
[3] **T**ycoon **o**bject-**o**riented **L**anguage

Anhang A

Das objektorientierte Paradigma

Die Implementierung der Zugriffskontrollbibliotheken ist in Anlehnung an ein objektorientiertes Zugriffskontrollmodell durchgeführt worden. Für das Verständnis der vorangegangenen Kapitel und der nachfolgenden Modellierungstechnik ist es daher erforderlich, die Grundzüge des objektorientierten Paradigmas einzuführen. Dieser Abschnitt erhebt dabei keinen Anspruch auf Vollständigkeit. Umfassendere Einführungen finden sich zum Beispiel in [Heu92] und [Boo91].

Im objektorientierten Ansatz ist die Trennung zwischen Daten und Funktionen, wie sie zum Beispiel im funktionalen und imperativen Programmierparadigma vorherrscht, aufgehoben. Ein Objekt ist eine diskrete unterscheidbare Entität [RBP+91]. Es besitzt einen Zustand, ein Verhalten und eine Identität. Der **Zustand** eines Objektes wird durch seine statischen Eigenschaften (Datenstruktur, Attribute) und die aktuellen (dynamischen) Werte dieser Eigenschaften beschrieben. Das **Verhalten** eines Objektes ist durch die Gesamtheit seiner Operationen (Methoden) charakterisiert. Methoden werden in Konstruktoren, Destruktoren, Mutatoren, Selektoren und Iteratoren klassifiziert. Daten können nur über die Methoden eines Objektes modifiziert werden (Datenkapselung). Auf diese Weise wird die strukturelle Komplexität eines Objektes versteckt. Im Vordergrund steht somit nur der Gebrauch der Daten selbst, nicht deren konkrete Implementierung [Tsa95].
Die **Objektidentität** ist die Eigenschaft, mit der ein Objekt von anderen Objekten unterschieden werden kann. Identitäten sind systemweit eindeutig, unveränderlich und nach außen nicht sichtbar. Zwei Objekte mit unterschiedlichen Identitäten sind somit auch dann verschieden, wenn sie den gleichen Zustand und das gleiche Verhalten besitzen.

In objektorientierten Datenbankmodellen umfaßt das Konzept der **Klasse** vier Gesichtspunkte [Mün94]: Eine Klasse liefert eine einheitliche Beschreibung für eine Menge von Objekten (intensionaler Aspekt des Klassenkonzeptes). Sie verwaltet eine Kollektion von Objekten, die eine gleiche Struktur und gleiches Verhalten aufweisen (extensionaler Aspekt des Klassenkonzeptes) [HK87]. Ein einzelnes Objekt dieser Menge bezeichnet man als **Instanz**, während die gesamte Kollektion, die konzeptuell unendlich viele Objekte enthalten kann, **Extension** der Klasse heißt. Ein Objekt ist stets an die Existenz einer Klasse gebunden, daher sind die Begriffe Objekt und Instanz austauschbar. Die Beschreibung eines Objektes kann sich im Laufe seiner Lebensdauer ändern, so daß es beispielsweise aus der Extension einer Klasse entfernt und einer anderen

hinzugefügt wird (Objektmigration). Auch die Extension einer Klasse besitzt somit einen Zustand, der sich ändern kann (modifizierender Aspekt des Klassenkonzeptes). Eine Klasse muß darüber hinaus die eindeutige Identifikation der Objekte ihrer Extension sicherstellen. Da die Identität eines Objektes nach außen nicht sichtbar ist, erfolgt seine Identifikation über die Werte seiner konkreten Beschreibung. Innerhalb einer Klassenextension muß daher Schlüsselintegrität gelten, das heißt es darf keine zwei Objekte geben, die die gleichen identifizierenden Werte besitzen (identifizierender Aspekt des Klassenkonzeptes).
Unter **Vererbung** versteht man die gemeinsame Nutzung von Attributen und Methoden durch verschiedene hierarchisch angeordnete Klassen. Klassen können in den nachfolgenden Beziehungen zueinander stehen:

Generalisierungsbeziehungen (*kind-of relationship***):** Klassen können Teil einer Hierarchie sein, die nach aufsteigender Spezialisierung geordnet ist: Generelle Klassen stehen im oberen Teil der Hierarchie, speziellere Klassen darunter. Eine spezielle Klasse (Subklasse) erbt alle Eigenschaften (Methoden und Attribute) einer generelleren Klasse (Superklasse) und fügt der Klassendefinition ihre eigenen Eigenschaften hinzu. Erbt eine Subklasse nur die Eigenschaften einer Superklasse, so spricht man von einfacher Vererbung; sind mehrere Superklassen involviert, handelt es sich um **multiple Vererbung**. Ändert sich die Semantik einer ererbten Eigenschaft, so kann diese durch eine Redefinition überschrieben werden (*overriding*). Klassendefinitionen sind somit wiederverwendbar; neuer Programmkode muß nur für zusätzliche Methoden geschrieben werden.

Aggregationsbeziehungen (*part-of relationship***):** Aggregationsbeziehungen dienen der Modellierung komplexer Objekte. Unter einem **komplexen Objekt** versteht man ein Aggregat, das aus mehreren Komponenten besteht. Diese Komponenten können ihrerseits wiederum komplex oder atomar sein. Ein **atomares Objekt** ist ein komplexes Objekt ohne Komponenten oder interne Beziehungen [Kel90]. Die Frage, welche Komponenten eines Systems atomar und welche komplex sind, hängt vom Abstraktionsgrad der betrachteten Anwendung ab [Boo91].
Komplexe Objekte können verschachtelt und überlappend sein und bilden eine Inklusionshierarchie. Ist ein Objekt Komponente zweier oder mehrerer komplexer Objekte, so bezeichnet man es als geteilt, anderenfalls als exklusiv. Im Gegensatz zu Vererbungsbeziehungen fügen aggregierte Klassen ererbten Eigenschaften in der Regel keine weiteren eigenen Eigenschaften hinzu.

Assoziationsbeziehungen: Assoziationen sind benannte, bidirektionale semantische Beziehungen zwischen Klassen. Die beteiligten Klassen einer Assoziation können mit Kardinalitäten versehen sein. Diese spezifizieren, wieviele Instanzen einer Klasse in Relation zu der mit ihr assoziierten Klasse stehen.

Metaklassenbeziehungen: Eine Metaklasse ist eine Klasse, deren Extension wiederum aus Klassen besteht. Auf diese Weise können Klassen, analog zu Instanzen, so behandelt werden, als seien sie modifizierbare Objekte.

Abbildung A.1 dient der Verdeutlichung einiger objektorientierter Konzepte. Dargestellt sind vier Klassen. Jede Klasse besitzt einen Namen, strukturelle Informationen (Attribute), Methoden, mit denen auf diese Attribute zugegriffen werden kann, und eine Extension. Attribute,

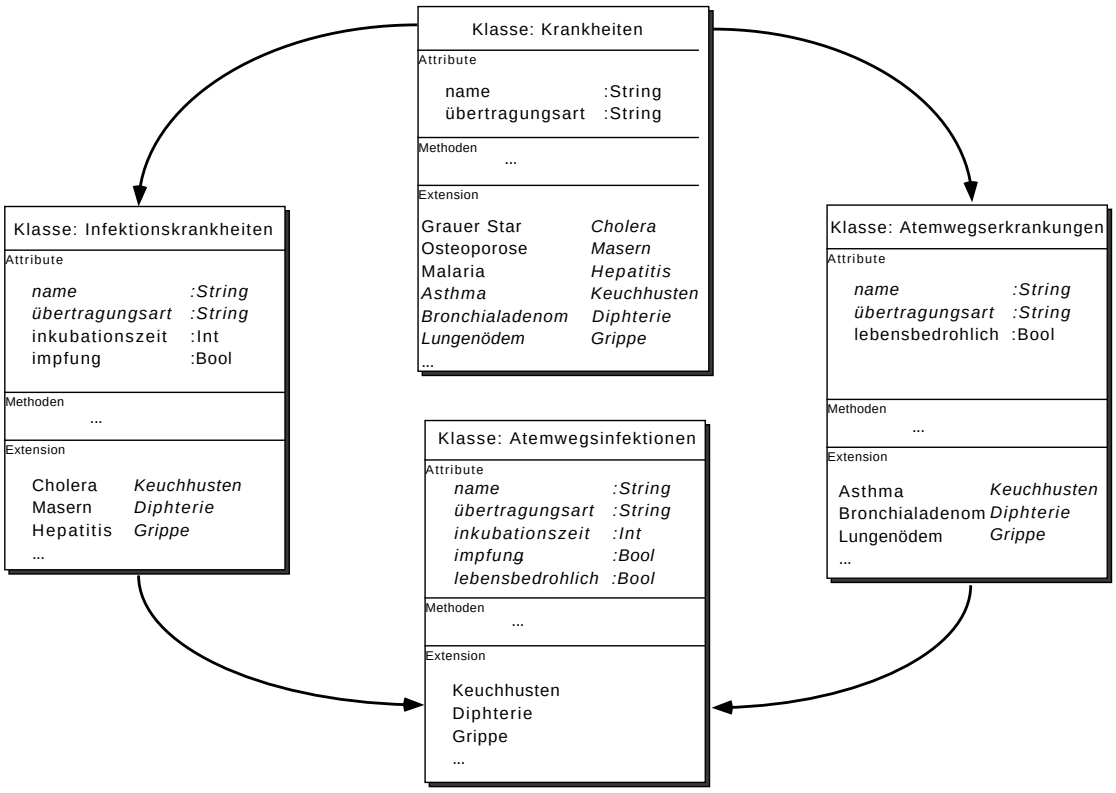


Abbildung A.1: Mehrfachvererbung

die in einer Klasse selbst definiert werden, sind in Normalschrift dargestellt, während ererbte Attribute kursiv geschrieben sind. Die Klasse *Krankheiten* ist eine Superklasse der Klassen *Atemwegserkrankungen* und *Infektionskrankheiten*, die wiederum Superklassen der Klasse *Atemwegsinfektionen* sind. Die vier Klassen stehen somit in Generalisierungsbeziehungen zueinander. Die Klassen *Infektionskrankheiten* und *Atemwegserkrankungen* erben die Attribute *name* und *übertragungsart* sowie die Methoden ihrer Superklasse *Krankheiten* (einfache Vererbung). Die Klasse *Atemwegsinfektionen* erbt die Eigenschaften der Klassen *Atemwegserkrankungen* und *Infektionskrankheiten* sowie implizit die Eigenschaften der Klasse *Krankheiten* (multiple Vererbung). Objekte, die in der Extension der Klasse *Atemwegsinfektionen* stehen, sind somit implizit auch in der Extension der anderen drei Klassen enthalten, da zum Beispiel eine Diphterie sowohl eine Atemwegsinfektion als auch eine Infektionskrankheit, eine Atemwegserkrankung und eine Krankheit ist.
Es zeigt sich also, daß die Kardinalität der Extension einer Klasse mit zunehmender Spezialisierung abnimmt, während die Zahl der zu ihrer Beschreibung notwendigen Attribute und Methoden zunimmt.

Anhang B

Visualisierung objektorientierter Systeme

Dieses Kapitel stellt überblicksartig eine graphische Notation für die Abbildung objektorientierter Systeme vor. Diese graphische Notation ist der Modellierungsmethode OMT[1] [RBP+91] entnommen und wird für die Darstellung objektorientierter Zugriffskontrollsysteme in Abschnitt 2.4 verwendet. OMT unterscheidet die folgenden drei Teilmodelle:

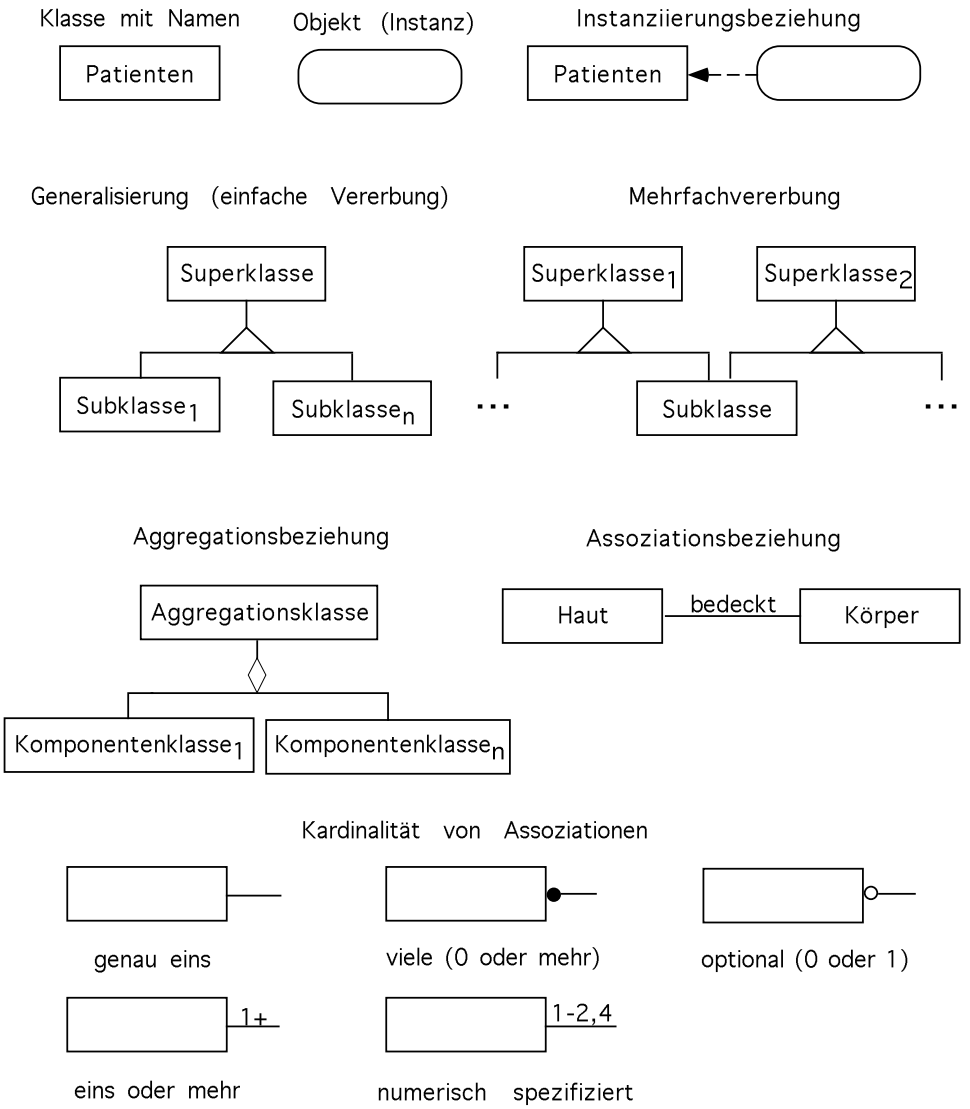
1. Das **Objektmodell** erlaubt die Spezifikation von Eigenschaften und Methoden von Objekten und Klassen sowie die Modellierung ihrer Beziehungen zueinander. Im Objektmodell werden somit die statischen Eigenschaften eines objektorientierten Systems beschrieben. Ziel des Modellierungsprozesses ist die Erstellung eines Objektdiagrammes. Dieses wird durch einen Graphen repräsentiert, dessen Knoten Klassen oder Objekte sind und dessen Kanten deren Beziehungen zueinander darstellen.

2. Aufgabe des **dynamischen Modells** ist es, Objektzustände und Zustandsübergänge, die durch Systemereignisse ausgelöst werden, zu modellieren.

3. Das **funktionale Modell** bildet Datenflüsse und Integritätsbedingungen ab. Es beschreibt die Veränderungen von Daten durch Prozesse unter Einhaltung der spezifizierten Bedingungen.

Das Objektmodell dient als Basis für die beiden anderen Modelle [Bös95]. Da alle verwendeten Beispiele objektorientierter Zugriffskontrolle nur deren statische Eigenschaften betreffen, genügt es, sich im folgenden auf das Objektmodell zu beschränken. Dabei werden primär die für das Verständnis der Beispiele relevanten Notationen des Objektmodells vorgestellt.

[1] **O***bject* **M***odelling* **T***echnique*



Aufbauend auf diesen Konstruktionsprimitiven veranschaulicht Abbildung B.1 die Modellierung einer konkreten Objekthierarchie. Diese ist dem Bereich der Anatomie entnommen und findet insbesondere in Abschnitt 2.4 Verwendung.
Der menschliche Körper wird als Aggregat aus einem Rumpf, einem Kopf, zwei Armen und zwei Beinen beschrieben. Der Rumpf besteht aus mehreren inneren Organe, zu denen das Herz, der Blinddarm und weitere Organe, die im konkreten Modell nicht von Interesse sind, gehören. Der Kopf besitzt einen Kiefer und ist Träger der sensorischen Organe. Diese umfassen unter anderem die Augen und Ohren. Der Kiefer schließlich besteht aus 32 Zähnen.
Dieses Beispiel verdeutlicht, daß es nicht erforderlich ist, alle Details der Realität abzubilden. Vielmehr hat sich die Modellierung auf genau die Aspekte zu beschränken, die später im objektorientierten Modell von Relevanz sind.

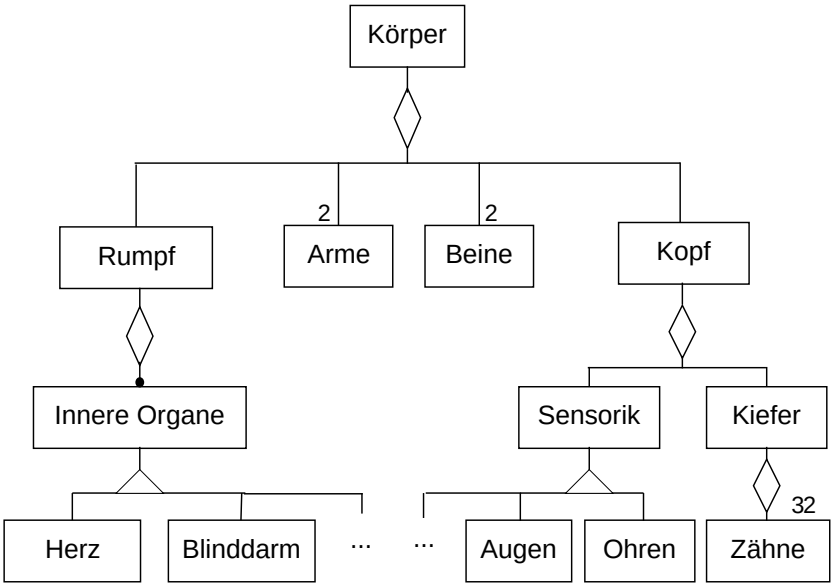


Abbildung B.1: Ein Modellierungsbeispiel

Anhang C

Visualisierung von Autorisierungsstrukturen

Die Komplexität eines Autorisierungssystems (oder dessen Teilkomponenten) im laufenden Betrieb kann am besten dadurch erfaßt werden, daß es graphisch dargestellt wird. Für diese Aufgabe wird auf die Funktionalität eines externen Diensterbringers, dem Visualisierungssystem *daVinci* [FW95, FW94], zugegriffen. *daVinci* wurde an der Universität Bremen entwickelt und ist ein interaktives Fenstersystem zur graphischen Repräsentation gerichteter Graphen. Zur Spezifizierung dieser Graphen stellt *daVinci* eine abstrakte Syntax zur Verfügung. Die konkrete Visualisierung von Komponenten des Autorisierungssystems, wie der Subjekthierarchie oder der Propagierungshistorie eines Rechtes, erfolgt über die Generierung ihrer abstrakten Syntax, die in einer Datei abgelegt wird. Aus dieser Datei konstruiert *daVinci* schließlich den Graphen und zeigt ihn an. Dieser Vorgang stellt aus der Sicht des Benutzers eine atomare Handlung dar; das Tycoon-System braucht dabei nicht verlassen zu werden.
Abbildung C.1 veranschaulicht diese drei Teilschritte synoptisch. Im oberen rechten Ausschnitt stehen die TL-Anweisungen, die den Subjektgraphen konstruieren. Die letzte Anweisung (*a.visualize*) generiert die darunter stehende abstrakte Syntaxbeschreibung für die Subjekthierarchie. Nachfolgend erfolgt ein Aufruf von *daVinci* mit der generierten Syntax und die Repräsentation des Subjektgraphen wird am Bildschirm angezeigt (linker Ausschnitt).

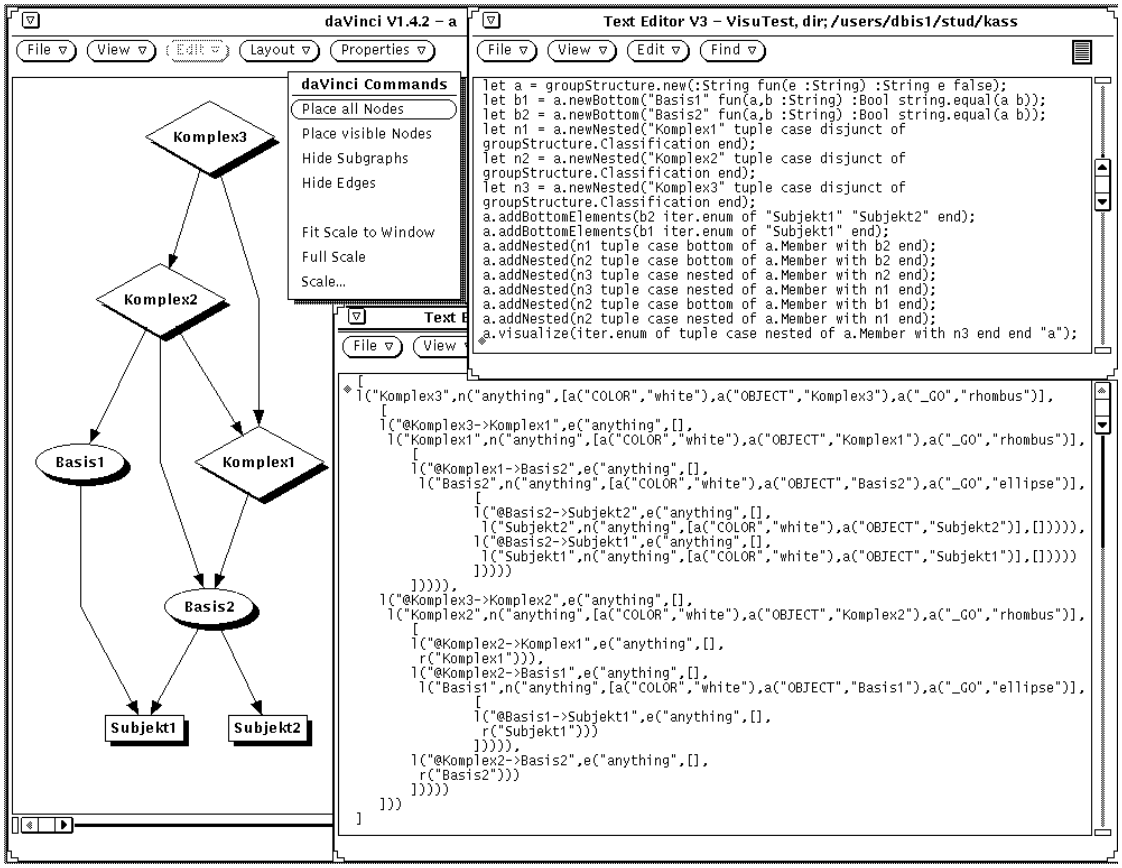


Abbildung C.1: Visualisierung einer Gruppenstruktur mit generierter Syntax

Anhang D

Ausgewählte Schnittstellen der Autorisierungsbibliothek

Die nachfolgenden Schnittstellen bilden eine kleine Auswahl der implementierten Module zur Zugriffskontrolle. Es handelt sich hierbei um die vier Komponenten der Zugriffskontrollbibliothek, aus denen sich der Anwender selbständig eigene Applikationen zusammenstellen kann.

D.1 Die Modulschnittstelle GroupStructure

Die Modulschnittstelle *GroupStructure* bietet Funktionen zum Aufbau und zur Verwaltung einer Subjekthierarchie beliebiger Komplexität an. Die Subjekthierarchie ist in Form eines abstrakten Datentyps implementiert; der Typ der verwalteten Subjekte ist frei wählbar.

```
interface GroupStructure
import iter group nameDB :GroupTypes
export
 Let Classification =
  Tuple
   case deepDisjunct
   case disjunct
  end
 (* defines a group policy for nested groups *)
 Let T(E <:Ok) = Tuple
  Nested <:Ok
  Bottom <:Ok
  error :Exception
  Let Member = Tuple
   case bottom with member :Bottom
```

case *nested* **with** *member :Nested*
end
(type for an arbitrary group (Bottom or Nested) *)*
setEqual(equalE(:Member :Member) :Bool) :**Ok**
newNested(name :String kind :Classification) :Nested
(creates a new empty group which can be filled with other groups *)*
newBottom(name :String equalE(:E :E) :Bool) :Bottom
(returns a new empty bottom group *)*
addNested(nestedGroup :Nested newMember :Member) :**Ok**
(adds an element to a nested group *)*
addNestedElements(nestedGroup :Nested elements :iter.T(Member)) :**Ok**
(adds a list of groups to a nested group *)*
addBottom(bottomGroup :Bottom newMember :E) :**Ok**
(adds an element to a bottom group *)*
addBottomElements(bottomGroup :Bottom elements :iter.T(E)) :**Ok**
(adds a list of elements to a bottom group *)*
subExisting(element :E) :Bool
(returns true if element is member of at least one bottom group *)*
groupsWithElement(element :E) :iter.T(Bottom)
(returns all bottom groups of which element is member *)*
memberNested(element :Member nestedGroup :Nested) :Bool
(tests if element is member of group *)*
memberBottom(element :E bottomGroup :Bottom) :Bool
(tests if element is member of group *)*
deepMember(element :E nestedGroup :Nested) :Bool
(tests if element of base type is member of any recursively reachable*
*bottom group *)*
deepGroups(nestedGroup :Nested) :iter.T(Member)
(returns all bottom groups of group *)*
deleteNested(element :Member nestedGroup :Nested) :**Ok**
(deletes an element from group *)*
deleteNestedElements(elements :iter.T(Member) nestedGroup :Nested): **Ok**
(deletes a list of elements from a bottom group *)*
deleteBottom(element :E bottomGroup :Bottom) :**Ok**
(deletes an element from a bottom group *)*
deleteBottomElements(elements :iter.T(E) bottomGroup :Bottom) :**Ok**
(deletes a list of elements from a bottom group *)*
nestedElements(nestedGroup :Nested) :iter.T(Member)
(returns all members of group *)*
bottomElements(bottomGroup :Bottom) :iter.T(E)
(returns all deep members of a bottom group *)*
deepElements(nestedGroup :Nested) :iter.T(E)
(returns all deep members of group *)*
getName(nestedGroup :Nested) :String
(returns the name of a group *)*
getBottomName(bottomGroup :Bottom) :String
(returns the name of a bottom group *)*

getByName(name :String): Member
(checks if any member with name exists and returns it. Raises error*
*if no such member exists *)*
getSuperGroups(nestedGroup :Nested) :iter.T(Nested)
(returns an iteration over all groups which have nestedGroup as a*
*subgroup *)*
transitiveSuperGroups(nestedGroup :Nested) :iter.T(Nested)
(returns an iteration over all groups which have nestedGroup as a*
*transitive reachable subgroup *)*
getBottomSuperGroups(bottomGroup :Bottom) :iter.T(Nested)
(returns an iteration over all groups which have bottomGroup as a*
*subgroup *)*
transitiveBottomSuperGroups(bottomGroup :Bottom) :iter.T(Nested)
(returns an iteration over all groups which have bottomGroup as a*
*transitive reachable subgroup *)*
getClassification(nestedGroup :Nested) :Classification
(returns the classification of a group *)*
changeClassification(nestedGroup :Nested
*newClassification :Classification) :***Ok**
(changes the classification of a group if possible *)*
size(nestedGroup :Nested) :Int
(returns the number of elements in group *)*
bottomSize(bottomGroup :Bottom) :Int
(returns the number of elements in a bottom group *)*
deepSize(nestedGroup :Nested) :Int
(returns the number of all base elements, transitiv reachable *)*
numberOfSubGroups(nestedGroup :Nested) :Int
(returns the number of all subGroups that are transitive reachable *)*
cyclesExisting() :Bool
(returns true if group structure contains a least one circle *)*
depth(nestedGroup :Nested) :Int
(returns the maximum depth of group, i.e. the number of hierachy levels *)*
empty(nestedGroup :Nested) :Bool
(checks if group is empty *)*
emptyBottom(bottomGroup :Bottom) :Bool
(checks if bottom group is empty *)*
*clear(nestedGroup :Nested) :***Ok**
(deletes all elements in group *)*
*clearBottom(bottomGroup :Bottom) :***Ok**
(deletes all elements in a bottom group *)*
copy(nestedGroup :Nested name :String) :Nested
(returns a group with exactly all elements of group *)*
copyBottom(bottomGroup :Bottom name :String) :Bottom
(returns a bottom group with exactly all elements of group *)*
disjunct(groupA, groupB :Nested) :Bool
(returns true if both groups don't have a common element *)*
disjunctBottom(groupA, groupB :Bottom) :Bool

```
  (* returns true if both bottom groups don't have a common element *)
  overlapping(groupA, groupB :Nested) :Bool
  (* returns true if both groups have a least one common element *)
  overlappingBottom(groupA, groupB :Bottom) :Bool
  (* returns true if both bottom groups have a least one common element *)
  included(minGroup, maxGroup :Nested) :Bool
  (* returns true if all elements of minbGroup are in maxGroup too *)
  includedBottom(minGroup, maxGroup :Bottom) :Bool
  (* returns true if all elements of minGroup are in maxGroup too *)
  isomorph(groupA, groupB :Nested) :Bool
  (* checks, if all subgroups of groupA and groupB are equal and have the
     same structure. Equality of groupA and groupB is not tested *)
  baseDisjunct(groupA, groupB :Nested) :Bool
  (* returns true if groupA and groupB do not have a common base element *)
  intermediateDisjunct(groupA, groupB :Nested) :Bool
  (* returns true if all subgroups of groupA and groupB are equal. Bottom-
     Groups and their elements are not tested for equality *)
  baseEqual(groupA, groupB :Nested) :Bool
  (* returns true if both groups have the same number and type of bottom
     groups *)
  intermediateEqual(groupA, groupB :Nested) :Bool
  (* returns true if all nested subgroups of groupA and groupB are equal *)
  visualize(groups :iter.T(Member)  file :String) :Ok
  (* opens a daVinci graph editor and visualizes the groupStructure with
     groups as the root nodes of the tree. A textual representation is written
     to file. This function can be used for cyclic and acyclic graphs. *)
end

new(E <:Ok  extractName(:E) :String  cyclic :Bool) :T(E)
(* creates a new group structure. Cyclic denotes whether the groups of
   the group structure are allowed to be composed with cycles *)

end;
```

D.2 Die Modulschnittstelle Authorization

Die Modulschnittstelle *Authorization* stellt Zugriffskontrollabstraktionen für die Modellierung geschützter Werte und deren Vererbungsbeziehungen zueinander bereit.

```
interface Authorization

import
  iter
export
  Let Element = Ok
  (* Identity of an Object that will be controlled by the security policy
     (normally a function) *)
  AuthStrategy <:Ok
  (* Strategy that is used iff there is neither a permission nor a prohibition
     for a user to access a function. Optimistic denotes that in such cases
     the access is allowed; pessimistic means to forbid the access. *)
  optimistic,
  pessimistic :AuthStrategy
  error :Exception
  Let T(Subject <:Ok) <:Ok =
    Tuple
      Template(E <:Ok) <:Ok
      Instance(E <:Ok) <:Ok
      Node(E <:Ok) <:Ok (* Type for Source-Valueguard *)
      Edge (E <:Ok) <:Ok (* Type for Implication *)
      CrossRef <:Ok
      InnerRef, IntraRef, InterRef <:Ok
      newTemplate(E <:Ok  auth :AuthStrategy) :Template(E)
      (* creates a new template *)
      isNodeOf(E <:Ok  t :Template(E)  node :Node(E)) :Bool
      (* checks, if node was created by template t *)
      getTemplate(E <:Ok  node :Node(E)) :Template(E)
      (* returns the template to which node (source valueguard) belongs *)
      getAuthStrategy(E <:Ok  node :Node(E)) :AuthStrategy
      (* returns the authorization strategy which is valid for node *)
      instances(E <:Ok  t :Template(E)) :iter.T(Instance(E))
      (* returns all instances that belong to t *)
      addInstance(E <:Ok  t :Template(E)  inst :E  name :String) :Instance(E)
      (* adds an instance to a template *)
      deleteInstance(E <:Ok  i :Instance(E)) :Ok
      (* deletes an instance from a template *)
      getInst(E <:Ok  i :Instance(E)) :E
      (* returns the element inst added with addInstance *)
      addElement(E <:Ok  t :Template(E)  e :Element  name :String
        owner :Subject) :Node(E)
      (* adds an element to a template and returns a handle for a source
```

```
    valueguard *)
  deleteElement(E <:Ok  node :Node(E)  owner :Subject) :Ok
  (* deletes a node (sourve valueguard) and all its refernces and
    invalidates the users handle *)
  addImplication(E <:Ok  source, destination :Node(E)
   name :String  pos :Bool) :Edge(E)
  (* connects two nodes (source valueguards) of a template, raises error if such a
    connection already exists. If pos is true only positive rights are inherited
    via this implication; otherwise only negative rights are inherited *)
  deleteImplication(E <:Ok  impl :Edge(E)) :Ok
  (* removes an implication an invalidates the user's handle *)
  insert,
  insertWeak,
  insertWeakNeg,
  insertNeg(E <:Ok  node :Node(E)  s :Subject  owner :Subject) :Ok
  (* inserts a subject into the (positive or negative) ACL of a template
    node (source valueguard). Its rights are valid for all corresponding
    nodes in the instances *)
  deleteWeakNeg,
  deleteNeg,
  deleteWeak,
  delete(E <:Ok  node :Node(E)  s :Subject  owner :Subject) :Ok
  (* deletes a subject from the ACL of a template node (source valueguard) *)
  exists,
  existsWeak,
  existsWeakNeg,
  existsNeg(E <:Ok  node :Node(E)  s :Subject) :Bool
  (* checks if an explicit right for a template node (source valueguard) exists *)
  existsNegInstTemp,
  existsInstTemp,
  existsNegInst,
  existsInst(E <:Ok  node :Node(E)  s :Subject  i :Instance(E)) :Bool
  (* checks if an explicit or implicit right for an instance node
    (valueguard) exists; suffix -Temp denotes that this right was inherited
    from the ACL of a template node (source valueguard); suffix -Inst means
    that it was inherited by the ACL of an instance node (valueguard)*)
 existsWeakInst,
 existsWeakInstNeg,
 existsWeakTemp,
 existsWeakTempNeg(E <:Ok  node :Node(E)  s :Subject  i :Instance(E)) :Bool
  addLink(E,F <:Ok  nodeFrom :Node(E)  nodeTo :Node(F)  pos :Bool) :CrossRef
  (* adds a crossreference between two nodes (source valueguards) of
    different templates, raises error if such a link already exists. If pos
    is true only positive rights are inherited via this crossreference;
    otherwise only negative rights are inherited *)
  deleteLink(c :CrossRef) :Ok
  (* removes a crossreference and invalidates the users handle *)
```

```
    allowed(E <:Ok  node :Node(E)  s :Subject  i :Instance(E)) :Bool
    (* important function that checks if a subject has a positive right (permission)
       to access the object associated with an instance node (valueguard) *)
    insertInstACL(E <:Ok  node :Node(E)  s:Subject  i :Instance(E)  pos :Bool
      owner :Subject) :Ok
    (* inserts a subject into the ACL of an instance node (valueguard) *)
    deleteInstACL(E <:Ok  node :Node(E)  s:Subject  i :Instance(E)
      owner :Subject) :Ok
    (* deletes a subject from the ACL of an instance node (valueguard) *)
    insertWeakInst(E <:Ok  node :Node(E)  s:Subject  i :Instance(E)
      pos :Bool  owner :Subject) :Ok
    (* inserts a subject into the ACL for weak rights of an instance node (valueguard) *)
    deleteWeakInst(E <:Ok  node :Node(E)  s:Subject  i :Instance(E)
      owner :Subject) :Ok
    (* deletes a subject from the ACL for weak rights of an instance node (valueguard) *)
    addInter(E,F <:Ok  nodeFrom :Node(E)  nodeTo :Node(F)  instFrom :Instance(E)
      instTo :Instance(F)  pos :Bool) :InterRef
    (* adds an inter-instance reference, i.e. a connection between two instance nodes
       (valueguards) belonging to different templates, raises error if such a
       connection already exists. If pos is true only positive rights are inherited
       via this instreference; otherwise only negative rights are inherited *)
    addIntra(E <:Ok  nodeFrom,nodeTo :Node(E)
      instFrom, instTo :Instance(E)  pos :Bool) :IntraRef
    (* adds an intra-instance reference, i.e. a connection between two
       instance nodes (valueguards) belonging to the same template but
       different instances. If pos is true only positive rights are inherited
       via this instreference; otherwise only negative rights are inherited *)
    addInner(E <:Ok  nodeFrom, nodeTo :Node(E)  inst :Instance(E)
      pos :Bool) :InnerRef
    (* adds an inner-instance reference, i.e. a connection between two
       instance nodes (valueguards) belonging to the same instance. If pos is
       true only positive rights are inherited via this instreference;
       otherwise only negative rights are inherited *)
    deleteInter(i :InterRef) :Ok
    (* deletes an inter-instance reference and invalidates the users handle *)
    deleteIntra(i :IntraRef) :Ok
    (* deletes an intra-instance reference and invalidates the users handle *)
    deleteInner(i :InnerRef) :Ok
    (* deletes an inner-instance reference and invalidates the users handle *)
    visualizeInstances(E <:Ok  t:Template(E)  file :String) :Ok
    (* opens a daVinci graph editor and shows the instances of t *)
  end

  new(Subject <:Ok  null :Subject  exName(:Subject) :String
    eq(:Subject  :Subject) :Bool) :T(Subject)

end;
```

D.3 Die Modulschnittstelle Granting

Propagierungshistorien diskreter Autorisierungen werden mit Hilfe der Modulschnittstelle *Granting* erzeugt und verwaltet.

```
interface Granting
import
export

  Let T(Subject, Object <:Ok) = Tuple
    Element <:Ok
    Key <:Ok
    grant(grantor, grantee :Subject  object :Object  grantOption :Bool) :Ok
    (* Grantor enables access on object for grantee. GrantOption determines
       whether grantee is allowed to grant access on object to further users *)
    grantOwnership(grantor :Subject  object :Object) :Ok
    (* grantor becomes owner of object *)
    getOwner(object :Object) :Subject
    (* returns the owner of object, raise error if no owner exists *)
    owner(subject :Subject  object :Object) :Bool
    (* returns true if subject has ownership for object *)
    revoke(grantor, grantee :Subject  object :Object) :Ok
    (* Grantor revokes access on object for grantee. All grants that were given
       by grantee will be deleted as well *)
    revokeOwnership(owner :Subject  object :Object) :Ok
    (* drops owner relation between owner and object and deletes all inherited
       access rights *)
    changeGrantOption(grantor, grantee :Subject  object :Object) :Ok
    (* grantor changes  grantOption attribute for access on object by
       grantee to false for all grants that have been given so far. *)
    changeOwnership(oldOwner, newOwner :Subject  object :Object
      keepUsers :Bool) :Ok
    (* replaces old owner of object with new owner. If keepUsers is true all
       transitive inherited access grants are kept, otherwise
       revokeOwnership(oldOwner object) and grantOwnership(newOwner object) is
       performed *)
    accessAllowed(user :Subject  object :Object) :Bool
    (* returns true if user has grant to use object *)
    visualize(object :Object  file :String  withTimeStamps :Bool) :Ok
    (* opens a daVinci graph editor that shows the propagation of a right on
       object.*)
  end

  error :Exception

  new(Subject, Object <:Ok  extractName1(:Subject) :String
     extractName2(:Object) :String) :T(Subject Object)
```

end*;*

D.4 Die Modulschnittstelle AuthGenerators

Generatorfunktionen, die aus einer ungeschützten Funktion und einem konkreten Autorisierungssystem eine geschützte Funktion erzeugen, bietet die Modulschnittstelle *AuthGenerators* an. Da Funktionen verschiedener Arität im allgemeinen nicht in einer Subtypbeziehung zueinander stehen, muß je eine Generatorfunktion für jede mögliche Anzahl von Eingabeparametern einer Funktion bereitgestellt werden.

interface *AuthGenerators*

import

authorization

export

error :**Exception**

generate0(E,N,Subject <:**Ok** *f() :E auth :authorization.T(Subject)*
node :auth.Node(N) convert() :N eq(:N :N) :Bool) :**Fun***(sub :Subject) :E*

generate1(E,F,N,Subject <:**Ok** *f(:E) :F auth :authorization.T(Subject)*
node :auth.Node(N) convert(:E) :N eq(:N :N) :Bool) :**Fun***(:E sub :Subject) :F*

generate2(E,F,G,N,Subject <:**Ok** *f(:E :F) :G auth :authorization.T(Subject)*
node :auth.Node(N) convert(:E :F) :N
eq(:N :N) :Bool) :**Fun***(:E :F sub :Subject) :G*

generate3(E,F,G,H,N,Subject <:**Ok** *f(:E :F :G) :H auth :authorization.T(Subject)*
node :auth.Node(N) convert(:E :F :G) :N
eq(:N :N) :Bool) :**Fun***(:E :F :G sub :Subject) :H*

generate4(E,F,G,H,I,N,Subject <:**Ok** *f(:E :F :G :H) :I*
auth :authorization.T(Subject) node :auth.Node(N) convert(:E :F :G :H) :N
eq(:N :N) :Bool) :**Fun***(:E :F :G :H sub :Subject) :I*

generate5(E,F,G,H,I,J,N,Subject <:**Ok** *f(:E :F :G :H :I) :J*
auth :authorization.T(Subject) node :auth.Node(N) convert(:E :F :G :H :I) :N
eq(:N :N) :Bool) :**Fun***(:E :F :G :H :I sub :Subject) :J*

end*;*

Literaturverzeichnis

[Abr93] Marshall D. Abrams. Renewed Understanding of Access Control Policies. In *Proceedings of the 16th National Computer Security Conference - Information System Security: User Choices*, Seite 87–96. National Institute of Standards and Technology, 1993.

[ACC81] M. P. Atkinson, K. J. Chisholm, und W. P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN notices*, 17(7), 1981.

[ACO85] A. Albano, L. Cardelli, und R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[BCCGY94] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, und K. Yazdanian. Multilevel Security in Object-Oriented Databases. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 79–89. 1994.

[BEG95] Rüdiger Buck-Emden und Jürgen Galimow. *Die Client/Server-Technologie des Systems R/3*. Addison Wesley Verlag, 1995.

[Ber93] Elisa Bertino. Architectural Issuses in OODBMS. Seminarunterlagen zur EDBT Summer School in Leysin (Schweiz), September 1993.

[Bib77] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technischer Bericht MTR-3153, MITRE Corporation, Bedford, Juni 1977.

[Bir94] Barry Bird. The Security Facilities of PCTE. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 25–42, 1994.

[BJS94] E. Bertino, S. Jajodia, und P. Samarati. Enforcing Mandatory Access Control in Object Bases. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 96–116. 1994.

[BL73] D. E. Bell und L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technischer Bericht ESD-TR-73-278, Vol. 1, The MITRE Corporation, Bedford, Massachusetts, 1973.

[BL76] D. E. Bell und L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technischer Bericht ESD-TR-75-306, The MITRE Corporation, Bedford, Massachusetts, 1976.

[Ble94] Gerrit Bleumer. Security for Decentralized Health Information Systems. In B. Barber, A. R. Bakker, und S. Bengtsson, Hrsg., *Caring for Health Information - Safety, Security, Secrecy*, Seite 139–146. Elsevier Science, 1994.

[BN89] D. F. C. Brewer und J. W. Nash. The Chinese Wall Security Policy. In *Proceedings 1989 IEEE Symposium on Security and Privacy*, Oakland, California, 1989. IEEE Computer Society Press.

[Boo91] Grady Booch. *Object Oriented Design With Applications.* Benjamin-Cummings Publishing Company Inc., 1991.

[BOS94] E. Bertino, F. Origgi, und P. Samarati. A New Authorization Model for Object-Oriented Databases. In J. Biskup, M. Morgenstern, und C.E. Landwehr, Hrsg., *Database Security.* Universität Hildesheim, August 1994.

[Bös95] Andreas Bösch. PolitIcon: Entwurf einer Bilddatenbank für den Index zur politischen Ikonographie. Studienarbeit am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, September 1995.

[Brü92] Hans H. Brüggemann. Rights in an Object-Oriented Environment. In C. E. Landwehr und S. Jajodia, Hrsg., *Database Security V*, Seite 99–115, 1992.

[Brü93] Hans H. Brüggemann. Object-Oriented Authorization. In J. Paredaens und L. Tenenbaum, Hrsg., *Advances in Database Systems, Implementations and Applications*, Band 347, Seite 139–160. 1993.

[Bry94] Ciarán Bryce. An Access Control Model for a Parallel Object-Based Programming Language. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 166–182, 1994.

[BS94a] E. Bertino und P. Samarati. Research Issues in Discretionary Authorizations for Object Bases. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 183–199, 1994.

[BS94b] Eike Born und Helmut Stiegler. Discretionary access control by means of usage conditions. In *Computers & Security*, Band 13, Seite 437–450. 1994.

[BW94] E. Bertino und H. Weigand. An approach to authorization modeling in object-oriented database systems. In P. P. Chen, Hrsg., *Data & Knowledge Engineering*, Band 12, Seite 1–29. 1994.

[Car89] Luca Cardelli. Typeful Programming. Technischer Bericht 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, Kalifornien, Mai 1989.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technischer Bericht, Digital Equipment Corporation, Systems Research Center, Palo Alto, Kalifornien, Juni 1994.

[CFMS95] Silvana Castano, Maria G. Fugni, Giancarlo Martella, und Pierangela Samarati. *Database Security.* Addison Wesley Verlag, 1995.

[CGG+77] D. D. Chamberlain, J. N. Gray, P. P. Griffiths, I. L. Traiger, und B. W. Wade. Data Base System Authorization. Technischer Bericht, IBM Research Division, 1977.

[CL90] L. Cardelli und G. Longo. A semantic basis for Quest. Technischer Bericht 55, Digital Equipment Corporation, Systems Research Center, Palo Alto, Kalifornien, März 1990.

[CMMS91] L. Cardelli, S. Martini, J. C. Mitchell, und A. Scedrov. An Extension of System F with Subtyping. In T. Ito und A.R. Meyer, Hrsg., *Theoretical Aspects of Computer Software, TACS 91*, Lecture Notes in Computer Science, Seite 750–770. Springer-Verlag, 1991.

[Cor91] J. R. Corbin. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer Verlag, 1991.

[CW85] L. Cardelli und P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. In *ACM: Computing Surveys*, Band 17(4), Seite 477–522, Dezember 1985.

[DCBM89] A. Dearle, R. Connor, F. Brown, und R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages*, Portland, Oregon, Juni 1989.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley-Verlag, 1982.

[DES77] Data Encryption Standard. Federal Information Processing Standards, no. 46, National Bureau of Standards, U.S. Department of Commerce, 1977.

[DHP89] K. R. Dittrich, M. Härtig, und H. Pfefferle. Discretionary Access Control in Structurally Object-Oriented Database Systems. In C. E. Landwehr, Hrsg., *Database Security II*, Seite 105–122, 1989.

[Dit83] Klaus R. Dittrich. *Ein universelles Konzept zum flexiblen Informationsschutz in und mit Rechensystemen.* Nummer 75 in Informatik-Fachberichte. Springer-Verlag, 1983.

[Dit94] Klaus R. Dittrich. Current Trends in Database Technology and Their Impact on Security Concepts. In J. Biskup, M. Morgenstern, und C. Landwehr, Hrsg., *Proceedings of the IFIP WG 11.3 Eighth Annual Working Conference on Database Security*, Universität Hildesheim, Institut für Informatik, August 1994.

[DLS+87] D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, und W. R. Heckman. A Multilevel Relational Data Model. In *Proceedings of the IEEE Symposium on Security and Privacy*, Seite 220–234, April 1987.

[DNP87] D. E. Denning, P. G. Neumann, und D. B. Parker. Social Aspects of Computer Security. In *10th National Computer Security Conference*, Seite 320–325, Baltimore, USA, 1987.

[DoD85] Trusted Computer Security Evaluation Criteria. Technischer Bericht, Department of Defense, National Computer Security Center, 1985.

[DoD87a] A Guide to Understanding Discretionary Access Control in Trusted Systems. Technischer Bericht S-228,576, Department of Defense, National Computer Security Center, Fort George G. Meade, Maryland, 1987.

[DoD87b] A Guide to Understanding Audit in Trusted Systems. Technischer Bericht, Department of Defense, National Computer Security Center, 1987.

[DRKJ85] D. D. Downs, J. R. Rub, C. K. Kung, und C. S. Jordan. Issues in Discretionary Access Control. In *Proceedings 1985 IEEE Symposium on Security and Privacy*, Seite 208–218, April 1985.

[Fag77] Ronald Fagin. On an Authorization Mechanism. Technischer Bericht, IBM Research Division, 1977.

[Fai94] Belinda Fairthorne. OMG White Paper on Security. Technischer Bericht, Object Management Group, Security Working Group, April 1994.

[FGS89] E. B. Fernández, E. Gudes, und H. Song. A Security Model for Object-Oriented Databases. In *Security and Privacy*, 1989.

[FGS91] E. B. Fernández, E. Gudes, und H. Song. Evaluation of Negative, Predicate and Instance-based Authorization. In C. E. Landwehr und S. Jojodia, Hrsg., *Database Security IV*, Seite 85–98, 1991.

[FH92] Simone Fischer-Hübner. IDA (Intrusion Detection and Avoidance System): Ein einbruchsentdeckendes und einbruchsvermeidendes System. Dissertation am Fachbereich Informatik der Universität Hamburg, Juli 1992.

[FH95] Simone Fischer-Hübner. Considering Privacy as a Security-Aspect: A Formal Privacy-Model. DASY Papers 5/95, Institut for Anvendt Datalogi og Systemvidenskab, Handelshøjskolen i København, Frederiksberg, Dänemark, 1995.

[FHB90] S. Fischer-Hübner und K. Brunnstein. Combining Verified and Adaptive System Components Towards More Secure Computer Architectures. In J. Rosenberg und J. L. Keedy, Hrsg., *Security and Persistence*, Workshops in Computing, Seite 301–308, Bremen, Mai 1990. Springer-Verlag.

[FK93] David Ferraiolo und Richard Kuhn. Role-Based Access Controls. In *Information Systems Security: Building Blocks to the Future, 15th Computer Security Conference*, Band II, Seite 137–155. National Institute of Standards and Technology, 1993.

[FKK90] B. Freisleben, P. Kammerer, und J. L. Keedy. Capabilities and Encryption: The Ultimate Defense Against Security Attacks? In J. Rosenberg und J. L. Keedy, Hrsg., *Security and Persistence*, Bremen, 1990. Springer Verlag.

[FLPG94] E. B. Fernández, M. M. Larrondo-Petrie, und E. Gudes. A Method-Based Authorization Model for Object-Oriented Databases. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 135–150, 1994.

[For94] Karen A. Forcht. *Computer Security Management.* Boyd & Fraser Verlag, 1994.

[FW94] Michael Fröhlich und Matthias Werner. The Graph Visualization System *daVinci*-A User Interface for Applications. Technischer Bericht 5/94, Universität Bremen, Fachbereich Informatik, September 1994.

[FW95] Michael Fröhlich und Matthias Werner. *daVinci V1.4 User Manual.* Universität Bremen, Juni 1995.

[FWF94] Eduardo B. Fernández, Jie Wu, und Minjie H. Fernández. User Group Structures in Object-Oriented Database Authorization. In J. Biskup, M. Morgenstern, und C. E. Landwehr, Hrsg., *Database Security.* Universität Hildesheim, August 1994.

[Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, 1995.

[Gas88] Morrie Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold Verlag, 1988.

[Gei95] Andreas Geisler. Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache. Diplomarbeit am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, Juni 1995.

[GLL89] R. Gapliardi, G. Lapis, und B. Lindsay. A flexible and efficient Database Authorization Facility. Technischer Bericht, IBM Research Division, 1989.

[GM94] Andreas Gawecki und Florian Matthes. The Tycoon Machine Language TML: An Optimizable Persistent Program Representation. Technischer Bericht FIDE/94/100, Fachbereich Informatik der Universität Hamburg, August 1994.

[GM95] Andreas Gawecki und Florian Matthes. TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification. Technischer Bericht, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich DBIS, 1995.

[GW76] P. P. Griffith und B. W. Wade. An Authorization Mechanism for a Relationale Database System. In *ACM Transactions on Database Systems*, Band 1(3), Seite 242–255, September 1976.

[Har81] H. Rex Hartson. Database Security-System Architectures. In *Information Systems*, Band 6, Seite 1–22, 1981.

[Heu92] Andreas Heuer. *Objektorientierte Datenbanken*. Addison-Wesley Verlag, 1992.

[HK87] R. Hull und R. King. Semantic Database Modelling: Survey Applications and Research Issues. In *ACM: Computing Surveys*, Band 19(3), 1987.

[Hof77] Lance J. Hoffman. *Modern Methods for Computer Security and Privacy*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Hof82] Christoph Hofman. *Group-Theoretic Algorithms and Graph Isomorphism.* Springer-Verlag, 1982.

[Hof90] Lance J. Hoffman. *Rogue Programs: Viruses, Worms and Trojan Horses*. Van Nostrand Reinhold Verlag, 1990.

[Hos92] Hilary H. Hosmer. The Multipolicy Paradigm. Technischer Bericht, Data Security Inc., 58 Wilson Road, Bedford, MA 01730, 1992.

[HRU76] M. A. Harrison, W. L. Ruzzo, und J. D. Ullman. Protection in Operating Systems. In *Communications of the ACM*, Band 19(8), Seite 461–471, August 1976.

[JKP93] M. Jurecic, U. Kohl, und E. Pelikan. Datenschutz und Datensicherheit für verteilte Klinikanwendungen. In *Entwicklung und Management verteilter Anwendungssysteme.* GI/ITG Fachgruppe Kommunikation und Verteilte Systeme, Universität Frankfurt, 1993.

[JMMN90] C. Jensen McCollum, J. R. Messing, und L. Notargiacomo. Beyond the Pale of MAC and DAC - Defining New Forms of Access Control. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, Seite 190–200, 1990.

[Kaß94] Thomas Kaß. Objektorientierte Datenmodellierung: Ein Klasseneditor zur Entwurfsunterstützung. Studienarbeit am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, Januar 1994.

[Kel90] Udo Kelter. Group-Oriented Discretionary Access Controls for Distributed Structurally Object-Oriented Database Systems. Technischer Bericht, Fernuniversität Hagen, Arbeitsbereich Praktische Informatik, 1990.

[Kel91] Udo Kelter. Discretionary Access Controls in a High-Performance Object Management System. In *Proceedings 1991 IEEE Symposium on Security and Privacy*, 1991.

[Kir94] Plamen Kiradjiev. Dynamische Optimierung in CPS-orientierten Zwischensprachen. Diplomarbeit am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, November 1994.

[KN93] J. Kohl und B. C. Neuman. The Kerberos Network Authentication Service (V5). working draft - currently released as Internet Draft, April 1993.

[Lan81] C. E. Landwehr. Formal models for computer security. In *ACM: Computing Surveys*, Band 13(3), September 1981.

[LH94] E. E. O. R. Lindgreen und I. S. Herschberg. On the validity of the Bell-LaPadula model. *Computers & Security*, 13(4):317–333, 1994.

[Lip91] S. B. Lipner. Criteria, evaluation and the international environment. In D. T. Lindsay, Hrsg., *IFIP Information Security*, 1991.

[LS87] Peter Lockemann und Joachim W. Schmidt. *Datenbankhandbuch*. Springer-Verlag, 1987.

[Lun89] T. F. Lunt. Access Control Policies for Database Systems. In C.E. Landwehr, Hrsg., *Database Security*, Band II, Seite 41–52. 1989.

[Lun91] T. F. Lunt. Security in Database Systems: A Researcher's View. In H. Lippold, P. Schmitz, und H. Kersten, Hrsg., *Sicherheit in Informationssystemen - Proceedings der 2. Deutschen Konferenz über Computersicherheit.* Vieweg-Verlag, 1991.

[Mat93] Florian Matthes. *Persistente Objektsysteme.* Springer-Verlag, 1993.

[MC84] J. K. Millen und C. M. Cerniglia. Computer Security Models. Technischer Bericht, The MITRE Corporation, Bedford, Massachusetts, September 1984.

[Mei93] Jens Meier. Grundlagen sicherer IT-Systeme und deren Umsetzung in Anwendungen mir relationalen Datenbanksystemen. Diplomarbeit am Fachbereich Informatik der Universität Hamburg, Mai 1993.

[Mey88] B. Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Mil84] R. Milner. A proposal for Standard ML. In *Proceedings of the ACM symposium on Lisp and Functional Programming*, 1984.

[MMS94] Florian Matthes, Sven Müßig, und Joachim W. Schmidt. Persistent Polymorphic Programming in Tycoon: An Introduction. Technischer Bericht FIDE/94/106, Fachbereich Informatik der Universität Hamburg, August 1994.

[MMS95a] Bernd Mathiske, Florian Matthes, und Joachim W. Schmidt. On Migrating Threads. Technischer Bericht, Universität Hamburg, Fachbereich Informatik, 1995.

[MMS95b] Bernd Mathiske, Florian Matthes, und Joachim W. Schmidt. Scaling Database Languages to Higher-Order Distributed Programming. Technischer Bericht, Universität Hamburg, Fachbereich Informatik, 1995.

[Mor91] Matthew Morgenstern. A Security Model for Multilevel Objects with Bidirectional Relationships. In C. E. Landwehr und S. Jajodia, Hrsg., *Database Security*, Band IV, Seite 53–72. 1991.

[MS92] Florian Matthes und Joachim W. Schmidt. The database programming language DBPL: Rationale and report. Technischer Bericht FIDE/92/46, Fachbereich Informatik der Universität Hamburg, Juli 1992.

[MS93a] Florian Matthes und Joachim W. Schmidt. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. Technischer Bericht, Fachbereich Informatik der Universität Hamburg, 1993.

[MS93b] Florian Matthes und Joachim W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In *Proceedings of the Euro-ARCH '93*, Seite 301–317. Springer Verlag, 1993.

[MS94a] Florian Matthes und Joachim W. Schmidt. The DBPL Project: Advances in Modular Database Programming. In *Information Systems*, Band 19, Seite 121–140, 1994.

[MS94b] Florian Matthes und Joachim W. Schmidt. Persistent Threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, Seite 403–414, Santiago de Chile, September 1994.

[MTH90] R. Milner, M. Tofte, und R. Harper. *The Definition of Standard ML*. MIT-Press, Cambridge, Massachusetts, 1990.

[Mül92] Kay Müller. Unterstützung von Gruppenkommunikation in offenen verteilten Systemen. Diplomarbeit am Arbeitsbereich Datenbanken und Informationssyteme der Universität Hamburg, September 1992.

[Mün94] Petra Münnix. Objektorientierte Datenmodellierung: Generierung statisch typisierter Repräsentationen. Diplomarbeit am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, Juli 1994.

[Nel91] G. Nelson, Hrsg. *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, 1991.

[Opp92] Rolf Oppliger. *Computersicherheit.* Vieweg-Verlag, 1992.

[OSF93] OSF. *OSF DCE Administration Guide - Core Components.* Prentice Hall Verlag, Englewood Cliffs, NewJersey, 1993.

[OW93] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, 1993.

[Par92] D. Parker. Seventeen information security myths debunked. In *Proceedings of the ISSA Security Conference*, 1992.

[Pfl89] Charles P. Pfleeger. *Security in Computing.* Prentice Hall-Verlag, 1989.

[Poh89] Hartmut Pohl. *Sicherheit der Informationstechnik.* Datakontext Verlag, 1989.

[PP92] A. Pfitzmann und B. Pfitzmann. Technical Aspects of Data Protection in Health Care Informatics. In J. Noothoven van Goor und J. P. Christensen, Hrsg., *Advances in Medical Informatics*, Seite 368–386. IOS Press, 1992.

[Pri90] Wolfgang Prinz. *Application of the X.500 Directory by Office Systems: Support for Representation, Authentication, Authorization and MHS.* Nummer 181

in GMD-Berichte. Gesellschall für Mathematik und Datenverarbeitung mbH, 1990.

[RBKW91] Fausto Rabitti, Elisa Bertino, Won Kim, und Darrell Woelk. A Model of Authorization for Next-Generation Database Systems. In *ACM: Transaction on Database Systems*, Seite 88–131, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, und W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International Editions, 1991.

[RMS95] Andreas Rudloff, Florian Matthes, und Joachim W. Schmidt. Security as an Add-On Quality in Persistent Object Systems. In J. Eder und L. A. Kalinichenko, Hrsg., *Second International East/West Database Workshop*, Workshops in Computing, Seite 90–108, Klagenfurt, Österreich, 1995. Springer Verlag.

[RSA78] R. Rivest, A. Shamir, und L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2), 1978.

[RWBM90] J. E. Roskos, S. Welke, J. Boone, und T. Mayfield. Integrity in the Department of Defense Computer Systems. Technischer Bericht Draft IDA Paper P-2316, Institute for Defense Analysis, Virginia, Juli 1990.

[Sch77] Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Toronto, Kanada, August 1977.

[Sen95] Ann Senn. *Open Systems for Better Business.* International Thomson Publishing Inc., 1995.

[Sie94] Bodo S. Sienkiewicz. *Computer-Sicherheit.* Addison-Wesley-Verlag, 1994.

[SL93] Hermann Strack und Kwok-Yan Lam. Context-Dependent Access Control in Distributed Systems. In E. G. Dougall, Hrsg., *Computer Security*, Seite 137–155, 1993.

[Smi89] D. Smith, Jeffrey. *Design and Analysis of Algorithms.* PWS-Kent Publishing Company, 1989.

[SNS88] J. G. Steiner, B. C. Neumann, und J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, Februar 1988.

[Sta95] William Stallings. *Sicherheit im Datennetz.* Prentice Hall-Verlag, 1995.

[Ste89] G. Steinke. Modelling Security Requirements in Information Systems. In M. Jarke, Hrsg., *Database Application Engineering with DAIDA*, Band I, Seite 201–220. 1989.

[Tal94] James A. Talvitie. An Object-Oriented Application Security Framework. In B. Thuraisingham, R. Sandhu, und T. C. Ting, Hrsg., *Security for Object-Oriented Systems*, Seite 55–75, 1994.

[Tha93] Georg Erwin Thaller. *Computersicherheit.* Nummer 18 in DuD Fachbeiträge. Vieweg-Verlag, Juni 1993.

[Thu91] Bhavani Thuraisingham. Multilevel Security for Multilevel Database Systems. In C. E. Landwehr und S. Jajodia, Hrsg., *Database Security*, Band IV, Seite 99–116. 1991.

[Tsa95] Thomas C. Tsai. *A Network of Objects.* International Thomson Publishing Inc., 1995.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, Hrsg., *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, Band 201, Seite 1–16, 1985.

[Wet94] Ingrid Wetzel. Programmieren in STYLE: Über die systematische Entwicklung von Programmierumgebungen. Dissertation am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg, 1994.

[Wir85] Niklaus Wirth. *Programmieren in Modula-2.* Springer-Verlag, 1985.

[Wir86] Niklaus Wirth. *Algorithmen und Datenstrukturen mit Modula-2.* Teubner Verlag, 1986.

[Wis89] Simon Wiseman. On The Problem of Security in Data Bases. Royal Signals & Radar Establishment Memorandum 4263, Ministry of Defence, Malvern, Worchestershire, Großbritannien, 1989.

[Wis90] Simon Wiseman. The Control of Integrity in Databases. In *Proceedings of IFIP WG11.3, Database Security Workshop*, Halifax, Yorkshire, England, September 1990.

[WL81] R. F Wilms und B. G. Lindsay. A Database Authorization Mechanism supporting individual and group authorization. Technischer Bericht, IBM Research Division, 1981.