Studienarbeit 03/01

Entwicklung einer Softwarekomponente zur Verwaltung von Dokumenthierarchien

von

Kai Schmitz-Hofbauer

Bochum, im August 2003

Betreuer: Prof. Dr.-Ing. habil. Helmut Balzert
Dipl.-Inform. Peter Ziesche

Zusammenfassung

Diese Studienarbeit beschäftigt sich mit der Entwicklung einer Komponente für den Windows-Client der *e-learning*-Plattform w3l. Mit dem w3l-Windows-Client werden zum einen administrative Tätigkeiten durchgeführt. Zum anderen dient er zur Erfassung und Verwaltung von Dokumenten von *e-learning*-Kursen. Die zu entwickelnde Komponente soll in erster Linie Autoren von e-learning-Kursen beim Erfassen und Verwalten ihrer Dokumente unterstützen. Zu diesem Zweck bietet die Komponente die Möglichkeit, eigene Verwaltungsstrukturen in Form von Ordnern anzulegen, in denen Autoren ihre persönlichen Dokumente ablegen können. Bei der Entwicklung von w3l wird das *CASE*-Werkzeug Janus der Firma oTRIs eingesetzt, mit dem es möglich ist, kaufmännisch/administrative Anwendungen auf Basis eines UML-OOA-Modelles zu generieren. Die Entwicklung der Komponente wird maßgeblich von diesem Werkzeug beeinflusst.

Schlüsselwörter: Komponente, Halbfabrikat, Softwarebaustein, Janus, Generatorsystem, w3l, e-learning, Autorenunterstützung, Baum, Hierarchie, Dokumenthierarchie, MFC

Abstract

This student research project deals with the development of a software-component for the windows-client of the e-learning-platform w3l. On the one hand with the w3l-windows-client it is possible to perform general adminstrative actions. On the other hand authors of e-learning courses can use the windows-client to gather and to administrate e-learning-documents. As a first priority this component should assist authors in gathering and administrating their personal documents. For this purpose the component offers the possibility to create administration structures in terms of folders. In these structures authors can store their personal documents. The development of w3l is based on the use of the CASE-tool Janus. Janus offers the possibility to generate business/administrative applications based on a UML-diagram. This tool has a significant influence on the development of the component.

keywords: component, semifinished product, software element, w3l, Janus, generator system, e-learning, author assistance, tree, hierarchy, document hierarchy, MFC

Inhaltsverzeichnis

1 Einleitung

1.1 Die e-learning-Plattform w3l

Diese Studienarbeit ist im Rahmen der Entwicklung der *e-learning*-Plattform w3l entstanden. Die *e-learning*-Plattform w3l ist ein gemeinsames Forschungsprojekt der Ruhr-Universität Bochum und der Fachhochschule Dortmund. Das Akronym w3l steht für **l**ebens**l**anges **L**ernen im **W**eb. Das zu erwerbende Wissen wird den Lernenden in Form von Kursen zur Verfügung gestellt, auf die sie über das Internet zugreifen können. Abbildung 1.1-1 zeigt den Aufbau des w3l-Systems.

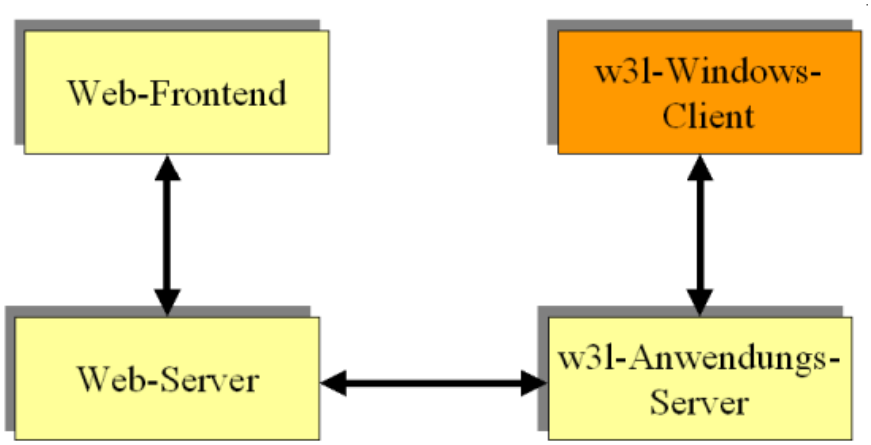

Abb. 1.1-1: Die Architektur von w3l

Das Web-Frontend ist die Schnittstelle, über die Lernende auf das System zugreifen. Es ist die einzige Komponente, die für sie sichtbar ist. Der Anwendungsserver kapselt die Anwendungslogik und die Datenhaltung. Mit dem w3l-Windows-Client werden zum einen administrative Tätigkeiten durchgeführt. Zum anderen dient er zur Erfassung und Verwaltung von *e-learning*-Kursen. Ziel dieser Studienarbeit ist es, eine Komponente zu entwickeln, die die Benutzerfreundlichkeit des w3l-Windows-Client für Autoren von *e-learning*-Kursen erhöht.

1.2 Das Generatorsystem Janus

Bei der Entwicklung von w3l wird das *CASE*-Werkzeug *Janus* der Firma oTRIs eingesetzt. Bei Janus handelt es sich um ein Generatorsystem, mit dem es möglich ist, kaufmännisch/administrative Anwendungen auf Basis eines UML-OOA-Diagrammes zu generieren. Janus ermöglicht das Erstellen von (lauffähigen) Client-/Server-Pilotsystemen mit

Datenbankanbindung und grafischer Benutzeroberfläche. Sowohl der w3l-Windows-Client als auch der Anwendungsserver werden mit diesem Werkzeug erstellt. Es ist möglich, den generierten Programmcode um eigenen Code, sog. USER CODE, zu erweitern. Hierzu stellt Janus eine eigene Klassenbibliothek zur Verfügung.

2 Zielsetzung

Wie bereits erwähnt, ist es Ziel dieser Studienarbeit, eine Komponente zu entwickeln, die das Arbeiten mit dem w3l-Windows-Client erleichtert. Die zu entwickelnde Komponente soll in erster Linie Autoren von *e-learning*-Kursen beim Erfassen und Verwalten ihrer Dokumente unterstützen. Im Folgenden soll nun zunächst umrissen werden, wie Autoren derzeit ihre Arbeit mit dem w3l-Client verrichten und auf welche Hürden sie dabei stoßen.

2.1 Arbeit mit dem w3l-Windows-Client

Abbildung 2.1-1 zeigt das Hauptfenster des w3l-Windows-Client.

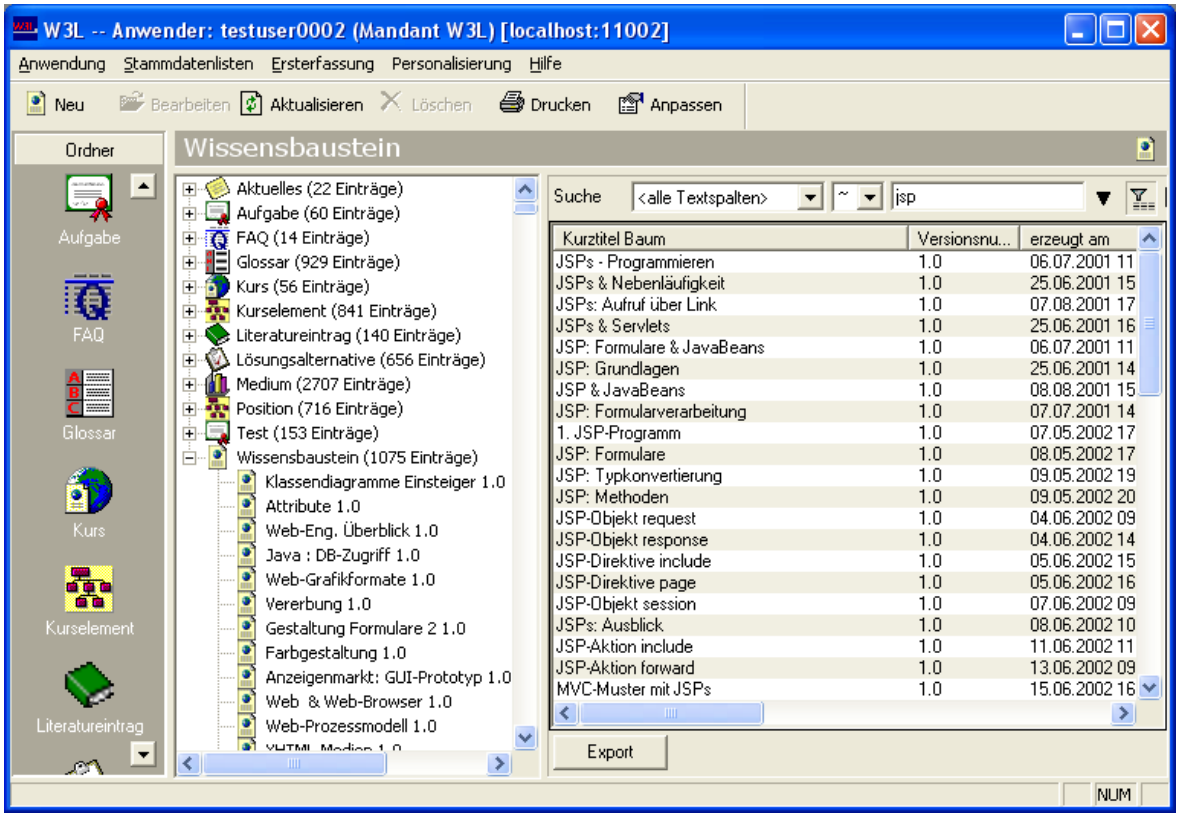

Abb. 2.1-1: Der w3l-Windows-Client

Das Hauptfenster ist grob in die drei Teile **Ordnerleiste** (links), **Baumstruktur** (Mitte) und den **rechten Fensterbereich** unterteilt. Die Ordnerleiste dient lediglich zur Schnellorientierung. In der Baumstruktur sind alle vorhandenen Objektgruppen (zum Beispiel Kurs) mit der Anzahl der vorhandenen Objekte aufgeführt. Im rechten Bereich wird eine Liste aller Objekte einer Objektgruppe angezeigt. Die Listenan-

sicht erlaubt auch das Filtern nach verschiedenen Kriterien, so dass bei Bedarf die Anzeige der Objekte auf eine Teilmenge der zu einer Objektgruppe gehörenden Objekte beschränkt werden kann. Die Listenansichten lassen sich auch über das Menü *Stammdatenlisten* in einem externen Dialogfenster außerhalb des Hauptfensters darstellen. Für jede Objektgruppe existiert ein Erfassungsdialog, der zur Erfassung der Daten eines Objektes dient. Dieser Dialog kann durch Auswählen eines Objektes aus der Listenansicht oder der Baumstruktur geöffnet werden. Falls die Daten eines neuen Objektes erfasst werden sollen, muss dieser Dialog über das Menü *Ersterfassung* aufgerufen werden. Für Autoren sind neben der bereits erwähnten Objektgruppe *Kurs* die Objektgruppen *Wissensbaustein*, *Medium*, *Glossarbegriff*, *Literatureintrag*, *Aufgabe*, *Test*, *Lösungsalternative* und *FAQ* von Bedeutung. Wissensbausteine sind elementare Wissenseinheiten, aus denen sich die w3l-Kurse zusammensetzen. Medien können Bilder, Animationen oder Video- und Tonaufzeichnungen sein. In w3l werden zwei Arten von Übungen unterschieden, nämlich Tests und Aufgaben. Tests sind Multiple-Choice-Aufgaben. Bei Aufgaben muss ein gestelltes Problem konstruktiv oder analytisch gelöst werden. Im Wesentlichen besteht der Unterschied zwischen beiden darin, dass Tests automatisch korrigiert werden können, Aufgaben jedoch nicht. Die Namen der restlichen Objektgruppen sind aussagekräftig und bedürfen daher keiner weiteren Erklärung. Im Folgenden wird im Zusammenhang der aufgezählten Objektgruppen der Begriff *Dokument* synonym für *Objekt* verwendet.

2.2 Hindernisse beim Erfassen und Verwalten von Dokumenten

Das Arbeiten mit dem w3l-Windows-Client gestaltet sich an einigen Stellen etwas unkomfortabel. Da das w3l-System seit einiger Zeit erfolgreich im Einsatz ist, haben immer mehr Autoren Dokumente erstellt. Die Anzahl der Dokumente ist stetig gewachsen. So ist zum Beispiel die Zahl der im System vorhandenen Wissensbausteine auf 1075 (vgl. Abbildung 2.1-1) gestiegen. Das Auswählen eines Dokuments (zum Bearbeiten) ist recht mühsam. Wird der Baumeintrag für Wissensbausteine aufgeklappt, so erscheinen dort alle 1075 Verwaltungs-Dokumente. Eine alphabetische Sortierung findet nicht statt. Es kann mehrere Minuten dauern, den gewünschten Wissensbaustein zu lokalisieren. Eine Alternative zu der Baumstruktur bietet die Listenansicht. Über sie

ist es zumindest möglich, gezielt nach Objekten zu suchen. Aber auch diese Variante ist für die tägliche Arbeit zu aufwendig. Gerade wenn über einen längeren Zeitraum regelmäßig an einem Dokument gearbeitet wird, ist dieses Vorgehen unkomfortabel. Ein weiterer Nachteil des w3l-Windows-Client ist, dass Objekte nur in ihrer Objektgruppe gespeichert werden können. Kurse werden mit anderen Kursen zusammen gespeichert, Wissensbausteine mit anderen Wissensbausteinen, Medien mit anderen Medien etc. Es ist nicht möglich, Dokumente projektbezogen zu speichern. Es wäre praktisch, wenn alle Dokumente, die zu einem Projekt gehören, in einer organisatorischen Einheit gespeichert werden könnten. So wäre es zum Beispiel möglich, alle zu einem Kurs gehörenden Dokumente, wie zum Beispiel Wissensbausteine, Aufgaben, Tests, Medien und Glossarbegriffe, zusammen abzulegen.

2.3 Verbesserungsmöglichkeiten

Die im vorangegangenen Kapitel erwähnten Nachteile des w3l-Windows-Client könnten eliminiert werden, indem man Benutzern die Möglichkeit gibt, eigene Verwaltungsstrukturen für ihre persönlichen Dokumente anzulegen. Diese Verwaltungsstruktur kann in Form einer Ordnerstruktur realisiert werden. So könnte ein Autor entsprechend den Projekten, an denen er derzeit arbeitet, eine Ordnerstruktur anlegen, in der er seine Wissensbausteine, Medien, Aufgaben etc. speichert. Dies hätte den Vorteil, dass das aufwendige Suchen nach den eigenen Dokumenten entfällt und zusammengehörende Dokumente auch zusammen gespeichert werden. Dieser „Ordneransatz“ wird im Weiteren verfolgt.

3 Modell zur Verwaltung der Dokumenthierarchie

In diesem Kapitel wird ein Modell zur Speicherung von Dokumenthierarchien vorgestellt.

3.1 Ausschnitt aus dem w3l-Klassendiagramm

Bevor aber näher auf das Modell eingegangen werden kann, muss zunächst ein Ausschnitt aus dem w3l-OOA-Modell vorgestellt werden. Dies ist für das weitere Verständnis unverzichtbar. Das folgende Klassendiagramm zeigt, wie die verschiedenen Dokumenttypen auf eine Klassenhierarchie abgebildet werden.

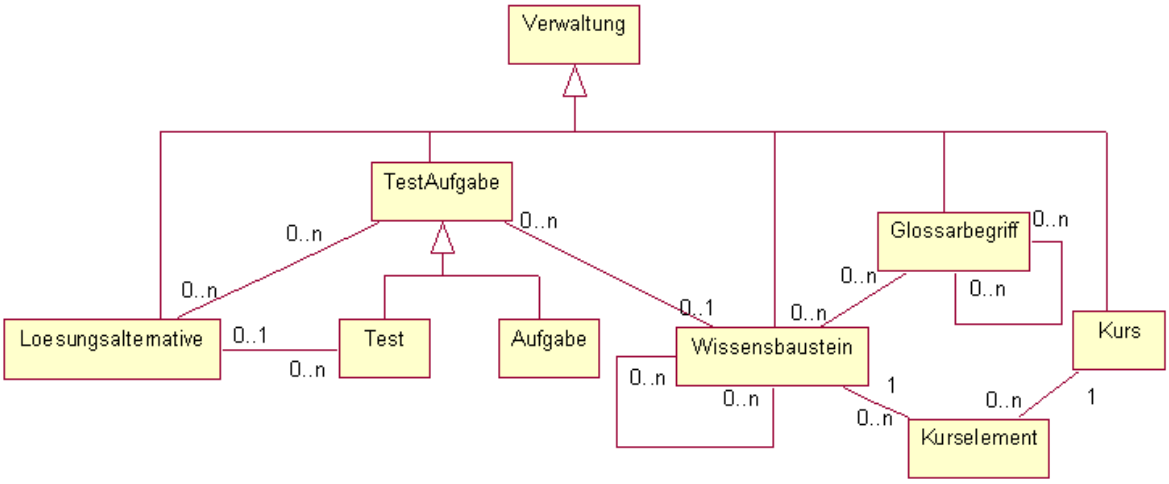

Abb. 3.1-1: Vererbungshierarchie der Verwaltungsklassen

Wie dem Diagramm zu entnehmen ist, existiert für jeden Dokumenttyp bzw. jede Objektgruppe eine eigene Klasse. Alle diese Klassen haben die gemeinsame Oberklasse `Verwaltung`. Die Klassen `Test` und `Aufgabe` haben zusätzlich die gemeinsame Oberklasse `TestAufgabe`. Im Folgenden werden diese Klassen auch als Verwaltungsklassen bezeichnet. Um das Diagramm übersichtlich zu gestalten, wurden nur die für Autoren wichtigsten Klassen aufgeführt. Zwischen den Klassen existieren natürlich komplexe Beziehungen. Die Darstellung der Assoziationen wurde im obigen Diagramm auf die wichtigsten beschränkt, da sie für diese Studienarbeit eine untergeordnete Rolle spielen und das Diagramm unnötig komplizieren würden.

3.2 Modell der Ordnerstruktur

Wie bereits angesprochen, sollen Autoren ihre Dokumente in Ordnern speichern können. Die Ordner müssen also Objekte vom Typ Verwaltung speichern. Das Klassendiagramm aus Abbildung 3.2-1 zeigt eine Lösung dieses Problems.

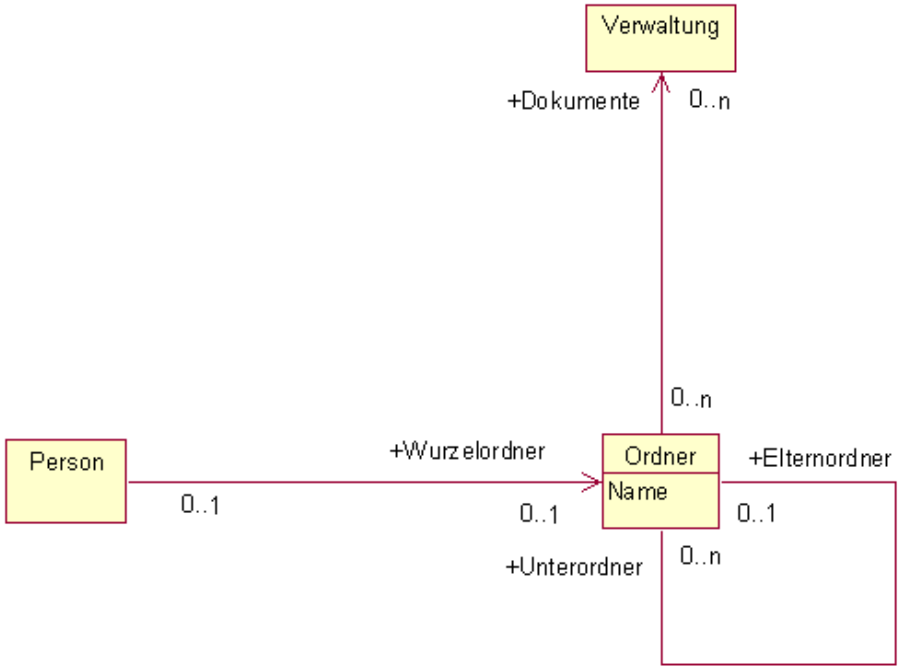

Abb. 3.2-1: Klassendiagramm der Ordnerstruktur

Die Klasse `Person` verwaltet die elementaren zu einer Person gehörenden Eigenschaften. Die Eigenschaften von Autoren werden zum Großteil ebenfalls von dieser Klasse verwaltet. Aus Gründen der Übersichtlichkeit wurde bei den Klassen `Person` und `Verwaltung` auf eine Auflistung der Attribute und Operationen verzichtet. Die dargestellten Assoziationen wurden auf das Nötigste beschränkt. Zur Speicherung von Verwaltungsobjekten wurde im Modell die Klasse `Ordner` hinzugefügt. In jedem Ordner können beliebig viele Verwaltungsobjekte abgelegt werden. Dies ist im Diagramm durch die Assoziation `Dokumente` dargestellt. Jeder Ordner kann neben Verwaltungsobjekten noch weitere Unterordner enthalten, jeder Ordner kann jedoch entweder nur einen oder keinen Elternordner haben. Dies wird im Diagramm durch die reflexive Assoziation `Unterordner/Elternordner` symbolisiert. Durch diese Art der Verknüpfung der Ordner entsteht eine **Baumstruktur**. Die so gewählte Baumstruktur ermöglicht eine hierarchische Verwaltung von Dokumenten. Weiterhin ist es notwendig, dass eine Person

Zugriff auf die Dokumentstruktur hat. Die Person benötigt dazu lediglich Zugriff auf das Wurzelelement der Baumstruktur. Dies wird durch die Assoziation `Wurzelordner` beschrieben. Über die vom Wurzelordner ausgehenden Assoziationen ist jedes Element des Baumes erreichbar.

4 Konzeption der Komponente

Dieses Kapitel beschäftigt sich damit, die Entwicklung der Komponente aufzuzeigen. Der erste Abschnitt gibt aber zunächst einen kurzen Überblick über das Thema *komponentenbasierte Softwareentwicklung* und erläutert, warum in dieser Studienarbeit die gewünschte Funktionalität in Form einer Komponente realisiert wird.

4.1 Entscheidungsgrundlage für eine Komponente

Softwarekomponenten werden häufig auch als Bausteine oder Halbfabrikate bezeichnet. Sie kapseln in der Regel eine bestimmte Funktionalität, die nach außen über Schnittstellen zur Verfügung gestellt wird. Nach /BALZERT 01/ sind wichtige Ziele, die bei der Entwicklung von Komponenten verfolgt werden, Wiederverwendbarkeit und Modularisierung. Komponenten werden meist so konzipiert, dass sie einen hohen Grad an Wiederverwendbarkeit aufweisen und in verschiedenen Kontexten eingesetzt werden können. Nach /BALZERT 98/ lässt sich durch Wiederverwendung von Software die Produktivität um den Faktor vier erhöhen. Auf diese Art und Weise kann ein Ziel schneller und damit, was in der industriellen Praxis besonders wichtig ist, zu wesentlich geringeren Kosten erreicht werden. Durch eine höhere Modularisierung eines Softwareproduktes lässt sich die Wartbarkeit entscheidend erhöhen. Für diese Studienarbeit ist der Aspekt der Wiederverwendbarkeit von entscheidender Bedeutung. Zum einen soll die zu entwickelnde Komponente natürlich möglichst universell einsetzbar sein. Zum anderen wird aber auch eine bereits existierende Komponente weiterverwendet. Dadurch konnte der Aufwand für die Entwicklung drastisch reduziert werden.

Für das Generatorsystem Janus wurde am Lehrstuhl für Softwaretechnik ein eigenes Komponentenmodell entwickelt. Es ermöglicht es, Komponenten über einen Editor grafisch mit dem Modell zu verknüpfen. Die Komponenten besitzen die Fähigkeit zur nachträglichen Anpassung. Dies geschieht über Eigenschaften. Diese Eigenschaften sind vergleichbar mit Attributen aus der objektorientierten Welt. Sie können entweder mit Hilfe des grafischen Editors auf einen festen Wert gesetzt werden oder mit einem Element aus dem Modell (z. B. dem Attribut einer bestimmten Klasse) verknüpft werden. So ist es möglich, den Wert einer Eigenschaft auf den Attributwert zu setzen. Beim Ge-

nerieren der Anwendung wird die Komponente mit in die Anwendung hineingeneriert und entsprechend den im Modell vorgenommenen Verknüpfungen an geeigneter Stelle in die Anwendung integriert. Das Janus-Komponentenmodell unterscheidet GUI-, Fachkonzept- und Business-Komponenten. Business-Komponenten bestehen aus einer GUI- und einer Fachkonzeptkomponente. Eine ausführliche Beschreibung des Janus-Komponentenmodelles würde den Rahmen dieser Studienarbeit sprengen. Es sei hier auf /ZWINTZSCHER 03-1/ verwiesen. Diese Studienarbeit beschäftigt sich mit der Entwicklung einer GUI-Komponente.

4.2 Anforderungen an die Komponente

Die zu entwickelnde Komponente soll möglichst große Ähnlichkeit mit dem *Windows-Explorer* aufweisen, da jeder Windows-Benutzer den Explorer kennt und damit umgehen kann. Damit soll erreicht werden, dass der Benutzer bekannte Strukturen wiedererkennt und die Komponente intuitiv richtig benutzen kann. Die neue Komponente soll, genau wie der Windows-Explorer, vertikal zweigeteilt sein. Auf der linken Seite soll sich eine Baumstruktur befinden, die eine Sicht auf die Ordnerstruktur enthält. Es soll dem Benutzer die Möglichkeit gegeben werden, eigene Ordner mit beliebig vielen Unterordnern anzulegen. Auf der rechten Seite soll sich eine Detailansicht befinden, die den Inhalt des Ordners, also die in ihm gespeicherten Dokumente, in Form einer Listendarstellung enthält. Über die Listenansicht soll es möglich sein, neue Dokumente anzulegen sowie vorhandene Dokumente zu bearbeiten und zu löschen. Die Funktion der Komponente lässt sich mit der des Home-Verzeichnisses unter den Betriebssystemen Linux bzw. Unix und Windows vergleichen, wo der Benutzer beliebige Unterordner anlegen und seine persönlichen Dokumente ablegen kann.

4.3 Weiterentwicklung der NodeTreeView-Komponente

Ein Teil der geforderten Funktionalität wird bereits von einer vorhandenen Komponente zur Verfügung gestellt, der *NodeTreeView*-Komponente. Für Kenner des w3l-Systems sei erwähnt, dass es sich dabei um die Komponente handelt, die auch für den Kurskonfigurator eingesetzt wird. Wenn bei der Entwicklung eines neues (Teil-)Produktes auf ein Wiederverwendungsarchiv zurückgegriffen werden kann, ist es üblich, dass zunächst überprüft wird, ob dieses Archiv etwas ent-

hält, was für das neue Produkt weiterverwendet werden kann. Diese Vorgehensweise hat *bottom-up*-Charakter, der natürlich im Gegensatz zu der gebräuchlichen *top-down*-Vorgehensweise steht. Dieser Sachverhalt ist nach /BALZERT 98/ typisch, falls bei der Softwareentwicklung auf ein Wiederverwendungsarchiv zurückgegriffen wird. Am Lehrstuhl für Softwaretechnik dient ein separates *cvs-Repository*, das ausschließlich Komponenten enthält, als Wiederverwendungsarchiv. Im Folgenden werden nun die wesentlichen Elemente der *NodeTreeView*-Komponente vorgestellt und auf eine Weiterentwicklung der Komponente eingegangen. Abbildung 4.3-1 zeigt den Aufbau dieser Komponente. Sie besteht aus zwei Teilen, einer Baumstruktur und einer Li-

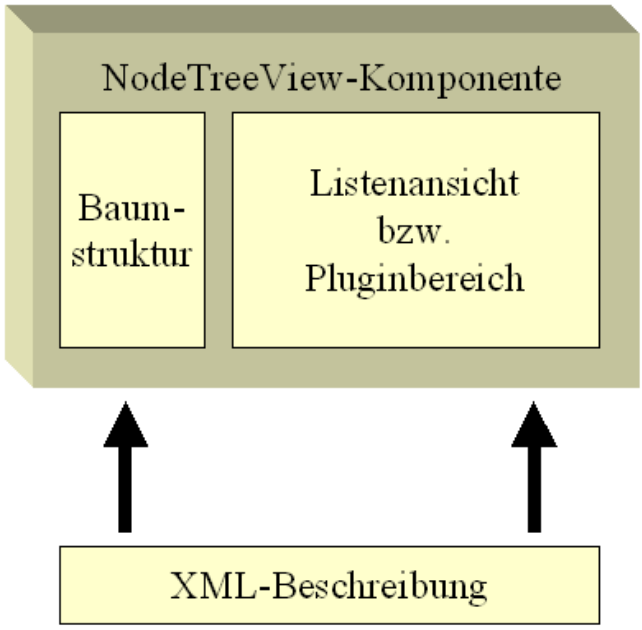

Abb. 4.3-1: Aufbau der NodeTreeView-Komponente

stenansicht bzw. einem Pluginbereich. Das Erscheinungsbild und das Verhalten der Komponente können in einer XML-Beschreibung festgelegt werden. Diese XML-Beschreibung wird der Komponente über die Eigenschaft `xmlData` übergeben. Die Struktur des Baumes ergibt sich aus dem angegebenen XML-Dokument. Standardmäßig wird neben dem Baum eine Liste angezeigt. Diese Liste enthält sämtliche Objekte ausgewählter Klassen. Die Auswahl dieser Klassen findet in der XML-Beschreibung statt und kann einem bestimmten Baumelement zugeordnet werden. Diese Liste kann für diese Studienarbeit nicht verwendet werden. Die *NodeTreeView*-Komponente bietet aber die Möglichkeit, in dem rechten Bereich eine beliebige Komponente als `Plugin` darzustellen. In diesem Fall müssen Name und Paket der Komponente in dem

XML-Dokument angegeben werden. Ein Plugin wird immer an ein Baumelement gebunden, d. h. sobald das Baumelement ausgewählt wird, wird die zugehörige Plugin-Komponente geladen und in dem rechten Bereich angezeigt. Dies bedeutet, dass für jedes Baumelement, das diesen Mechanismus nutzen soll, ein entsprechender Eintrag in der XML-Beschreibung vorgenommen werden muss. Kapitel 4.5 beschäftigt sich mit der Entwicklung einer Plugin-Komponente. Die Baumstruktur der *NodeTreeView*-Komponente kann zum größten Teil wiederverwendet werden. Mit ihr kann eine Sicht auf die Ordnerstruktur realisiert werden. Leider hat sie zwei gravierende Schwächen. Zum einen bietet diese Komponente keine Möglichkeit, Baumelemente über ein Kontextmenü neu anzulegen oder umzubenennen. Zum anderen gibt es keine Möglichkeit, das Bearbeiten des Baumes zusätzlich mit eigener, kontextabhängiger Fachlogik zu verbinden. Für das konkrete Beispiel der Ordnerstruktur bedeutet dies, dass, sobald ein Element aus der Baumstruktur entfernt wird, auch das zugehörige Ordner-Objekt (dauerhaft) gelöscht werden muss. Diese beiden Schwächen mussten eliminiert werden und zwar so, dass die Funktionalität der Komponente in ihren derzeitigen Einsatzgebieten unverändert blieb. Um dies zu gewährleisten wurde die Komponente um die boolsche Eigenschaft `customizableTree` erweitert. Der Funktionsumfang der Komponente wurde nun so erweitert, dass, falls diese Eigenschaft gesetzt ist, es Benutzern über ein Kontextmenü möglich ist, neue Baumelemente anzulegen oder vorhandene Baumelemente umzubenennen oder zu löschen. Ist diese Eigenschaft nicht gesetzt, verhält sich die Komponente so wie vor der Erweiterung. Weiterhin wurde der *NodeTreeView*-Komponente die Schnittstelle `INodeTreeCustomizer` hinzugefügt. Abbildung 4.3-2 zeigt diese Schnittstelle.

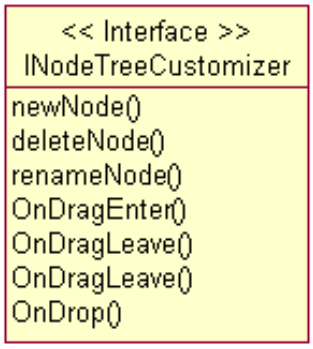

Abb. 4.3-2: Die Schnittstelle INodeTreeCustomizer

Die Schnittstelle stellt einen Satz von *Callback*-Operationen zur Verfü-

gung, die von der *NodeTreeView*-Komponente an eine Klasse, die diese Schnittstelle implementiert, delegiert werden. Auf diese Art und Weise ist es möglich, die *NodeTreeView*-Komponente um eigene Funktionalität zu ergänzen. Eine Instanz der implementierenden Klasse muss der *NodeTreeView*-Komponente natürlich bekannt gemacht werden. Zu diesem Zweck wurde die Komponente um die Operation `setNodeTreeEditor` erweitert. Alle Operationen dieser Schnittstelle werden natürlich nur aufgerufen, falls die Eigenschaft `customizableTree` gesetzt ist. Die ersten drei Operationen werden beim Erstellen, Löschen und Umbenennen eines Baumknotens aufgerufen. Die restlichen Operationen kapseln Reaktionen auf *drag & drop*-Ereignisse. Diese werden von der *NodeTreeView*-Komponente ebenfalls an die die Schnittstelle implementierende Klasse weitergeleitet. Dies ist notwendig, da die Reaktionen auf *drag & drop*-Ereignisse hochgradig abhängig von dem jeweiligen Anwendungsfall sind. Das Thema *drag & drop* wird in Kapitel 5 ausführlich behandelt werden.

4.4 Die PersonalTreeView-Komponente

Dieser Abschnitt beschäftigt sich mit der Entwicklung einer Komponente, die die im vorangegangenen Abschnitt beschriebene Schnittstelle `INodeTreeCustomizer` implementiert. Diese Komponente soll es Benutzern ermöglichen, persönliche Ordnerstrukturen in Form eines Baumes anzulegen. Aus diesem Grund wird die Komponente *PersonalTreeView* genannt. Abbildung 4.4-1 zeigt den logischen Aufbau der *PersonalTreeView*-Komponente auf Basis der beteiligten Klassen.

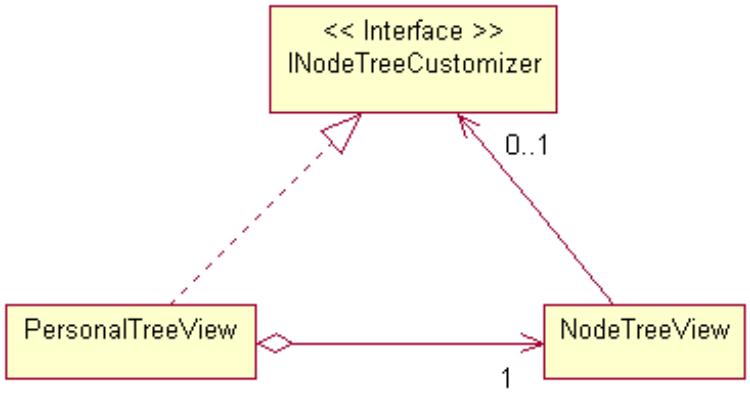

Abb. 4.4-1: Logische Architektur der PersonalTreeView-Komponente

In jeder Janus-Komponente gibt es standardmäßig eine Klasse, die den gleichen Namen trägt wie die zugehörige Komponente. So ist die Klas-

se NodeTreeView Bestandteil der *NodeTreeView*-Komponente und die Klasse PersonalTreeView Bestandteil der *PersonalTreeView*-Komponente. Zu einer Janus-Komponente gehören in der Regel mehrere Klassen. Wie bereits mehrfach erwähnt wird die beschriebene Schnittstelle von der *PersonalTreeView*-Komponente implementiert. Da diese Komponente die vollständige Baumfunktionalität nach außen zur Verfügung stellen soll, aggregiert sie die *NodeTreeView*-Komponente. Um die *Callback*-Operationen aufrufen zu können assoziiert die *NodeTreeView*-Komponente eine Instanz, die diese Schnittstelle implementiert, in diesem konkreten Fall die *PersonalTreeView*-Komponente. Abbildung 4.4-2 verdeutlicht noch einmal, wie die beiden Komponenten über die Schnittstelle INodeTreeCustomizer miteinander verknüpft sind.

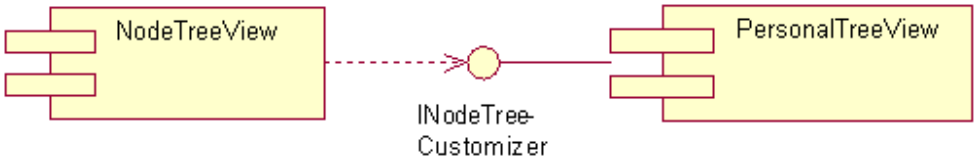

Abb. 4.4-2: Nutzung der Schnittstelle INodeTreeCustomizer

Die *NodeTreeView*-Komponente ruft die bei der *PersonalTreeView*-Komponente implementierten *Callback*-Operationen über die Schnittstelle auf.

4.5 Die Folder-Detail-View-Komponente

Dieser Abschnitt beschäftigt sich mit der Konzeption einer *Plugin*-Komponente für die *NodeTreeView*-Komponente. Diese Komponente soll die Detailansicht eines Ordners zur Verfügung stellen. Wie bereits erwähnt soll die Detailansicht den Inhalt des Ordners, also die in ihm gespeicherten Dokumente, in Form einer Listenansicht darstellen. Über die Listenansicht soll es Benutzern möglich sein, neue Dokumente zu erstellen sowie vorhandene Dokumente zu bearbeiten und zu löschen. Da diese Komponente die Detailansicht eines Ordners kapselt, wird sie *FolderDetailView*-Komponente genannt. Damit eine Komponente als Plugin eingesetzt werden kann, muss sie die Schnittstelle ITreeView-Plugin implementieren. Diese Schnittstelle enthält nur die eine Operation Init. Diese Operation wird von der *NodeTreeView*-Komponente aufgerufen, sobald der zugehörige Knoten im Baum ausgewählt und die Komponente geladen wurde. Dabei werden mit einem Parameter

verschiedene Informationen über den ausgewählten Knoten übergeben. Im Wesentlichen enthält dieser Parameter die Informationen, die auch in der XML-Beschreibung angegeben sind. Abbildung 4.5-1 verdeutlicht noch einmal die Verknüpfung der beiden Komponenten.

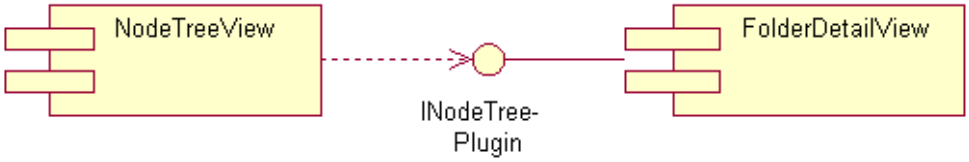

Abb. 4.5-1: Nutzung der Schnittstelle INodeTreePlugin

4.6 Gesamtüberblick

Die vorangegangenen Abschnitte haben sich mit der Neu- bzw. Weiterentwicklung von drei Einzelkomponenten beschäftigt. Alle drei Komponenten zusammen stellen die geforderte Funktionalität zur Verfügung. Es wurde zwar schon darauf eingegangen, wie die Komponenten paarweise miteinander in Verbindung stehen, es wurde aber noch nicht thematisiert, wie alle drei Komponenten miteinander verknüpft werden, um die geforderte Funktionalität zu erfüllen. Abbildung 4.6-1 zeigt die physische Architektur der Komponente.

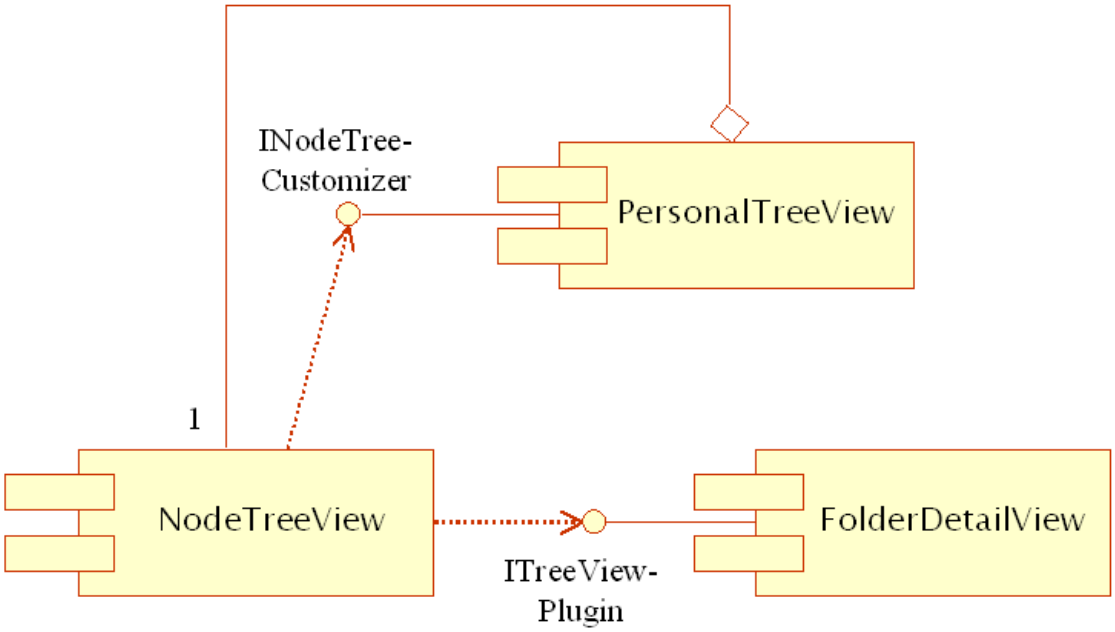

Abb. 4.6-1: Verknüpfung der beteiligten Komponenten

Das Diagramm verdeutlicht noch einmal abschließend das „Zusammenspiel“ der einzelnen Komponenten. Es sei noch einmal betont, dass bei der späteren Verwendung lediglich die *PersonalTreeView*-Komponente eingebunden werden muss. Alle anderen Komponenten müssen zwar installiert werden, die Verwendung gestaltet sich aber transparent.

5 Implementierung

Dieses Kapitel beschäftigt sich mit der Realisierung der Komponente sowie mit der Integration der Komponente in den w3l-Windows-Client. Da sich dieses Kapitel intensiv mit Programmierung befasst, werden im nächsten Abschnitt zunächst einige Details zum Janus-Generatorsystem sowie zur Verwendung der Janus-Klassenbibliothek erwähnt. Die darauffolgenden Abschnitte setzen sich dann intensiv mit der Implementierung selbst auseinander.

5.1 Die Janus-Klassenbibliothek

Mit dem Janus-Generatorsystem ist es möglich, kaufmännisch/administrative Anwendungen auf Basis eines UML-OOA-Diagrammes zu generieren. Für das w3l-Projekt wird dabei C++-Code erzeugt. Zusätzlich werden Projektdateien für die Entwicklungsumgebung *Microsoft Visual C++* erstellt, so dass sich der generierte Code direkt und ohne Hindernisse kompilieren lässt. Der generierte Code ist natürlich um eigenen Code, sog. `User Code`, erweiterbar. Allerdings kann diese Erweiterung nur an fest vorgegebenen Stellen erfolgen. Diese Stellen sind im generierten Code durch die Kommentare `//* START USER CODE` und `//* END USER CODE` gekennzeichnet. Nur an diesen gekennzeichneten Stellen vorgenommene Erweiterungen werden bei einer erneuten Generierung nicht überschrieben. Janus erstellt auch eine Datenbankanbindung, im Normalfall für eine relationale Datenbank. Dabei wird aus dem OOA-Modell direkt das Datenbankschema generiert. In dem w3l-Projekt wird das relationale Datenbanksystem MySQL eingesetzt. Zur Implementierung eigener Fachlogik steht eine eigene Laufzeitbibliothek zur Verfügung. Diese Klassenbibliothek kapselt auch den Zugriff auf die Datenbank und ermöglicht es, objektorientiert auf die relationale Datenbank zuzugreifen. Es ist nicht mehr notwendig, die Datenbank über SQL-Befehle anzusprechen. Dazu führt das Janus-System ein *Mapping* von den Tabellen der Datenbank auf Objekte der modellierten Klassen durch. Auf diese Art und Weise wird der sonst eintretende Strukturbruch zwischen objektorientierter Programmierung und Datenhaltung mit relationalen Datenbanken vermieden. Im Folgenden werden die wichtigsten Klassen der Janus-Laufzeitbibliothek vorgestellt. Dabei werden nur die Klassen berücksichtigt, die auch für diese Studienarbeit eingesetzt werden konnten. Zunächst sollen die Klassen `PDClass`, `PDObject`, `PDMeta` und `PDIterator` vorgestellt werden. Die

Abkürzung PD steht für *Problem Domain*, was sich mit Fachkonzept übersetzen lässt. Mit ihrer Hilfe kann auf den w3l-Anwendungsserver und somit auch auf den Datenbestand zugegriffen werden.

`PDObject` ist eine der wichtigsten Klassen des Janus-Laufzeitsystemes. Sie dient als Basisklasse für alle Fachkonzeptklassen. Fachkonzeptklassen sind alle Klassen, die in dem UML-OOA-Modell, das der Generierung zu Grunde liegt, enthalten sind. Diese Klasse bietet unter anderem die Möglichkeit, auf Attribute zuzugreifen, zum Herstellen und Auflösen von Beziehungen sowie dem Aufruf von Operationen. Änderungen an einem Objekt sowie Änderungen an Beziehungen zu anderen Objekten lassen sich direkt mit der Datenbank synchronisieren.

`PDIterator` Diese Klasse bietet die Möglichkeit, selektiv über den vorhandenen Datenbestand unter Ausnutzung des Iterator-Musters zu navigieren. So lassen sich zum Beispiel Iteratoren erzeugen, die es ermöglichen, (iterierend) auf alle Objekte einer Klasse zuzugreifen, die eine bestimmte Bedingung erfüllen.

`PDClass` Bei dieser Klasse handelt es sich um eine *Utility*-Klasse, deren primäres Ziel es ist, Fachkonzeptobjekte zu verwalten. Sie bietet unter anderem die Möglichkeit, neue Objekte zu erstellen sowie vorhandene zu löschen oder zu suchen. Außerdem sind in dieser Klasse einige Operationen zum Verwalten des Servers bei einer Client/Server-Anwendung definiert, zum Beispiel zum Anmelden eines Benutzers oder zum Ändern des Passwortes. Weiterhin können mit Hilfe von PDClass Iteratoren erzeugt und gelöscht werden.

`PDMeta` Diese Klasse dient zum Auslesen von Metainformationen. Sie stellt einen Satz von Operationen zur Verfügung mit deren Hilfe Informationen über das OOA-Modell der generierten Anwendung ausgelesen werden können. Es ist zum Beispiel möglich, Informationen über die Vererbungshierarchie zu erfragen. Es lässt sich einfach feststellen, ob eine bestimmte Klasse A die Oberklasse von einer anderen Klasse B ist.

Außerdem existiert eine eigene Janus-Klasse zur Verwaltung von Zeichenketten, die Klasse **`StringJ`**.

Weiterhin gibt es eine eigene Bibliothek für die Programmierung des GUI, das JAF (*Janus Application Framework*). Das JAF realisiert eine spezielle Anbindung an das Fachkonzept. Die Klassen des JAF kommen unter anderem bei der generierten Benutzeroberfläche zum Einsatz. Falls aber auch eigene GUI-Funktionalität entwickelt werden soll, kann auf diese Klassen zurückgegriffen werden. Es sei noch erwähnt, dass die derzeitige Version des JAF auf der MFC (*Microsoft Foundation Classes*) aufsetzt. Für diese Studienarbeit ist vor allem die Klasse **`PDDialog`** von Bedeutung. Sie dient als Basisklasse aller generierten Dialoge und stellt die grundlegenden Funktionen zur Kommunikation mit dem hinterlegten Fachkonzept zur Verfügung.

Die in diesem Abschnitt vorgestellten Klassen bieten natürlich nur einen kleinen Einblick in die Janus-Klassen- Bibliothek. Für detaillierte Informationen sei hier auf /OTRIS 03/ und /OTRIS 01/ verwiesen.

5.2 Ausgestaltung der Komponente

Bei der Entwicklung von Janus-Komponenten können die Klassen des JAF nicht eingesetzt werden, da eine komponentenseitige Verwendung der JAF-Klassen nur sehr eingeschränkt erlaubt ist. Die zu entwickelnden Komponenten wurden unter Ausnutzung des MFC-Rahmenwerkes realisiert. Bei der MFC handelt es sich um eine in C++ geschriebene Klassenbibliothek, die ein Grundgerüst für die Erstellung von Windows-Anwendungen zur Verfügung stellt. Das Gerüst definiert zum einen den Aufbau einer Anwendung und stellt zum anderen Standardimplementierungen der Benutzeroberfläche zur Verfügung. Die MFC selbst basiert auf der Windows-API, die auch in MFC-Anwendungen verwendet werden kann. Informationen zur MFC und der Windows-API können /MICROSOFT 98/ und /KRUGLINSKI 98/ entnommen werden. Für die Realisierung des GUI der *FolderDetailView*-Komponente kam unter anderem die MFC-Klasse `CListCtrl` zum Einsatz. Die NodeTreeView-Komponente basiert im Wesentlichen auf der MFC-Klasse `CTreeCtrl`. Die Realisierung von Baum- und Listenansichten sind Standardprobleme, die mit der MFC gelöst werden können. In der angegebenen Literatur wird diese Thematik ausführlich behandelt. Im weiteren Verlauf sollen nur die Implementierungen erläutert werden, bei denen von solchen Standard-Problemlösungen abgewichen wird.

5.3 Zugriff auf den w3l-Server

Aus den Janus-Komponenten kann nicht direkt auf das Janus-Laufzeitsystem zugegriffen werden. Der Zugriff ist nur über eine Zugriffsschicht möglich. Diese Zugriffsschicht heißt **PDAccess**. Aufrufe an diese Zugriffsschicht werden an die darunterliegende Laufzeitumgebung delegiert. Die Benutzung der Klassen und Schnittstellen dieser Zugriffsschicht gestaltet sich nahezu identisch mit der der Laufzeitumgebung, so dass auf diese Zugriffsschicht nicht näher eingegangen wird. Diese Thematik wird in /ZWINTZSCHER 03-1/ und /ZWINTZSCHER 03-2/ ausführlich behandelt.

Es wurde eine Klasse entwickelt, die den Zugriff auf den w3l-Server kapselt und die benötigte Fachlogik implementiert. Diese Klasse kann sowohl zum Verwalten der Ordner selbst als auch zum Verwalten des Ordner-Inhaltes eingesetzt werden. Sie ist also für das Verwalten eines persönlichen Arbeitsbereiches (bestehend aus Ordner und zugehörigem Inhalt) zuständig. Aus diesem Grund wird diese Klasse `PersonalWorkspace` genannt. Da die Klasse für die Organisation des gesamten Arbeitsbereiches zuständig ist, wird sie sowohl in der *PersonalTreeView*-Komponente als auch in der *FolderDetailView*-Komponente eingesetzt. Abbildung 5.3-1 zeigt diese Klasse.

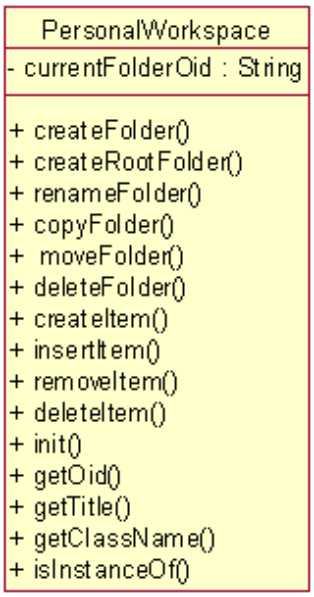

Abb. 5.3-1: Die Klasse PersonalWorkspace

Die Implementierung dieser Klasse steht im engen Zusammenhang mit dem Modell aus Abbildung 3.2-1. Das Attribut `currentFolderOid` spei-

chert die Oid des Ordner-Objektes von dem Ordner, der derzeit in Bearbeitung ist. Bei der Oid handelt es sich um eine eindeutige Objektidentität, mit deren Hilfe ein Objekt jederzeit identifiziert werden kann. Sie wird vom Janus-System automatisch vergeben. Auf das Attribut kann lesend und schreibend mit Hilfe der dazugehörenden get- und set-Methoden zugegriffen werden. Die ersten sechs Operationen dienen dem Erzeugen, Löschen, Umbenennen, Kopieren und Verschieben von Ordnern. CreateRootFolder erzeugt einen neuen Wurzelordner, der zusätzlich dem Person-Objekt bekannt gemacht wird. Die Operation createItem erzeugt ein neues Verwaltungsobjekt und fügt es dem aktuellen Ordner hinzu, die Operation deleteItem löscht ein Verwaltungsobjekt. InsertItem ermöglicht es, ein bereits vorhandenes Verwaltungsobjekt dem aktuellen Ordner hinzuzufügen. RemoveItem entfernt ein Objekt aus einem Ordner, ohne das Objekt aus dem Datenbestand zu löschen. Das Objekt wird dann nicht mehr von der Komponente verwaltet. Um auf das Objekt zuzugreifen, muss dann die Janus-Listenansicht oder die Janus-Baumstruktur (vgl. Abbildung 2.1-1) herangezogen werden. Init dient dem Initialisieren einer Instanz dieser Klasse. Über diese Operation werden der Klasse unter anderem Zeiger auf ein PDClass-und ein PDMeta-Objekt übergeben. Die Verwendung dieser Klassen wird später noch an einem Beispiel verdeutlicht. Auf die restlichen Operationen wird bei Bedarf eingegangen. Zum besseren Verständnis der Arbeitsweise dieser Klasse sollen im Folgenden beispielhaft für diese Klasse ausgewählte Codeausschnitte vorgestellt werden. Abbildung 5.3-2 zeigt die Implementierung der Operation createRootFolder. Diese Operation erhält als Parameter einen Zeiger auf das Person-Objekt, für die der Wurzelordner angelegt werden soll und eine Zeichenkette, die den Namen des neuen Ordners beinhaltet. In den ersten Zeilen wird zunächst ein Transaktionsobjekt (pdoOrdnerTrans) und anschließend ein Datenbankobjekt (pdoOrdner) von der Klasse Ordner erzeugt. Transaktionsobjekte sind Hilfsobjekte, die der Transaktionssteuerung dienen. Im Folgenden wird der Attributwert für den Namen beim Transaktionsobjekt gesetzt. Im Anschluss wird mit connect() unter Angabe des Assoziationsnamen (Wurzelordner) die Beziehung zwischen der Person und dem Wurzelordner hergestellt. Nachdem alle Änderungen vorgenommen wurden muss die Transaktion abgeschlossen werden. Dies geschieht mit commitAndDelete. Dabei werden alle Modifikationen am Transaktionsobjekt in das Datenbankobjekt übernommen. Im Anschluss wird sofort das Transaktionsobjekt gelöscht. Falls beim Anlegen der Objekte

```
int PersonalWorkspace::
    createRootFolder(PDAccess::PDObject *pdoPerson, const char *name)
{
  int error = 0;
  //
  // Transaktionsobjekt erzeugen...
  //
  PDAccess::PDObject *pdoOrdnerTrans;
  pdoOrdnerTrans = pdClass->newObject("Ordner", error, true, true);
  if(error != 0)
    return error;
  //
  // Datenbankobjekt erzeugen...
  //
  PDAccess::PDObject *pdoOrdner;
  pdoOrdner = pdClass->newObject("Ordner", error, false, false);
  if (pdoOrdnerTrans && pdoOrdner)
  {
    //
    // Bezeichner des Wurzelordners setzen...
    //
    pdoOrdnerTrans->setAttribute("Name",name);
    //
    // Beziehung zum Person-Objekt herstellen...
    //
    int res = pdoPerson->connect("Wurzelordner", pdoOrdner);
    if ( res  < 0)
      error = res;

    pdClass->commitAndDelete (pdoOrdnerTrans,pdoOrdner);
    pdClass->sync();

    pdoOrder->release();
  }
  return error;
}
```

Abb. 5.3-2: Codebeispiel - Die Operation createRootFolder

ein Fehler aufgetreten sein sollte, so gibt diese Operation einen Fehlercode zurück, im Normalfall wird der Wert 0 zurückgegeben. Wie auf den Fehler reagiert wird, wird der darüberliegenden Schicht überlassen. Die Implementierung der Operation `createFolder` gestaltet sich analog. Allerdings wird dort das neu erzeugte Ordnerobjekt nicht mit einem Personobjekt konnektiert, sondern mit dem Elternordnerobjekt. Beim Hinzufügen von Verwaltungsobjekten zu einem Ordner kommt ebenfalls dieser Mechanismus zum Einsatz. Der Codeausschnitt aus Abbildung 5.3-3 zeigt die technische Umsetzung in der Operation `insertItem`. Die Operation `ptr` ermöglicht das gezielte Suchen nach Objekten auf Basis der Oid. Die Operation gibt, falls vorhanden, einen Zeiger auf das mit Oid angegebene Objekt zurück. In einem letzten Codeausschnitt dieser Klasse soll noch die Operation `isInstanceOf` vorgestellt werden (vgl. Abbildung 5.3-4). Diese Operation überprüft, ob das Objekt mit der angegebenen Oid eine Instanz einer angegebenen Klasse (hier Klasse Verwaltung) bzw. einer Unterklasse der angege-

```
.
.
.
//
// Das Ordner-Objekt holen...
//
PDAccess::PDObject *pdoOrdner;
pdoOrdner = (PDAccess::PDObject*) pdClass->ptr(currentFolderOid);
//
// Das Verwaltungs-Objekt holen...
//
PDAccess::PDObject *pdoVerwaltung;
pdoVerwaltung = (PDAccess::PDObject*) pdClass->ptr(verwaltungOid);
.
.
.
//
// Beziehung herstellen...
//
pdoOrdner->connect("Dokumente", pdoVerwaltung);
.
.
.
```

Abb. 5.3-3: Codebeispiel - Hinzufügen von Verwaltungsobjekten

benen Klasse ist. In Janus gibt es zu jeder Klasse eine numerische Id. Die zu einem Klassennamen gehörende Id kann mit Hilfe der Operation `getId()` aus `PDMeta` ermittelt werden. Über `GetBaseId()` aus der Klasse `PDMeta` kann die Id der Basisklasse ermittelt werden. Durch Vergleich der numerischen Ids lässt sich so feststellen, ob das Objekt eine Instanz einer Unterklasse von Verwaltung ist. Es muss hier die gesamte Vererbungslinie berücksichtigt werden. Aus diesem Grund gestaltet sich der Aufruf der Operation `isInstanceOf(int baseClassId, int classId)` rekursiv.

5.4 Aggregation der NodeTreeView-Komponente

Aus Abbildung 4.6-1 geht hervor, dass die *PersonalTreeView*-Komponente die *NodeTreeView*-Komponente aggregiert. In diesem Abschnitt soll nun die technische Realisierung erläutert werden. Die Klasse `MfcAdapter` (vgl. /ZWINTZSCHER 03-2/) ermöglicht es, Komponenten ineinander einzubetten. Das Beispiel aus Abbildung 5.4-1 zeigt die Verwendung der Klasse. Bei `m_MfcAdapter` handelt es sich um ein Attribut der Klasse NodeTreeView. Mit `create` wird die angegebene Komponente geladen und angezeigt. Die Operaton `publishProperty` veröffentlicht die Eigenschaft einer eingebetteten Komponente. Dadurch wird diese Eigenschaft wie eine Eigenschaft der einbettenden Komponente behandelt. Für den Benutzer der Komponente entsteht so der Eindruck, dass die Eigenschaften `xmlData` und `customizableTree` zur *PersonalTree-*

```
bool PersonalWorkspace::isInstanceOf(StringJ className, StringJ oid)
{
  PDAccess::PDObject *pdo =  (PDAccess::PDObject*) pdClass->ptr(oid);
  if (pdo)
  {
    int id = pdo->GetId();
    pdo->release();
    int baseClassId = pdMeta->getId(className);
    return isInstanceOf(baseClassId,id);
  }
  return false;
}

bool PersonalWorkspace::isInstanceOf(int baseClassId, int classId)
{
  if(classId == baseClassId)
    return true;

  int directUpperClassId = pdMeta->getBaseId(classId);

  if (classId != directUpperClass)
    return isInstanceOf(baseClassId,directUpperClass);

  return false;
}
```

Abb. 5.3-4: Codebeispiel - Die Operation isInstanceOf

```
.
.
.
m_MfcAdapter.setControlAdapter((jc::JCControlAdapter*)getAdapter());
if (!m_MfcAdapter.create("swt", "NodeTreeView", this))
{
  AfxMessageBox("Die NodeTreeView-Komponente konnte
                              nicht geladen werden!");
}
//
// Wichtige Eigenschaften der NodeTreeView-Komponente
// nach aussen zur Verfuegung stellen...
//
m_MfcAdapter.publishProperty("xmlData");
m_MfcAdapter.publishProperty("customizableTree");
.
.
.
```

Abb. 5.4-1: Codebeispiel - Aggregation der NTV-Komponente

View- Komponente gehören. Der Tatbestand, dass es eine eingebettete Komponente gibt und dass die beiden genannten Eigenschaften eigentlich zu der eingebetteten Komponente gehören, bleibt für den Benutzer verborgen.

5.5 Integration der Komponente in die Anwendung

Im Regelfall werden die Janus-Komponenten über einen grafischen Editor mit dem Modell verküpft. Beim Generieren der Anwendung wird die Komponente mit in die Anwendung hineingeneriert und entsprechend den im Modell vorgenommenen Verknüpfungen in einem der generierten **Erfassungsdialoge** an geeigneter Stelle platziert. Auf diese Art und Weise können die Standard-Janus-GUI-Elemente durch eigene Komponenten ersetzt werden. Allerdings ist die eingebettete Komponente dann nur (lokal) in dem einbettenden Dialog verfügbar. Das ist für den Großteil der Anwendungsfälle eine sehr komfortable Möglichkeit, die generierten Erfassungsdialoge nach eigenen Wünschen zu gestalten. Die *PersonalTreeView*-Komponente kann natürlich auf diese Art und Weise auch in einen Erfassungsdialog eingebettet werden. Dies ist aber nicht optimale Platzierung der Komponente. Da sie zum Verwalten von persönlichen Dokumenten dient, sollte man möglichst schnell und einfach auf sie zugreifen können. Daher sollte sie an einer globaleren Stelle in die Anwendung integriert werden und nicht an den Erfassungsdialog einer bestimmten Klasse gebunden sein. Für das Platzieren der Komponente in dem w3l-Windows-Client existieren im Wesentlichen zwei Möglichkeiten. Die erste Alternative ist, die Komponente in einem externen Fenster darzustellen. Dies hat den Vorteil, dass die Handhabung des Hauptfensters unverändert bleibt. Nachteilig ist, dass ein Anwender dann immer ein zweites Fenster öffnen muss, um mit der Komponente arbeiten zu können. Die andere Alternative besteht darin, die Komponente in das Hauptfenster zu integrieren. Dieser Lösungsweg hat den Vorteil, dass der Anwender nur mit einem Fenster arbeiten muss. Für beide Alternativen gibt es Vor- und Nachteile. Es ist anzunehmen, dass jeder Autor diesbezüglich seine eigenen Vorlieben hat. Aus diesem Grund wurden beide Möglichkeiten realisiert. Dazu wurde ein neuer Menüeintrag in den w3l-Windows-Client (**`Personalisierung`**) aufgenommen. Über ihn ist es sowohl möglich, die Komponente in einem externen Fenster als auch eingebettet ins Hauptfenster darzustellen. Die Bildschirmschnappschüsse aus Abbildung 5.5-1 und 5.5-2 zeigen die Realisierung der beiden Möglichkei-

ten. Im Folgenden wird nun die technische Umsetzung beschrieben. Bei der *PersonalTreeView*-Komponente handelt es sich im Wesentlichen um ein GUI-Element, das in einen Dialog eingebettet werden muss. Bei dem Dialog kann es sich entweder um einen generierten Dialog oder um einen selbst entwickelten handeln. Zur Darstellung der Komponente wurde ein eigener Dialog erstellt. Dieser Dialog wurde in der Klasse `DlgPersonalTree` realisiert. Diese Klasse ist eine direkte Unterklasse der Klasse `PDDialog` (vgl. auch Abschnitt 5.1). Das Codebeispiel aus Abbildung 5.5-3 zeigt, wie die Komponente in den Dialog eingebettet wird. In der ersten Zeile wird eine neue Instanz des `UIAdapters` erzeugt und in dem Attribut `personalTreeView` gespeichert. Die Klasse UIAdapter übernimmt Vermittlungsaufgaben zwischen der Benutzeroberfläche und einer GUI-Komponente. Die GUI-Komponente wird im Konstruktor angegeben. Exemplare dieser Klasse können auf die gleiche Art und Weise wie GUI-Elemente des JAF verwendet werden. Es wird anstelle des JAF-GUI-Elementes die jeweils angegebene Komponente in das GUI integriert. Über den Konstruktor werden dem `UIAdapter` das einbettende GUI-Element sowie die geometrischen Maße der Komponente mitgeteilt.

Das Janus-System bietet von Hause aus die Möglichkeit, einen Dialog innerhalb des Hauptfensters darzustellen. Zu diesem Zweck stellt die Klasse `UIWorkspace`, die die Funktionalität des Hauptfensters kapselt, die Operation `setPaneWindow` zur Verfügung. Ihr kann als Parameter unter anderem ein Exemplar des `DlgPersonalTree` übergeben werden. Jeder Dialog wird aber standardmäßig in einem separaten Fenster dargestellt.

Es wurde bereits thematisiert, dass die Komponente eine XML-Beschreibung der Baumstruktur als Eingabe erwartet. Zur Speicherung dieses XML-Dokuments wurde die Klasse `Person` um das Attribut `PersonalXMLTree` erweitert. Der Attributwert von `PersonalXMLTree` dient der Komponente als Eingabe. Zu diesem Zweck wird die Eigenschaft `xmlData` mit Hilfe der Operation `connect()` mit diesem Attribut verbunden. Mit `setProperty` ist es möglich, Eigenschaften einen festen Wert zuzuweisen. Hier werden Breite und Höhe sowie die Eigenschaft `customizableTree` auf `TRUE` gesetzt.

Falls die Komponente in einem externen Fenster dargestellt werden soll, so muss dieses auch größenveränderbar sein. Der Anwender muss die geometrischen Maße des Fensters per Ziehoperation mit der Maus ändern können. Bei Janus-Dialogen ist dies jedoch nicht möglich. Das JAF selbst bietet keine Möglichkeit, Dialoge größenveränderbar zu ma-

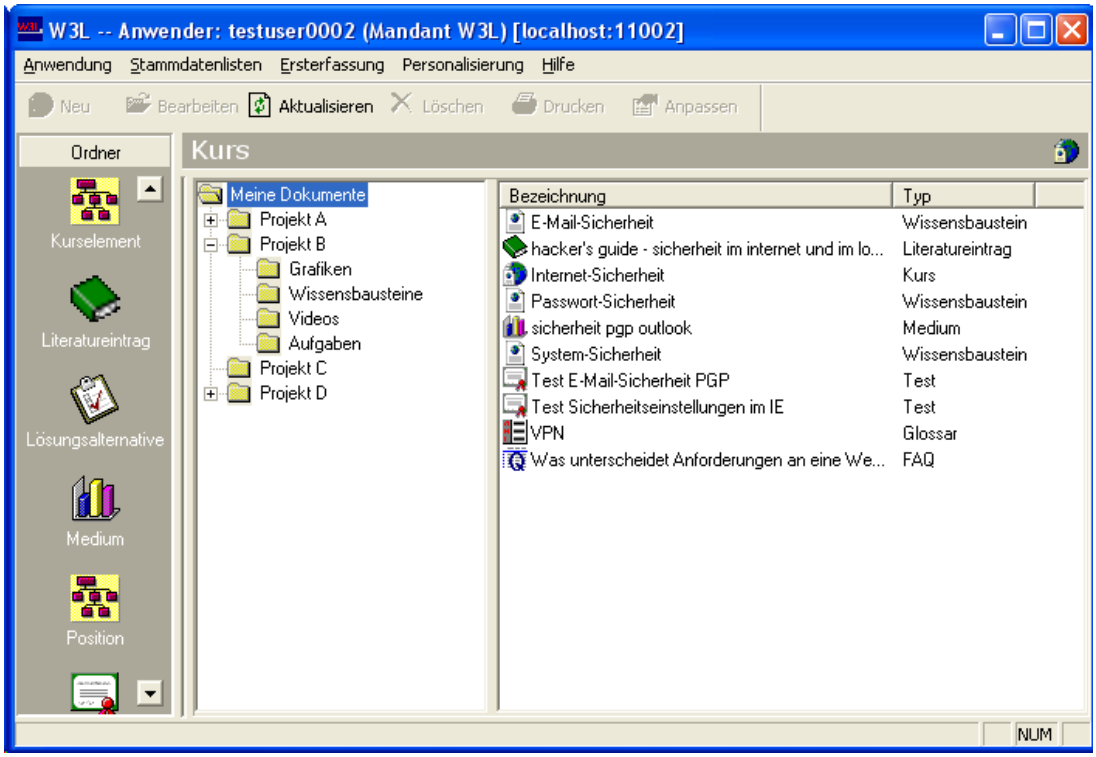

Abb. 5.5-1: Integration der Komponente in den Arbeitsbereich

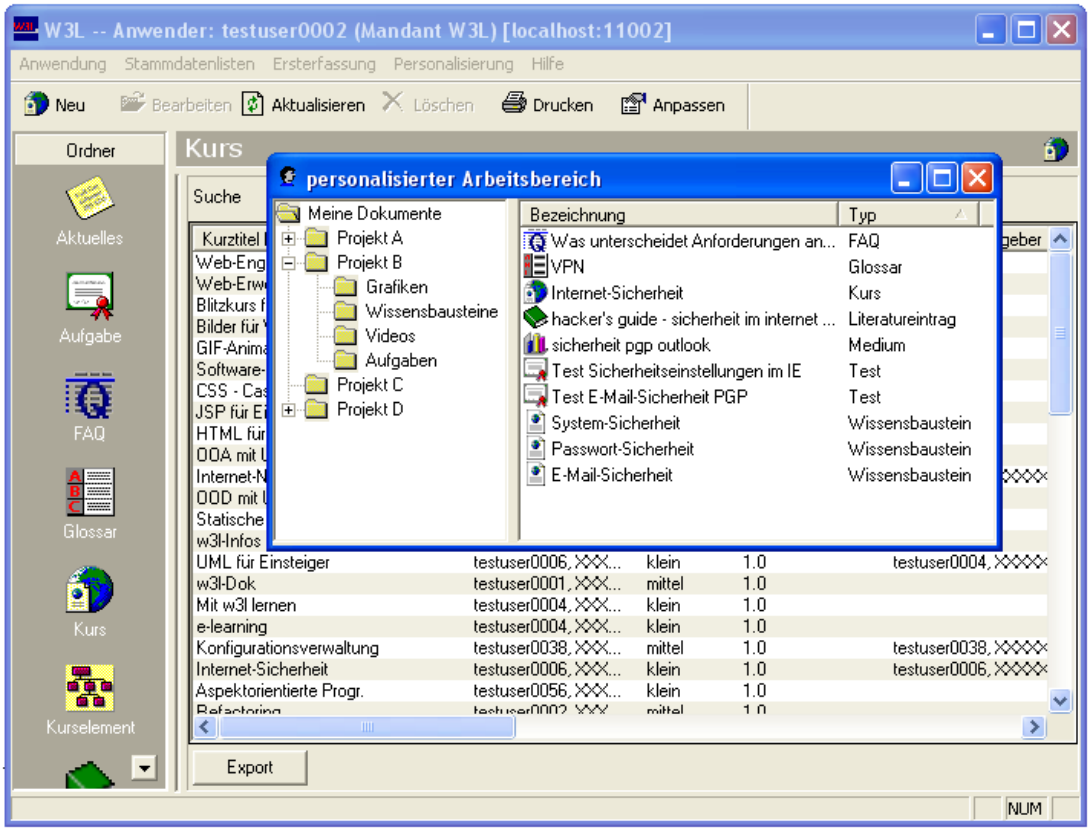

Abb. 5.5-2: Darstellung der Komponente in einem externen Fenster

```
.
.
.
personalTreeView = new UIAdapter("swt", "PersonalTreeView",
                    this, new UIRes(0,0,res->width, res->height));
personalTreeView->connect("PersonalXmlTree",
                    "xmlData", "Person::PersonalXmlTree", false);
personalTreeView->setProperty("customizableTree", "TRUE");
personalTreeView->setProperty("WIDTH", toString(res->width));
personalTreeView->setProperty("HEIGHT", toString(res->height));
.
.
.
```

Abb. 5.5-3: Codebeispiel - Einbettung der Komponente

chen. Wie bereits angesprochen, liegt dem JAF die MFC zu Grunde. Das JAF ermöglicht es aber, einen Zeiger auf das dem Dialog zu Grunde liegende MFC-Element zu ermitteln. Es kann dann das MFC-Element so modifiziert werden, dass man die Größe des Dialoges ändern kann. Abbildung 5.5-4 zeigt die technische Umsetzung. Mit `getNative` läßt sich

```
.
.
.
CWnd *wnd = (CWnd*) getNative();
DWORD style = wnd->GetStyle();
wnd->ModifyStyle(style,  WS_THICKFRAME | WS_CAPTION | WS_BORDER |
         WS_SYSMENU | WS_MAXIMIZEBOX | WS_MINIMIZEBOX,
         SWP_DRAWFRAME | SWP_NOZORDER);

style = wnd->GetExStyle();
wnd->ModifyStyleEx(style,WS_EX_APPWINDOW,SWP_DRAWFRAME|SWP_NOZORDER);

CMenu* pMenu = wnd->GetSystemMenu(FALSE);
if (pMenu)
{
  pMenu->InsertMenu(0, MF_BYPOSITION | MF_STRING,
                    SC_MINIMIZE, _T("Minimieren"));
  pMenu->InsertMenu(0, MF_BYPOSITION | MF_STRING ,
                    SC_MAXIMIZE, _T("Maximieren"));
  pMenu->InsertMenu(0, MF_BYPOSITION | MF_STRING,
                    SC_SIZE, _T("Größe ändern"));
  pMenu->InsertMenu(0, MF_BYPOSITION | MF_STRING,
                    SC_RESTORE, _T("Wiederherstellen"));
}
.
.
.
```

Abb. 5.5-4: Codebeispiel - Verwendung von getNative

das zu Grunde liegende MFC-Element ermitteln. In diesem konkreten Fall wird ein Zeiger zurückgegeben, der vom Typ `CWnd` ist. Die MFC-Klasse `CWnd` stellt Fensterfunktionalität zur Verfügung. Einige grundlegende Eigenschaften von Fenstern werden durch den Fensterstil (*window style*) beschrieben. Dieser lässt sich mit `ModifyStyle` modifizieren. Der Fensterstil `WS_THICKFRAME` ermöglicht es, dass ein Benutzer

die Fenstergröße per Ziehoperation mit der Maus manipulieren kann. Der Stil WS_CAPTION ermöglicht, einen Fenstertitel zu vergeben, WS_MINIMZE und WS_MAXIMIZE ermöglichen, dass ein Fenster minimiert bzw. maximiert werden kann. Erweiterte Eigenschaften können durch den erweiterten Fensterstil (*extended window style*) beeinflusst werden. Hier wurde dem Fenster mit ModifyStyleEx der Stil WS_EX_APPWINDOW hinzugefügt. Dadurch wird für das Fenster ein eigener Eintrag in der Windows-Taskleiste vorgenommen. Weiterhin muss das Systemmenü des Fensters angepasst werden. Dabei handelt es sich um das Menü, das sich bei den meisten Fenstern in der oberen linken Ecke befindet. In der MFC kapselt die Klasse CMenu die Menüfunktionalität. Die Operation GetSystemMenu ermittelt einen Zeiger auf das CMenu-Objekt, das das Systemmenü des Fensters beschreibt. Diesem Menü müssen die Einträge für die Größenveränderbarkeit hinzugefügt werden. Dies ist zwingend notwendig, da ansonsten die Modifikationen des Fensterstiles ohne Wirkung bleiben. Diese Lösung hat den Nachteil, dass sie zwingend voraussetzt, dass das JAF auf der MFC aufsetzt. Wenn in einer neuen Version des JAF eine andere GUI-Bibliothek als die MFC zum Einsatz kommt, ist diese Lösung nicht mehr funktionsfähig. Allerdings konnte keine Alternative zu dem gewählten Lösungsweg gefunden werden.
Zuletzt soll in diesem Abschnitt noch kurz umrissen werden, wie die in dem Attribut PersonalXmlTree der Klasse Person gespeicherte XML-Beschreibung erstellt wird. Bei diesem Attribut handelt es sich um ein abgeleitetes Attribut. Der Attributwert eines abgeleiteten Attributes wird jedes Mal, wenn auf das Attribut zugegriffen wird, neu bestimmt. Jedes Mal wenn auf das Attribut PersonalXmlTree zugegriffen wird, wird die vorhandene Ordner- und Dokumentstruktur analysiert und in eine XML-Beschreibung überführt. Zu diesem Zweck wurde eigens die Klasse PersonalTreeParser entwickelt.

5.6 Fehlerbehandlung

Beim Arbeiten mit der Komponente kann es vorkommen, dass Fehler auftreten. So ist es zum Beispiel möglich, dass ein Autor versucht, ein Objekt zu löschen, das er auf Grund fehlender Berechtigungen nicht löschen darf. Dem Benutzer muss dieser Sachverhalt mitgeteilt werden. Zum Löschen von Verwaltungsobjekten steht die Operation deleteItem der Klasse PersonalWorkspace zur Verfügung. Diese Operation gibt im fehlerfreien Fall den numerischen Wert 0 zurück, im Fehler-

fall einen Fehlercode. Die Einheit, die diese Klasse verwendet, muss nun den Fehler erkennen, die zu dem Fehlercode gehörende Fehlermeldung ermitteln und diese dem Benutzer mitteilen. Abbildung 5.6-1 zeigt, wie die Fehlerbehandlung beim Löschen eines Verwaltungsobjektes von der *FolderDetailView*-Komponente durchgeführt wird. Mit Hilfe von `errorMessage()` aus `PDMeta` ist es möglich, die zu einem Fehlercode gehörende Fehlermeldung zu ermitteln. `AfxMessage` zeigt

```
.
.
.
int errorCode = theWorkspace.deleteItem(theItem);
if (errorCode != 0)
{
//
// Fehlermeldung ermitteln und ausgeben...
//
PDAccess::PDMeta *pdMeta = getAdapter()->getPDMetaAcc();
if(pdMeta)
{
  CString errorMessage = pdMeta->errorMessage(errorCode);
  CString msg = "Beim Löschen des Objektes ist ein Fehler
                         aufgetreten.\n\nFehlergrund:\n";
  msg = msg + errorMessage;
  AfxMessageBox(msg);
}
.
.
.
```

Abb. 5.6-1: Codebeispiel - Fehlerbehandlung

einen Meldungsdialog mit dem angegebenen Text an. Abbildung 5.6-2 zeigt eine solche Fehlermeldung. In diesem Fall wurde versucht, ein Objekt zu löschen, für das nur Leserechte vorhanden waren und somit das Löschen nicht gestattet war. Nach dem gleichen Prinzip wird an sämtlichen Stellen die Fehlerbehandlung durchgeführt.

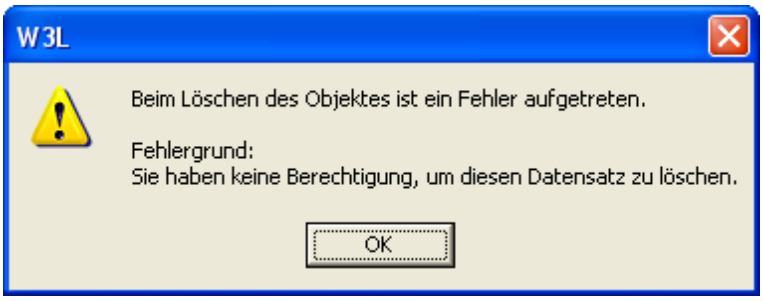

Abb. 5.6-2: Fehlermeldung

5.7 Datenaustausch der Komponente mit dem w3l-Windows-Client

Mit der Komponente sollen auch vorhandene Objekte verwaltet werden können. Um dies zu ermöglichen, ist es notwendig, dass sie der Komponente bekannt gemacht werden. Es muss also ein Datenaustausch zwischen der Komponente und dem w3l-Windows-Client stattfinden. Dazu wurde der Datenaustausch mit Hilfe von *drag & drop* realisiert. So wurde es ermöglicht, dass Benutzer vorhandene Dokumente aus der Baumstruktur oder der Listenansicht per Ziehoperation mit der Maus einem Ordner hinzugefügt werden können. Dabei können die gedraggten Objekte sowohl in der Detailansicht als auch auf einem Ordner abgelegt (*gedropped*) werden. Das entsprechende Objekt wird dann diesem Ordner hinzugefügt und in die Verwaltungsstruktur integriert. Abbildung 5.7-1 zeigt mögliche Quellen und Senken für den Datenaustausch mittels *drag & drop*. Der *drag & drop*-Vorgang wird also in einer

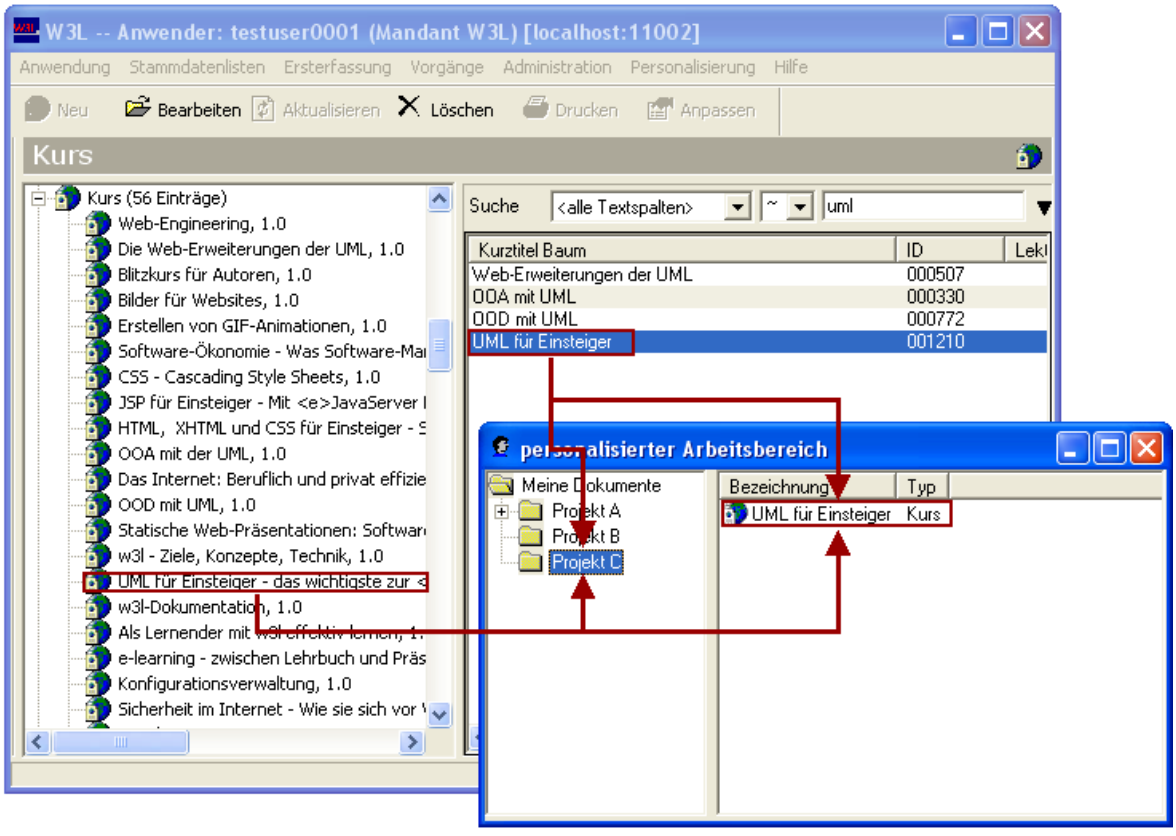

Abb. 5.7-1: Datenaustausch mit dem w3l-Client

mit Janus generierten Anwendung gestartet und endet in einer Janus-Komponente. Bei einer mit Janus generierten Anwendung kommt als GUI-Bibliothek das JAF zum Einsatz, bei einer Janus-Komponente die

MFC. Beide handhaben *drag & drop*-Vorgänge auf unterschiedliche Art und Weise. Zur Realisierung des *drag & drop* mussten beide Mechanismen kombiniert werden.

Auf der Janus-Seite müssen GUI-Elemente zunächst einem Ereignisab-

```
void LstKurs::itemDragged (UIDragDropEvent &event)
{
  UIObject *dragSource = event.getSource();
  if(event.getID() == UIEvent::DRAG_START)
  {
   PDTable *theTable = (PDTable*) dragSource;
   PDObject *pdo = theTable->getSelectedObject();
   if(pdo)
   {
     //
     // OID des ausgewaehlten Listenelementes ermitteln...
     //
     OID        oid = pdo->GetPDObjectId();
     dragOid = toString(oid);
     //
     // Handle auf den globalen Speicherblock ermitteln, in dem die
     // Drag & Drop-Informationen gespeichert werden...
     //
     HGLOBAL hgData = GlobalAlloc(GPTR,sizeof(DragDropInfo));
     ASSERT(hgData!=NULL);
     //
     // Speicherblock sperren und Zeiger auf Beginn
     // des Speichers ermitteln...
     //
     DragDropInfo *dragDropInfo = (DragDropInfo*) GlobalLock(hgData);
     ASSERT(dragDropInfo!=NULL);
     //
     // Sperre des Speicherblockes aufheben...
     //
     GlobalUnlock(hgData);
     //
     // Informationen der Drag-Quelle setzen...
     //
     dragDropInfo->theOid    = dragOid;
     dragDropInfo->ctrlType  = DRAGSRC_JANUS_LIST_DIALOG;
     dragDropInfo->srcWin    = getNative();
     //
     // Daten speichern...
     //
     COleDataSource dataSrc;
     dataSrc.CacheGlobalData(CF_TEXT, hgData);
     //
     // Drag & Drop-Vorgang starten...
     //
     DROPEFFECT dropEffect = dataSrc.DoDragDrop(DROPEFFECT_LINK,NULL);
   }
  }
}
```

Abb. 5.7-2: Codebeispiel - Operation itemDragged

hörer (`DragDropListener`) bekannt gemacht werden. Zu diesem Zweck stellen *drag & drop*-fähige GUI-Elemente die Operation `addDragDrop-Listener` zur Verfügung. Bestandteil eines `DragDropListener` ist die Funktion `itemDragged`. Diese Operation wird beim Start eines *drag & drop*-Vorganges bei allen registrierten `DragDropListenern` aufge-

rufen. Diese Operation dient dazu, den Vorgang vorzubereiten. Dazu müssen zunächst für den Austausch relevante Daten an zentraler Stelle gespeichert werden. Wenn der Vorgang abgeschlossen wird, können diese Daten weiterverwendet werden. Für den Datenaustausch kamen die dafür vorgesehenen MFC-Mechanismen zum Einsatz. Abbildung 5.7-2 zeigt die Implementierung der Operation `itemDragged` aus der Klassse `LstKurs`. Diese Klasse stellt die Funktionalität des Janus-Listendialoges für Kurse zur Verfügung. Wenn auf einem Listeneintrag mit der Maus eine Ziehoperation eingeleitet wird, wird diese Operation aufgerufen. Hier wird mit `getSelectedObject()` zunächst das Listenelement ermittelt, das in den Vorgang involviert ist. Im Anschluss wird die Oid dieses Objektes ermittelt. Für die Speicherung der auszutauschenden Daten wurde ein eigener strukturierter Datentyp (*struct*) entwickelt (`DragDropInfo`). Dieser ermöglicht es, eine Oid als String (`theOid`), eine Konstante, die den Typ des GUI-Elementes kennzeichnet, bei dem der Vorgang gestartet wurde (`ctrlType`), und das GUI-Element, bei dem der Vorgang begonnen wurde (`srcWin`), zu speichern. Für den Datenaustausch selber wird die MFC-Klasse `COleDataSource` eingesetzt. Sie ermöglicht es, mit `cacheGlobalData` die für den Datenaustausch benötigten Daten global zur Verfügung zu stellen. Es werden hier die Daten im globalen Speicherblock (Variable `hgData` vom Typ `HGLOBAL`) übergeben. In diesem Speicherblock befinden sich die auszutauschenden Daten. Mit der letzten Anweisung in dieser Operation `doDragDrop` wird der *drag & drop*-Vorgang gestartet. Die Operation `itemDragged` wurde nicht durch das Generatorsystem erzeugt, sondern manuell im `User Code`-Bereich der Klasse `LstKurs` hinzugefügt. Im Folgenden soll nun der *Drop*-Vorgang in der *FolderDetailView*-Komponente betrachtet werden. Bei der Verarbeitung des *drop*-Ereignisses kamen nur MFC-Hilfsmittel zum Einsatz. Bestandteil der *FolderDetailView*-Komponente ist unter anderem die Klasse `CFolderOleDropTarget`. Diese Klasse ist eine direkte Unterklasse der MFC-Klasse `COleDropTarget`. Sie stellt Operationen zur Verfügung, mit denen auf *drop*-Ereignisse reagiert werden kann. Eine Instanz dieser Klasse muss zuvor das GUI-Element registrieren, das das *drop*-Ereignis verarbeiten soll. Es wird nun beschrieben, wie der *drag & drop*-Vorgang mit Hilfe der Operation `OnDrop` aus `CFolderOleDropTarget` abgeschlossen wird. `OnDrop` wird automatisch aufgerufen, wenn ein Objekt auf dem GUI-Element abgelegt wird. Die Operation erhält unter anderem einen Zeiger auf das GUI-Element, bei dem der Vorgang abgeschlossen wurde, und einen Zeiger auf ein Objekt der Klasse `CO-`

```
BOOL CFolderOleDropTarget::OnDrop(CWnd* pWnd,
     COleDataObject* pDataObject, DROPEFFECT dropEffect,
     CPoint point)
{
  if(!pDataObject->IsDataAvailable(CF_TEXT))
    return FALSE;
  //
  // Handle auf den globalen Speicherblock ermitteln, in dem die
  // Drag & Drop-Informationen gespeichert sind...
  //
  HGLOBAL   hGlobal;
  hGlobal = pDataObject->GetGlobalData(CF_TEXT);
  //
  // GlobalLock sperrt das angegebene Speicherobjekt und gibt
  // einen Pointer auf das erste Byte des Speicherblockes zurueck...
  //
  DragDropInfo* pData = (DragDropInfo*) GlobalLock(hGlobal);
  if (pData==NULL)
    return false;
  //
  // Sperre des Speicherbereiches aufheben...
  //
  GlobalUnlock(hGlobal);

  StringJ theOid = "0";
  if (pData->ctrlType == DRAGSRC_JANUS_TREE
      || pData->ctrlType == DRAGSRC_JANUS_LIST_DIALOG)
    theOid = pData->theOid;
  //
  // Einen Zeiger auf das Fenster holen....
  //
  CFolderListCtrl *listCtrl = (CFolderListCtrl*) pWnd;
  //
  // Nachricht an das Elternfenster (FolderDetailView) schicken...
  //
  listCtrl->GetParent()->SendMessage(MSG_NEWITEM, 0 ,(long) &theOid);

  return TRUE;
}
```

Abb. 5.7-3: Codebeispiel - Operation OnDrop

leDataObject. Über ihn ist der Zugriff auf die zuvor mit cacheGlobal-Data gespeicherten Daten möglich. Mit Hilfe von GetGlobalData lässt sich der globale Speicherblock ermitteln, in dem die Daten gespeichert sind. Mit GlobalLock wird im Anschluss der Speicherblock gesperrt und ein Zeiger auf diesen Speicherbereich ermittelt. Da es sich bei den zwischengespeicherten Daten um ein Objekt vom Typ DragDropIn-fo handelt, kann dieser Zeiger in einen DragDropInfo-Zeiger gecastet werden. Über diesen Zeiger kann nun auf die in dem struct gespeicherten Daten zugegriffen werden. So lässt sich die Oid des gedraggten Objektes mit theOid = pData->theOid ermitteln. Das zu dieser Oid gehörende Objekt muss nun der Ordnerstruktur hinzugefügt werden. Zu diesem Zweck wird hier die Nachricht (MSG_NEWITEM) an das übergeordnete Fenster gesendet. Die Oid wird der Nachricht als Parameter angefügt. Das übergeordnete GUI-Element, in diesem Fall die Folder-

TreeView-Klasse, kann nun auf diese Nachricht reagieren und das neue Objekt in die Ordnerstruktur integrieren. Zu diesem Zweck fügt sie das Objekt mit `theWorkspace.insterItem(theOid)` dem aktuellen Ordner hinzu. Die notwendigen Veränderungen am GUI werden ebenfalls vorgenommen. Der *drag & drop*-Vorgang und somit auch der Datenaustausch zwischen dem w3l-Windows-Client und der Komponente sind damit erfolgreich abgeschlossen. Abbildung 5.7-4 zeigt noch einmal die zeitliche Abfolge des *drag & drop*- Vorganges. Außer für den Lis-

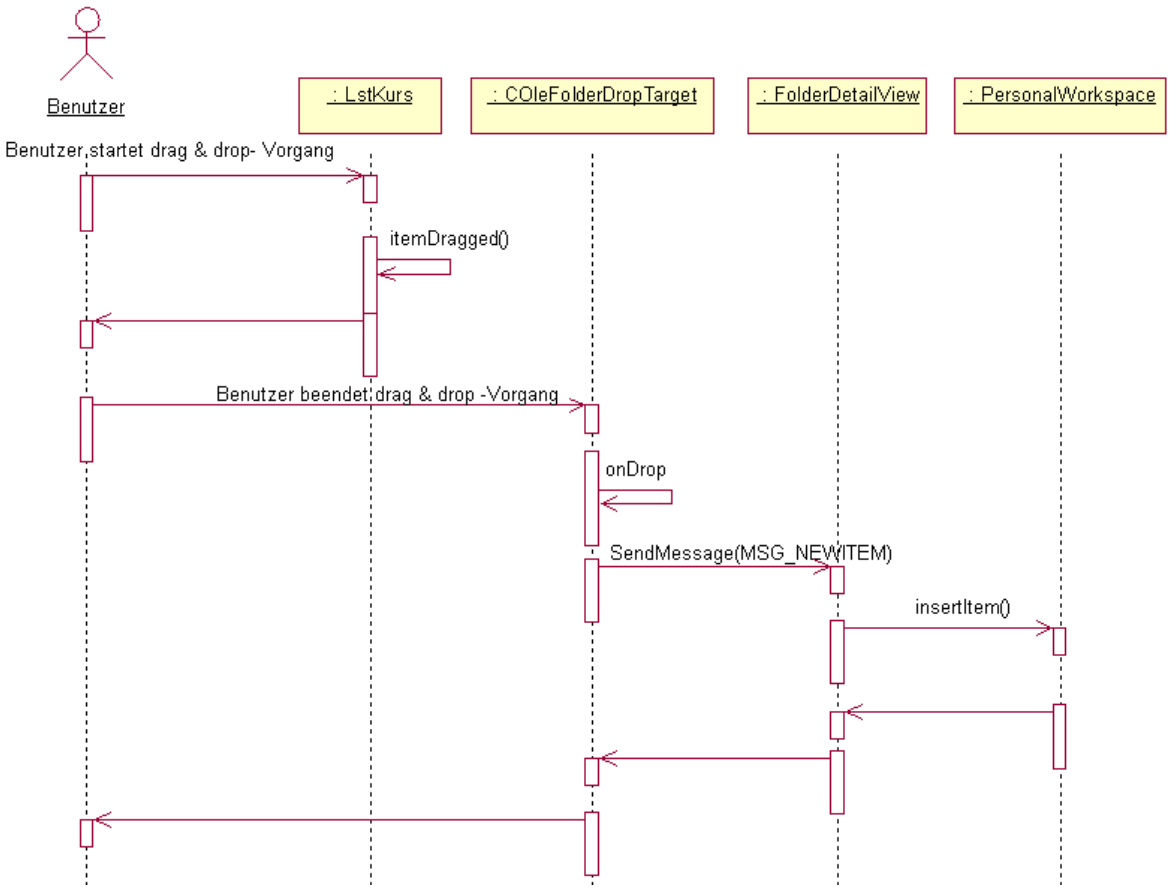

Abb. 5.7-4: Sequenzdiagramm drag & drop

tendialog `LstKurs` wurde für die Listendialoge `LstAufgabe`, `LstTest`, `Medium`, `LstLoesungsalternative`, `LstGlossarbegriff`, `LstLiteratureintrag`, `LstVerwaltung`, `LstFAQ` und `LstWissensbaustein` ebenfalls die *drag & drop*-Unterstützung auf die gleiche Art und Weise realisiert. Die *drag & drop*-Unterstützung für die Janus-Baumstruktur und die Baumstruktur der Komponente gestaltet sich analog. Neben der Möglichkeit des Datenaustausches lässt sich per *drag & drop* auch die Zusammensetzung des Baums beeinflussen. So können Ordner mit ihren Unterordnern innerhalb des Baumes beliebig kopiert und versch-

ben werden. Außerdem ist es möglich, Verwaltungsdokumente aus der Listenansicht per *drag & drop* in einen beliebigen Ordner zu kopieren oder zu verschieben. Dabei wird, wie in Windows-Anwendungen üblich, die Kopierfunktionalität durch zusätzliches Betätigen der Strg- bzw. Ctrl-Taste aktiviert. Standardmäßig werden Dokumente und Ordner bei *drag & drop*-Vorgängen verschoben.

6 Ausblick

Mit der im Rahmen dieser Studienarbeit entwickelten Komponente lassen sich Dokumente von *e-learning*-Kursen wesentlich einfacher verwalten. Es wird Autoren die Möglichkeit gegeben, ihre Dokumente hierarchisch und projektbezogen zu speichern. Dazu können sie eigene Verwaltungsstrukturen in Form von Ordnern anlegen, in denen sie Dokumente entsprechend ihren eigenen Wünschen ablegen können. Außerdem gestaltet sich das Anlegen neuer Dokumente unkomplizierter als auf die herkömmliche Art und Weise. Allerdings ist diese Komponente zur Zeit nur in der Lage, mit `Verwaltungs`-Objekten zu arbeiten. Es ist nicht möglich, beliebige Objekte zu verwalten. In einem weiteren Entwicklungszyklus könnte die Komponente so verallgemeinert werden, dass sie auch in beliebigen Kontexten einsetzbar ist. So ist es denkbar, dass die Komponente um eine zusätzliche Eigenschaft erweitert wird, in der eine Basisklasse für die zu handhabenden Objekte angegeben wird. Um das in dieser Studienarbeit beschriebene Verhalten der Komponente zu bewirken, müsste in dieser Eigenschaft die Klasse `Verwaltung` angegeben werden. Es wäre dann möglich, die Dokumente von den Klassen, die von `Verwaltung` erben (zum Beispiel `Wissensbaustein`, `Kurs` etc.), zu verwalten. Es könnte dann aber auch jede andere Klasse angegeben werden, so dass sich das Einsatzgebiet der Komponente drastisch erhöhen würde und nicht auf `Verwaltungs`-Objekte beschränkt bliebe.
Weiterhin lässt sich der Teil, der die eigentliche Fachlogik enthält - die Klasse `PersonalWorkspace` - in eine eigene Fachkonzeptkomponente auslagern. Aus dem GUI- und Fachkonzeptteil könnte dann eine *Business*-Komponente konstruiert werden. Dies trägt zu einer erhöhten Modularisierung bei. Damit ließe sich die Wartbarkeit und Flexibilität entscheidend verbessern.
Vor allem der praktische Einsatz wird zeigen, welche Anforderungen und Wünsche noch unerfüllt geblieben sind und wie sich die Weiterentwicklung der Komponente in Zukunft gestalten wird.

Abbildungsverzeichnis

Literatur

/BALZERT 01/
Balzert, Helmut, Lehrbuch der Softwaretechnik I - 2.Auflage, Spektrum Akademischer Verlag, 2001

/BALZERT 98/
Balzert, Helmut, Lehrbuch der Softwaretechnik II, Spektrum Akademischer Verlag, 1998

/KRUGLINSKI 98/
Kruglinski, David; Sheperd, George; Wingo, Scot, Inside Visual C++ 6, Microsoft Press 1998

/MICROSOFT 98/
msdn Library, Microsoft, 1998

/OTRIS 01/
oTRIs Software AG, Janus-Programmierhandbuch, 2001

/OTRIS 03/
oTRIs Software AG, Janus-API-Dokumentation, 2003

/W3L 02/
w3l GmbH, e-learning-Kurs *Blitzkurs für Autoren*, 2002

/ZWINTZSCHER 03-1/
Zwintzscher, Olaf, Komponentenbasierte und generative Softwareentwicklung - Generierung komponentenbasierter Software aus erweiterten UML-Modellen, Ruhr-Universität Bochum, Lehrstuhl für Softwaretechnik, 2003

/ZWINTZSCHER 03-2/
Zwintzscher, Olaf, Janus-Components-API-Documentation, 2003

Erklärung

Hiermit erkläre ich, dass der vorliegende Text im Rahmen einer selbstständig durchgeführten Studienarbeit am Lehrstuhl für Software-Technik der Fakultät für Elektrotechnik und Informationstechnik der Ruhr-Universität Bochum entstanden ist.

Bochum, 08.08.2003