JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Rechtesysteme von Betriebssystemen - Eine Bestandsaufnahme und deren Umsetzung im virtuellen Dateisystem der Telecoaching - Plattform WeLearn

Diplomarbeit zur Erlangung des akademischen Grades

Diplom - Ingenieur

in der Studienrichtung *Informatik*

Angefertigt am

Institut für *Informationsverarbeitung und Mikroprozessortechnik (FIM)*

Betreuung:

o. Univ.-Prof. Dr. Jörg R. Mühlbacher

Von:

Richard Leitner

Linz, April 2003

Danksagung

Zuallererst möchte ich meinen Eltern danken, dass sie in den Jahren meines Studiums nie das Vertrauen in mich verloren. Sie gewährten mir finanzielle und ideelle Unterstützung, wo immer sie nötig war, und stellten dafür ihre eigenen Bedürfnisse in den Hintergrund, um mir eine akademische Ausbildung zu ermöglichen.

Auch meiner Lebensgefährtin Barbara gebührt ein großer Dank. Sie hat an mich geglaubt und mich in manch schwerer Stunde wieder aufgebaut. Sie war auch stets bemüht, die Probleme und Verpflichtungen des Alltags möglichst von mir fernzuhalten, damit ich mich auf die Beendigung meines Studiums konzentrieren konnte.

Den Kolleginnen und Kollegen des Instituts für Informationsverarbeitung und Mikroprozessortechnik danke ich für ihre Unterstützung und für das hervorragende Arbeitsklima, das einer erfolgreichen Mitarbeit sehr förderlich war.

Besonderer Dank gilt meinem Betreuer o.Univ.-Prof. Dr. Jörg R. Mühlbacher, der mir die Mitarbeit an der Entwicklung der Telecoaching-Plattform WeLearn und dadurch die Erstellung dieser Arbeit ermöglichte. Er hatte auch immer ein offenes Ohr für Fragen und nahm sich stets die Zeit, Probleme und Anregungen zu diskutieren.

Kurzfassung

Die vorliegende Diplomarbeit beschreibt verschiedene Implementierungsmöglichkeiten von Rechtesystemen in gängigen Betriebssystemen und vergleicht sie mit der Implementierung in der Distanceteaching / Distancecoaching / Distancelearning-Plattform WeLearn Release 2. Dabei werden die Besonderheiten der einzelnen Implementierungen hervorgehoben und die Vor- und Nachteile gegenübergestellt.

Ein gut durchdachtes Rechtesystem ist sowohl für Betriebssysteme, als auch für andere Systeme mit integrierter Benutzerverwaltung von enormer Bedeutung, um den Zugriff auf die Ressourcen zu steuern. Dass dies keine triviale Angelegenheit ist, versucht diese Diplomarbeit darzustellen. Sie bietet Einblick in die Überlegungen, die bei der Implementierung des Rechtesystems von WeLearn Release 2 angestellt wurden, um ein möglichst ausgereiftes Konzept zu erstellen.

Eine detaillierte Beschreibung der Berechtigungen und der Verwaltung derselben liefert gemeinsam mit der Darstellung der technischen Umsetzung einen Überblick über die Entwicklung dieses wichtigen Teiles von WeLearn Release 2.

Abstract

This diploma thesis describes how the rights-systems of current operating systems are implemented and compares them with the implementation of the distanceteaching / distancecoaching / distancelearning platform WeLearn Release 2. Therefore it shows the special features of each implementation and opposes advantages and disadvantages.

To control access to the resources has immense importance for both operating systems and systems with integrated user administration. This diploma thesis provides an overview on concepts which have been used for implementing the rights-system of WeLearn Release 2 in order to get a well engineered result.

The detailed description of the permissions and the administration of them shows in addition to the technical background a survey of the development of this important part of WeLearn Release 2.

Inhaltsverzeichnis

1 Einleitung und Motivation

In der heutigen Zeit ist die computerbasierte Informationsverarbeitung allgegenwärtig (Stichwort „Ubiquitous Computing“). Ein Arbeiten ohne die Unterstützung durch Computer ist sowohl in der Wirtschaft als auch in der öffentlichen Verwaltung, und ebenso in der Wissenschaft, nicht mehr vorstellbar. Durch diese weite Verbreitung und auch durch die weltweite Vernetzung der Computersysteme kommen jedoch auf den Anwender im allgemeinen, auf den Systemadministrator im speziellen, neue Anforderungen zu, die es früh zu erkennen und auch zu beachten gilt. Die Daten, die jeder einzelne auf seinem Rechner und im Netzwerk ablegt, würden ausgedruckt wahrscheinlich schon unzählige Bände füllen. Diese Daten sollen aus verschiedensten Gründen nicht für jedermann zugänglich sein. Die persönlichen Daten bedürfen ohne Zweifel eines besonderen Schutzes. Auch Unternehmen haben viele Dokumente, die ihnen den Wettbewerbsvorteil gegenüber den Mitbewerbern garantieren und daher nicht in deren Hände gelangen dürfen. Neue Entwicklungen müssen geheim gehalten werden, da diese oft durch enorme finanzielle Anstrengungen ermöglicht wurden.

Leider gibt es auch Menschen, die nicht nur das Erlangen von fremder Information anstreben, sondern die zusätzlich deren Zerstörung bzw. Veränderung als Ziel vor Augen haben.

Diese unautorisierten Zugriffe auf Dateien erfolgen jedoch nicht nur über das Netzwerk (Internet und Intranet), sondern können auch auf dem eigenen Rechner durchgeführt werden. Zugriffe können nun durch die Vergabe verschiedener Zugriffsberechtigungen gesteuert werden. Hier wird unterschieden zwischen allgemeinen Berechtigungen, die der Systemadministrator vergibt, und solchen, die jeder einzelne Benutzer auf seine Dateien vergibt. Mit diesen versucht der Benutzer, den Zugriff von Personen, die ebenfalls auf demselben System arbeiten, einzuschränken. Der Systemadministrator hat natürlich das Ziel, eine möglichst stabile Umgebung auf dem Rechner zur Verfügung zu stellen, um den Wartungsaufwand für sich gering zu halten und den Benutzer nicht von seinen üblichen Tätigkeiten abzuhalten. Damit dies gewährleistet wird, ist es erforderlich, gewisse Bereiche des Systems vor unautorisiertem Zugriff zu schützen. Von besonderer Bedeutung sind die Dateien des Betriebssystems (Systemdateien). Führt der Benutzer in für die Stabilität des Systems unverzichtbaren Dateien Änderungen ohne jegliche

Kenntnis des Hintergrundes durch, kann dies zur Folge haben, dass der Rechner nicht mehr startet und eventuell der Verlust von Daten nicht mehr vermieden werden kann.
Ähnlich verhält es sich mit den Dateien eines Benutzers. Dieser möchte zum Beispiel manche Dateien vor anderen Benutzern verbergen. Es kann aber auch wünschenswert sein, nur die Art des Zugriffes einzuschränken. Beispielsweise brauchen Lektoren eines in Bearbeitung befindlichen Buches nur einen Lesezugriff; Schreiben dürfen nur die Autoren.

Diese Arbeit befasst sich mit der Analyse der Zugriffsberechtigungen auf Dateien von ausgewählten Betriebssystemen und der Umsetzung solcher Zugriffsberechtigungen in der Telecoaching-Plattform „WeLearn Release 2“. Dabei wird auch auf die speziellen Anforderungen eingegangen, die eine solche Telecoaching-Plattform an das System der Zugriffskontrolle stellt. Hier sind die verschiedenen Sichtweisen auf die Lehrunterlagen und Systemkomponenten von besonderer Bedeutung (Student/Schüler – Lehrer/Coach – Administrator).

Übersicht

Nach der kurzen Einführung in die Thematik wird in Kapitel 2 näher auf die allgemeinen Anforderungen an ein Rechtesystem in Betriebssystemen eingegangen. Es werden verschiedene Theorien zu diesem Thema exemplarisch dargestellt und Möglichkeiten der Implementierung aufgezeigt.
In Kapitel 3 werden konkret einige ausgewählte Betriebssysteme beleuchtet und deren Vorgehensweise in Bezug auf die Zugriffsberechtigungen auf Dateien dargestellt. Dabei wird sich ein großer Teil Microsoft Windows XP widmen, da dies die umfangreichsten Funktionen im Bereich der Zugriffseinschränkung und -überwachung zur Verfügung stellt. Auch Unix/Linux und Apple´s Mac OS X werden untersucht.
Im Anschluss daran wird die Telecoaching-Plattform WeLearn (Release 2) vorgestellt, die am Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM) der Johannes Kepler Universität Linz entwickelt wurde. Dabei wird speziell auf die Implementierung des virtuellen Dateisystems eingegangen, wobei die Verwaltung der Zugriffsberechtigungen den Schwerpunkt bildet.

Ein Vergleich des Rechtesystems von WeLearn Release 2 mit dem von WeLearn Release 1 einerseits, mit denen der besprochenen Betriebssysteme andererseits, rundet die Arbeit ab.
Im letzten Abschnitt werden einige Programmauszüge (JAVA) vorgestellt und erläutert, um die konkrete Umsetzung im Rahmen des Projektes zu verdeutlichen.

2 Rechtesystem - Eine Beschreibung

2.1. Allgemeines

Die zentrale Aufgabe eines Rechtesystems betreffend das Filesystem in einem Betriebssystem ist die Zugriffskontrolle. Bei Systemen, die den Zugriff auf einzelne Dateien anderer Benutzer nicht verbieten können, gibt es zwei Möglichkeiten der Dateizugriffskontrolle:
Entweder den Zugriff auf das ganze System gewähren, oder das ganze System verschließen. Da dies aber aufgrund der Benutzerfreundlichkeit und Handhabung nicht wünschenswert ist, hat der Zugriff etwas differenzierter zu erfolgen. Somit ist ein kontrollierter Zugriff erforderlich.

2.2. Benutzeridentifikation als Fundamentalaufgabe

Um die Sicherheit der Ressourcen gewährleisten zu können und den Missbrauch von Rechnerkapazität und Information zu verhindern, ist es erforderlich, nur legitimierten Benutzern den Zugang zu Betriebsmitteln (Computer, Dateien, Programme, Transaktionen) zu erlauben. Das Ziel einer Zugriffkontrolle ist es daher, den Benutzer in seinen Anwendungen (Lesen, Schreiben, Löschen und Umbenennen von Dateien) einzuschränken. Somit hat der Benutzer vor einem Zugriff auf Ressourcen seine Berechtigung gegenüber dem System nachzuweisen. Diese Identifikation erfolgt im Allgemeinen durch das Betriebssystem. Dieser Prozess wird in folgende zwei Schritte unterteilt [Kra89]:

1. Identifikation: Durch die Identifikation wird zwischen dem Benutzer und dem System ein eindeutiger Bezug hergestellt, indem der Benutzer den Anmeldenamen in eine dafür vorgesehene Anmeldemaske eingibt. Die Identifikation ist die Behauptung der Identität [Che00], gibt aber keinen Hinweis, ob diese Behauptung auch richtig ist.
2. Echtheitsprüfung/Authentifizierung: Dadurch wird überprüft, ob der Benutzer tatsächlich der ist, für den er sich ausgibt. Die Authentifizierung ist der Nachweis der Identität [Che00]. Diese Erkennung kann auf eine der folgenden Möglichkeiten basieren, wobei erst eine Kombination aus mehreren dieser Möglichkeiten eine akzeptable Authentifizierungsmaßnahme darstellt:
    - Wissen des Benutzers (Passwort)

- Besitz des Benutzers (Magnetkarte, Schlüssel)
- Eigenschaft des Benutzers (Iris-Scan, Fingerabdruck, Gesichtsgeometrie)
- Fähigkeit des Benutzers (Spracherkennung)
- Kombination aus obigen Möglichkeiten

In der Praxis werden beide Schritte gemeinsam durchgeführt, wobei nur diejenige Möglichkeit auf Betriebssystemebene unterstützt werden kann, die auf dem Wissen des Benutzers basiert. Diese wird als „Passwortverfahren“ bezeichnet.

Abbildung 1 - Anmeldedialog eines MS Windows 2000 Servers

Da bei diesem Verfahren das Passwort eingegeben werden muss, bedarf es bei der Eingabe und der Übertragung zur Überprüfung eines besonderen Schutzes. Einer unbefugten Person darf es nicht möglich sein, das Passwort bei der Eingabe zu lesen. Die eingegebenen Buchstaben sind daher durch Symbole unkenntlich zu machen.
Um sich gegen das Abhören des Passwortes bei der Übertragung zu sichern, wird es verschlüsselt übertragen.
An dieser Stelle sollte auch noch einmal auf die sorgfältige Auswahl des Passwortes hingewiesen werden. Dieses muss sich durch eine ausreichende Länge, die Verwendung von Zahlen und Sonderzeichen und durch eine regelmäßige Änderung auszeichnen, um nur einige Kriterien zu nennen. [Hoe02]

2.3. Aufgaben

Das POSIX-6-Komitee hat folgende gezielte Maßnahmen vorgeschlagen, um Fehlfunktionen von Programmen sowie bewusste Angriffe von Benutzern im Bereich der Zugriffsrechte auf Dateien möglichst schwierig zu machen [Bra98]:

- *„das Prinzip, für die Erfüllung einer Aufgabe nur die geringstmöglichen Rechte einzuräumen (least privilege);*
- *die Zugriffskontrolle durch diskrete Angaben zu ergänzen (discretionary access control); Also z. B. durch Zugangs- oder Kontrolllisten (Access Control Lists ACL), in denen alle Leute enthalten sind, die Zugriff auf diese Datei haben, sowie ihre genau spezifizierten Rechte. Dies sind zusätzliche Spezifikationen zu den einfachen Zugriffsrechten.*
- *eine verbindliche Zugangskontrolle, unabhängig vom Erzeuger (mandatory access control); Der Zugang zu einem Objekt erfolgt nur von Prozessen mit umfangreicheren Berechtigungen.*
- *die Aufzeichnungen von Zuständen eines Objekts (audit trail), um bei missbräuchlicher Verwendung die Ursachen und Personen herausfinden zu können.“*

Diese Maßnahmen werden jedoch von gängigen Betriebssystemen nur eingeschränkt implementiert.

2.4. Funktionalität

Wie diese Zugriffsbeschränkungen oder -berechtigungen aussehen, ergibt sich aus den Möglichkeiten, wie ein Zugriff stattfinden kann. Je nach Art des Zugriffes wird der Zugang gewährt oder verboten. Es sind hier verschiedene Arten des Zugriffs zu unterscheiden, z. B.:

- Read: Lesen von der Datei
- Write: Schreiben in die Datei
- Execute: Ausführen der Datei
- Append: Anhängen neuer Informationen an das Ende der Datei
- Delete: Löschen der Datei
- List: Auflisten des Namens und der Dateiattribute

Einige dieser Zugriffsoperationen haben je nach dem, ob sie auf Verzeichnisse oder auf Dateien ausgeführt werden, unterschiedliche Funktionalitäten. Manche sind auf Verzeichnissen nicht definiert (z.B. Append), andere wiederum auf Dateien nicht (z.B. List).

Natürlich ist dies nur ein kleiner Ausschnitt aus möglichen Zugriffsoperationen. Er stellt jedoch die grundlegenden Operationen dar, auf denen die übrigen, komplexeren Operationen aufbauen. So kann eine Copy-Operation auf die Leseoperation zurückgeführt werden, denn wenn man eine Datei lesen darf, wird man sie im Normalfall auch kopieren oder drucken dürfen. Der Schutz der Dateien ist also nur auf der untersten Schicht der Operationen definiert, da die höheren Operationen auf die unteren aufbauen. Einzelne Softwarepakete bieten zusätzliche Operationen auf ihnen zugehörige Dokumente an. Es können spezifische Berechtigungen auf diese Dateien vergeben werden, die die Berechtigungen des Dateisystems noch zusätzlich einschränken. Ein Beispiel für diese erweiterten Berechtigungen ist Adobe Acrobat. PDF-Dateien können mit Einschränkungen erstellt werden, damit der Leser die Datei zum Beispiel nicht drucken kann. Auf diese zusätzlichen Berechtigungen hat das Rechtesystem des Filesystems keinen Einfluss. Dateien, die durch die Berechtigungen des Filesystems vom Benutzer gelesen werden können, können durch die Einschränkung in Adobe Acrobat möglicherweise trotzdem nicht gedruckt werden.

2.5. Arten der Zugriffskontrolle

Die am häufigsten verwendete Art der Zugriffskontrolle ist die, den Zugriff von der Identität des Benutzers abhängig zu machen. Da verschiedene Benutzer auch unterschiedliche Zugriffsmöglichkeiten auf Dateien benötigen, scheint diese Vorgehensweise auch sinnvoll zu sein.

2.5.1. Access Control Lists

Zumeist wird die personenbezogene Zugriffskontrolle durch „Access Control Lists (ACL)“ implementiert. Das heißt: jeder Ressource wird eine Liste zugeordnet, in der die Benutzer und deren jeweiligen Zugriffsrechte auf diese Ressource eingetragen sind. Ein solcher Eintrag in diese Liste, der den Benutzer und dessen Zugriffsrecht enthält, wird als „Access Control Entry (ACE)“ bezeichnet.

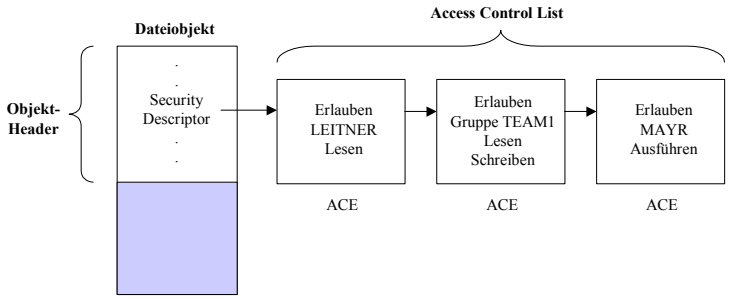


Abbildung 2 - Access Control List

Der Ablauf beim Zugriff auf eine Datei ist wie folgt:
Ein Benutzer versucht, auf eine Datei zuzugreifen. Das Betriebssystem überprüft nun anhand der ACL dieser Datei, ob der Benutzer die Berechtigung besitzt, die gewünschte Operation mit der Datei auszuführen. Ist dies nicht der Fall, wird dem Benutzer sein Vorhaben verweigert.

2.5.2. Zugriffsmodus

Eine andere Art der Zugriffskontrolle ist folgende: die Zugriffsberechtigten werden in drei Kategorien eingeteilt:
Eigentümer (owner), **Gruppenmitglieder** (user group member) und **alle anderen** (others). Diese drei Kategorien erhalten nun jeweils eine Kombination aus den drei zur Verfügung stehenden Zugriffsvariablen (Flags): Lesen (R), Schreiben (W) und Ausführen (X). Je nach Erteilung der Rechte kann nun der Benutzer auf die entsprechende Datei zugreifen.
Diese Vorgehensweise wird von Unix-/Linux-Betriebssystemen verwendet. Bei jedem Zugriff eines Unix-Prozesses auf eine Datei werden die Zugriffsberechtigungen überprüft. Für ausführbare Dateien gibt es eine Besonderheit: das sogenannte s-Bit. Hierbei handelt es sich um ein Verfahren, mit dem die Rechte des Aufrufers eines Programms während der Ausführung erweitert werden können. Diese s-Bits können für den Eigentümer und für die Gruppe gesetzt werden. Normalerweise übernimmt der mit der ausführbaren Datei gestartete Prozess die Zugriffsberechtigungen des aufrufenden Prozesses. Durch ein gesetztes s-Bit wird der Prozess unter der Benutzer- und/oder Gruppen-

nummer des Eigentümers bzw. der Gruppe des Objektes ausgeführt, und nicht unter der des aufrufenden Prozesses.

Ein Beispiel für die Verwendung des s-Bits sei anhand der Passwortdatei */etc/passwd* erläutert. Schreibrechte auf diese Datei besitzt nur der Systemverwalter. Es muss jedoch jedem nicht privilegierten Benutzer möglich sein, sein Passwort, das in dieser Datei hinterlegt ist, zu ändern. Dies geschieht mit dem Programm *passwd*. Eigentümer dieses Programms ist der Systemverwalter. Es ist mit dem s-Bit für Besitzer und Gruppe versehen. Somit wird bei der Ausführung des Programms die Benutzer- bzw. Gruppennummer des Systemverwalters eingesetzt und daher schreibend auf die Datei */etc/passwd* zugegriffen. Die Überprüfung, dass dort nur das eigene Passwort verändert wird, übernimmt das Programm selbst.

Dieser Mechanismus birgt eine Vielzahl von Gefahren. Daher sollte auf die Verwendung verzichtet werden, wenn dies möglich ist.

3 Analyse von Rechtesystemen in ausgewählten Betriebssystemen

3.1. Microsoft Windows XP

Da Microsoft Windows XP mehrere Dateisystemtypen unterstützt, werden in den nächsten Kapiteln die wichtigsten behandelt und speziell auf deren jeweilige Umsetzung des Rechtesystems eingegangen.

Grundsätzlich ist das Dateisystem eines Datenträgers festgelegt. Je nach Wahl des Dateisystems sind bestimmte Möglichkeiten gegeben, die Dateien strukturiert abzulegen oder auch den Zugriff auf sie einzuschränken. Auch gibt es speziell bei Microsoft Windows Unterschiede, wie die jeweiligen Versionen mit den Dateisystemen zurechtkommen. Man muss sich daher sehr genau überlegen, für welches Dateisystem man sich bei der Installation von Microsoft Windows XP entscheidet. Ausschlaggebend werden die jeweiligen persönlichen Bedürfnisse des Anwenders bzw. des Administrators sein.

Abbildung 3 - Merkmale der Dateisysteme NTFS und FAT/FAT32

3.1.1. Das FAT/FAT32-Dateisystem

3.1.1.1.Grundlagen

FAT (File Allocation Table) ist das einfachste Dateisystem von Windows XP. Wie der Name schon sagt, wird in einer Tabelle abgelegt, in welchen Sektoren der Festplatte sich welcher Teil einer jeden Datei befindet.

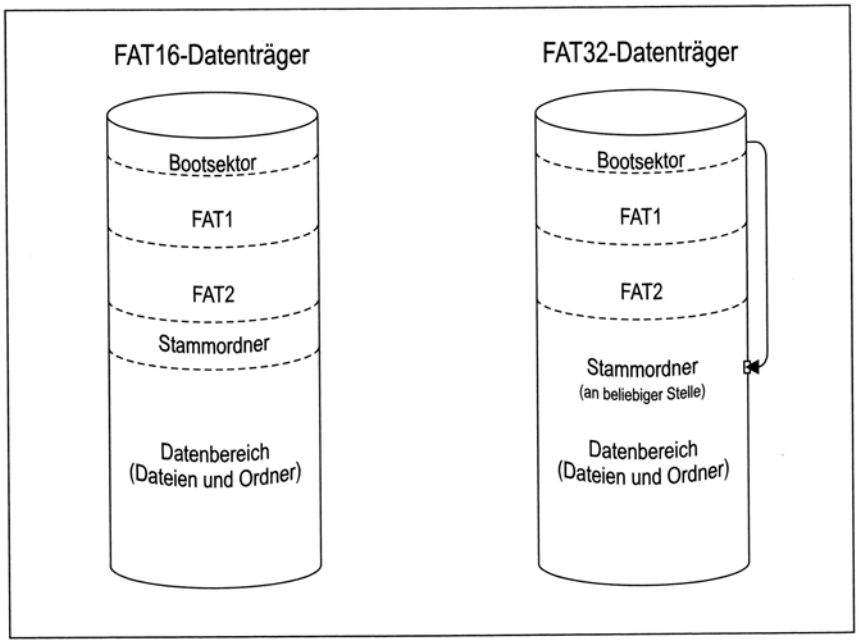


Abbildung 4 - Vergleich FAT16/FAT32-Datenträger [Mic02]

Dieses Dateisystem sollte ursprünglich nur kleine Festplatten mit relativ einfachen Ordnerstrukturen verwalten. Es wurde jedoch an die Bedürfnisse der Betriebssysteme angepasst, wodurch sich die Benennung der neueren Versionen ergab. Die Nummern - es gibt FAT12, FAT16 und FAT32 - beziehen sich auf die Länge der Adressen, mit denen auf die Dateien verwiesen wird. Aus dieser Länge ergibt sich auch die maximale Anzahl der Cluster, die in einer FAT adressierbar sind. Bei systembedingter maximaler Clustergröße ergibt sich damit auch die maximal nutzbare Kapazität einer Platte.

Eine Festplatte mit FAT-Dateisystem kann auch von anderen Betriebssystemen verwendet werden; es ist sogar ein Dual-Boot-System möglich, also die Installation zweier Betriebssysteme auf einer Festplatte. Beim Systemstart kann zwischen den beiden Betriebssystemen gewählt werden, welches schließlich verwendet werden soll.

Wenn die Größe einer Partition einer Festplatte 2 Gigabyte überschreitet, so kann nur FAT32 damit umgehen. Die älteren Versionen von FAT können aufgrund der eingeschränkten Adressierung keine Partitionen jenseits dieser Grenze verwalten.

3.1.1.2.Zugriffsberechtigungen auf Dateien und Ordner

In FAT-Dateisystemen ist es nicht möglich, direkt auf eine Datei oder ein Verzeichnis Zugriffsberechtigungen zu vergeben. Es sind einige wenige Dateiattribute vorhanden, die nur sehr geringe und leicht zu übergehende Beschränkungen erlauben. Diese Dateiattribute sind:

- Schreibgeschützt: Dieses Attribut verhindert das Überschreiben bzw. Verändern einer Datei.
- Versteckt: Ist dieses Attribut gesetzt, so wird die Datei oder das Verzeichnis bei einer Auflistung der Dateien nicht angezeigt. Dies gilt sowohl für die Anzeige in der MS-DOS-Eingabeaufforderung durch den Befehl „dir“, als auch im Windows Explorer.
- System: Durch dieses Attribut wird die Datei sowohl versteckt, als auch schreibgeschützt. Es wird als besonderer Schutz für Betriebssystemdateien angesehen und vorwiegend dafür verwendet.
- Archiv: Durch dieses Attribut wird der Zugriff auf diese Datei gar nicht eingeschränkt. Es wird häufig von Datensicherungsprogrammen verwendet, die damit diese Datei als bereits gesichert kennzeichnen und bei einem erneuten Sicherungsvorgang nicht mehr berücksichtigen. Wird die Datei jedoch verändert, so wird auch dieses Attribut wieder gelöscht, um die Datei bei einem neuerlichen Sicherungsvorgang wieder zu archivieren.

Es ist allen Benutzern möglich, diese Attribute zu ändern oder zu löschen. Sie stellen also keinen wirksamen Schutz vor unbefugtem Zugriff dar. Rechte auf Benutzerebene werden von FAT nicht unterstützt, was in Bezug auf die Datensicherheit ein großes Manko darstellt.

3.1.1.3.Weitere Eigenschaften

Die Gewährleistung der Datenintegrität ist ein weiterer Punkt, bei dem das FAT-Dateisystem Mängel aufweist. So sollte ein Dateisystem mit Störungen geregelt umge-

hen können und nach einem Systemausfall die Daten wiederherstellen können. In [Bün02] werden folgende Punkte aufgelistet, die das FAT-Dateisystem sehr anfällig gegenüber Datenfehlern machen:

- „*Der Bootsektor wird nicht gesichert, sodass bei seiner Beschädigung kein Startvorgang mehr möglich ist.*
- *Wird die Dateizuordnungstabelle beschädigt, wird nicht automatisch die Sicherheitskopie benutzt. Diese kann erst ein externes Reparaturprogramm wie beispielsweise Chkdsk aktivieren (in FAT32 behoben).*
- *Eine integrierte Dateisystemsicherheit wie bei NTFS steht unter FAT nicht zur Verfügung. Eine unterbrochene Schreibaktion, die beispielsweise eine alte Datei durch eine neue ersetzen soll, hat so den Verlust der alten* ***und*** *der neuen Datei zur Folge*.“

Demgegenüber kann man natürlich auf einen mit FAT formatierten Datenträger ohne weiteres mit einer MS-DOS-Bootdiskette zugreifen, was bei NTFS nicht mehr möglich ist. Auf diesen FAT-Datenträger können in weiterer Folge im Fehlerfall auch die verschiedensten Reparaturwerkzeuge angewendet werden.

Der Umstand, dass bei FAT kaum Einfluss auf die Zugriffsberechtigungen auf Dateien besteht, ist im Zusammenhang mit Rechten einer der größten und wichtigsten Unterschiede zum NTFS, welches im folgenden Abschnitt erklärt wird.

3.1.2. Das NT File System (NTFS)

Das NTFS (New Technology File System) wurde das erste Mal mit Microsoft Windows NT 3.1 ausgeliefert [Mic98]. In Windows XP kommt nun schon Release 5 zur Anwendung (NTFSv5). In dieser Version sind auch schon eigene Funktionen für Kompression und Verschlüsselung direkt im Dateisystem integriert. Außer Microsoft Windows 2000 und Microsoft Windows XP können keine anderen Betriebssysteme auf mit NTFSv5 formatierte Datenträger zugreifen (Microsoft Windows NT 4.0 mit Service Pack 4 auch nur eingeschränkt).

3.1.2.1.Grundlagen

Die folgende Abbildung stellt die Grundstruktur eines NTFS-Datenträgers dar. Bis auf den Bootsektor können alle Elemente variabel auf dem Datenträger angeordnet werden. Der Bootsektor enthält einen Verweis auf den *Master File Table* (MFT), der die wichtigste Organisationsstruktur eines NTFS-Datenträgers darstellt.

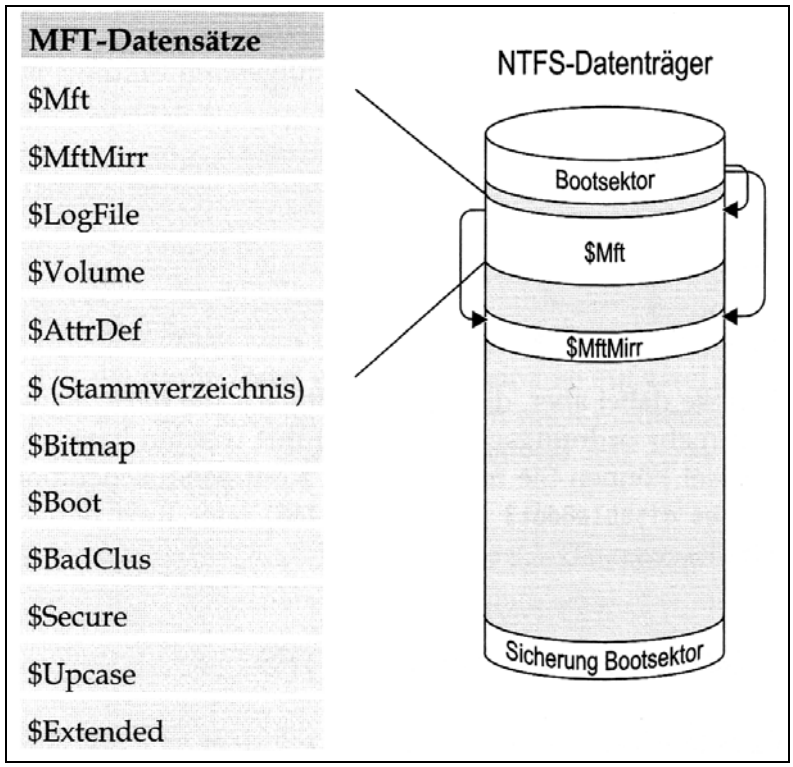


Abbildung 5 - Layout eines NTFS-Datenträgers

Wie in obiger Abbildung zu sehen ist, existieren sowohl vom Bootsektor, als auch vom MFT Sicherungskopien, die bei einem festgestellten Fehler der Originaldatei verwendet werden. Im Fall des MFT enthält diese Sicherungskopie jedoch nur die wichtigsten Elemente.

Jeder Datensatz eines MFT hat eine Größe von 2 KB. Für jede Datei und für jedes Verzeichnis wird ein solcher Datensatz erstellt. Handelt es sich um eine sehr kleine Datei (<1500 Bytes), so wird diese gleich im Datensatz selbst gespeichert, was zu einem sehr schnellen Zugriff führt und Speicherplatz spart. Ist die Datei größer, so werden mehrere Datensätze belegt und der erste speichert die Zeiger auf die Restlichen.

Verzeichnisse werden intern als ganz normale Dateien gehandhabt. Kleine Verzeichnisse mit wenigen Einträgen werden wiederum in einem Datensatz abgelegt. Dabei werden nur die Zeiger auf den jeweiligen Eintrag der Datei in der MFT gespeichert. Bei größeren Verzeichnissen, die nicht mehr komplett in einen MFT-Datensatz passen, wird eine B-Baumstruktur aufgebaut, deren Knoten wiederum die Verweise auf die Einträge der Dateien in der MFT enthalten.

3.1.2.2.Zugriffsberechtigungen auf Dateien und Ordner

Im NTFS ist es tatsächlich möglich, Zugriffsberechtigungen auf Dateien und Ordner zu vergeben. Diese sind sowohl lokal, als auch bei Zugriffen über das Netzwerk gültig. Enthalten sind die einzelnen Berechtigungen im Dateiattribut *Sicherheitsbeschreibung*.

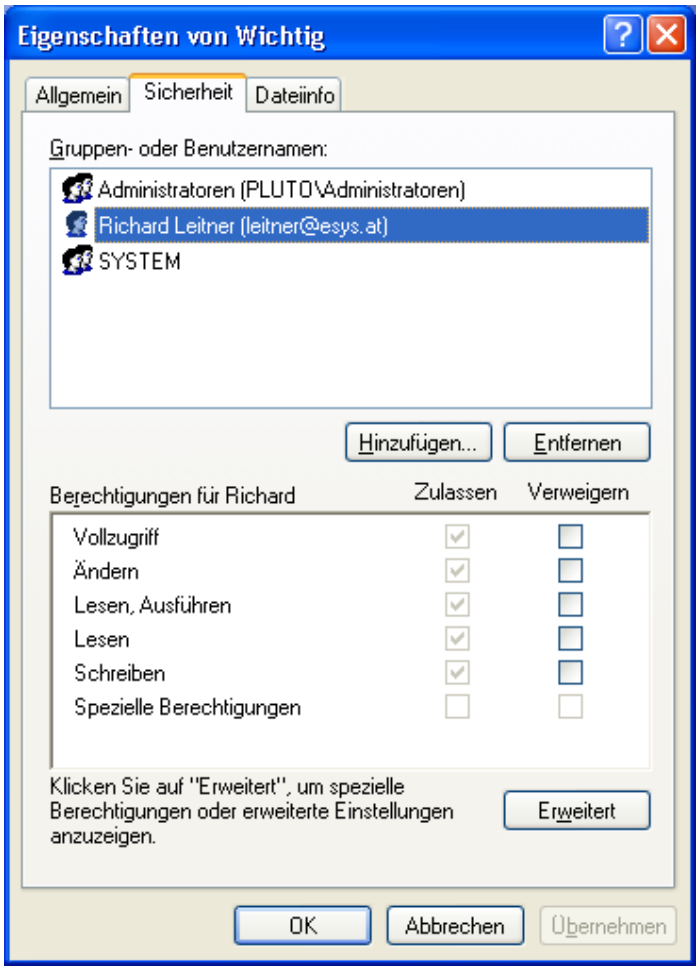


Abbildung 6 - Dialog zum Einstellen der Zugriffsberechtigungen

Um diese Berechtigungseinträge des NTFS-Attributs *Sicherheitsbeschreibung* standardmäßig sehen bzw. ändern zu können, ist es notwendig, in eine Active Directory-Domäne eingebunden zu sein. Nicht in eine Active Directory-Domäne eingebundene oder nur in ein Arbeitsgruppen-Netzwerk eingebundene Rechner können nur auf eine vereinfachte Rechtvergabe zugreifen. In diesem Fall ist die Registerkarte Sicherheit ausgeblendet.

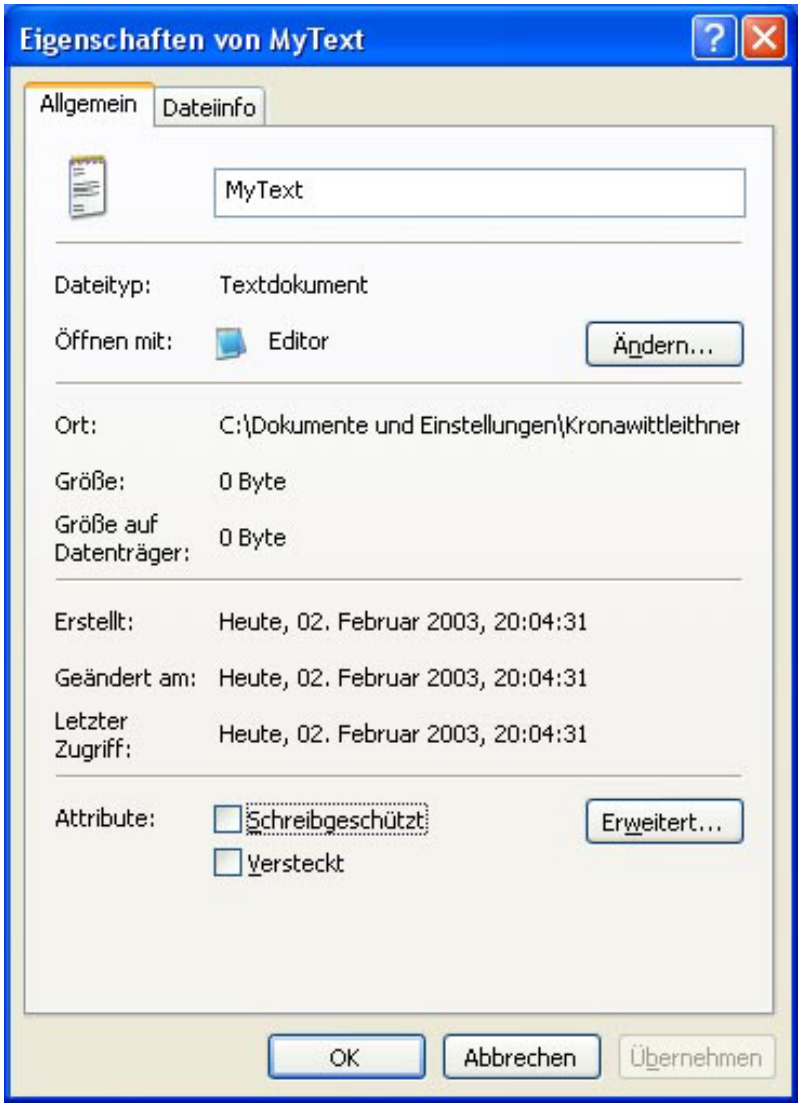


Abbildung 7 - Standarddialog für Dateieigenschaften (einfache Rechtvergabe)

Ein Rechner, der nicht in eine Active Directory-Domäne eingebunden ist, kann dennoch in die erweiterte Zugriffsverwaltung umschalten. Dazu steht im Windows Explorer unter dem Menüpunkt *Extras* das Dialogfenster *Ordneroptionen* zur Verfügung. Auf der Registerkarte *Ansicht* ist einfach die Option *Einfache Dateifreigabe verwenden* zu deaktivieren. Durch diese Veränderung sind sofort für alle Dateien und Ordner die erweiterten NTFS-Zugriffsrechte aktiv.

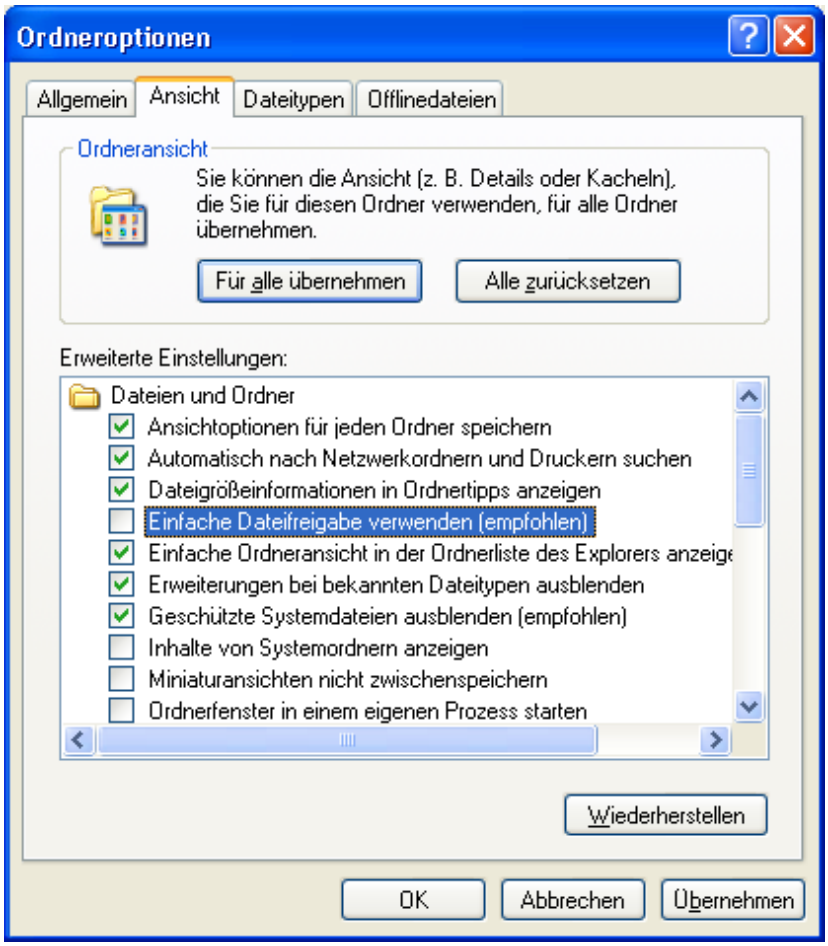


Abbildung 8 - Aktivieren der erweiterten NTFS-Zugriffsrechte

Im Anschluss wird nun die generelle Logik der NTFS-Zugriffsrechte näher erläutert. Wir setzen voraus, dass die erweiterte Sicht der Zugriffsrechte aktiviert ist. Grundsätzlich ist noch zu erwähnen, dass Änderungen, die in der erweiterten Sicht durchgeführt werden, auch dann noch wirksam sind, wenn wieder in den vereinfachten Modus zurückgeschaltet wird. Hier ist eine Fehlerquelle vorhanden, die in der Praxis zu kaum nachvollziehbaren Auswirkungen und zu Überraschungen führen kann.

Es gibt zwei Grundprinzipien, auf die die NTFS-Zugriffsrechte aufgebaut sind:

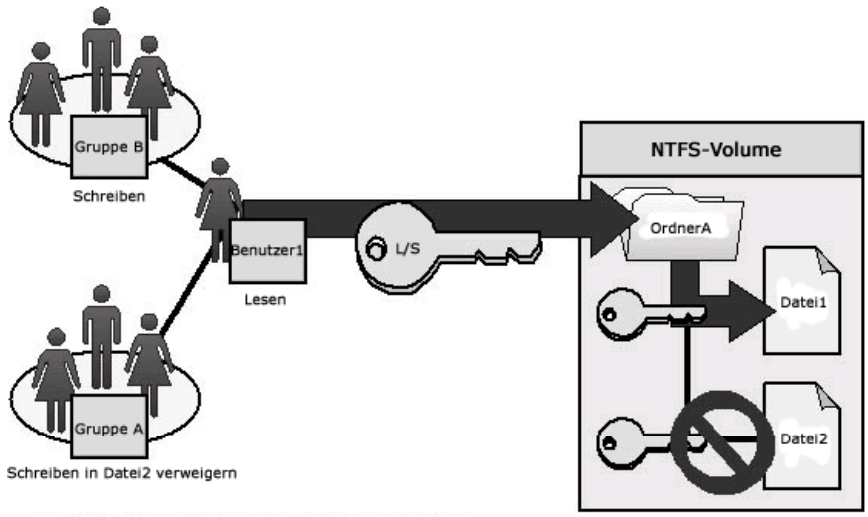
- Verweigern hat Vorrang vor Zulassen
- Vererbung von Rechten

Verweigern hat Vorrang vor Zulassen

Bei der Auswertung der NTFS-Berechtigungen folgt Windows XP folgendem Prinzip: Je nach erteilten Berechtigungen wird einem Benutzer oder einem Gruppenmitglied vol-

ler oder eingeschränkter Zugriff auf eine Datei oder ein Verzeichnis gewährt oder auch verweigert. Dabei wird in zwei Schritten vorgegangen:
Zuerst werden die negativen Berechtigungen (Verweigerungen) ausgewertet, da diese mehr Gewicht haben als die positiven Berechtigungen (Genehmigungen). Dies ist auch der Grund, warum man mit expliziten Verweigerungen eher sparsam umgehen sollte. Wird einer Gruppe beispielsweise das Leserecht auf einen Ordner verwehrt, kann keinem einzelnen Mitglied der Gruppe nachträglich ein Leserecht auf diesen Ordner gewährt werden.
Im zweiten Schritt werden die positiven Berechtigungen ausgewertet. Wird dem Benutzer nach Auswertung der negativen Berechtigungen der Zugriff noch nicht verwehrt, werden in weiterer Folge die Zugriffsgenehmigungen auf die Datei oder das Verzeichnis ausgewertet.
Ist der Benutzer oder die Gruppe, in der der Benutzer Mitglied ist, nicht in der Zugriffsliste vorhanden, so hat er generell keinen Zugriff auf dieses Objekt. Das heißt, dass kein Benutzer oder Gruppenmitglied dieses Rechners oder in der gesamten Domäne Zugriff auf die Datei oder den Ordner hat, der nicht explizit in der Liste angeführt ist.

Natürlich ist es auch möglich, dass ein Benutzer Mitglied mehrerer Gruppen ist. In diesem Fall kann es auch vorkommen, dass sich die Zugriffsberechtigungen der Gruppen widersprechen. Die Berechtigungen werden kumuliert, wobei eine Verweigerung schwerer wiegt als eine Genehmigung und somit bevorzugt wird.



•NTFS-Berechtigungen sind kumulativ
•Dateiberechtigungen überschreiben Ordnerberechtigungen
•Eine Zugriffsverweigerung setzt anderweitig erteilte Berechtigungen außer Kraft

Abbildung 9 - Mehrere NTFS-Berechtigungen

Vererbung von Rechten

Mit Windows 2000 geht Microsoft einen neuen Weg in der Vererbung von Rechten. In früheren Versionen von Windows werden die Berechtigungen bei der Erzeugung eines Objektes vom übergeordneten Objekt kopiert. Die Vererbung der Berechtigungen findet bei der Erzeugung statt. Das bedeutet, dass nachträglich Änderungen der Berechtigungen am übergeordneten Objekt keine Auswirkungen auf darunter liegende Objekte haben (*„create time inheritance“*). Es besteht jedoch die Möglichkeit, die Änderungen an einem Objekt auch auf alle in der Hierarchie darunter liegenden Objekte anwenden zu lassen, indem man eine entsprechende Option bei der Änderung der Berechtigungen aktiviert.

Windows 2000 kopiert bei der Erzeugung eines Objektes ebenfalls die Berechtigungen des übergeordneten Objektes. Es werden jedoch Veränderungen an den Berechtigungen der übergeordneten Objekte automatisch auch an die darunter liegende Objekte weitergegeben (*„automatic propagation of permissions“*) [SAT00]. Diese automatische Weitergabe der Berechtigungen kann unterbrochen werden, was später noch beschrieben wird.

Die Möglichkeit der Vererbung erleichtert die Zuweisung spezieller Rechte für einen ganzen Verzeichnisbaum.

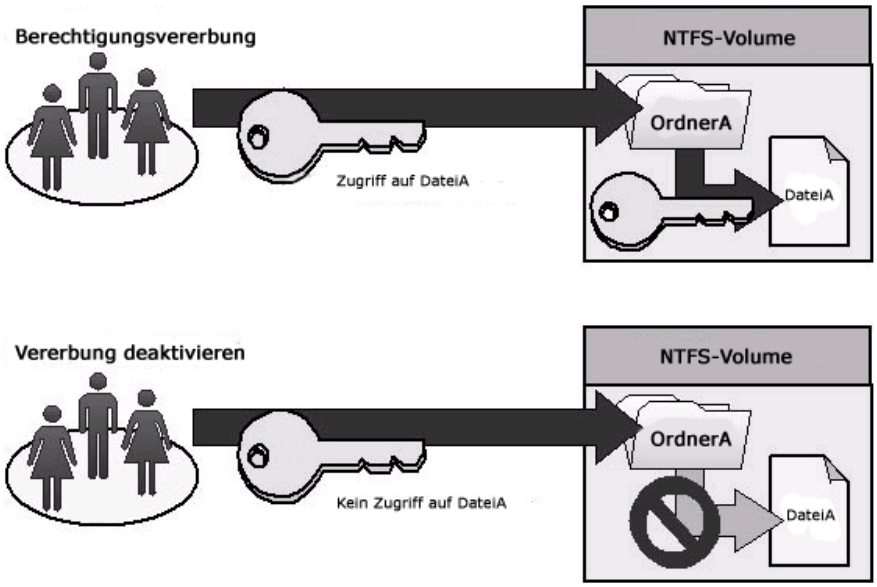


Abbildung 10 - Vererbung (aktiviert u. deaktiviert) [Mic02]

Abbildung 10 zeigt ein Beispiel. Im oberen Teil der Abbildung erbt „DateiA“ die Berechtigungen von „OrdnerA“. Der Gruppe von Benutzern, die links zu sehen ist, wird Zugriff auf „OrdnerA“ gewährt. Somit hat diese Gruppe auch Zugriff auf die in „OrdnerA“ enthaltene Datei „DateiA“.
Im unteren Teil ist die Weitergabe der Berechtigung an das Kindobjekt unterbrochen. „DateiA“ erbt nicht mehr die Berechtigungen von „OrdnerA“ und es können somit eigene Berechtigungen vergeben werden, in diesem Fall eine Verweigerung. Der Gruppe von Benutzern ist der Zugriff auf „DateiA“ daher verwehrt.

Selbstverständlich kann für jede Datei und für jeden Ordner festgelegt werden, ob sie bzw. er die Berechtigungen vom übergeordneten Objekt erben soll. Diese Weitergabe der Berechtigungen muss unterbrochen werden, wenn individuelle Berechtigungen vergeben werden sollen. Soll ein Objekt die Berechtigungen nicht mehr vom übergeordneten Objekt erben und eigene Berechtigungen zugewiesen bekommen, muss dem System mitgeteilt werden, wie es mit den bisherigen, geerbten Berechtigungen umgehen soll. Folgendes Dialogfenster wird dem Benutzer zur Entscheidung über die weitere Vorgehensweise angezeigt:

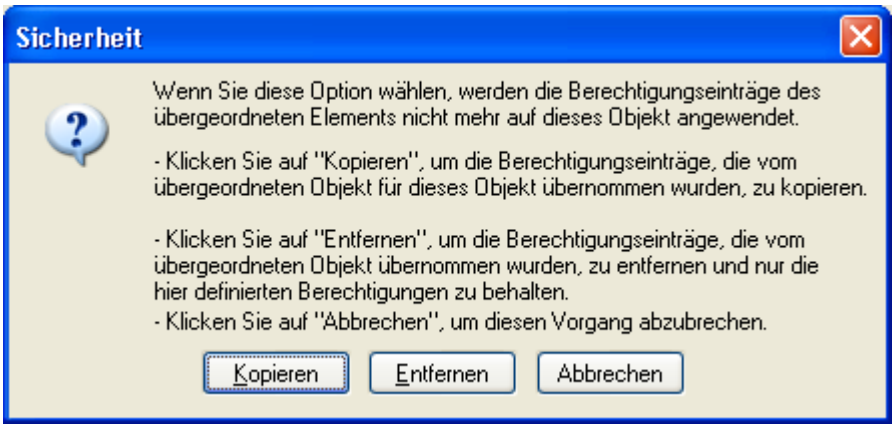


Abbildung 11 – Weitergabe der Berechtigungen aufheben

- Kopieren der Berechtigungseinträge:
  Die bisherigen Berechtigungen werden in das Attribut „Sicherheitsbeschreibung“ kopiert. Allfällige Änderungen (Hinzufügen oder Entfernen einzelner Berechtigungseinträge) können anschließend durchgeführt werden. Diese Alternative wird wahrscheinlich die am häufigsten verwendete sein.
- Entfernen der Berechtigungseinträge:
  Bei dieser Alternative werden alle Einträge entfernt und müssen im Anschluss für dieses Objekt völlig neu aufgebaut werden. Sollte dieses Objekt ein Ordner sein, werden wiederum die Berechtigungen dieses Ordners an alle Elemente, die darin enthalten sind, weitergegeben, sofern die Weitergabe nicht auch für diesen Ordner unterbrochen wird.

Eine weitere Besonderheit ist das Verhalten beim Kopieren oder Verschieben von Ordnern und Dateien. Wird ein Ordner oder eine Datei, der bzw. die die Berechtigungen vom übergeordneten Ordner erbt, kopiert, so erbt er/sie die Berechtigungen des Zielordners. Beim Verschieben ist hier ein weiterer Unterschied zu bemerken: Wird ein Ordner oder eine Datei zwischen NTFS-Ordnern desselben Datenträgers verschoben, so bleiben die ursprünglichen Sicherheitseinstellungen erhalten. Wird jedoch auf einen anderen Datenträger verschoben, so wird das Verschieben wie das Kopieren behandelt und die Sicherheitseinstellungen des Zielordners werden geerbt.
Voraussetzung für jegliches Kopieren oder Verschieben ist natürlich, dass der Benutzer die erforderlichen Berechtigungen sowohl für die zu kopierende Datei, als auch für den Zielordner besitzt. Ist dies nicht der Fall, ist der Vorgang ohnehin nicht möglich.

Eine Sonderrolle nimmt der persönliche Ordner (Home Directory) eines Benutzers ein. Für jeden Benutzer, der an einem System angelegt wird, wird im Verzeichnis „Dokumente und Einstellungen“ ein eigener Ordner für die lokale Speicherung der benutzerspezifischen, „persönlichen“ Daten angelegt. Hier werden beispielsweise auch die individuellen Startmenüeinträge oder die Dokumente des Benutzers abgelegt, die er im Ordner „Eigene Dateien“ speichert. In weiterer Folge werden wir diesen Ordner den „persönlichen Ordner“ nennen.

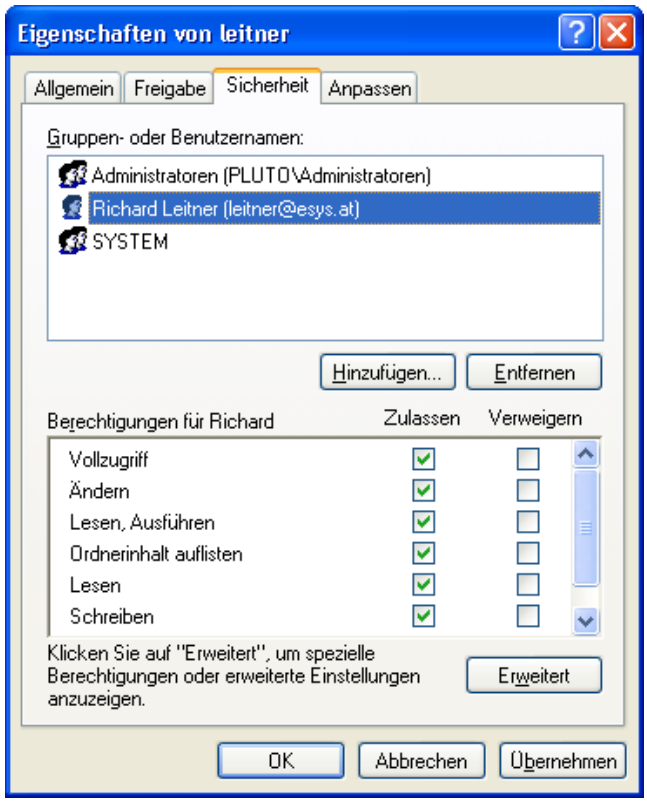


Abbildung 12 - Sicherheitseinstellungen für einen persönlichen Ordner (Home-Directory)

Auf den persönlichen Ordner soll nur der Besitzer Zugriffsrechte haben. Außer diesem hat selbstverständlich auch der Administrator alle Berechtigungen auf diesen Ordner. In Abbildung 12 sind die Berechtigungen für einen persönlichen Ordner abgebildet, worin die Berechtigungen deutlich zu erkennen sind. Die persönlichen Ordner erben nicht die Berechtigungen vom übergeordneten Ordner, sondern erhalten bei der Erstellung des Benutzers (z. B. mit dem Managementkonsolen-Snap-In „Lokale Benutzer und Gruppen“) eine eigene Access Control List. Darin werden nur dem Besitzer des persönlichen

Ordners Zugriffsberechtigungen erteilt. Sämtliche Dateien, die in diesem Ordner vom Benutzer oder von Anwendungsprogrammen erstellt bzw. in diesen Ordner kopiert werden, erben die Berechtigungen des persönlichen Ordners. Somit ist es nur für den jeweiligen Benutzer möglich, auf diese Dateien und Ordner je nach gesetzten Berechtigungen zuzugreifen (mit Ausnahme des Administrators). Alle anderen Benutzer haben keine Berechtigungen auf diese Dateien und Ordner.

3.1.2.3.Überprüfung der Zugriffsberechtigung

Wie schon erwähnt, muss das System die Authentizität des Benutzers eindeutig feststellen können, um überhaupt die Bearbeitung von Dateien und Verzeichnissen kontrollieren zu können. Damit dies sichergestellt werden kann, ist bei Windows XP ein „Authenticated Logon“ erforderlich, bevor auf Systemressourcen zugegriffen werden kann.
Fordert nun ein Thread den Zugriff auf ein Objekt an, wird mittels der Sicherheitsidentifikation des Rufers entschieden, ob er den angeforderten Zugriff ausführen darf oder nicht.
Bevor ein Thread Zugriff auf ein Objekt bekommt, muss er beim Öffnen des Objektes bekannt geben, auf welche Weise er auf das Objekt zugreifen möchte. Das Sicherheitssystem überprüft anschließend, ob der Thread die Berechtigung für die geforderte Aktion besitzt und gewährt bzw. verweigert einen „Handle“ auf das Objekt. Der Objektmanager zeichnet alle gewährten Zugriffsberechtigungen in der „Handle“-Tabelle eines Prozesses auf. Dabei werden der Methode *ObCheckObjectAccess* die Sicherheitsrichtlinien („Security Credentials“) des auf ein Objekt zugreifenden Threads, die auszuführenden Zugriffsoperation und ein Zeiger auf das Objekt übergeben. Diese Methode sperrt zuerst die Sicherheitsdaten des Objektes und den Sicherheitskontext des Threads (lock). Dadurch wird verhindert, dass ein anderer Thread im System die Sicherheitsrichtlinien des Objekts ändert, während die Sicherheitsüberprüfung durchgeführt wird.
Wenn ein Thread ein existierendes Objekt mittels dessen Namen öffnen will, wird der Objektmanager aktiv. Er sucht den entsprechenden Namen zuerst in seinem eigenen Namensraum. Wird er dort nicht fündig, muss eine interne Funktion aufgerufen werden, die das Objekt lädt und einen Eintrag in der „Handle“-Tabelle des Prozesses erzeugt, der auf das geladene Objekt verweist. Diese Funktion wird jedoch nur ausgeführt, wenn der Objektmanager die dafür notwendigen Berechtigungen festgestellt hat.

Der „Security Identifier“

Um in Windows XP eindeutig identifiziert werden zu können, erhalten Subjekte wie zum Beispiel Benutzer oder Computer eine eindeutige Identifikationsnummer – den „Security Identifier“ (SID). Ein SID ist ein numerischer Wert variabler Länge, der aus mehreren Teilen besteht. Der erste Teil ist eine „SID structure revision number“. Anschließend folgt ein 48-bit „Authority Value“, der den Dienst identifiziert, der diesen SID erstellt hat. Dies ist typischerweise ein Windows-System oder eine Domäne. Darauf folgt eine variable Anzahl von „Subauthorities“ und/oder ein „Relative Identifier“ (RID). „Subauthorities“ identifizieren „Trustees“, die zu einer bestimmten „Authority“ gehören. „Trustees“ sind vertrauenswürdige Bereiche in Windows-Netzwerken. RIDs geben Windows ganz einfach die Möglichkeit, verschiedene eindeutige Identifizierer auf der Basis eines SIDs zu erstellen.

Beispiel SID:

S-1-5-21-1463437245-1224812800-863842198-1128

Wird ein SID textuell dargestellt, so wird er mit einem S am Beginn eingeleitet. Im obigen Beispiel stellt die darauf folgende 1 die Revisionsnummer dar. 5 ist der „Authority Value“ (Windows Security Authority). Anschließend folgen vier „Subauthorities“, sowie der RID 1128.

Wie entsteht nun ein SID? Bei der Installation von Windows wird dem System durch das Setup-Programm ein SID zugewiesen. Die lokalen Benutzerkonten erhalten durch das Windows System ihre SIDs, wobei für jedes Benutzerkonto und für jede Gruppe der SID des lokalen Systems durch einen RID erweitert wird. Die RIDs für Benutzer und Gruppen beginnen mit 1000 und werden jeweils um eins erhöht. Der Administrator- und der Gast-Account haben vordefinierte RIDs: 500 für den Administrator und 501 für den Gast. Es gibt auch noch einige andere vordefinierte Werte. Ebenso wie für einen lokalen Rechner wird für jede Domäne ein eigener SID erzeugt.

Damit der „Security Reference Monitor“ (SRM) die Sicherheitsmerkmale eines Threads eindeutig identifizieren kann, verwendet er ein Objekt, das „Access Token“ genannt wird. In diesen Sicherheitsmerkmalen stehen Informationen über Berechtigungen, Be-

nutzerkonten und Gruppen, die mit dem Thread verbunden sind. Bei der Anmeldung an ein Windows XP System wird ein Token erzeugt, der den Benutzer repräsentiert. Anschließend wird der Token dem Logon-Shell-Prozess des Benutzers zugewiesen. Alle Programme, die vom Benutzer gestartet werden, erben eine Kopie des Token.

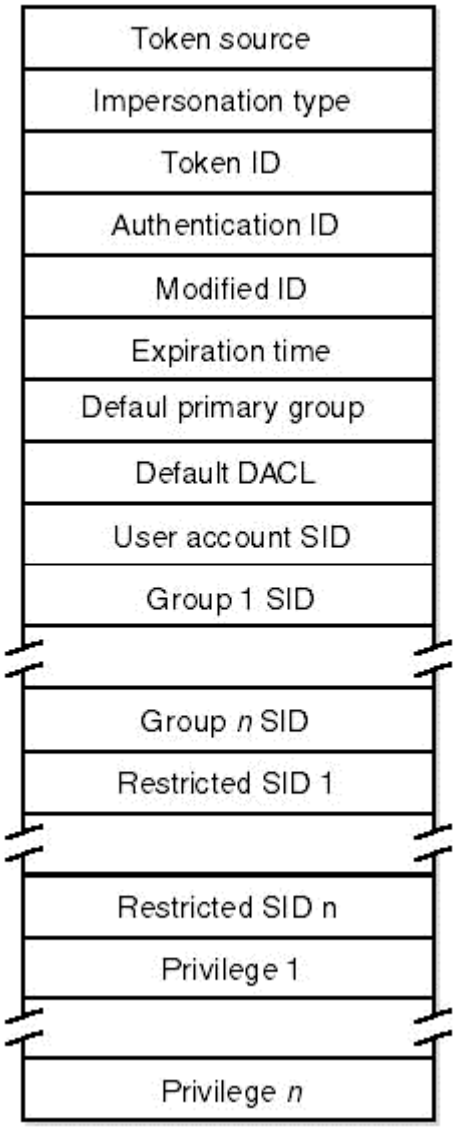


Abbildung 13 - Access Token [Mic02]

Zwei Komponenten eines Token werden vom Windows Sicherheitsmechanismus benützt, um festzustellen, was dem Thread erlaubt ist.

Einerseits werden die Felder mit den SIDs des Benutzerkontos und der Gruppen vom SRM verwendet, um die Zugriffsberechtigungen auf die zu schützenden Dateien zu überprüfen.
Andererseits werden die Rechte des Threads aus dem Token ausgelesen. Diese sind in einem Array im Token abgelegt. Ein Beispiel für ein solches Recht ist die Erlaubnis den Rechner herunterfahren zu dürfen (Name des Rechts: *SeShutdown*).

Security Descriptor und Access Control

So wie der Access Token dem Sicherheitssystem von Windows XP über die Berechtigungen des Benutzers informiert, so hält der Security Descriptor eines Objektes Informationen für das Sicherheitssystem darüber bereit, wer auf ein Objekt zugreifen kann und auf welche Weise dieser Zugriff geschehen darf. Diese Informationen werden direkt bei einem Objekt abgelegt. Das Objekt speichert also mit, wer welche Operationen auf ihm ausführen darf. Die Datenstruktur, die diese Informationen speichert, wird „Security Descriptor“ genannt.
Dieser „Security Descriptor“ besteht aus folgenden Elementen [Mic02]:

- Revisionsnummer: enthält die Version des SRM-Sicherheitsmodells, welches für die Erzeugung des Descriptors verwendet wurde;
- Flags: diese beschreiben das Verhalten bzw. die Charakteristik des Descriptors (optional; Beispiel: Vererbung nicht gestattet);
- SID des Besitzers
- SID der Gruppe: SID der „Hauptgruppe“ des Objektes;
- „Discretionary access-control list (DACL): legt fest, wer welche Zugriffsberechtigungen auf das Objekt hat;
- „System access-control list (SACL): wird für Protokollierung verwendet (welche Operationen von welchen Benutzern mitprotokolliert werden sollen);

Eine „Access Control List“ (ACL) besteht aus einem Header und einer beliebigen Anzahl an „Access Control Entries“ (ACE). Es ist auch möglich, dass eine ACL keinen ACE enthält.
Ein ACE besteht aus einem SID, einer Zugriffsmaske und einigen Flags, die hier jedoch nicht näher erläutert werden sollen.

Es gibt vier verschiedene Typen von ACEs: access-allowed, access-denied, allowed-object, denied-object. Access-allowed erlaubt den Zugriff und access-denied verweigert ihn. Die beiden anderen Typen sind nur innerhalb eines „Active Directory“ von Bedeutung. Bis auf einen kleinen Unterschied im Aufbau verhalten sie sich jedoch wie access-allowed und access-denied.
Windows XP verwaltet seine Berechtigungen in so genannten DACLs. Sollte ein Objekt keine DACL besitzen (null DACL), so ist jedem der Zugriff auf dieses Objekt gestattet. Gibt es jedoch eine DACL ohne ACEs, dann hat niemand eine Berechtigung zum Zugriff.

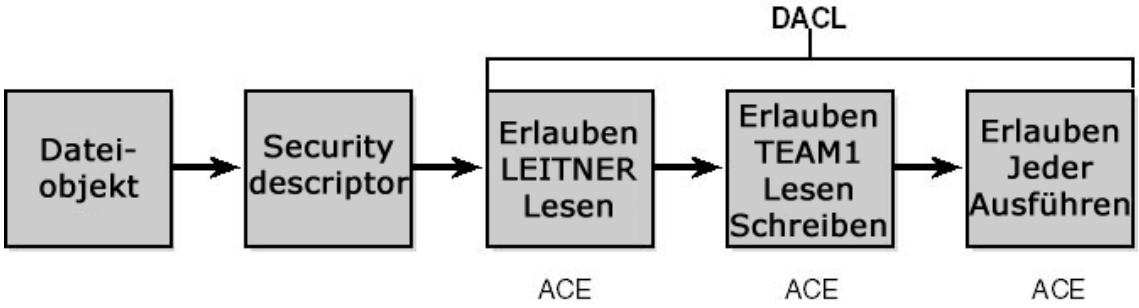


Abbildung 14 – DACL [Mic02]

Entscheidung über den Zugriff

Es werden zwei Algorithmen verwendet, die darüber entscheiden, ob auf ein Objekt der Zugriff gestattet wird oder nicht:

- Der Erste überprüft wie umfangreich die Berechtigungen zum Zugriff auf ein Objekt sind. Er bestimmt also die weitestgehende Erlaubnis. Dies kann im „User Mode“ beispielsweise durch den Aufruf der Win32-Funktion *GetEffectiveRightsFromAcl* realisiert werden.
- Ein zweiter Algorithmus überprüft, ob der gewünschte Zugriff auf ein Objekt erlaubt ist. Diese Überprüfung wird durch den Aufruf der Funktion *AccessCheck* bzw. *AccessCheckByType* durchgeführt.

Der erste Algorithmus geht nach folgendem Ablauf vor:

1. Hat das Objekt keine DACL, gibt es keine Einschränkungen und voller Zugriff wird gewährt.

2. Hat der Rufer das Recht, sich als Besitzer dieses Objektes einzutragen („take-ownership privilege“), so wird das Ändern des Besitzers erlaubt und anschließend die DACL weiter überprüft.
3. Ist der Rufer gleichzeitig der Besitzer des Objektes, werden die Berechtigungen „read-control“ und „write-DACL“ erteilt.
4. Scheint ein SID, der im Token des Rufers übergeben wird, in einem ACE vom Typ „access-denied“ auf, so werden die Zugriffsberechtigungen, die diesen ACE abdecken, aus der Liste der erlaubten Zugriffe entfernt.
5. Scheint ein SID in einem ACE vom Typ „access-allowed“ auf, so werden die Zugriffsberechtigungen dieses ACE zu der Liste der erlaubten Berechtigungen hinzugefügt, es sei denn sie wurden vorher schon durch eine Verweigerung daraus entfernt.

Der zweite Algorithmus überprüft, basierend auf der geforderten Liste an Zugriffsberechtigungen des Rufers, ob ihm der gewünschte Zugriff gestattet ist oder nicht. Um dies festzustellen, müssen folgende Schritte durchgearbeitet werden (diese Vorgehensweise ist etwas vereinfacht dargestellt):

1. Gibt es für das Objekt keine DACL, so gibt es auch keine Beschränkungen und es wird sofort der gewünschte Zugriff gestattet.
2. Hat der Rufer das Recht, sich als Besitzer dieses Objektes einzutragen, wird ihm eine Berechtigung zum Ändern des Besitzers gewährt und die DACL weiter überprüft.
3. Ist der Rufer der Besitzer des Objektes, so erhält er die Berechtigung, die Eigenschaften zu lesen und die DACL zu verändern. Sollte eine dieser beiden Berechtigungen die zu Überprüfende gewesen sein, so wird die DACL nicht weiter beachtet; ansonsten wird weiter überprüft.
4. Jeder ACE einer DACL wird überprüft, und zwar von der ersten bis zur letzten. Hier ist jedoch auch die Reihenfolge der ACEs von entscheidender Bedeutung. Wird der geforderte Zugriff nun im ACE vom Typ „access-allowed“ gefunden, so wird dieser Zugriff gewährt. Sollten alle geforderten Zugriffe gefunden werden, so ist die Überprüfung der Berechtigungen erfolgreich und Zugriff wird gewährt, auch wenn in einem ACE, der später in der DACL steht, ein solcher Zugriff verboten wird. Wird jedoch bei der Überprüfung ein ACE vom Typ „ac-

cess-denied“ vorgefunden, so wird beim ersten Vorkommnis einer der gewünschten Berechtigungen der Vorgang abgebrochen und der Zugriff nicht genehmigt.

5. Ist nun das Ende der DACL erreicht und es wurden noch nicht alle geforderten Zugriffstypen genehmigt, so wird der Zugriff ebenfalls verweigert.

Bei beiden Algorithmen ist die Reihenfolge der ACEs in der DACL von entscheidender Bedeutung. Bei der Überprüfung der Berechtigungen wird nicht darauf geachtet, ob sich weiter hinten in der DACL noch ein ACE befindet, der dem vorliegenden Ergebnis widerspricht. Es ist also möglich, dass ein Benutzer auf ein Objekt vollen Zugriff erhält, nur weil der erlaubende Eintrag vor dem verweigernden Eintrag in der Liste steht.
Um diesen in einem Sicherheitssystem nicht tragbaren Zustand zu verhindern, werden ab Windows 2000 die ACEs in einer DACL beim Anlegen in die gewünschte Reihenfolge gebracht. Es werden alle ACEs vom Typ „access-denied“ vor denen vom Typ „access-allowed“ gereiht, damit die Sicherheit in sinnvoller Weise gewährleistet ist.
Diese Sortierung muss auch bei jeder Änderung der Zugriffsberechtigungen wiederhergestellt werden. Diese Aufgabe übernimmt das System selbst.

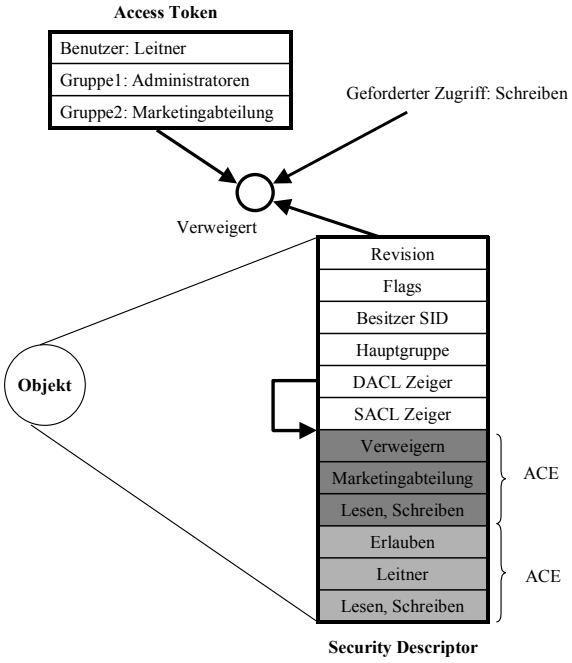


Abbildung 15 - Beispiel für eine Berechtigungsprüfung

Im obigen Beispiel ist deutlich die Vorgangsweise bei der Überprüfung eines Zugriffes zu erkennen. Der Benutzer „Leitner“ möchte auf ein Objekt zugreifen. „Leitner“ ist Mitglied der Gruppen „Administratoren“ und „Marketingabteilung“. Nun versucht er einen Schreibzugriff auf das vorliegende Objekt durchzuführen. Der DACL-Zeiger verweist auf die ACEs, die in vorgegebener Reihenfolge am Ende des Access Tokens angehängt wurden. Der erste ACE, der überprüft wird, ist der ACE der Gruppe „Marketingabteilung“. Dieser verweigert dieser Gruppe den Lese- und den Schreibzugriff auf die gewünschte Datei. Somit darf der Benutzer „Leitner“ nicht auf das Objekt zugreifen, obwohl ihm der zweite ACE dies ausdrücklich erlauben würde. Er befindet sich jedoch in der Reihenfolge hinter dem Verweigerungseintrag und wird daher nicht mehr ausgewertet.

3.2. Unix/Linux

Bei der Entwicklung von Unix wurde der Weg eingeschlagen, in den Attributen jeder Datei einen Zugriffsmodus mit abzuspeichern. Dieser Zugriffsmodus wird von einer 16-bit Zahl dargestellt.

Unter den Attributen, die einer Unix-Datei zugeordnet sind, befinden sich auch jene, die für die Feststellung der Zugriffsberechtigungen verwendet werden. Es sind dies die Attribute „Eigentümer UID“ und „Gruppen ID“.

3.2.1. Benutzer und Gruppen

Unix speichert die Benutzer und Gruppen in eigenen Dateien, woraus bei der Anmeldung die jeweiligen Informationen geladen werden. Die Dateinamen können in einzelnen Distributionen von Unix/Linux abweichen. Die folgenden Ausführungen können daher nur als Beispiel für die interne Speicherung der Benutzer- und Gruppeninformationen angesehen werden.

Die Informationen über die einzelnen Benutzerkennungen sind in der Datei „/etc/passwd“ gespeichert. Das Recht, solche Benutzerkennungen zu erzeugen, ist dem Administrator (Benutzer „root“) vorbehalten, dem die Benutzernummer 0 zugewiesen wird. Mit dieser hat er bei der Überprüfung der Berechtigungen einige Vorzüge, die im Abschnitt „Zugriff auf Dateien“ näher erläutert werden. Meist erfolgt die Erzeugung einer neuen Benutzerkennung menügeführt.

Auszug aus einer Datei „/etc/passwd“:

```
huber:Cwt508Dw0SerY:4:1:Hans Huber, 21054
marina:NTLnC5R2n9ykc:6:5:Marina Müller, 34589
manual:UwkaAxDJVtyko:9:7:Manualredakteur
```

Jede Benutzerkennung wird in eine eigene Zeile geschrieben, wobei die einzelnen Teile eines Eintrages durch Doppelpunkte getrennt sind. Der erste Eintrag stellt die Benutzerkennung dar, gefolgt vom verschlüsselt gespeicherten Passwort. Anschließend folgt die Benutzernummer, die intern den zugehörigen Benutzer repräsentiert. Durch diese Nummer wird der Benutzer im System eindeutig identifiziert. Sie kann also mit dem SID in Microsoft Windows XP verglichen werden. Der nächste Eintrag ist die Grup-

pennummer. Jeder Benutzer muss mindestens einer Gruppe zugeordnet werden. Dieser ist er nach der Anmeldung an das System zugeordnet. Anschließend folgt ein Kommentarfeld, in dem meist der vollständige Name des Benutzers eingetragen wird. Optional können noch zwei weitere Einträge folgen. Der erste benennt das Dateiverzeichnis, das als Login-Dateiverzeichnis des Benutzers dienen soll. Dieses Verzeichnis wird nach der Anmeldung am System automatisch zum aktuellen Verzeichnis („home“). Der zweite optionale Eintrag betrifft das Programm, das nach der Anmeldung des Benutzers ausgeführt werden soll. Wird in diesem Feld nichts eingetragen, wird automatisch ein Kommando-Interpreter („Shell“) gestartet.

Die Gruppen werden wiederum vom Systemadministrator („root“) in der Datei „etc/group“ definiert. Die Erstellung neuer Gruppen wird meist durch ein menügeführtes Programm unterstützt. Für jede neue Gruppe wird eine Zeile in die Datei eingefügt, die folgenden Aufbau hat:

```
root::0:root,daemon,cron
manual::6:huber,meier
kermit::8:mueller,huber
```

Jede Zeile besteht aus vier Elementen. Das erste Element gibt den Namen der Gruppe an. Das zweite Element stellt das verschlüsselt gespeicherte Passwort der Gruppe dar, welches bei den meisten Systemen jedoch nicht unterstützt wird. Dieses Passwort müsste ein Benutzer eingeben, wenn er die Gruppe wechseln wollte. Ob diese Funktion zur Verfügung steht, kommt auf das verwendete Unix-System an. Einerseits gibt es die Möglichkeit, dass ein Benutzer, der mehreren Gruppen angehört, gleichzeitig als Mitglied aller Gruppen gilt. Andererseits gibt es in manchen Systemen die Möglichkeit, mit dem Befehl „newgrp“ in eine andere Gruppe zu wechseln.
Das dritte Element gibt die eindeutige Nummer der Gruppe an. Diese Nummer wird auch in der Datei der Benutzerkennungen verwendet, um einen Benutzer einer Gruppe zuzuordnen. Der vierte Element einer Zeile ist eine Liste der Benutzer, die zu dieser Gruppe zugeordnet sind. Die einzelnen Einträge werden durch ein Komma voneinander getrennt. Wie im obigen Beispiel zu sehen ist, kann ein Benutzer mehreren Gruppen zugeordnet werden.

Üblicherweise werden jene Benutzer in einer Gruppe zusammengefasst, die gemeinsam an einem Projekt arbeiten oder einer Abteilung angehören und somit auf dieselben Daten im System zugreifen müssen. Es besteht aber auch die Möglichkeit, einer Gruppe nur einen Benutzer zuzuordnen, um ihm spezielle Rechte im Dateisystem zu geben.

Jede Unix-Datei hat einen Eigentümer, der ihr zumeist bei der Erstellung der Datei zugeordnet wird. Nur der Administrator ist berechtigt, den Eigentümer zu ändern. Diesem Eigentümer können eigene Zugriffsberechtigungen auf eine Datei zugeordnet werden. Genauso verhält es sich mit der „Gruppen ID“. Jede Unix-Datei ist auch einer Gruppe zugeordnet, die wiederum völlig unabhängig vom Eigentümer der Datei ist; es ist also nicht notwendig, dass der Eigentümer Mitglied der Gruppe ist.

3.2.2. Zugriffsberechtigungen auf Dateien

Bei Dateien kann zwischen drei Zugriffsberechtigungen unterschieden werden: Lese- (r), Schreib- (w) und Ausführungsberechtigung (x). Diese können natürlich kombiniert werden oder es wird keine von ihnen gesetzt. Auch kann man eigene Berechtigungen für den Eigentümer, für die Gruppe und für alle übrigen Benutzer festlegen.
Die 16-bit Zahl, die für die Speicherung der Zugriffsberechtigungen verwendet wird, wird dafür in fünf Werte aufgeteilt. Die ersten vier Bits legen den Dateityp fest (z.B.: d für Verzeichnis, - für eine normale Datei, l für einen Verweis (Link)). Die restlichen Bits werden in vier Werte zu drei Bits unterteilt, die nun die Werte für die Rechte enthalten.
Die Zuordnung auf die Bits erfolgt folgendermaßen:

Recht	r	w	x
Bit-Wert	4	2	1

Die ersten drei Bits beziehen sich auf die Ausführung von Programmen und haben im engeren Sinn nichts mit den Rechten zu tun. Es handelt sich dabei um einen vorgegebenen Wert, der eine Prozess-ID vorgibt, unter der das Programm im Unix-System gestartet werden soll. Es wird in weiterer Folge nicht näher darauf eingegangen.
Die übrigen 3x3 Bits legen nun die Rechte für den Eigentümer, für das Gruppenmitglied und für alle übrigen Benutzer fest. Die Werte der einzelnen Rechte für jeden Teil wer-

den nun addiert, sodass dadurch Oktalziffern zur leichteren Handhabung entstehen (0-7).
Diese Vorgehensweise soll anhand eines Beispiels näher erläutert werden. Dazu soll uns ein Ausschnitt einer Auflistung eines Verzeichnisses dienen, der den Eintrag für eine einzelne Textdatei darstellt. Um eine solche Auflistung in Unix mit der Darstellung aller Dateiattribute zu erhalten, ist in der Shell (Kommando-Interpreter) der Befehl „ls -l“ einzugeben:

```
-rw-r----- 1 leitner fim … test.txt
```

In obiger Darstellung wurden einige Details ausgeblendet (…), die diesem Beispiel nicht weiter dienlich sind. Die ersten zehn Zeichen stellen die Berechtigungen dar.

```
Dateiart  Benutzer  Gruppe  Rest
    -        rw-      r--    ---
```

Dabei ist ersichtlich, dass es sich um eine reguläre Datei handelt. Würde es sich zum Beispiel um ein Verzeichnis handeln, wäre in der Spalte Dateiart ein „d“ eingetragen. Der Eigentümer der Datei ist der Benutzer „leitner“, der Lese- und Schreibberechtigungen hat (rw-). Die Mitglieder der Gruppe „fim“ dürfen die Datei nur lesen (r--). Alle sonstigen Benutzer des Unix-Systems sind nicht berechtigt, Operationen auf dieser Datei auszuführen.
Numerisch können die Rechte folgendermaßen dargestellt werden:

```
rw- = 4 + 2 + 0 = 6
r-- = 4 + 0 + 0 = 4
--- = 0 + 0 + 0 = 0
```

Der numerische Wert der gesamten Zugriffsberechtigungen ist also 640. Dieser Wert wird auch beim Setzen der Berechtigungen verwendet, was durch die Eingabe des Befehls „chmod“ durchgeführt wird. Die Rechte wurden bei obigem Beispiel also durch folgenden Befehl gesetzt: *chmod 640 test.txt*.

Ein Benutzer, der von diesen Einschränkungen nicht betroffen ist und alle Berechtigungen hat, ist der Superuser (Benutzernummer = 0). Dieser ist somit der Administrator oder der Benutzer „root“, wie dieser in Unix genannt wird. Das Passwort, das diesem Benutzer gegeben wird, sollte natürlich sehr sorgfältig ausgewählt werden und nur dem Systemverwalter und dessen Stellvertreter bekannt sein.

Eine Besonderheit im Bezug auf die Rechtvergabe stellen Verzeichnisse dar. Um sich die Auflistung der Dateien eines Verzeichnisses ansehen zu können, reicht es nicht aus, die Leseberechtigung zu haben. Dafür ist die Ausführungsberechtigung notwendig. Schreibzugriff gestattet auch Veränderungen am Verzeichnis.
Diese Art der Zugriffskontrolle erlaubt nun einige Dinge, die besonderer Beachtung bedürfen. Das Löschen einer Datei in einem Verzeichnis erfordert nur Schreibrechte auf das Verzeichnis, nicht jedoch auf die Datei selbst, da dies als Modifikation des Verzeichnisses interpretiert wird. Über Umwege ist dadurch jedoch eine Modifikation der Datei selbst möglich, indem man die Originaldatei löscht und durch eine modifizierte Kopie der Datei ersetzt. Beide Operationen erfordern nur Schreibzugriff auf das Verzeichnis. Man sieht, dass auch bei der Administration eines Unix-Systems besondere Vorsicht geboten ist.

3.2.3. Zugriff auf Dateien

Beim Zugriff eines Benutzers auf die Datei werden generell vier Schritte durchlaufen [Gro91]:

1. Ist die Benutzernummer des laufenden Prozesses (Unix spricht nach der Anmeldung am System von einem „laufenden Prozess“ mit einer gewissen Benutzerkennung.) gleich 0, so ist aktuell der Benutzer „root“ - also der Systemverwalter - angemeldet, und es wird sofort jeglicher Zugriff auf die Datei gewährt.
2. Sollte dies nicht der Fall sein, so kommt die Eigentümer-Identifikationsnummer der Datei ins Spiel. Es wird nun überprüft, ob die Benutzernummer des laufenden Prozesses mit der Eigentümer-Identifikationsnummer der Datei übereinstimmt. Ist dies der Fall, gelten die Zugriffsrechte des Dateieigentümers.
3. Stimmen die Benutzernummern nicht überein, so wird im nächsten Schritt die Gruppennummer des Prozesses mit der Gruppennummer der Datei verglichen.

Gibt es hier eine Übereinstimmung, so richten sich die Zugriffsrechte nach den Berechtigungen, die die Gruppe erhalten hat, der die Datei zugeordnet wurde.

4. Falls auch diese Gruppennummer nicht übereinstimmt, so werden im letzten Schritt die Zugriffsrechte der sonstigen Benutzer überprüft. Sollte hier ein Zugriff gestattet sein, wird er gewährt. Sollten andernfalls keine Rechte für sonstige Benutzer gesetzt sein, wird der Zugriff nicht gestattet.

Die folgende Abbildung soll diesen Ablauf der Prüfung der Zugriffsrechte noch einmal graphisch verdeutlichen:

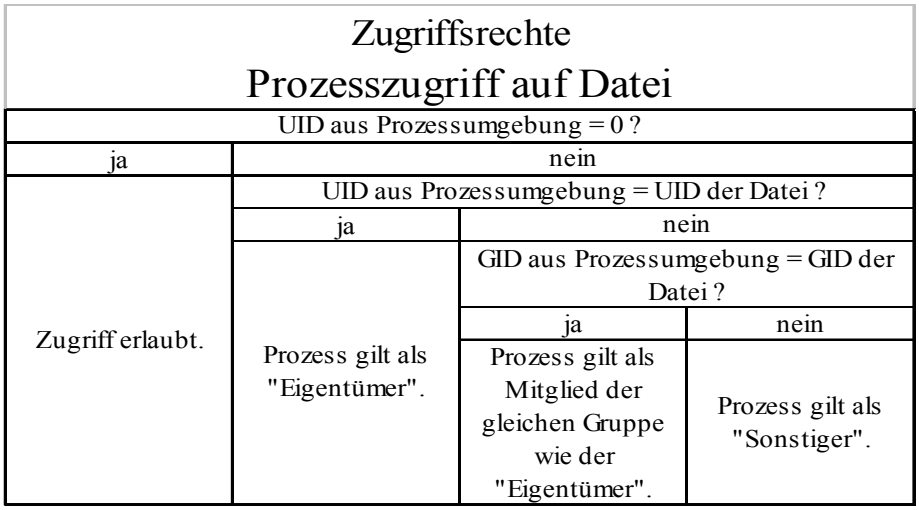


Abbildung 16 - Steuerung des Dateizugriffs am Beispiel Unix

3.3. Mac OS X

Laut den Ausführungen in [App02] basiert Mac OS X - das Betriebssystem, das die Firma Apple für ihre Rechner verwendet - auf BSD (Berkley Software Distribution), einer Unix-Variante. Es ist daher nicht sehr verwunderlich, dass Mac OS X die Verwaltung der Zugriffsberechtigungen fast identisch regelt wie Unix. Somit fällt die Beschreibung der Zugriffssteuerung von Mac OS X sehr kurz aus.

Mac OS X kann die Berechtigungen ebenfalls für Besitzer, Gruppen und Sonstige vergeben. Auch die Arten der Berechtigungen sind gleich (rwx). Die Verwaltung der Zugriffsberechtigungen unterscheidet sich daher nicht von Unix.

Der einzige Unterschied zu Unix liegt in der Funktion des Benutzers „root“. In Unix und Linux ist der Benutzer „root“ jener Benutzer, der alle Berechtigungen hat und somit auch alle Einstellungen der Dateien und des Systems ändern kann. Mac OS X blendet jedoch den Benutzer „root“ aus. Stattdessen gibt es eine Gruppe mit dem Namen „admin“, der alle Administratoren zugeordnet werden. Diese Administratoren haben - nicht wie unter Unix - keinen uneingeschränkten Zugriff auf alle Dateien des Systems (System Domain), da es ihnen zum Beispiel verboten ist, grundlegende Systemdateien zu manipulieren. Auch sonst sind ihnen im Vergleich zum Benutzer „root“ unter Unix einige Einschränkungen auferlegt. Ansonsten können sie jedoch genauso handeln, wie der Systemverwalter eines Unix-Systems (Benutzer und Gruppen erzeugen, Berechtigungen auf Dateien und Verzeichnissen vergeben etc.). Durch diese Einschränkungen wird einerseits verhindert, dass unbeabsichtigt schwerwiegende Veränderungen am System durchgeführt werden und es somit instabil wird. Andererseits wird damit verhindert, dass vom Netzwerk Unbefugte die Kontrolle über das gesamte System übernehmen können und die regulären Benutzer ausgesperrt werden. Dies ist bei Unix dann der Fall, wenn sich ein Unbefugter des „root“-Passwortes bemächtigt und es ändert. Sobald dies geschehen ist, kann der lokale Systemverwalter nur noch das System herunterfahren (falls er das noch darf) und neu installieren, da er keinerlei Veränderungen an der Berechtigungsstruktur mehr durchführen kann.

Um dennoch die Möglichkeit zu haben, im absoluten Notfall die Kontrolle über ein System mit Mac OS X zu übernehmen, kann ein Mitglied der Gruppe „admin“ den „root“-Benutzer aktivieren. Dies ist jedoch nur lokal am System erlaubt und kann somit nicht von einem Angreifer über das Netzwerk durchgeführt werden. Da dieser Benutzer dann

- wie in Unix - alle Berechtigungen besitzt, sollte nur das Notwendigste erledigt werden, um den „root“-Benutzer anschließend sofort wieder zu deaktivieren.

4 Das Distanceteaching / Distancecoaching / Distancelearning-Framework WeLearn Release 2

4.1. Beschreibung

WeLearn („Web Environment for Learning“) ist eine webbasierte Lernumgebung. Das bedeutet, dass Lehrinhalte von Kursanbietern online für die Teilnehmer eines Kurses verfügbar gemacht werden. Dazu gehört nicht nur das Anbieten von Content (Inhalt) via HTML, sondern es ist auch möglich, Kursmaterial in verschiedensten Dateiformaten über die Webplattform zugänglich zu machen. Unter anderem unterstützt WeLearn pdf- (Adobe Acrobat Reader), ppt- (Microsoft PowerPoint) und doc-Dateien (Microsoft Word). Vorraussetzung ist nur, dass der entsprechende Viewer auch auf dem Computer des Kursteilnehmers vorhanden ist.

Dadurch, dass die zu verwendenden Dateiformate nicht vorgegeben sind, ist eine universelle Einsetzbarkeit möglich. Es sei hier nur an die Möglichkeit gedacht, WeLearn als Basis für ein Wissensmanagement-System zu verwenden. Dabei ist auch die einfache Erweiterbarkeit von Vorteil, da man neu benötigte Objekttypen ohne weiteres in das System integrieren kann.

Von großer Bedeutung ist auch, dass WeLearn Kursmaterial darstellen kann, welches nach der Content Packaging Specification (CPS) des IMS Global Learning Consortium standardisiert erstellt worden ist. Somit kann man auch Lehrmaterial verwenden, das eigentlich für andere Plattformen erstellt worden ist, sofern diese ihrerseits CPS implementieren. Näheres dazu ist im Kapitel 4.1.2. nachzulesen.

4.1.1. Designziele

Ein wesentliches Ziel bei der Entwicklung von WeLearn war es, ein universell einsetzbares Framework zu schaffen, um jegliche Art von Kursen vollständig oder größtenteils online abhalten zu können. Dabei ist WeLearn nicht nur für Universitäten und Schulen geeignet, sondern kann von jedem anderen Anbieter verwendet werden.

Damit WeLearn möglichst einfach an die Bedürfnisse unterschiedlicher Kursanbieter angepasst werden kann, ist eine leichte Erweiterbarkeit notwendig. Daher fiel die Wahl für die Implementierung von WeLearn auf Java-Servlets, da man dadurch die Möglichkeit einer Erweiterung durch Java-Klassen bietet. Diese müssen lediglich den vorgege-

benen Schnittstellen entsprechen und können auch zur Laufzeit in das System eingebunden werden.
Die Implementierung von WeLearn mittels Java bietet die Möglichkeit, sowohl Unix- als auch Windowsplattformen auf der Serverseite einzusetzen. Clientseitig gibt es ohnehin keine Einschränkungen bezüglich des Betriebssystems. Hier ist nur ein HTML-fähiger Browser erforderlich.

Ein weiteres überaus wichtiges Designziel von WeLearn war es, dass alle verwendeten Komponenten frei verfügbar sind, was dem Benutzer (Schulen, Universitäten, etc.) oft sehr hohe Lizenzkosten erspart. Das WeLearn-System selbst wurde daher eingeschränkt auf bestimmte Länder als Open Source Projekt konzipiert.

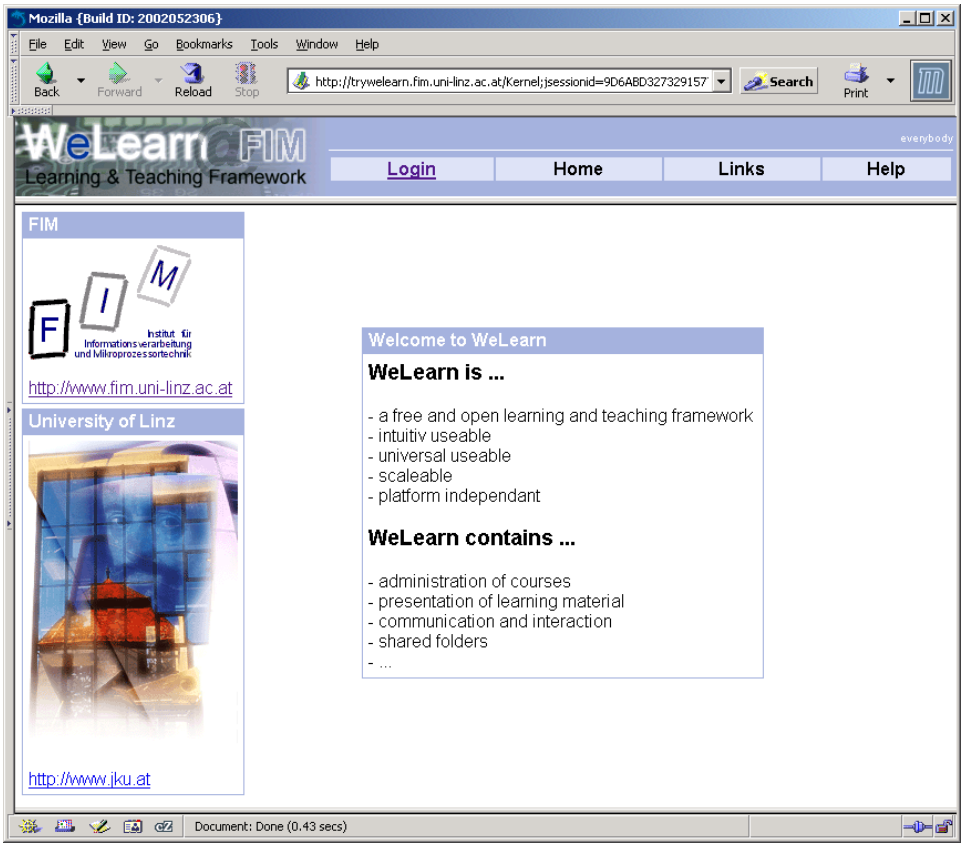


Abbildung 17 - Das WeLearn System Release 2

Als äußerst positiven Aspekt bei der Implementierung des WeLearn-Systems ist zu vermerken, dass die Installation keine zusätzliche Hard- und Software sowie keine Än-

derungen in der bestehenden Netzwerktopologie erfordern darf, was nämlich mit meist sehr hohen Anschaffungskosten verbunden wäre.

Auch die Benutzerfreundlichkeit muss bei der Entwicklung einer Distancecoaching-Plattform berücksichtigt werden. Denn erst eine möglichst einfache und intuitive Bedienung fördert die Akzeptanz bei den Benutzern. Dabei ist zu beachten, dass diese Forderung für die Lernenden und für die Unterrichtenden gleichermaßen zutrifft.

WeLearn ermöglicht nicht nur die Verwaltung der Kursinhalte, es integriert auch die vollständige Administration eines Kurses und die Verwaltung der Benutzeraccounts. Der Administrator (in vielen Fällen der Coach des Kurses) legt für jeden Kursteilnehmer einen Account im WeLearn-System an. Mit diesem kann sich der Teilnehmer am System anmelden und bekommt nur jene Teile des Systems zu sehen, für die er vom Administrator die Berechtigung erhalten hat. Somit ist hier durchaus auch ein Rechtesystem vorhanden, dass die Administration und den Zugang zum WeLearn-System regelt. Durch diese Funktionalität können mehrere Kurse mit unterschiedlichen Teilnehmergruppen auf einem Server betrieben werden. Jede Gruppe sieht nur jene Teile des Lehrinhalts, die für sie gedacht sind. Der Lehrer kann jedoch in einem System alle seine Kurse verwalten und muss sich nicht mit verschiedenen Eingabemasken auseinandersetzen.

Die gesamte Administration der Kurse und der Benutzer erfolgt ebenfalls webbasiert. Der Administrator vergibt auf die jeweiligen Elemente (Verzeichnisse, Foren, Kurse, etc.) Rechte für die Benutzer. Somit können sie Angaben zu Übungen zum Beispiel nur lesen, nicht jedoch löschen oder verändern.

Jedem Benutzer kann auch ein eigener, privater Ordner im virtuellen Dateisystem zur Verfügung gestellt werden. In diesem kann er seine Dateien individuell verwalten und ablegen, wobei nur er auf sie zugreifen darf.

Ein zusätzlicher positiver Aspekt ist die Vereinfachung der Verwaltung bei der Abwicklung eines Kurses. Alle Kursteilnehmer sind im System erfasst und haben einen eindeutigen Benutzernamen. Über speziell adaptierte Foren oder Verzeichnisse kann man ein Prüfungsanmeldesystem simulieren, um die Prüfungsanmeldung möglichst benutzerfreundlich und einfach zu gestalten. Durch die gute Erweiterbarkeit ist es auch kein Problem, ein eigenes Objekt zu implementieren, das diese Funktionalität zur Verfügung stellt.

Doch WeLearn ist nicht nur eine Plattform zur Präsentation von Lehrinhalten. Es unterstützt sowohl die Kommunikation zwischen den Kursteilnehmern untereinander, als auch die Kommunikation zwischen Kursteilnehmer und Coach über spezielle Foren. In diesen Foren kann über vorgegebene Themen diskutiert und mit dem Coach für Fragen und Anmerkungen Kontakt aufgenommen werden.

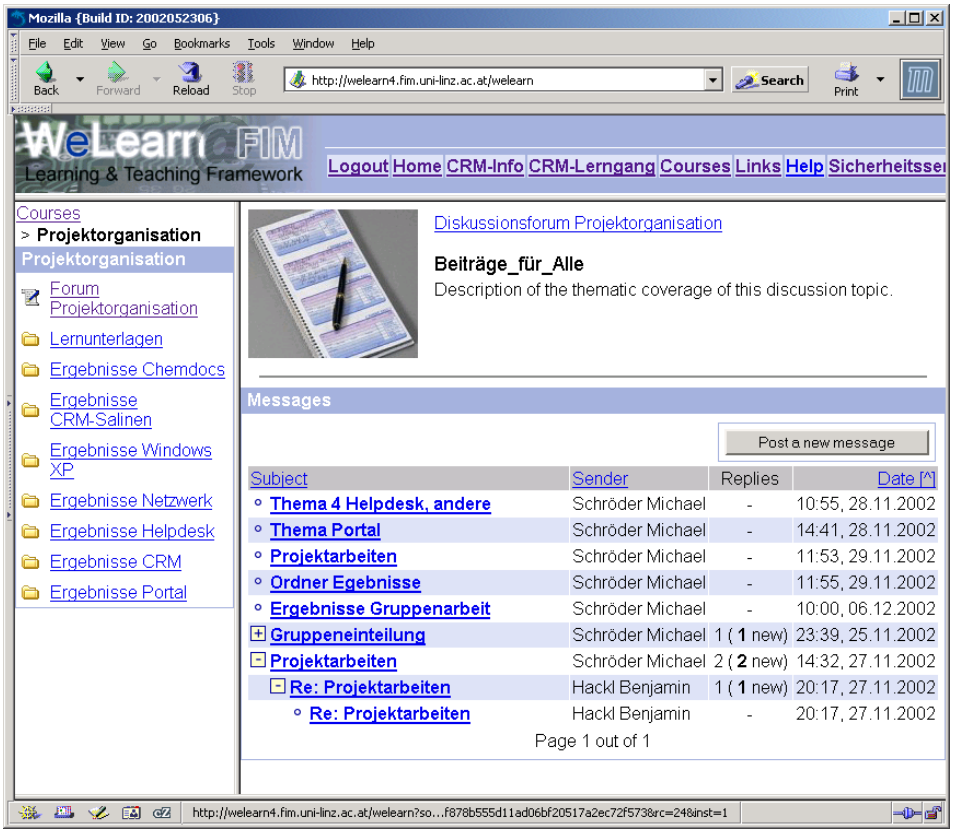


Abbildung 18: WeLearn Release 2 - Diskussionsforum

4.1.2. IMS Content Packaging Specification

Die WeLearn-Plattform versteht es auch, Lehrinhalte korrekt anzuzeigen, die nach den „Standards“ des „*IMS Content Packaging Information Model*“ [IMS01] erstellt wurden. Diese Spezifikationen fördern den Austausch von Lehrinhalten zwischen verschiedenen Lehr-/Lernplattformen, da sie den Aufbau dieser Inhalte standardisieren und in ein XML-Schema verpacken. Dieser Aspekt war einer der Hauptschwerpunkte der Entwicklung der webbasierten Telecoaching-Plattform WeLearn.

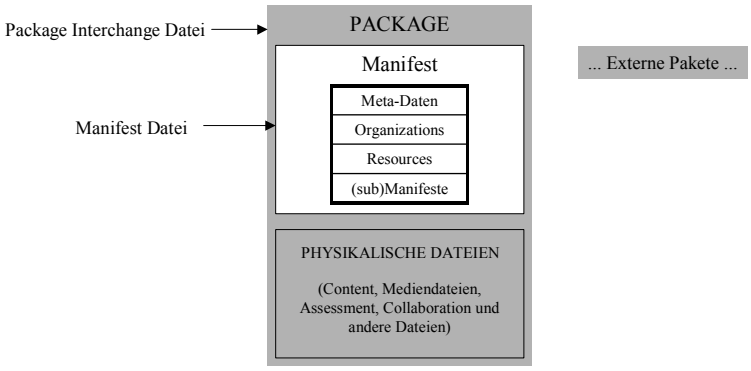


Abbildung 19 - Der Umfang des IMS Content Packaging [IMS01]

Ein IMS-Paket besteht grundsätzlich aus zwei Teilen:

1. eine XML-Datei, die die Struktur des Paketes und die Verknüpfungen zu den Inhalte enthält (IMS-Manifest);
2. physikalische Dateien, die durch das Manifest beschrieben werden und den konkreten Inhalt enthalten.

Die Datei, die den Aufbau beschreibt und die Verknüpfung zu den einzelnen Dateien enthält, wird IMS-Manifest genannt. Diese Datei hat einen standardisierten Namen („imsmanifest.xml“). Diese ist der Einstiegspunkt für jede Applikation, die IMS-Pakete verarbeiten bzw. erstellen kann. Sie besteht wiederum aus mehreren Teilen, wobei nicht immer alle enthalten sein müssen:

- Metadaten: Diese XML-Elemente beschreiben das Paket als Ganzes und geben somit Applikationen, die dieses Paket verwenden, wichtige Informationen über verschiedenste Eigenschaften des Pakets. Diese Metadaten sind optional, bieten jedoch die Möglichkeit, detailliertes Wissen über den Inhalt und den Aufbau des Pakets zu sammeln.
- Organisationen: Um die Struktur des Inhaltes eines Pakets abbilden zu können, wird das XML-Element „Organisation“ verwendet. Es unterstützt den Verfasser, inhaltlich zusammenhängende Bereiche in einen Unterbereich zusammenzufassen (z.B.: Kapitel, Lektionen).
- Ressourcen: Diese XML-Elemente enthalten Verweise auf physikalische Dateien, die den Inhalt des Kurses darstellen.

- Sub-Manifeste: Es besteht auch die Möglichkeit, bereits bestehende Manifeste in das neue Paket mit einzubinden, was natürlich die Wiederverwendbarkeit von Kursmaterial sehr unterstützt.

Der zweite Teil eines Pakets enthält alle physikalischen Dateien, auf die im Ressourcen-Bereich der Manifest-Datei verwiesen wird. Dabei ist der Dateityp völlig irrelevant (Text-, Musik-, Videodateien, Präsentationen), da es hier nur um den strukturellen Aufbau geht. Um die korrekte Anzeige der Dateien hat sich dann die Applikation zu kümmern, mit deren Hilfe dieses Paket angezeigt werden soll.

Ist ein Kurs fertig erstellt, werden alle Dateien - Manifest und physikalische Dateien („Content“) - in eine einzige Datei zusammengefügt, die „Package Interchange File“ genannt wird. Diese Datei wird meist auch komprimiert, um die Weitergabe zu vereinfachen.

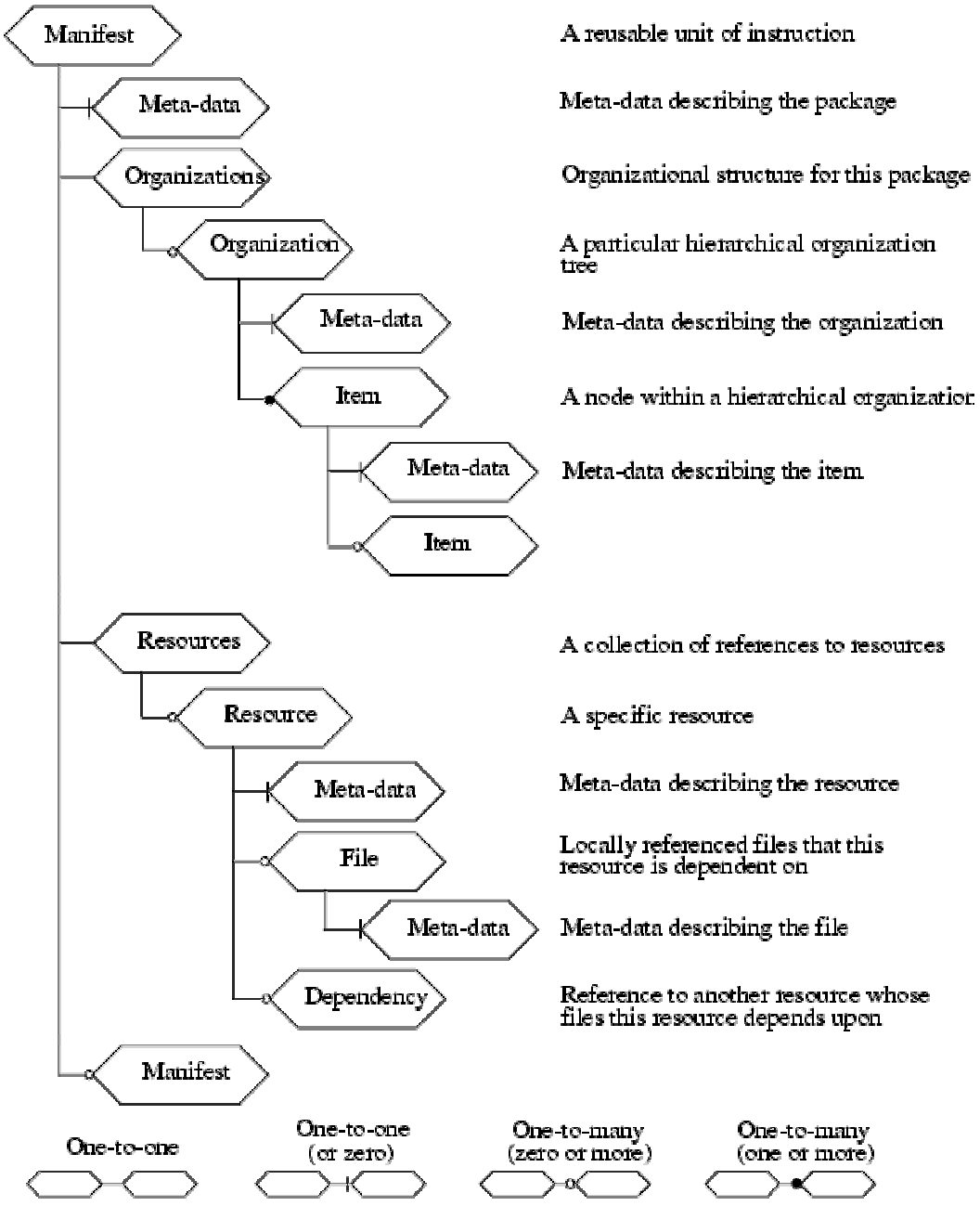


Abbildung 20 - Elemente eines Manifests [IMS01]

4.2. Bedeutung eines Rechtesystems für ein Distanceteaching / Distancecoaching-Framework

Ein Rechtesystem ist ein zentraler Bestandteil eines Distanceteaching / Distancecoaching-Frameworks. Erst dadurch wird es möglich, in seinem Dateisystem verschiedene Sichten auf die Struktur zu gewähren. Dadurch kann man deutlich den Administrator eines Kurses (der Coach) von den Teilnehmern organisatorisch trennen. Der Coach hat alle Möglichkeiten, um die vorhandenen Elemente nach seinem didaktischen Konzept einzusetzen. Die Teilnehmer können nur jene Bereiche des Systems erreichen und verändern, die vom Administrator für sie freigegeben wurden. Dadurch wird die Komplexität des gesamten Systems vor ihnen verborgen und sie können sich voll und ganz auf das Studium der Lehrinhalte konzentrieren.

4.2.1. Anforderungen an ein Rechtesystem in einem Distanceteaching / Distancecoaching-Framework

Das Rechtesystem eines Distanceteaching / Distancecoaching-Frameworks muss natürlich einigen konkreten Anforderungen genügen, die hier kurz erläutert werden.

Der Administrator hat die Möglichkeit, den Benutzern nur jene Teile des Systems zugänglich zu machen, die für sie vorgesehen sind. Es können somit für alle Objekte folgende Berechtigungen vergeben werden: Zugriff erlaubt („Grant“) oder Zugriff verweigert („Deny“). Werden vom Administrator keine Angaben gemacht, befinden sich alle Berechtigungen im Zustand „Not set“. Bei der Überprüfung der Zugriffsberechtigungen wird in diesem Fall der Zugriff ebenfalls verweigert. Dieser Zustand ist nur von Bedeutung, wenn ein Benutzer Mitglied in mehreren Gruppen ist. Wird einer dieser Gruppen der Zugriff auf ein Objekt gewährt, so werden bei der Überprüfung der Zugriffsberechtigungen die übrigen Gruppen, deren Berechtigungen nicht gesetzt sind, ignoriert und der Zugriff gewährt. Sollte in einer Gruppe jedoch eine Verweigerung gesetzt sein, so wird diesem Benutzer der Zugriff auf dieses Objekt nicht gewährt.

Wünschenswert ist es, wenn man diese Berechtigungen nicht für jeden einzelnen Benutzer extra vergeben muss, sondern wenn die Benutzer in Gruppen eingeteilt werden können und man die Vergabe der Berechtigungen über diese Gruppen steuern kann. Die

Berechtigungsvergabe soll jedoch nicht nur über die Gruppen stattfinden, sondern es ist auch wünschenswert, bei jedem einzelnen Mitglied einer Gruppe noch eine individuelle Berechtigung zu setzen. Dies kann zum Beispiel der Fall sein, wenn ein Mitglied einer Gruppe noch zusätzlich Informationen zu einem Thema für die Ausarbeitung einer Arbeit benötigt, die für die anderen Benutzer jedoch nicht von Bedeutung sind. Hier ist es viel benutzerfreundlicher, wenn man diesem einen Benutzer nun die entsprechende Leseberechtigung geben und ihn trotzdem in seiner Gruppe belassen kann.
Mit diesem Aufbau des Rechtesystems ist es nun möglich, dass ein Benutzer mehreren Gruppen zugeteilt ist und sich die Berechtigungen dieser Gruppen gegenseitig widersprechen. Hier ist eine eindeutige Vorgehensweise wünschenswert, um Inkonsistenzen zu vermeiden und einem Benutzer nicht irrtümlich die falschen Berechtigungen zu erteilen. Eine Lösung dazu ist, dass die Verweigerung des Zugriffs mehr Gewicht hat als die Erlaubnis. Zum Beispiel ist Benutzer „Hofer“ Mitglied von Gruppe „Marketing“ und Gruppe „Controlling“, wobei der Gruppe „Marketing“ der Zugriff auf den Ordner „Umsätze“ verweigert ist. Der Gruppe „Controlling“ ist der Zugriff jedoch ausdrücklich erlaubt. Möchte Benutzer „Hofer“ auf den Ordner „Umsätze“ zugreifen, ist ihm das nicht gestattet, da er Mitglied der Gruppe „Marketing“ ist, der der Zugriff auf den Ordner „Umsätze“ ausdrücklich verboten ist. Im Normalfall sind Verweigerungen zu vermeiden, um undurchsichtige Konstrukte und nicht erwünschtes Verhalten bei der Vergabe von Berechtigungen im Rechtesystem zu verhindern. Wäre die Berechtigung für die Gruppe „Marketing“ nicht gesetzt („Not set“), hätten Benutzer, die ausschließlich Mitglieder der Gruppe „Marketing“ sind, ebenfalls keinen Zugriff, Benutzer „Hofer“ jedoch schon, da ihm durch die Mitgliedschaft in Gruppe „Controlling“ der Zugriff gewährt wird.

Jeder Kursteilnehmer soll ein Verzeichnis als persönlichen Ordner zur Verfügung gestellt bekommen, auf das er uneingeschränkt Zugriff hat, also sowohl Schreib-, als auch Lese- und Ausführungsberechtigungen. Hier kann er beispielsweise Arbeitskopien zwischenspeichern. Dieses Verzeichnis, im allgemeinen „Home“-Verzeichnis genannt, wird jedoch in der Größe beschränkt sein müssen, damit nicht zu große Datenmengen auf den Server geladen werden.

4.2.2. Problematik von Lehr-Settings

Durch das Rechtesystem ist es dem Lehrenden möglich, den Ablauf des Kurses nach seinem Konzept zu steuern. Er hat die Möglichkeit, bereits den gesamten Kursinhalt im System zu platzieren, ihn jedoch noch von den Kursteilnehmern zu verbergen. Je weiter der Kurs nun inhaltlich fortschreitet, desto mehr Informationen und Unterlagen werden vom Kursleiter für die Teilnehmer freigegeben. Dadurch können sehr gezielt didaktische Konzepte umgesetzt werden, um den Lernerfolg möglichst groß werden zu lassen.
Durch verschiedene Abstufungen der Einschränkungen können auch Tutoren in das System eingebunden werden. Diese stehen den Kursteilnehmern in den Diskussionsforen Rede und Antwort, sind jedoch nicht berechtigt, am Kursinhalt etwas zu verändern. Ihre Aufgabe besteht jedoch nicht darin, sofort auf alle im Forum aufgeworfenen Fragen zu antworten. Vielmehr sollten sie versuchen, unter den Kursteilnehmern eine Diskussion aufkommen zu lassen, die in der selbständigen Beantwortung der Fragen mündet. Dieser Diskussionsprozess führt zu einer intensiven Verinnerlichung des Stoffes bei den Diskussionsteilnehmern, der den Lernerfolg überaus positiv beeinflusst.
Auch wird durch die gegenseitige Beantwortung der Fragen bzw. durch die Unterstützung durch die Tutoren der Kursleiter entlastet und braucht sich nur mehr mit den auch für die Tutoren zu schweren Fragen befassen bzw. kann sich auf die Erstellung neuer Kursinhalte konzentrieren.

4.3. Das Rechtesystem im virtuellen Dateisystem von WeLearn Release 2

Die Bezeichnung „virtuelles Dateisystem“ wird vor allem im Unix-Bereich verwendet. Damit wird verdeutlicht, dass es für die einzelnen Prozesse keine Rolle spielt, welches konkrete Dateisystem auf dem Speichermedium verwendet wird. Das virtuelle Dateisystem dient als Schnittstelle zwischen den Prozessen und den konkreten Dateisystemen (z. B. reiserFS oder ext2). Dabei verwenden die Prozesse eine standardisierte Schnittstelle, die das virtuelle Dateisystem zur Verfügung stellt. Dieses führt schließlich die Anpassung an das konkrete Dateisystem und damit die Speicherung der Dateien durch.
In WeLearn Release 2 ist das sehr ähnlich. Wurden in Release 1 noch alle Objekte in einer Datenbank abgelegt, so erfolgt die Speicherung in Release 2 auf eine viel flexiblere Art und Weise. Die Speicherung in einer Datenbank setzt voraus, dass eine Datenbank vorhanden ist. Da die Installation und Konfiguration einer Datenbank nicht trivial ist und es auch keine leistungsfähigen, kostenlosen Produkte gibt, die den Anforderungen von WeLearn gerecht werden, wurde in Release 2 das Konzept eines virtuellen Dateisystems aufgegriffen. Dabei werden sämtliche Objekte durch den Serialisierungsmechanismus von Java serialisiert und auf der Festplatte abgespeichert, wobei für jedes Objekt eine eigene Datei mit dem „Object Identifier“ als Namen angelegt wird.
Die Objekte von WeLearn Release 2 sind in einer Baumstruktur angeordnet. Ausgehend von einem Wurzelobjekt werden alle Objekte als Knoten oder Blätter in dieser Baumstruktur abgebildet. Diese Hierarchie findet sich nur im virtuellen Dateisystem wieder, wo sie im File-Manager als Verzeichnisstruktur abgebildet wird. Sämtlichen serialisierten Objekte werden in einem einzigen Ordner auf der Festplatte abgelegt und man kann außerhalb vom System die Hierarchie nicht erkennen.

In der Folge wird konkret die Implementierung des Rechtesystems in WeLearn betrachtet. Dabei achten wir zuerst auf die konzeptionelle Sicht, um sich später mit den technischen Hintergründen näher zu befassen.

4.3.1. Arten von Berechtigungen

Im aktuellen WeLearn-System gibt es sieben verschiedene Arten von Berechtigungen, die gesetzt werden können. Alle Berechtigungen sind in der Klasse „Permission“ implementiert. Diese Klasse enthält nur ein Feld, welches die Art der Berechtigung repräsentiert. Eine Vergleichsmethode ermöglicht es, zwei Berechtigungen miteinander zu vergleichen und so die Art festzustellen. Es ist ohne weiteres möglich, neue Arten von Berechtigungen zu erzeugen.

4.3.1.1.Die Berechtigung „Sichtbar“ (VISIBLE)

Mit dieser Berechtigung ist es dem Benutzer möglich, die Datei in der Auflistung des Dateimanagers zu sehen. Hat er dieses Recht nicht, haben auch die übrigen Berechtigungen keinen Sinn. Denn es ist unmöglich Aktionen auf Dateien auszuführen, wenn es nicht einmal gestattet ist, sie zu sehen.

4.3.1.2.Die Berechtigung „Lesen“ (READ)

Ist diese Berechtigung gesetzt, so ist es dem Benutzer erlaubt, das Objekt als solches zu lesen. Das heißt, er kann das Objekt zwar markieren, jedoch nicht ansehen. Daher ist es dem Benutzer auch gestattet, eine Kopie dieses Objektes zu erstellen. Diese Kopie erbt die Berechtigungen des übergeordneten Ordners im virtuellen Dateisystem, wo es eingefügt wurde, und hat nicht die ACL des Originalobjektes. Die Leseberechtigung sagt nichts darüber aus, ob man auch den Inhalt einer Datei betrachten darf. Dies wird durch die Ausführberechtigung geregelt.

4.3.1.3.Die Berechtigung „Schreiben“ (WRITE)

Diese Berechtigung erlaubt es dem Benutzer, verändernd auf das Objekt einzuwirken. Damit ist jedoch nicht gemeint, dass der Inhalt des Objektes editiert werden kann, sondern es ist möglich, die Eigenschaften (Name, angezeigtes Icon) zu verändern oder das Objekt zu löschen. Ist der Schreibzugriff verweigert, kann die Datei auch nicht gelöscht werden. Die Änderung der Zugriffsberechtigungen ist mit dieser Berechtigung jedoch auch nicht möglich (Berechtigung „Rechte ändern“).

4.3.1.4.Die Berechtigung „Ausführen“ (EXECUTE)

Diese Berechtigung hat im WeLearn-System einen besonderen Stellenwert, denn sie hat für jedes Objekt eine andere Bedeutung. Hat ein Benutzer diese Berechtigung, dann wird ihm der Dateiname im Dateimanager als Link dargestellt und er ist in der Lage, diese Datei anzuklicken und sie damit auszuführen. Dieses Ausführen schaut nun je nach Dateityp anders aus. Ein Ordner beispielsweise hat das Auflisten seines Inhaltes als Ausführoperation. Klickt man also auf einen Ordner, geht man in der Dateihierarchie eine Stufe tiefer und die Liste der Dateien, die im Ordner enthalten sind, wird angezeigt. Eine Textdatei wird so wie die meisten üblichen Dateiformate beim Ausführen geöffnet, sofern der entsprechende Viewer auf dem Rechner verfügbar ist. Ist dies nicht der Fall, wird die Datei zum Download angeboten.

4.3.1.5.Die Berechtigung „Rechte ändern“ (CHANGE_RIGHTS)

Wie der Name schon sagt, gestattet es diese Berechtigung, die Berechtigungen der Benutzer für eine Datei oder ein Verzeichnis zu ändern.

4.3.1.6.Die Berechtigung „Editieren“ (EDIT)

Diese Berechtigung hat - genauso wie die nachfolgende - einen Sonderstatus, da sie nur in Diskussionsforen in Verwendung ist. Die Edit-Berechtigung erlaubt es, einen Foreneintrag nachträglich zu ändern. Diese Anforderung wurde bei der Implementierung gewünscht, um etwaige Tippfehler noch korrigieren zu können. Ob das Ändern eines Foreneintrags im Nachhinein überhaupt möglich sein sollte, sei dahingestellt.

4.3.1.7.Die Berechtigung „Anhängen“ (ATTACH)

Auch diese Berechtigung findet nur in den Diskussionsforen ihre Anwendung. Ist dieses Recht gesetzt, so kann man an einen Beitrag im Forum eine Datei (Attachment) anhängen. Diese Anforderung wurde ebenfalls während der Implementierung hinzugefügt.

4.3.2. Verwaltung der Rechte

Die Implementierung des Rechtesystems in WeLearn Release 2 ist ähnlich der in Microsoft Windows XP, jedoch bei weitem einfacher. Jedes Objekt verwaltet seine Berechtigungen in einer eigenen Liste (*Access Control List* (ACL) genannt). Dadurch kann

beim Zugriff auf ein Objekt jederzeit festgestellt werden, ob der zugreifende Benutzer die Berechtigung besitzt oder nicht.

Beim Anlegen eines Objektes wird zuerst die „Access Control List“ des im virtuellen Dateisystem übergeordneten Objektes geerbt. Es ist somit auch nicht möglich, die Berechtigungen auf dieses Objekt zu verändern. Es muss zuerst eine eigene ACL angelegt werden, um dem Objekt spezielle Berechtigungen zu geben. Für jeden Benutzer, der eine spezielle Berechtigung auf das Objekt besitzt, wird nun ein Eintrag in der Liste erzeugt. Diese Operationen werden in einem speziellen Dialog durchgeführt, dem Rechte-Dialog, der vom File-Manager aus aufgerufen werden kann, sofern dem Benutzer für den Zugriff auf das Objekt die Berechtigung „ChangeRights“ gegeben wurde.

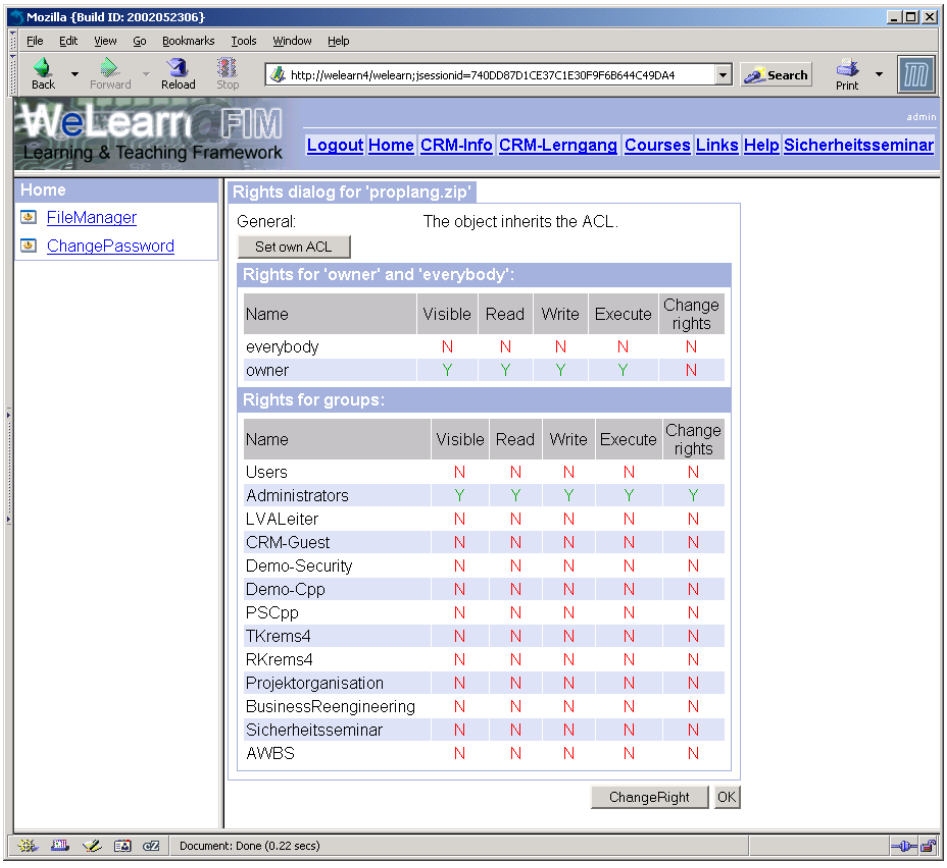


Abbildung 21 - Der Rechte-Dialog

Wurde für ein Objekt eine eigene ACL erzeugt, so werden die Berechtigungen, die vorher vom übergeordneten Objekt geerbt wurden, kopiert und in die eigene ACL eingetragen. Anschließend können eigene Berechtigungen für dieses Objekt vergeben werden, die wiederum an untergeordnete Objekte weitervererbt werden können.

Nehmen wir ein Objekt an, für das eine eigene ACL erzeugt wurde. Eine Berechtigung für dieses Objekt kann einen von drei Stati annehmen: nicht gesetzt („Not set“), erlaubt („Grant“) oder verweigert („Deny“). Wurde für eine Gruppe oder einen Benutzer noch keine Berechtigung gesetzt (die Berechtigung für diese Gruppe oder diesen Benutzer ist im Status „Not set“), so wurde noch keine ACE erzeugt und der Zugriff auf das entsprechende Objekt wird für diese Gruppe oder diesen Benutzer verweigert. Ein Mitglied einer Gruppe hat nur Zugriff auf ein Objekt, wenn dieser Gruppe ausdrücklich der Zugriff auf dieses Objekt erlaubt wurde, sofern das Objekt eine eigene ACL hat.
Um einer Gruppe oder einem Benutzer den Zugriff auf ein Objekt zu gewähren, muss der Status der gewünschten Berechtigung auf „Grant“ gesetzt werden.
In seltenen Fällen wird die Berechtigung auf „Deny“ gesetzt. Damit ist der Zugriff auf das entsprechende Objekt für die Gruppe oder den Benutzer verweigert. Befindet sich ein Benutzer in mehreren Gruppen, kann es zu unerwünschten Effekten kommen, die schon weiter oben beschrieben wurden.

Sämtliche Objekte im WeLearn System stehen zueinander in einer hierarchischen Beziehung. Ausgehend von einem Wurzelobjekt wird eine Baumstruktur aufgebaut, die alle Objekte des virtuellen Dateisystems enthält. Da beim Start des Systems noch keine Authentifizierung erfolgt ist, kann das System noch nicht auf einzelne Benutzerberechtigungen reagieren. Daher wurde eine spezielle Gruppe eingeführt, in der alle Benutzer des Systems standardmäßig Mitglied sind (die Gruppe „Users“). Dieser Gruppe ist der Zugriff auf die grundlegenden Objekte des Systems - wie zum Beispiel das Login-Modul - gestattet. Damit ist es möglich, auch die Authentifizierung innerhalb des WeLearn-Frameworks durchzuführen. Sämtliche Objekte, auf die die Gruppe „Users“ eine Zugriffsgenehmigung hat, können von den Benutzern je nach erteilter Berechtigung verwendet werden. Durch eine Verweigerung des Zugriffs auf ein Objekt ist allen angelegten Benutzern der Zugriff auf dieses Objekt verwehrt. Daher sollte man bei der Veränderung der Berechtigungen der Gruppe „Users“ besondere Vorsicht walten lassen.

Grundsätzlich kann das Rechtesystem von WeLearn Release 2 so adaptiert werden, dass für jeden Benutzer Rechte auf ein Objekt gesetzt werden können. Um jedoch die Administrierbarkeit und die Übersichtlichkeit zu fördern, können die Benutzer auch in Gruppen eingeteilt werden, denen für alle ihre Mitglieder Berechtigungen vergeben werden

können. In einem durchschnittlichen Anwendungsbereich dieses Distanceteaching / Distancecoaching-Systems werden durchaus einige hundert Benutzer angelegt und so ist es ein großer Vorteil, diese Möglichkeit der vereinfachten Rechteverwaltung nützen zu können. Daher wurde in der aktuellen Version darauf verzichtet, einem einzelnen Benutzer eigene Berechtigungen vergeben zu können. Dies vereinfacht die Administration erheblich, schränkt deswegen jedoch die Erzeugung komplexer Berechtigungskonstrukte nicht ein. Denn es ist möglich, Gruppen mit nur einem Benutzer zu erzeugen und diese mit den gewünschten Berechtigungen auszustatten. Damit ist derselbe Effekt erreicht und nur dieser eine Benutzer hat die entsprechenden Berechtigungen.
Um die Berechtigungen nur für einen Benutzer vergeben zu können, ohne den Schritt über eine eigene Gruppe gehen zu müssen, wäre eine Erweiterung der Klasse *RightsView.java* notwendig, da dieser Dialog zur Zeit ausschließlich die Verwaltung von Gruppenberechtigungen unterstützt. Nur die Berechtigungen der Benutzer „Jeder“ und „Besitzer“ können individuell gesetzt werden.
In der Implementierung wurde folgendes Konzept übernommen: Jedes Objekt enthält selbst die Informationen, wer Berechtigungen zum Zugriff hat. Das heißt, jedem Objekt wird eine Liste mit den Gruppen zugewiesen, denen vorher Berechtigungen erteilt wurden. Somit entscheidet jedes Objekt selbst, ob ein Zugriff erlaubt ist oder nicht.
Diese Liste wird jedoch nicht im Objekt selbst abgespeichert. Sie wird in einem separaten Ordner abgelegt. Jedes Objekt erhält den Verweis auf seine gespeicherte Liste der Berechtigungen (ACL).

4.3.3. Die technische Umsetzung

4.3.3.1. Der Rechte-Manager (RightsManager.java)

Diese Klasse stellt den allgemeinen Anlaufpunkt für alle Anfragen bezüglich der Rechtvergabe und Rechtprüfung dar. Von dieser Klasse darf es selbstverständlich nur ein Exemplar geben, welches alle Anfragen entgegennehmen muss. Diese Forderung wird durch die statische Methode *instance()* erreicht, die bei ihrem Aufruf entweder das bereits existierende oder ein neu angelegtes Exemplar des Rechte-Managers zurückliefert, jedoch auf jeden Fall kein zweites Exemplar erzeugt. Im ganzen System wird nur über diese Methode auf das Exemplar des Rechte-Managers zugegriffen, was eine mehrmalige Erzeugung unmöglich macht.
Zwei Bereiche werden von dieser Klasse abgedeckt:

1. Sie stellt Methoden zur Verfügung, um die Rechte, die ein Benutzer auf ein Objekt hat, zu managen. Es gibt also Methoden, die eine Berechtigung oder eine Verweigerung für einen Benutzer auf ein bestimmtes Objekt setzen (*setPermission(...)*) oder auch solche, die Berechtigungen oder Verweigerungen wieder löschen (*removePermission(...)*).
2. Sie bietet Methoden an, die es dem System erlauben festzustellen, ob eine Anforderung an ein Objekt – sei es nun eine Lese-, eine Schreib- oder eine Ausführen-Anforderung – erlaubt ist oder ob der Zugriff verweigert werden soll.

Im Hinblick auf die mögliche Erweiterung des Frameworks gibt es eine Methode, die als Parameter die Berechtigung übergeben bekommt, die zu überprüfen ist. Diese Methode (*checkPermission(...)*) stellt somit eine Vorlage zur Verfügung, die bei zukünftigen Erweiterungen die Prüfung der Rechte übernehmen kann. Dadurch können jederzeit neue Berechtigungen implementiert werden, die durch diesen Mechanismus überprüft und ohne Probleme in das Framework integriert werden können.

Aufgrund der Struktur von WeLearn war es erforderlich, von dieser Methode drei Versionen zur Verfügung zu stellen, die unterschiedliche Parameterlisten haben. Einer dieser Parameter muss das Objekt sein, das überprüft werden soll. Dieser ist in jeder Methode enthalten. Der zweite Parameter kann nun folgende drei Objekte enthalten:

- Person: Die einfachste Variante ist natürlich, dass ein Benutzer auf ein Objekt zugreifen möchte. Daher wird in dieser Variante die Berechtigung für eine Person überprüft.
- ObjectIdentifier: Vielfach werden die Objekte – auch Personen sind Objekte im WeLearn-System – über ihre eindeutige Kennung – den ObjectIdentifier – angesteuert. Daher war es notwendig, auch die Möglichkeit zu bieten, über diesen Parameter die Berechtigungsprüfung durchzuführen.
- SystemObject: Diese Möglichkeit wurde noch zusätzlich geboten, da an mehreren Stellen im Quellcode die Überprüfung der Berechtigungen durch die Übergabe des Systemobjekts zweckmäßiger schien. Natürlich gab es auch hier die Überlegung, diese Option aus dem Quellcode zu entfernen. Mit zunehmender Komplexität des Projektes entschieden wir uns dennoch, diese Möglichkeit auch weiterhin zur Verfügung zu stellen.

Als dritter Parameter wird die Berechtigung („permission“) übergeben, die durch den Aufruf der Methode überprüft werden soll.

4.3.3.2. Die Berechtigungen (Permission.java)

Diese Klasse legt fest, welche Berechtigungen in WeLearn Release 2 zur Verfügung stehen. Hier werden die möglichen Berechtigungen vordefiniert und je nach Bedarf ein Objekt mit dem entsprechenden Wert erzeugt. Dieser wird in eine Zustandsvariable gespeichert, um bei einer Überprüfung das entsprechende Recht abfragen und vergleichen zu können. Hier sind die sieben Berechtigungstypen eingerichtet, die zur Zeit in WeLearn verfügbar sind.

4.3.3.3. Access Control List (Acl.java)

WeLearn Release 2 implementiert das Rechtesystem auf der Basis von Access Control Listen (ACL). Dabei wird jedem Objekt, das im System erzeugt wird, eine ACL zugewiesen. Zu Beginn ist diese noch vom Vaterobjekt geerbt, das heißt, man kann die Berechtigungen auf dieses Objekt vorerst nicht ändern. Es ist notwendig, für jedes Objekt eine eigene ACL zu erzeugen, wenn es individuell Berechtigungen erhalten soll. Darin sind nun Access Control Entries (ACE) enthalten. Jeder Benutzer, der eine spezielle Berechtigung auf ein Objekt hat, erzeugt einen ACE in der ACL des betreffenden Objektes.

Derzeit sind nur einige Berechtigungen implementiert. Diese bilden die Grundlage für ein lauffähiges System. Durch den objektorientierten Programmieransatz ist es jederzeit möglich, das vorhandene System durch eigene Berechtigungen zu erweitern, wie dies anhand der „Edit“- und „Attach“-Berechtigung in Foren bereits geschehen ist.

Alle Einträge einer ACL - die „Access Control Entries“ - werden in einer ArrayList gespeichert, die bei der Erzeugung einer ACL angelegt wird. Natürlich existieren die verschiedensten Zugriffsmethoden auf diese ACL. Ein neuer Eintrag wird mit der Methode *addAce(Ace newAce)* hinzugefügt. Dabei wird zuerst überprüft, ob für den Benutzer, für den ein neuer Eintrag erzeugt werden soll, bereits ein Eintrag in der ACL existiert. Ist dies der Fall, so kann kein neuer Eintrag hinzugefügt werden, sondern der bereits existierende ist zu manipulieren. Bei dieser Aktion ist es auch erforderlich, dass rund um den ganzen Ablauf eine Transaktion geöffnet wird, um gleichzeitige Zugriffe auf die ACL und somit mögliche Inkonsistenzen zu vermeiden. Der Persistenz-Manager, der für diese Aktion verantwortlich ist, muss die ACL für das Objekt also sperren.

Um beim Zugriff auf ein Objekt den entsprechenden Eintrag für den zugreifenden Benutzer laden zu können, sind Zugriffsmethoden erforderlich, die einen Eintrag aus der Liste auslesen. Dies sind die Methoden mit den Namen *getAce(...)*, die in zwei unterschiedlichen Parameterlisten implementiert sind. Die erste Variante dieser Methode benötigt zwei Parameter: einerseits die Objekt-ID des Benutzers, der auf das Objekt zugreifen möchte; andererseits einen Boolean-Wert, der bestimmt, ob er die Verweigerungen oder die Genehmigungen laden will. Wird dieser Wert auf „Wahr" gesetzt, so werden die Genehmigungen übergeben, bei „Falsch" die Verweigerungen. Es wird also immer nur eine der beiden gespeicherten Einträge zurückgegeben, je nach dem, nach welchem gefragt wurde. Die zweite Variante der Methode ist ein Spezialfall der ersten, denn es wird nur die Objekt-ID des Benutzers übergeben. Intern wird die erste Variante aufgerufen, wobei der zweite Parameter auf „Wahr" gesetzt wird und somit die Genehmigungen abgefragt werden.

Natürlich muss es auch möglich sein, einen ACE wieder zu entfernen. Dafür wurde die Methode *removeAce(...)* entwickelt, von der es wiederum zwei Varianten gibt. Einerseits können mit *removeAce(ObjectIdentifier holder)* gleich beide Einträge eines Benutzers in einer ACL entfernt werden (Genehmigungen und Verweigerungen), andererseits kann man auch als zweiten Parameter angeben, welchen der beiden Einträge man entfernen möchte (*removeAce(ObjectIdentifier holder, boolean positive)*). Die erste Variante stützt sich auf die zweite, da mit zweimaligem Aufruf der zweiten Variante beide Einträge gelöscht werden können (zweiter Parameter einmal „Wahr", einmal „Falsch").

Eine weitere Methode, die in der Klasse *Acl* implementiert wurde, ist *cleanUp()*. Diese Methode sucht jene Einträge aus der Liste, bei denen die Liste der Berechtigungen oder Verweigerungen leer ist. Dies bedeutet, dass für den Benutzer, dem dieser ACE zugeordnet ist, keine Berechtigungen oder Verweigerungen mehr existieren und somit dieser Eintrag der Liste ohne Komplikationen gelöscht werden kann. Um Speicherplatz zu sparen, wird dies in weiterer Folge auch gemacht.

4.3.3.4.Access Control Entry (Ace.java)

Wie in Abbildung 22 gezeigt, muss ein Eintrag in der ACL einige Informationen gespeichert haben, um ihn eindeutig identifizieren zu können.

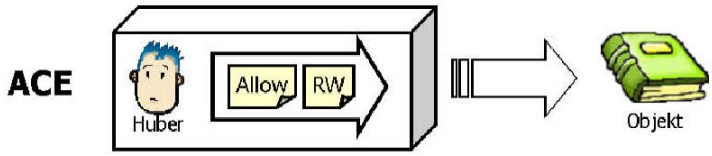


Abbildung 22 - Access Control Entry

Natürlich muss der Benutzer im Eintrag abgespeichert werden, da dieser bei der Suche nach den Berechtigungen eines Benutzers auf ein Objekt den Schlüssel darstellt (*ObjectIdentifier holder*). Werden die Berechtigungen eines Benutzers überprüft, weil er einen Zugriff auf ein Objekt durchführen will, wird in der ACL des Objektes nach dem entsprechenden Benutzer gesucht. Ist ein ACE mit der entsprechenden Kennung vorhanden, so wird weiters in der Liste nach der geforderten Berechtigung gesucht. Daraus folgt, dass auch eine Liste mit den für einen Benutzer festgelegten Genehmigungen oder Verweigerungen existieren muss (*List permissions*). In der Rechteverwaltung von WeLearn Release 2 ist vorgesehen, dass für die Genehmigungen und für die Verweigerungen je ein eigener Eintrag in der ACL erstellt wird. Daher ist noch ein weiteres Feld vom Typ Boolean im ACE enthalten, das festlegt, ob der Eintrag die Verweigerungen oder die Genehmigungen eines Benutzers auf das entsprechende Objekt enthält (*boolean positive*).

Die Klasse *Ace* stellt zwei Konstruktoren zur Verfügung. Mit dem ersten ist es möglich, durch Übergabe des Benutzers als Parameter einen ACE für die Genehmigungen zu erzeugen. Bei der zweiten ist ein zweiter Parameter erforderlich, der ein Wert vom Typ Boolean ist, um festzulegen, ob dieser ACE Genehmigungen oder Verweigerungen enthalten soll.

Neben den Methoden zum Auslesen und Setzen der Felder des Objektes gibt es einige Methoden, um Berechtigungen in einem ACE einzufügen bzw. solche zu entfernen. Zum Hinzufügen einer Berechtigung zu einem ACE wird die Methode *addPermission(Permission p)* verwendet, die die entsprechende Berechtigung in den ACE einträgt, sofern eine solche noch nicht vorhanden ist. Durch *removePermission(Permission p)* wird eine Berechtigung entfernt. Mit der Methode *isEmpty()* mit einem Rückgabewert vom Typ Boolean kann überprüft werden, ob noch Berechtigungen im ACE enthalten sind. Diese Methode wird von der Klasse *Acl* verwendet, um festzustellen, ob leere Einträge vorhanden sind, die gelöscht werden können.

Zur Überprüfung einer Berechtigung für einen Benutzer wird die Methode *checkPermission(Permission p)* verwendet, die ebenfalls einen Wert vom Typ Boolean retourniert.
Eine Methode, die bei einer Anfrage für die Unterscheidung von Verweigerungen und Genehmigungen verwendet wird, ist *isPositive()*. Diese liefert „Wahr“ zurück, wenn der ACE Genehmigungen enthält, „Falsch“ bei Verweigerungen.
Weitere Methoden oder Felder sind für die Klasse *Ace* nicht erforderlich.

4.3.3.5.Rechte ändern (ChangeRight.java)

Sollen Berechtigungen auf ein Objekt verändert werden, so führt die konkrete Veränderung in der ACL die Klasse „ChangeRight“ aus. Diese Veränderung kann einerseits durch das graphische Benutzerinterface durchgeführt werden. Andererseits kann die Klasse „ChangeRight“ auch über den „Command Line Interpreter“ (CLI) direkt als Kommando verwendet werden. Dieser CLI ist vergleichbar mit der Eingabeaufforderung in Windows. Einfache Kommandos können über dieses Interface abgesetzt werden.
Bei Veränderungen an einer ACL ist vom System immer darauf zu achten, dass diese Operation in einem nach außen hin abgeschlossen Bereich ausgeführt wird. Dazu wird eine eigene Transaktion gestartet, die die zu verändernde ACL sperrt und somit einen parallelen Zugriffsversuch verhindert.

4.4. Vergleich zum Rechtesystem von Release 1

Das Rechtesystem von WeLearn Release 1 geht einen völlig anderen Weg. Alle Objekte werden in einer Datenbank abgelegt. Genauso verhält es sich auch mit den Berechtigungen. Es werden also nicht die einzelnen Objekte direkt abgelegt, wie das in WeLearn Release 2 der Fall ist, sondern die Speicherung wird von einer Datenbank übernommen. Hier stellten sich bei der Implementierung einige Schwierigkeiten in Bezug auf die Antwortzeiten ein. Bei der Anzeige einer Seite sind meist mehrere Datenbankabfragen notwendig (teilweise bis zu 40), bis alle notwendigen Berechtigungen festgestellt sind. Dies führte zu erheblichen Verzögerungen und beeinträchtigte die Benutzerfreundlichkeit. Um dieses Problem in den Griff zu bekommen, wurde ein Cache entwickelt, der die schon einmal verwendeten Berechtigungen im Speicher hält und somit den Zugriff erheblich beschleunigt. Ein „Rightsvalidator“ steuert die Anfragen an diesen

Cache und lenkt sie zur Datenbank um, falls die benötigte Berechtigung nicht im Cache zwischengespeichert ist. Die Entwicklung dieses Cache war von großen Schwierigkeiten bezüglich Funktionalität und Performanz begleitet und bedurfte mehrere Entwürfe, bis schließlich eine lauffähige und performante Version fertig gestellt werden konnte.
Die in WeLearn Release 1 verfügbaren Berechtigungen sind:

- Visible (Sichtbar)
- Readable (Lesbar)
- Writeable (Schreibbar)
- Executeable (Ausführbar)

Für jede dieser Berechtigungen wird gespeichert, ob sie erlaubt (JA), verweigert (NEIN) oder vom übergeordneten Objekt geerbt (Erben) ist. Andere Möglichkeiten oder Berechtigungen wurden nicht implementiert.

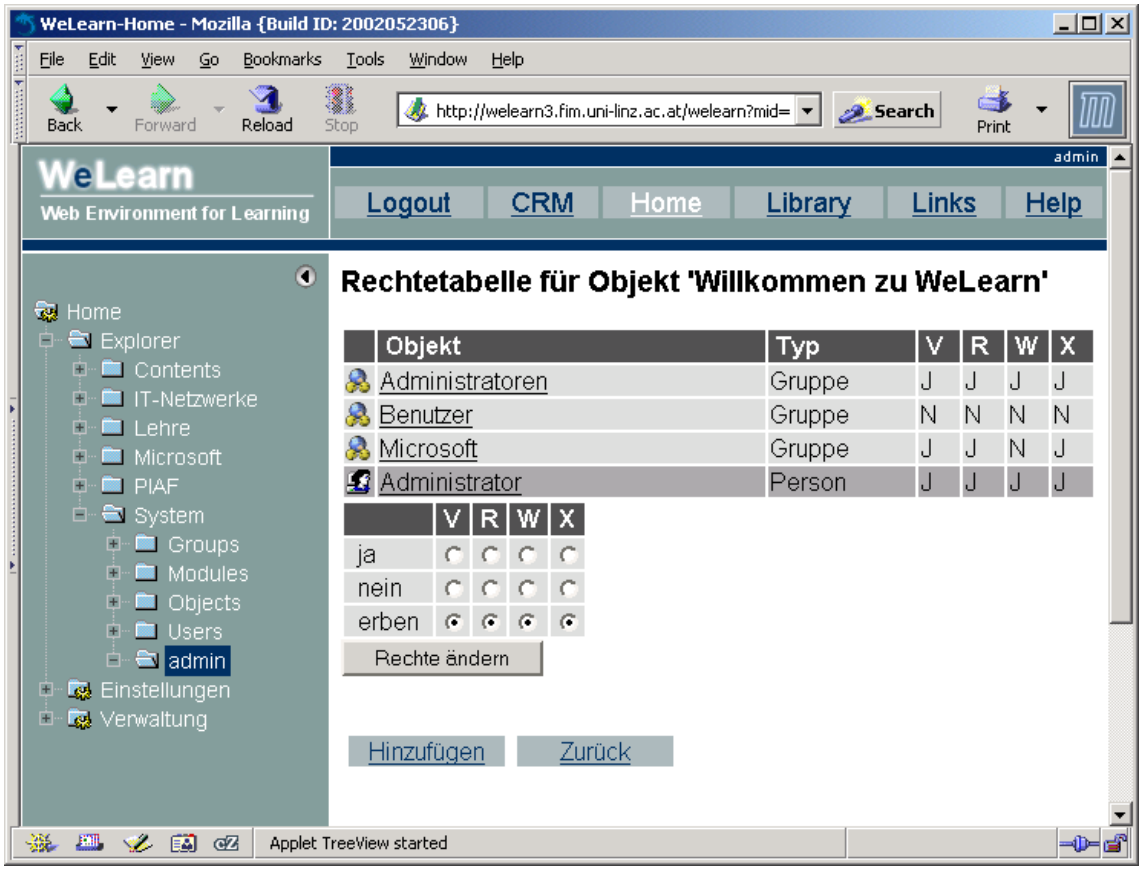


Abbildung 23 - Rechtetabelle von WeLearn Release 1

Bei WeLearn Release 1 existiert die Möglichkeit, Benutzer in Gruppen einzuteilen und die Berechtigungen dann auf diese Gruppen festzulegen. Diese Vorgehensweise bringt eine deutliche Vereinfachung der Administration mit sich, da auf ein Telecoaching-Framework oft hundert und mehr Benutzer Zugriff haben. Die Strategie, wie mit widersprüchlichen Berechtigungen eines Benutzers durch mehrfache Gruppenzugehörigkeit umgegangen wird, ist jedoch eine völlig andere. In Release 2 haben - ebenso wie in Windows XP - die Verweigerungen mehr Gewicht als die Genehmigungen. Ist ein Benutzer Mitglied mehrerer Gruppen, wobei die eine Gruppe eine Berechtigung hat, auf einen bestimmten Ordner zuzugreifen, einer anderen Gruppe der Zugriff jedoch verweigert ist, so hat der Benutzer keinen Zugriff auf diesen Ordner. Release 1 geht in diesem Fall anders vor: Hier zählt die Genehmigung mehr. Der Benutzer hat in Release 1 Zugriff auf einen Ordner, auch wenn ihm durch eine zweite Gruppenmitgliedschaft dieser verweigert würde. Durch diese Vorgehensweise ist in Release 1 der Administrator viel mehr gefordert, die Vergabe der Berechtigungen genau durchzuführen. Natürlich muss auch ein Administrator von WeLearn Release 2 sehr sorgsam bei der Vergabe der Berechtigungen umgehen, hat jedoch noch die Sicherheit, dass ein Benutzer auf eine Ressource nicht zugreifen darf, wenn ihm das irrtümlich durch eine weitere Gruppenmitgliedschaft erlaubt würde (sofern ihm der Zugriff durch eine andere Gruppenmitgliedschaft bereits explizit verweigert wird.). Dieser Mechanismus ist ein zusätzlicher Sicherheitsaspekt, der bei der Administration des WeLearn Release 2 Systems behilflich ist.

Ein weiterer Unterschied zu Release 1 ist die Vergabe der Berechtigungen auf einzelne Benutzer, die in Release 2 nicht möglich ist. In Release 1 ist dies durchaus möglich. Dabei wiegen die Berechtigungen, die ein einzelner Benutzer auf ein Objekt bekommt, mehr als die Gruppenberechtigungen, die sich aus der Mitgliedschaft des Benutzers in einer Gruppe ergeben. Diese Einschränkung kann jedoch in Release 2 umgangen werden, indem für diesen einen Benutzer eine eigene Gruppe angelegt wird und diese die entsprechenden Berechtigungen erhält.

Release 1 lehnt sich bei der Einteilung der Berechtigungen in „Benutzer“, „Gruppen“ und „Jeder“ an die Vorgehensweise von Unix an, wo diese Einteilung auch so ähnlich vorgenommen wird. Anstatt des „Benutzers“ gibt es in Unix den „Besitzer“, was wiederum in WeLearn Release 2 ebenso implementiert wurde. Auch hier gibt es den Status „Besitzer“, der spezielle Berechtigungen auf ein Objekt haben kann.

4.5. Vergleich zu anderen Rechtesystemen

Das Rechtesystem von WeLearn Release 2 folgt der Strategie von Windows XP und NTFS. Jedes Objekt und jeder Benutzer ist durch eine eindeutige Kennung identifiziert. Nur über diesen „Object Identifier“ ist ein Zugriff möglich. Dieser OID wird auch zur Verwaltung der Zugriffsberechtigungen verwendet. Jedes Objekt im virtuellen Dateisystem in WeLearn Release 2 erhält eine eigene ACL, in der die Berechtigungen für die Benutzer in ACEs abgelegt werden. Dabei ist es in WeLearn jedoch nicht so, dass für die Genehmigungen und die Verweigerungen dieselbe ACL verwendet wird, sondern es wird für beide Arten der Berechtigung eine eigene ACL für jedes Objekt angelegt. Dies ist ein Unterschied zu NTFS, der im Ergebnis jedoch keine sehr große Bedeutung hat. Dadurch erspart man sich die Sortierung der ACEs in der ACL, muss aber bei einer Abfrage der Berechtigungen immer zwei Methodenaufrufe durchführen, nämlich einen für die Genehmigungs- und einen für die Verweigerungs-ACL.

Eine ACL wird in WeLearn Release 2 nicht direkt beim zugehörigen Objekt abgelegt, sondern es gibt einen eigenen Ordner dafür. Dort werden die ACLs aller Objekte abgelegt. Beim Objekt selbst wird nur ein Verweis auf die beiden zugehörigen Listen im Rechte-Ordner gespeichert. Dieser Ordner ist genauso wie jeder andere Ordner Teil des virtuellen Dateisystems von WeLearn Release 2, ist jedoch dem Benutzer verborgen und kann nicht geöffnet werden, da eine direkte Manipulation an den ACLs nicht erlaubt ist.

Eine weitere Vereinfachung im Vergleich zu Windows XP, die bei der Entwicklung von WeLearn Release 2 vorgenommen wurde, ist die Einschränkung auf Gruppen. Eine Adaptierung in diese Richtung ist jedoch ohne weiteres möglich und bedarf nur eines geringen Entwicklungsaufwandes.

Ein weiterer Punkt, der auch in Microsofts Betriebssystemen Anwendung findet, ist der Vorrang der Verweigerung vor der Genehmigung. Diese Vorgehensweise ist die einzige, die im Fall der Zugriffsberechtigungen auf Dateien die erforderliche Sicherheit gewährleistet. Sonstige Strategien bergen die Gefahr, dass bei der Manipulation der Berechtigungen Fehler auftreten, die einen ungerechtfertigten Zugriff erlauben können.

Mit den Implementierungen der Rechtesysteme von Unix und Mac OS X bestehen kaum Gemeinsamkeiten. Lediglich die Einteilung in „Besitzer“, „Gruppen“ und „sons-

tige Benutzer“ ist von Unix übernommen. Ansonsten verwendet WeLearn Release 2 eine völlig andere Strategie, um die Berechtigungen der Benutzer einzuschränken.

4.6. Codefragmente

4.6.1. RightsManager.java

```
/*
 * RightsManager.java
 *
 * @author Richard Leitner
 *
 * This class does the checking of all permissions. It also offers the
 * only instance of the RightsManager class.
 */

package at.jku.fim.welearn.system.rights;

import at.jku.fim.welearn.system.object.*;
import at.jku.fim.welearn.reference.collections.*;
import at.jku.fim.welearn.persistence.PersistenceException;
import at.jku.fim.welearn.objects.user.Person;
import at.jku.fim.welearn.persistence.PersistenceManager;
import at.jku.fim.welearn.objects.HardLink;

public class RightsManager {

    /** The only instance of RightsManager. */
    private static RightsManager _instance = null;

    /** Constructor which creates a new instance of RightsManager */
    private RightsManager() {
    }

    /**
     * Returns the only instance of RightsManager.
     * If the instance doesn't exist yet, it is created within this
     * method.
     * @return The instance of RightsManager.
     */
    public static RightsManager instance() {
        if ( _instance == null ) {
            _instance = new RightsManager();
        }
        return _instance;
    }

    /**
     * This method checks whether the SystemObject who has the
     * specified permission on the SystemObject what.
     * @param who The SystemObject to test the permission with
     * @param what The SystemObject to test the permission on
     * @return True if the SystemObject who has the specified
     * permission to the SystemObject what.
     */
    public boolean checkPermission(ObjectIdentifier whoId,
                   SystemObject what, Permission permission)
                   throws PersistenceException
    {
        PersistenceManager pm = PersistenceManager.instance();
        SystemObject who = pm.load( whoId );
        if( who instanceof Person ) {
```

```
            return checkPermission( (Person) who, what, permission );
        } else {
            return checkPermission( who, what, permission );
        }
    }

    /**
     * This method checks if the SystemObject who has the specified
     * permission on the SystemObject what.
     * @param who The SystemObject for that the permission should be
     * tested.
     * @param what The SystemObject to test the permission on.
     * @return True if the SystemObject who has specified permission
     * to the SystemObject what.
     */
    public boolean checkPermission(SystemObject who,
            SystemObject what, Permission permission)
        throws PersistenceException
    {
        // special behaviour for HardLinks
        if( what instanceof HardLink ) {
            SystemObject obj = ((HardLink) what).getObject();
            if( obj != null ) {
                 boolean access = checkPermission( who, obj,
                                                   permission );
                 if( !access ) {
                    return false;
                 }
            }
        }

        Acl acl = what.getAcl();

        if( acl == null ) {
            return false;
        }

        Ace aceGrant = acl.getAce(who.getOid(), true);
        Ace aceDeny = acl.getAce(who.getOid(), false);

        if( (aceDeny != null) && aceDeny.checkPermission(permission) )
        {
            return false;
        }
        if((aceGrant != null) && aceGrant.checkPermission(permission))
        {
            return true;
        }

        return false;
    }

    /**
     * This Method checks if the Person who has the specified
     * permission on the SystemObject what.
     * @param who The Person to test the permission for.
     * @param what The SystemObject to test the permission on.
     * @return True if the Person who has specified permission
     * to the SystemObject what.
     */
```

```
public boolean checkPermission(Person who, SystemObject what,
                               Permission permission)
throws PersistenceException
{
    // special behaviour for HardLinks
    if( what instanceof HardLink ) {
        SystemObject obj = ((HardLink) what).getObject();
        if( obj != null ) {
             boolean access = checkPermission( who, obj,
                                               permission );
             if( !access ) {
                return false;
             }
        }
    }

    Acl acl = what.getAcl();

    if( acl == null ) {
        return false;
    }

    Ace aceGrant, aceDeny;
    SystemObject owner = what.getOwner();
    if ( owner != null ) {
        if( who.getOid().equals( owner.getOid() ) ) {
            /**
             * check the permissions of the owner
             * The owner´s rights are the strongest ones.
             */
            aceGrant = acl.getAce(ObjectIdentifier.USER_OWNER,
                                  true);
            aceDeny = acl.getAce(ObjectIdentifier.USER_OWNER,
                                 false);

            // check denies
            if ( (aceDeny != null) &&
                  aceDeny.checkPermission(permission) )
            {
                return false;
            }
            // check grants
            if ( (aceGrant != null) &&
                  aceGrant.checkPermission(permission) )
            {
                return true;
            }
        }
    }

    /**
     * determine the permissions for the person
     * person rights are stronger than group rights.
     */
    aceGrant = acl.getAce(who.getOid(), true);
    aceDeny = acl.getAce(who.getOid(), false);

    if((aceDeny != null) && aceDeny.checkPermission(permission))
    {
        return false;
```

```
        }
        if((aceGrant != null) && aceGrant.checkPermission(permission))
        {
            return true;
        }

        // Last check the rights for the groups where the person
        // is member of
        SystemObject group;
        SystemObjectList groups = who.getGroups();
        SystemObjectIterator it = groups.iterator();
        while( it.hasNext() ) {
            group = it.next();
            aceGrant = acl.getAce(group.getOid(), true);
            aceDeny = acl.getAce(group.getOid(), false);
            if( (aceDeny != null) &&
                 aceDeny.checkPermission(permission) )
            {
                return false;
            }
            if( (aceGrant != null) &&
                 aceGrant.checkPermission(permission) )
            {
                return true;
            }
        }

        // deny access if no permissions were set (not set)
        return false;
    }

    /**
     * Sets the specified permission on the SystemObject what for
     * the SystemObject what.
     * @param who The SystemObject for which the permission should be
     * tested.
     * @param what The SystemObject on which the permission should be
     * tested.
     * @param permission The permission to set.
     * @param positive Specifies if the permission shall be granted or
     * denied.
     */
    public void setPermission(SystemObject who, SystemObject what,
                              Permission permission, boolean positive)
    throws PersistenceException, AccessViolationException, Permission-
Exception
    {
        setPermission(who.getOid(), what, permission, positive);
    }

    /**
     * Sets the specified permission on the SystemObject what for
     * the SystemObject what.
     * @param who The ObjectIdentifier of the SystemObject for which
     * the permission should be tested.
     * @param what The SystemObject on which the permission should be
     * tested.
     * @param permission The permission to set.
     * @param positive Specifies if the permission shall be granted or
     * denied.
```

```
     */
    public void setPermission(ObjectIdentifier whoId,
      SystemObject what, Permission permission, boolean positive)
    throws PersistenceException, AccessViolationException, Permission-
Exception
    {
        // For changing the permissions the object must be locked.
        // no complications and side effects
        PersistenceManager pm = PersistenceManager.instance();
        pm.beginTransaction();
        boolean commit = false;
        try {
            pm.lock( what );

            Ace ace;
            Acl acl = what.getAcl();

            if( acl != null ) {
                if( what.hasOwnAcl() ) {
                    acl = (Acl) acl.mutableInstance();
                    pm.lock( acl );

                    ace = acl.getAce(whoId, positive);
                    if( ace != null ) {
                        ace.addPermission(permission);
                    } else {
                        ace = new Ace(whoId, positive);
                        ace.addPermission(permission);
                        acl.addAce(ace);
                    }
                } else {
                    throw new PermissionException(
                        "The permission couldn't be set, because the
object inherits the acl." );
                    }
            } else {
                acl = new Acl();
                ace = new Ace(whoId, positive);
                ace.addPermission(permission);
                acl.addAce(ace);

                what = what.mutableInstance();
                what.setAcl(acl);

                SystemObject rightsFolder = pm.load( ObjectIdenti-
fier.RIGHTS_FOLDER, true );
                rightsFolder.addChild( acl );
            }

            commit = true;
        } finally {
            pm.endTransaction(commit);
        }
    }

    /**
     * Removes the specified permission on the SystemObject what for
     * the SystemObject who.
     * @param who The SystemObject for which the permission should be
     * removed.
```

```
     * @param what The SystemObject from which the permission should
     * be removed.
     * @param permission The permission to remove.
     * @param positive Specifies if the permission to remove is
     * granted or denied.
     */
    public void removePermission(SystemObject who, SystemObject what,
                Permission permission, boolean positive)
    throws PersistenceException, AccessViolationException, Permission-
Exception
    {
        PersistenceManager pm = PersistenceManager.instance();

        pm.beginTransaction();
        boolean commit = false;
        try {
            pm.lock( what );

            Acl acl = what.getAcl();
            Ace ace;

            if ( acl != null ) {
                if ( what.hasOwnAcl() ) {
                    acl = (Acl) acl.mutableInstance();
                    pm.lock( acl );
                    ace = acl.getAce( who.getOid(), positive );
                    if ( ace != null ) {
                        ace.removePermission( permission );
                    }
                } else {
                    throw new PermissionException(
                        "The permission couldn't be removed, because
the object inherits the acl." );
                }
            }
            commit = true;
        } finally {
            pm.endTransaction(commit);
        }
    }
}
```

4.6.2. Permission.java

```
/*
 * Permission.java
 *
 * This class offers all available permissions.
 */

package at.jku.fim.welearn.system.rights;

/**
 *
 * @author Richard Leitner
 * @author Alexandros Paramythis
 */
public class Permission implements java.io.Serializable {

    static final long serialVersionUID = 528123206375229953L;

    /** Used to generate the internal representaion value. */
    private static int count = 0;

    /** The predefined <code>Permission</code> <i>READ</i>. */
    public static final Permission READ = new Permission(count++);

    /** The predefined <code>Permission</code> <i>WRITE</i>. */
    public static final Permission WRITE = new Permission(count++);

    /** The predefined <code>Permission</code> <i>EXECUTE</i>. */
    public static final Permission EXECUTE = new Permission(count++);

    /** The predefined <code>Permission</code> <i>VISIBLE</i>. */
    public static final Permission VISIBLE = new Permission(count++);

    /** The predefined <code>Permission</code> <i>CHANGE_RIGHTS</i>.
      */
    public static final Permission CHANGE_RIGHTS = new Permis-
sion(count++);

    /** The predefined <code>Permission</code> <i>EDIT</i>. */
    public static final Permission EDIT = new Permission(count++);

    /** The predefined <code>Permission</code> <i>ATTACH</i>. */
    public static final Permission ATTACH = new Permission(count++);


    /** The internal representation. */
    private int value;

    /**
     * Creates new instance of Permission.
     * @param value the internal representation of this enumeration
     */
    private Permission(int value) {
        this.value = value;
    }

    /**
     * Indicates whether some other object is "equal to" this one.
     * @param obj The reference object with which to compare.
```

```
     * @return <code>true</code> if this object is the same
     * as the obj argument; <code>false</code> otherwise.
     */
    public boolean equals(Object obj) {
        if ((obj == null) || (getClass() != obj.getClass())) {
            return false;
        }

        return this.value == ((Permission) obj).value;
    }

    /**
     * Returns a hash code value for the object.
     * @return A hash code value for this object.
     */
    public int hashCode() {
        return this.value;
    }

}
```

4.6.3. Acl.java

```
/*
 * Acl.java
 *
 * This class provides the access control list which contains
 * all permissions for accessing an object.
 */

package at.jku.fim.welearn.system.rights;

import java.util.*;
import at.jku.fim.welearn.system.object.*;
import at.jku.fim.welearn.persistence.PersistenceManager;
import at.jku.fim.welearn.persistence.PersistenceException;

/**
 *
 * @author Richard Leitner
*/

public class Acl extends SystemObject {

    static final long serialVersionUID = 6627328805172758530L;

    /** List with all access control entries. */
    private List entries;

    /** Creates a new instance of Acl */
    public Acl() {
        this.entries = new ArrayList();
    }

    /**
     * Sets the acl of this <code>SystemObject</code>.
     * <p><strong>Note:</strong>If there is no surrounding
     * transaction, a call to this method will result in persisting
     * this object immediately.</p>
     * @param sysobj the SystemObject of which the new acl should be
     * set.
     * @throws PersistenceException in case of any error doing a
     * persistence operation.
     * @throws AccessViolationException is this
     * <code>SystemObject</code> is immutable.
     */
    public void setAcl(SystemObject sysobj)
        throws PersistenceException, AccessViolationException
    {
        // Acl objects may not have an acl.
        throw new AccessViolationException( "Acl objects may not have
an acl." );
    }

    /**
     * Sets the acl of this <code>SystemObject</code>.
     * <p><strong>Note:</strong>If there is no surrounding
     * transaction, a call to this method will result in persisting
     * this object immediately.</p>
     * @param acl the new acl of this <code>SystemObject</code>
     * @throws PersistenceException in case of any error doing a
```

```
     * persistence operation.
     * @throws AccessViolationException is this
     * <code>SystemObject</code> is immutable.
     */
    public void setAcl(Acl acl)
        throws PersistenceException, AccessViolationException
    {
        // Acl objects may not have an acl.
        throw new AccessViolationException( "Acl objects may not have
an acl." );
    }

    /**
     * Adds a new entry to the ACL of the object
     * @param newACE The ACE that should be added.
     */
    public void addAce(Ace newAce)
    throws PersistenceException, AccessViolationException, Permission-
Exception {
        PersistenceManager pm = PersistenceManager.instance();

        pm.beginTransaction();
        boolean commit = false;
        try {
            pm.lock(this);

            Ace ace;
            Iterator it = this.entries.iterator();
            while( it.hasNext() ) {
                ace = (Ace) it.next();
                if( (ace.getHolder().equals( newAce.getHolder() )) &&
                    (ace.isPositive() == newAce.isPositive()) ) {
                    throw new PermissionException( "Adding ace not
possible - acl contains a conflicting ace." );
                }
            }
            this.entries.add(newAce);

            commit = true;
        } finally {
            System.out.println( "end addace - commit=" + commit );
            pm.endTransaction(commit);
        }

    }

    /**
     * Gets the positive <CODE>Ace</CODE> for the <CODE>SystemObject
     * holder</CODE>
     * @param holder The <CODE>ObjectIdentifier</CODE> of the object
     * whose <CODE>Ace</CODE> should be returned.
     * @return The <CODE>Ace</CODE> of the <CODE>SystemObject
     * holder</CODE>.
     */
    public Ace getAce(ObjectIdentifier holder) {
        return getAce(holder, true);
    }

    /**
     * Gets the <CODE>Ace</CODE> for the
```

```
     * <CODE>SystemObjectholder</CODE>
     * @param holder The <CODE>ObjectIdentifier</CODE> of the object
     * whose <CODE>Ace</CODE> should be returned.
     * @param positive If the positive or the negative ace shall be
     * returned.
     * @return The <CODE>Ace</CODE> of the
     * <CODE>SystemObject holder</CODE>.
     */
    public Ace getAce(ObjectIdentifier holder, boolean positive) {
        Ace ace;
        Iterator it = this.entries.iterator();
        while( it.hasNext() ) {
            ace = (Ace) it.next();
            if( ace.getHolder().equals(holder) && (ace.isPositive() ==
positive) ) {
                return ace;
            }
        }

        return null;
    }


    /**
     * Removes both <CODE>Ace</CODE> for the specified holder.
     * @param holder The <CODE>ObjectIdentifier</CODE> of the object
     * whose <CODE>Ace</CODE> should be removed.
     */
    public void removeAce(ObjectIdentifier holder)
        throws PersistenceException, AccessViolationException {
        PersistenceManager pm = PersistenceManager.instance();

        pm.beginTransaction();
        boolean commit = false;
        try {
            pm.lock(this);
            removeAce(holder, true);
            removeAce(holder, false);
            commit = true;
        } finally {
            pm.endTransaction(commit);
        }
    }

    /**
     * Removes the specified <CODE>Ace</CODE> for the specified
     * holder.
     * @param holder The <CODE>ObjectIdentifier</CODE> of the object
     * whose <CODE>Ace</CODE> should be removed.
     * @param positive specifies if the positive or the negative ace
     * should be removed.
     */
    public void removeAce(ObjectIdentifier holder, boolean positive)
        throws PersistenceException, AccessViolationException {
        PersistenceManager pm = PersistenceManager.instance();

        pm.beginTransaction();
        boolean commit = false;
        try {
            pm.lock(this);
```

```
            Ace ace;
            Iterator it = this.entries.iterator();
            while( it.hasNext() ) {
                ace = (Ace) it.next();
                if( (ace.getHolder().equals(holder)) &&
(ace.isPositive() == positive) ) {
                    this.entries.remove(ace);
                    return;
                }
            }

            commit = true;
        } finally {
            pm.endTransaction(commit);
        }
    }

    /**
     * Removes all empty aces from this acl.
     */
    public void cleanUp()
    throws PersistenceException, AccessViolationException {
        Ace ace;
        Iterator it = this.entries.iterator();
        while( it.hasNext() ) {
            ace = (Ace) it.next();
            if( ace.isEmpty() ) {
                removeAce( ace.getHolder() );
            }
        }
    }

    /**
     * Restores the state of this <code>SystemObject</code> according
     * to the specified memento.
     * @param m The memento holding a previous state of this object.
     */
    public void setMemento(Memento m) {
        MyMemento myMemento = (MyMemento) m;

        myMemento.restore(this);
        super.setMemento(myMemento.memento);
    }

    /**
     * Creates a memento of the current state of this
     * <code>SystemObject</code>.
     * @return a memento of the current state of this
     * <code>SystemObject</code>.
     */
    public Memento getMemento() {
        Memento m;

        m = super.getMemento();
        return new MyMemento(this, m);
    }

    /**
     * Stores the state of an <CODE>Person</CODE>.
```

```
     */
    private static class MyMemento implements Memento {

        /** The entries of this <CODE>Acl</CODE>.*/
        private List entries;

        /** The memento of the super class. */
        private Memento memento;

        /**
         * Creates a new instance of MyMemento.
         * @param o the source of this memento.
         */
        private MyMemento(Acl o, Memento m) {
            this.memento   = m;
            this.entries = o.entries;
        }

        /**
         * Restores the saved state inside this memento to the
         * specified <CODE>Person</CODE>.
         * @param o the <CODE>Person</CODE> to which the state should
         * be restored.
         */
        private void restore(Acl o) {
            o.entries = this.entries;
        }
    }
}
```

4.6.4. Ace.java

```
/*
 * Ace.java
 *
 * This class represents one element in an access control list.
 * One access control entry contains the permissions for one person.
 *
* @author Richard Leitner
 */

package at.jku.fim.welearn.system.rights;

import java.util.*;
import java.io.Serializable;

import at.jku.fim.welearn.system.object.*;

public class Ace implements Serializable {

    static final long serialVersionUID = 777722064120094892L;

    /** Determines whether the contained permissions are positive
     * ones. */
    private boolean positive;

    /** Determines the holder of this ace. */
    private ObjectIdentifier holder;

    /** The contained permissions */
    private List permissions;

    /** Creates a new instance of Ace */
    public Ace(ObjectIdentifier holder) {
        this(holder, true);
    }

    /** Creates a new instance of Ace */
    public Ace(ObjectIdentifier holder, boolean positive) {
        this.holder = holder;
        this.positive = positive;
        this.permissions = new ArrayList();
    }

    /**
     * Checks if the specified permission is part of the permission
     * set in this ace.
     * @param p Permission to be checked for.
     * @return true if the permission is part of the permission set in
     * this ace, false otherwise..
     */
    public boolean checkPermission(Permission p) {
        return this.permissions.contains(p);
    }

    /**
     * Checks if the <CODE>Ace</CODE> is empty.
     * @return <CODE>true</CODE> if empty.

     */
```

```
    public boolean isEmpty() {
        return this.permissions.isEmpty();
    }

    /**
     * Removes a <CODE>Permission-Object</CODE>.
     * @param p The permission to remove.
     */
    public void removePermission(Permission p) {
        this.permissions.remove(p);
    }

    /**
     * Adds the permission p to the Ace.
     * @param p The permission to add.
     */
    public void addPermission(Permission p) {
        if( !this.permissions.contains(p) ) {
            this.permissions.add(p);
        }
    }

    /**
     * Getter for property positive.
     * @return Value of property positive.
     */
    public boolean isPositive() {
        return positive;
    }

    /**
     * Setter for property positive.
     * @param positive New value of property positive.
     */
    public void setPositive(boolean positive) {
        this.positive = positive;
    }

    /**
     * Getter for property holder.
     * @return Value of property holder.
     */
    public ObjectIdentifier getHolder() {
        return holder;
    }

    /**
     * Setter for property holder.
     * @param holder New value of property holder.
     */
    public void setHolder(ObjectIdentifier holder) {
        this.holder = holder;
    }

}
```

4.6.5. ChangeRight.java

```
/*
 * ChangeRight.java
 *
 * This class represents the command to change the permissions
 * of a object.
 *
 * @author Richard Leitner
 */

package at.jku.fim.welearn.objects.utilities;

import at.jku.fim.welearn.system.object.command.*;
import at.jku.fim.welearn.system.object.AccessViolationException;
import at.jku.fim.welearn.system.object.SystemObject;
import at.jku.fim.welearn.system.object.ObjectIdentifier;
import at.jku.fim.welearn.system.SystemPath;
import at.jku.fim.welearn.system.InvalidPathException;
import at.jku.fim.welearn.objects.user.*;
import at.jku.fim.welearn.persistence.*;
import at.jku.fim.welearn.system.rights.*;
import at.jku.fim.welearn.reference.collections.SystemObjectIterator;
import java.util.*;

public class ChangeRight extends SystemObject {

    /** Default name for using the command in the CLI. */
    public static final String DEFAULT_NAME = "chgright";

    /** Constants defining the status */
    private static final int DONT_SET = 0;
    private static final int GRANT    = 1;
    private static final int DENY     = 2;
    private static final int RESET    = 3;

    /** Changing the read permission */
    transient int setReadable = DONT_SET;

    /** Changing the write permission. */
    transient int setWriteable = DONT_SET;

    /** Changing the execute permission. */
    transient int setExecutable = DONT_SET;

    /** Changing the visible permission. */
    transient int setVisible = DONT_SET;

    /** Changing the changerights permission. */
    transient int setChangeRights = DONT_SET;

    /** Changing the edit permission. */
    transient int setEdit = DONT_SET;

    /** Changing the attach permission. */
    transient int setAttach = DONT_SET;

    /** Determines if the object should inherit the acl. */
    transient boolean inheritAcl = false;
```

```
    /** Determines if the object should get its own acl. */
    transient boolean ownAcl = false;

    /** Defines the SystemObject to change the rights for. */
    transient SystemObject sysobj = null;

    /** Defines the parent of the SystemObject to change the rights
for. */
    transient SystemObject parent = null;

    /** The person or group for which the right is changed. */
    transient SystemObject who = null;

    /** Creates a new instance of ChangeRight */
    public ChangeRight() {
    }

    /**
     * Handles the capabilities command.
     * @param capabilities The <code>capabilities</code>
     * command to handle.
     */
    public void doCommand(Capabilities capabilities) {
        // Call our super class to get the default capabilities.
        super.doCommand( capabilities );

        // Move has no gui and only processes command line tasks.
        capabilities.setExecutable( true );
    }

    /**
     * This method handles the execute command to do the changing.
     * @param execute The <code>Execute</code> command to handle.
     * @throws AccessViolationException If this object is read-only.
     * @throws PersistenceException In case of any error during a
     * load or store operation. CommandLineExecute
     */
    public void doCommand(Execute execute)
        throws PersistenceException, AccessViolationException
    {
        PersistenceManager pm = PersistenceManager.instance();
        SystemObject rightsFolder = pm.load
              ( ObjectIdentifier.RIGHTS_FOLDER, true );

        execute.setSuccessful(true);
        interpretParameters(execute);
        if( !execute.isSuccessful() ) {
            return;
        }

        execute.setSuccessful(false);
        if( (sysobj == null) ||
            (!inheritAcl && !ownAcl && (who == null)) )
        {
            execute.setResult( "The object or the person/group doesn't
exist\n" +
                               usage(execute.getParameters() ) );
            return;
        }
```

```
        // Check whether the current user is allowed to change the
        // rights for the specified object.
        RightsManager rm = RightsManager.instance();
        if( !rm.checkPermission((Person)pm.load(execute.getUserId()),
                sysobj, Permission.CHANGE_RIGHTS ) )
        {
            execute.setResult( "The current user is not allowed to
                 change the rights for the specified object." );
            execute.setSuccessful(false);
            return;
        }

        // look if the object has an acl.
        Acl acl = sysobj.getAcl();

        // look if the inheritance should be changed.
        if( inheritAcl ) {
            if( (acl != null) && !sysobj.hasOwnAcl() ) {
                execute.setResult( "The object inherits the acl al-
already\n" );
                return;
            } else {
                if( (parent == null) || (parent.getAcl() == null) ) {
                    execute.setResult( "The object can't inherit the
acl, parent doesn't exist or doesn't have an acl.\n" );
                    return;
                }

                sysobj.setAcl( parent );
                execute.setSuccessful(true);
                return;
            }
        }

        // look if the object should get its own acl.
        if( ownAcl ) {
            if( sysobj.hasOwnAcl() ) {
                execute.setResult( "The object has already its own
acl.\n" );
                return;
            } else {
                if( parent == null ) {
                    execute.setResult( "The object can't get a copy of
the parent's acl, the parent doesn't exist.\n" );
                    return;
                }

                pm.beginTransaction();
                boolean commit = false;
                try {
                    acl = parent.getAcl();
                    Duplicate duplicate = new Duplicate(rightsFolder);
                    acl.doCommand( duplicate );

                    acl = (Acl) pm.load( duplicate.getDuplicateOid()
);
                    sysobj.setAcl( acl );

                    commit = true;
                    execute.setSuccessful(true);
```

```
                    return;
                } finally {
                    pm.endTransaction(commit);
                }
            }
        }

        // check if the systemobject has its own acl.
        if( (acl != null) && !sysobj.hasOwnAcl() ) {
            execute.setResult( "The object inherits the acl, it can't
be changed; you have to set an own acl for this object first.\n" +
                               usage(execute.getParameters() ) );
            return;
        }

        pm.beginTransaction();
        boolean commit = false;
        try {
            try {
                if( setReadable == GRANT ) {
                    rm.setPermission( who, sysobj,
                       Permission.READ, true );
                } else if( setReadable == DENY ) {
                    rm.setPermission( who, sysobj,
                       Permission.READ, false );
                } else if( setReadable == RESET ) {
                    rm.removePermission( who, sysobj,
                       Permission.READ, true );
                    rm.removePermission( who, sysobj,
                       Permission.READ, false );
                }

                if( setWriteable == GRANT ) {
                    rm.setPermission( who, sysobj,
                       Permission.WRITE, true );
                } else if( setWriteable == DENY ) {
                    rm.setPermission( who, sysobj,
                       Permission.WRITE, false );
                } else if( setWriteable == RESET ) {
                    rm.removePermission( who, sysobj,
                       Permission.WRITE, true );
                    rm.removePermission( who, sysobj,
                       Permission.WRITE, false );
                }

                if( setExecutable == GRANT ) {
                    rm.setPermission( who, sysobj,
                       Permission.EXECUTE, true );
                } else if( setExecutable == DENY ) {
                    rm.setPermission( who, sysobj,
                       Permission.EXECUTE, false );
                } else if( setExecutable == RESET ) {
                    rm.removePermission( who, sysobj,
                       Permission.EXECUTE, true );
                    rm.removePermission( who, sysobj,
                       Permission.EXECUTE, false );
                }

                if( setVisible == GRANT ) {
                    rm.setPermission( who, sysobj,
```

```
                        Permission.VISIBLE, true );
                } else if( setVisible == DENY ) {
                    rm.setPermission( who, sysobj,
                        Permission.VISIBLE, false );
                } else if( setVisible == RESET ) {
                    rm.removePermission( who, sysobj,
                        Permission.VISIBLE, true );
                    rm.removePermission( who, sysobj,
                        Permission.VISIBLE, false );
                }

                if( setChangeRights == GRANT ) {
                    rm.setPermission( who, sysobj,
                        Permission.CHANGE_RIGHTS, true );
                } else if( setChangeRights == DENY ) {
                    rm.setPermission( who, sysobj,
                        Permission.CHANGE_RIGHTS, false );
                } else if( setChangeRights == RESET ) {
                    rm.removePermission( who, sysobj,
                        Permission.CHANGE_RIGHTS, true );
                    rm.removePermission( who, sysobj,
                        Permission.CHANGE_RIGHTS, false );
                }

                if( setEdit == GRANT ) {
                    rm.setPermission( who, sysobj, Permission.EDIT,
                                      true );
                } else if( setEdit == DENY ) {
                    rm.setPermission( who, sysobj, Permission.EDIT,
                                      false );
                } else if( setEdit == RESET ) {
                    rm.removePermission( who, sysobj, Permission.EDIT,
                                         true );
                    rm.removePermission( who, sysobj, Permission.EDIT,
                                         false );
                }

                if( setAttach == GRANT ) {
                    rm.setPermission( who, sysobj, Permission.ATTACH,
                                      true );
                } else if( setAttach == DENY ) {
                    rm.setPermission( who, sysobj, Permission.ATTACH,
                                      false );
                } else if( setAttach == RESET ) {
                    rm.removePermission( who, sysobj,
                                         Permission.ATTACH, true );
                    rm.removePermission( who, sysobj,
                                         Permission.ATTACH, false );
                }

                commit = true;
                execute.setSuccessful(true);

            } catch( PermissionException e ) {
                execute.setSuccessful(false);
                throw new AccessViolationException( "Rights could not
be changed.\n" );
            }
        } finally {
            pm.endTransaction(commit);
```

```
        }


    }

    /**
     * Parses the parameters of the command and interprets them to
     * fill the corrseponding properties.
     * @param parameters The parameters to interpret.
     */
    private void interpretParameters(Execute execute)
        throws PersistenceException
    {
        PersistenceManager pm = PersistenceManager.instance();
        UsersFolder usersFolder = (UsersFolder) pm.load( ObjectIdenti-
fier.USERS_FOLDER );
        SystemPath currentPath  = execute.getCurrentPath();

        List parameters = execute.getParameters();
        Iterator it = parameters.iterator();
        String param;
        boolean error = false;

        // Ignore the first parameter, because this is the command
        // name.
        if ( it.hasNext() ) {
            it.next();
        }

        while ( it.hasNext() && !error ) {
            param = (String) it.next();
            if( "-grant".equals( param ) ) {
                if( it.hasNext() ) {
                    param = (String) it.next();
                    if( "chgrights".equals( param ) ) {
                        setChangeRights = GRANT;
                    } else if( "edit".equals( param ) ) {
                        setEdit = GRANT;
                    } else if( "attach".equals( param ) ) {
                        setAttach = GRANT;
                    } else {
                        if( param.indexOf('r') != -1 ) {
                            setReadable = GRANT;
                        }
                        if( param.indexOf('w') != -1 ) {
                            setWriteable = GRANT;
                        }
                        if( param.indexOf('x') != -1 ) {
                            setExecutable = GRANT;
                        }
                        if( param.indexOf('v') != -1 ) {
                            setVisible = GRANT;
                        }
                    }
                } else {
                    error = true;
                }
            } else if("-deny".equals( param )) {
                if( it.hasNext() ) {
                    param = (String) it.next();
```

```
            if( "chgrights".equals( param ) ) {
                setChangeRights = DENY;
            } else if( "edit".equals( param ) ) {
                setEdit = DENY;
            } else if( "attach".equals( param ) ) {
                setAttach = DENY;
            } else {
                if( param.indexOf('r') != -1 ) {
                    setReadable = DENY;
                }
                if( param.indexOf('w') != -1 ) {
                    setWriteable = DENY;
                }
                if( param.indexOf('x') != -1 ) {
                    setExecutable = DENY;
                }
                if( param.indexOf('v') != -1 ) {
                    setVisible = DENY;
                }
            }
        } else {
            error = true;
        }
    } else if("-reset".equals( param )) {
        if( it.hasNext() ) {
            param = (String) it.next();
            if( "chgrights".equals( param ) ) {
                setChangeRights = RESET;
            } else if( "edit".equals( param ) ) {
                setEdit = RESET;
            } else if( "attach".equals( param ) ) {
                setAttach = RESET;
            } else {
                if( param.indexOf('r') != -1 ) {
                    setReadable = RESET;
                }
                if( param.indexOf('w') != -1 ) {
                    setWriteable = RESET;
                }
                if( param.indexOf('x') != -1 ) {
                    setExecutable = RESET;
                }
                if( param.indexOf('v') != -1 ) {
                    setVisible = RESET;
                }
            }
        } else {
            error = true;
        }
    } else if( "-inheritacl".equals( param ) ) {
        inheritAcl = true;
    } else if( "-ownacl".equals( param ) ) {
        ownAcl = true;
    } else if("-what".equals( param )) {
        if( it.hasNext() ) {
            param = (String) it.next();
            try {
                sysobj = currentPath.findObject( param, true,
                                execute.getEnvironment() );
            } catch( InvalidPathException e ) {
```

```
                        error = true;
                    }

                    String path = SystemPath.extractPath( param );
                    try {
                        if( path != null ) {
                            parent = currentPath.findObject( path );
                        } else {
                            parent = currentPath.currentDirectory();
                        }
                    } catch( InvalidPathException e ) {
                        parent = null;
                    }
                } else {
                    error = true;
                }
            } else if("-person".equals( param )) {
                if( it.hasNext() ) {
                    param = (String) it.next();
                    who = usersFolder.findUser(param);
                } else {
                    error = true;
                }
            } else if("-group".equals( param )) {
                if( it.hasNext() ) {
                    param = (String) it.next();
                    try {
                        who = currentPath.findObject( param, true );
                        if( !(who instanceof Group) ) {
                            error = true;
                        }
                    } catch( InvalidPathException e ) {
                        error = true;
                    }
                } else {
                    error = true;
                }
            } else if ("-owner".equals( param )) {
                who = usersFolder.findUser( "owner" );
            } else {
                error = true;
            }
        }

        if( error || (parameters.size() < 2) ) {
            execute.setSuccessful( false );
            execute.setResult( usage( parameters) );
        }
    }
}
```

5 Literaturverweise

Bei Hyperlinks wird in Klammer ein Datum angegeben, an dem diese Seite das letzte Mal besucht wurde und somit noch online war.

[And95] Andleigh, Prabhat K., „UNIX Systemarchitektur“, Carl Hanser Verlag & Prentice-Hall International 1995, ISBN 3-446-16151-1

[App02] Apple Computer, Inc: “Inside Mac OS X”, http://developer.apple.com /techpubs/macosx/Essentials/SystemOverview/, Juli 2002 (2003-01-28)

[Bra98] Brause, Rüdiger: „Betriebssystem - Grundlagen und Konzepte“, Springer-Verlag Berlin Heidelberg 1998, ISBN 3-540-62929-7

[Bün02] Bünning, Uwe; Krause, Jörg: „Windows XP Professional“, Grundlagen und Strategien für den Einsatz am Arbeitsplatz und im Netzwerk, Carl Hanser Verlag München Wien, 2002, ISBN 3-446-21939-0

[Che00] Cheswick W., Bellovin S.: “Firewall and Internet Security – Repelling the Wily Hacker”, Addison-Wesley, 2000, ISBN 0201-633-574

[Cus93] Custer, Helen: „Inside Windows NT“, Microsoft Press, 1993, ISBN 1-55615-481-X

[Div02] Divotkey, Roman; Remplbauer, Doris: „Design und Implementierung einer webbasierten Lernumgebung“, Diplomarbeit am Institut für Informations-verarbeitung und Mikroprozessortechnik der Johannes Kepler Universität Linz, Februar 2002

[Gro91] Groll, Manfred: „Das UNIX Sicherheits-Handbuch“, Vogel Verlag Würz-burg, 1991, ISBN 3-8023-0380-6

[Hel01] Helml, Thomas: „Darstellung von Rechtestrukturen in Verzeichnisdiensten (am Beispiel Active Directory)“, Diplomarbeit am Institut für Informationsverarbeitung und Mikroprozessortechnik der Johannes Kepler Universität Linz, November 2001

[Hoe02] Hörmanseder, Rudolf; Kofler, Christoph: “Security Management: Authentication“, Unterlagen zur Lehrveranstaltung „Sicherheitsmanangement in der Informatik“, Sommersemester 2002, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz

[IMS01] IMS Global Lerning Consrotium, Inc: “IMS Content Packaging Information Model”, http://www.imsproject.org/content/packaging/cpv1p1p2/imscp_infov1p1p2.html, August 2001 (2003-01-30)

[IMS02] IMS Global Learning Consortium, Inc: „IMS Content Packaging Specification“, http://www.imsproject.org/content/packaging/index.cfm, November 2002 (2003-01-30)

[Jäg00] Jäger, M.: “UNIX-Dateisystem - Eine Einführung”, FH Giessen-Friedburg, 10. Oktober 2000, http://homepages.fh-giessen.de/~hg52/lv/bs1/skripten/dateisystem/pdf/dateisystem.pdf (2003-01-30)

[Kal01] Kalhammer, F: „Linux-Praxis: Grundlegende Eigenschaften der Unix-Dateisysteme“, 2001, http://www.linux-praxis.de/linux1/filesystem1.html (2003-02-12)

[Kof99] Kofler, Michael: “Linux - Installation, Konfiguration, Anwendung” , Addison-Wesley 1999, 4. Auflage, ISBN 3-8273-1475-5

[Kra89] Krallmann, Hermann: „EDV Sicherheitsmanagement“, Integrierte Sicherheitskonzepte für betriebliche Informations- und Kommunikationssysteme, Erich Schmidt Verlag 1989, ISBN 3-503-02858-7

[Lin00] LinuxFocus Magazine: „Unix Basics: File Access Permissions“, 31. Oktober 2000, http://www.tldp.org/linuxfocus/English/January1999/article77.html (2003-02-12)

[Mic98] Microsoft Corporation: “A Brief History of the Windows NT Operating System”, Oktober 1998, http://www.microsoft.com/presspass/features/1998/winntfs.asp (2003-03-21)

[Mic00] Microsoft Corporation: “Microsoft Windows 2000 Security Technical Reference”, Microsoft Press, 2000, ISBN 0-7356-0858-X

[Mic02] Microsoft Corporation: „MSCE Training Kit - Microsoft Windows XP Professional“, Microsoft Press, 2002, ISBN 3-86063-930-7

[Noa90] Noack, Claus; Hennig, Dietmar: “Systemsicherheit unter UNIX”, Carl Hanser Verlag München Wien 1990, ISBN 3-446-15988-6

[SAT00] SAT-Team: „SAT – NTFS Security Administration Tool“, Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM), Johannes Kepler Universität Linz, http://www.fim.uni-linz.ac.at/Research/sat/default.htm (2003-03-25)

[Sil02] Silberschatz, A.; Galvin, P. B.; Gagne, G.: „Operating System Concepts“, 6. Auflage, John Wiley & Sons, Inc. 2002, ISBN 0-471-41743-2

[Sol00] Solomon, David A.; Russinovich, Mark: “Inside Microsoft Windows 2000”, Microsoft Press, 2000, ISBN 0-7356-1021-5

6 Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am

7 Curriculum Vitae

Persönliche Daten

Name	Richard Leitner
E-Mail	*leitner@esys.at*
Geboren am	15. November 1974 in Vöcklabruck, OÖ
Eltern	Annemarie, geb. 1943, Gemeindebeamtin i. R. August, geb. 1937, Postbeamter i. R.
Bruder	Florian, geb. 1980, Student
Familienstand	ledig

Schulausbildung

1981 - 1985	Volksschule Timelkam
1985 - 1989	Bundesgymnasium Vöcklabruck
1989 - 1993	Bundesrealgymnasium Vöcklabruck; Matura: Juni 1993 (Fachbereichsarbeit Informatik: „Stack - Queue - Tree“; Betreuer: Mag. Schwarz Günther)

Präsenzdienst

Okt. 1993 - Mai 1994	Präsenzdienst, Schwarzenbergkaserne Salzburg

Studium

Seit Oktober 1994	Studium der Informatik an der Johannes Kepler Universität Linz
Seit Juli 1999	Studienrichtungsvertreter Informatik Mitglied diverser Gremien und Kommissionen der Universität Linz bzw. der Hochschülerschaft

Berufspraxis

Nov. 1999 - Mai 2001	freier Mitarbeiter der Rosenbauer International AG

Seit Mai 2001	Gesellschafter der eSYS Mayrhofer & Leitner Informationssysteme OEG
März - Dez. 2002	Projektmitarbeiter am Institut für Informationsverarbeitung und Mikroprozessortechnik der Johannes Kepler Universität Linz
Seit Okt. 2002	Prokurist der eSYS Informationssysteme GmbH
	Diverse Ferialpraktika (u. a. bei Post und Telekom Austria AG (nunmehr Post AG), SPS Technik GmbH)