Alles Weitere verschieben wir für diese beiden Windows ebenfalls auf später, genauso wie die Kommunikation zwischen den Fenstern, also das Aufrufen der entsprechenden Suchfenster und die Datenübergabe der ausgewählten Werte.

3.2 Ein typisches Verwaltungsfenster

Die zweite Seite unseres **4fF**-Fensters wird ein wenig individueller gehalten, denn sie bearbeitet den einzelnen Datensatz. In Kernel-Technologie würde sie ähnlich einem der gängigen Browser für das Internet aufgebaut werden, links die Übersicht der einzelnen Felder des betrachteten Datensatzes, rechts die Arbeitsfläche, in der der Inhalt eines ausgewählten Feldes angepackt werden kann. Mit Studio lässt sich die Übersicht bequem über Listen und ihre vielseitige Notation organisieren, die Arbeitsfläche wird über **Paged Panes** geregelt, die es für jeden berücksichtigten Feldtyp unterschiedlich zu geben hat.

Doch das sei nur der Vollständigkeit halber erwähnt.

Vollständigkeit ist aber im Augenblick nicht unser Bestreben, wir halten uns lieber an Einfachheit. Der Einfachheit halber führe ich Ihnen also ein Fenster der gängigen Bauart vor, für jede Datei individuell gestrickt. Doch Kernel-Technologie ließe sich nicht machen, wenn es nicht für jeden Schnittstellen-Typ Gemeinsamkeiten gäbe.

Für die Verwaltung eines einzelnen Datensatzes gibt es drei Aufgaben. Erstens die Anzeige der Auswahl, unter der dieser Datensatz aus irgendwelchen Gründen eingeordnet wird, zweitens Erklärungen, vor allem Beschriftungen für die Datenfelder, deren Verarbeitung die Anwender durchzuführen haben, sowie Beschreibungen der Inhalte, die die Anwender diesen Datenfeldern zugeteilt haben und last not least die Hilfestellung bei der Auffindung der Inhalte und die Prüfung auf die Korrektheit dieser Inhalte.

Dementsprechend sind die gängigen Verwaltungsfenster mit einem oft „Kopf“ oder „Header“ genannten Areal ausgerüstet, in dem sie die definierenden Angaben des Datensatzes darstellen, und einem Verarbeitungsteil, in dem die zu verwaltenden Felder mit ihren Beschriftungen und Hilfetexten angeboten werden. Je nach Feld stehen dort auch die Möglichkeiten zur Verfügung, mit denen der Inhalt aufgefunden oder geprüft werden kann. Damit ergibt sich eine Zweiteilung des Fensters, da die Dynamik eines Prozesses zwangsläufig schlecht auf bildhaften Oberflächen abgebildet werden kann. Man denke bloß an die sogenannten „process bars“, in denen der aktuelle Verarbeitungsfortschritt durch Laufbalken oder Ähnliches kenntlich gemacht wird, damit die Anwender sehen können, dass sich überhaupt etwas tut. Der dritte Anteil der Verwaltungsaufgabe, die Hilfestellung, wird meist über Rechte Maus oder graphische Symbole neben dem betrachteten Feld ermöglicht, historisch bedingt auch über Menü und Funktionstasten. Die Prüfung auf Korrektheit und Datenkonsistenz erfolgt automatisch nach Veränderung eines Feldinhaltes und benötigt deshalb kein graphisches Pendent außer im Fehlerfall, wo in der Regel Meldungen ausgegeben werden, die zumeist Aktivitäten von den Anwendern verlangen.

Da wir ein ganz „normales“ Verwaltungsfenster planen, halten wir uns auch an diese Bauart. Wie im Falle des Suchfensters werde ich auf alle optischen Bestandteile verzichten, um den Kern des Problems darzustellen - und um keine Geschmäcker zu verletzen. Doch wie im Falle des Suchfensters wird auch dieser Teil derart gestaltet sein, dass Sie keine Probleme haben sollten, das Fenster ganz Ihren Vorstellungen anzupassen. Unsere neuen Felder werden die Feldobjektliste wieder verändern, sodass wir die Weichenstellungen anpassen müssen.

Die Hilfestellung hinsichtlich Erläuterungen bleibt auf **TOOLTIPS** und Texten in der Statusleiste beschränkt. Das liegt aber nicht an Omnis, es bietet Ihnen sehr differenzierte Möglichkeiten, Ihren Anwendern unter die Arme zu greifen. Sehen Sie mal unter dem Stichwort „Help“ in der Hilfe nach, besonders unter **HELP PROJECT MANAGER**, in dessen Angebot auch ein **WIZARD** vorliegt. Denken Sie aber bei Hilfefunktionen immer daran, dass es Textfunktionen sind, die in allen Sprachen vorzuliegen haben, die Sie zur Verfügung stellen wollen, und dass Sie

bei der üblichen Hilfe die Dokumentenform benötigen, die meist von speziellen Software-Tools bearbeiteten werden müssen, so auch bei Studio. Außerdem sollten solche Dokumente, die den Anwendern die Bedienung Ihres Systems nahebringen sollen, nie leichtfertig erstellt werden, es bedarf einer genauso wohldurchdachten Struktur, um ein gutes Dokument zu erstellen wie bei einer guten Software-Anwendung, zumal auch hier die Aspekte „Übersichtlichkeit, Wartung und Wiederverwendbarkeit“ zu berücksichtigen sind. Dies sprengt aber den Rahmen unserer kleinen Applikation, deshalb begnügen wir uns mit **Tooltips** und Statusleisten, die die aktuelle Umgebung, in der sich unsere Anwender aufhalten, erläutern.

Was soll unser Fenster nun tun können?

Es muss die Datei und den Feldnamen des Satzes anzeigen, der hier zur Verarbeitung angeboten wird. Bei der Datei sollen aus der **Files**-Datei die Angaben über den Typ und das Typgewicht angezeigt werden, um den ausgewählten Fall näher zu beschreiben. Eine Verzweigung aus diesem Fenster in die Dateiverwaltung der Dateien benötigen wir nicht, da die Datei des aktuellen Datensatzes nicht verändert werden kann. Für das Dateifeld selbst ist eine Anzeige zusätzlicher Beschreibungen nicht erforderlich, exakt das ist schließlich unsere eigene Aufgabe - wir haben hier ja den ganzen Datensatz vor uns zur Bearbeitung. Die Verwaltung ist nur erforderlich für die Felder „Eigenschaft“, „Werte“, „Redundanz“ und „Distanzen“, die anderen Dateifelder werden daraus über die Funktionen der individuellen Methoden bestimmt.

Welche Anforderungen stellt nun das Feld „Eigenschaft“? Es muss geprüft werden, ob die für unser Dateifeld beantragte Eigenschaft in unserer Datei bereits verwendet wurde, was nicht erlaubt wäre. Darüberhinaus muss diese Eigenschaft aber auch in der **Fields**-Datei vorliegen und wenn nicht, dann mit unseren aktuellen Werten angelegt werden. Für die Prüfung, ob diese Eigenschaft bereits vorliegt und mit welchen Bedingungen, können wir die **$CheckValues** der individuellen Methoden der **Fields**-Datei verwenden und haben damit sofort die weiteren Angaben über diese Eigenschaft, die wir zur näheren Erläuterung ebenfalls anbieten wollen. Hier interessieren vor allem die Distanzen, die Verwendungen und die Typen. Doch es fehlt noch etwas - wir müssen bedenken, dass es für Dateisysteme sehr viele Eigenschaften geben kann. Also werden es unsere Anwender wohl als sehr nützlich empfinden, wenn sie Eigenschaften suchen können, um sie bei Bedarf auszuwählen oder eben neu anzulegen. Das bedeutet, dass wir eine Eigenschaftssuche auch auf dieser Seite unseres Windows benötigen.

Das Feld „Werte“ ist dagegen einfach - es nimmt nur Zahlenwerte an, die völlig aus der Beurteilung der Anwender stammen, für die wir keinerlei automatische Unterstützung anbieten können. Einzige Bedingung - sie müssen ganzzahlig und größer als Null sein.

Das Feld „Redundanz“ unterliegt strengeren Regeln, erlaubt uns jedoch eine sehr bequeme Hilfestellung, denn es enthält einen Feldnamen unserer eigenen Datei, wenn überhaupt. Redundanz eines Feldes bedeutet nichts weiter als Überflüssigkeit dieses Feldes. Überflüssig ist das Feld deshalb, weil sein Inhalt sich aus anderen Angaben einwandfrei und widerspruchslos herleiten läßt. Zumeist betrifft es Attribute von Komponentenobjekten, die bei relationalen Dateiverwaltungen in andere Dateien zwecks Bequemlichkeit oder Übersichtlichkeit mit übernommen wurden, obwohl der Identifikator des Komponentenobjektes ausreichen würde, um seine Attribute zu beschaffen.. Diesen Identifikator soll unser Feld „Redundanz“ aufnehmen, falls das aktuell bearbeitete Dateifeld tatsächlich redundant ist.

Als Hilfestellung können wir hier also bei Bedarf eine Liste all unserer eigenen Dateifelder anbieten und müssen kein freies Eingabefeld tolerieren. Damit entfallen auch weitere Prüfungen, denn ob ein Feld redundant ist oder nicht, liegt ebenfalls außerhalb unseres Einflussbereiches in der Beurteilung unserer erfahrenen Benutzer.

Für die Bestimmung der Distanzen haben wir uns schon früher auf ein ganz spezielles Verfahren festgelegt, erinnern Sie sich? Bei der Bearbeitung der Distanzwege in den individuellen Methoden setzten wir voraus, dass wir

nichts weiter als einen Vorgänger bei den Portal-Distanzen und einen Nachfolger bei den Exit-Distanzen verarbeiten müssen. Dies müssen Dateifelder sein, die aus anderen als unserer eigenen Datei stammen, die aber ihrerseits in unserer kleinen Applikation bereits als Datenbestand vorliegen müssen. Dateifelder von beliebigen Dateien, die außerhalb unseres Kontrollbereichs sind, haben für uns keine Bedeutung, für die liegen keine Angaben vor und deshalb können wir keinerlei Aussagen darüber treffen. Das heißt aber auch, dass unsere Anwender nicht jeden x-beliebigen Wert hier als Vorläufer bzw. Nachfolger angeben dürfen, es müssen Dateifelder sein, die wir bereits kennen. Hier wäre eine hierarchische Liste sehr hilfreich, die sich mit den Omnis-Menüs recht einfach erstellen lassen, wenn man berücksichtigt, dass der Menü-Inhalt bekannt sein muss, bevor das erste der hierarchischen Menüs instanziert wird. Außerdem existiert hier die Beschränkung auf zwanzig Zeilen, die sowohl in der Dateianzahl als auch in den Dateifeldern überschritten werden kann.

Doch zurück zu unseren Anwendern, die gerade einen Vorläufer bzw. Nachfolger unserem Datensatz verordnen wollen. Wenn die Auswahl eines bestimmten Datei getroffen wurde, können wir den Rest automatisch erledigen, freie Eingabe ist hier nicht erforderlich, da wir die Feldauswahl zur Auswahl anbieten können.

Alle weiteren Datenfelder dienen dann nur der Darstellung der Konsequenzen, die die Eingaben mit sich ziehen. Wir müssen also die Anzahl der Tentakel ermitteln, die sich aus unserer Eigenschaft herleitet, die Portal- und Exit-Distanz, die sich über die Portal- und Exit-Wegelisten aus den Angaben von Vorläufern und Nachfolgern als „Weglänge“ ergibt und dann die sechs Gewichte aus diesen Zahlenwerten, aus denen dann wiederum die feld- und aufgabenbezogenen Typen hergeleitet werden können. Das Feld **d_4fF_UsedInList** zeigen wir natürlich auch an, doch sein Wert wird von unserer **$NoticeMe** nur dann gesetzt, wenn das betreffende Feld selbst als Vorläufer oder Nachfolger ausgewählt worden war, Sie erinnern sich?

Als erstes kopieren wir also die drei Textfelder von der Suchseite mit **copy&paste,** denn die Beschriftung soll überall gleich aussehen und wird auch auf dieselbe Art ermittelt. Ausrichten und anordnen erfolgt wie gehabt, kennen Sie schon zur Genüge. Positionieren Sie sie erst einmal in die Mitte des Ganzen.

Die Felder, die den Dateinamen und den Feldnamen nur als Text anzeigen sollen, können wie vieles in Omnis auf mehrere Arten erzeugt werden. Sie können ein editierbares Feld nehmen und es in den „gelähmten“ Zustand „enabled“ = **kFalse** versetzen, Sie können ein normales Textfeld nehmen, das keine **Tooltips** kennt oder eines der **background objects**, die für Texte vorgesehen sind oder gar ein **String Label**, ein **background component** wie das **Wash Control**, das Methoden akzeptieren und ausführen kann wie ein Button. Das haben wir hier zwar nicht vorgesehen, ich habe freilich ein echtes Platzproblem mit den vielen Angaben, die ich bieten möchte und verzichte bei diesen puren Anzeigen damit auf Beschriftungsfelder. In Fällen, in denen mit einer unterschiedlichen Zahl von Angaben oder mit Erweiterungen des Informationsangebotes zu rechnen ist, benutze ich meist auch für solche nichtaktiven Präsentationen Listen, die als nichteditierbar und nicht selektierbar bestimmt sind und bei denen die Einstellung „active“ in Auswahl „General“ auf Nein steht, doch bei unserer individuellen Fenstergestaltung sind Einzelfelder abwechslungsreicher und optisch interessanter gestaltbar. Werden Listen nämlich mit der Aufbereitung „effect“ = **kFlat** eingestellt, Auswahl „Appearance“, so wirken sie zwar nicht aufdringlich, lassen sich flexibel aufbereiten und geben darüberhinaus noch die Möglichkeit der Scrollbalken, doch sie sehen eben meist nur wie eine Aneinanderreihung von Texten aus, auch wenn ich dort natürlich jede Menge erklärender Beschreibungen unterbringen kann.

Aus Platzmangel verzichte ich also auf die erläuternde Beschriftung bei den automatisch errechneten Feldern, damit komme ich aber in Erklärungsnotstand bei solchen Anwendern, die noch keine Übung mit dem Fenster oder der Methode haben. Als Ausweg bieten sich die raumsparenden Icons und **Tooltips** an. Ich könnte also Buttons verwenden, deren Events ich unberücksichtig lasse oder auch editierbare Felder, die ich wie meine Listen auf „enabled“ = **kFalse** setze, damit ich **Tooltips** verwenden kann. Da ich dieses Problem jedoch auch bei den edi-

tierbaren Datenfeldern der **4fF**-Datei habe, die ich ebenso deutlich und aussagekräftig wie die Angaben der definierenden Datei und den Namen des Datenfeldes aufbereiten möchte, verschiebe ich die Problematik auf später und mache mich erst einmal an den wesentlichen Teil - die Aktivitäten der Anwender.

Bei den Eigenschaften benötige ich eine Möglichkeit für die Anwender, Daten ganz variabel einzugeben. Das klingt zwar wie ein normaler Eingabevorgang, doch habe ich zusätzlich eine Suche über die Datei aller Eigenschaften vorgesehen. An dieser Stelle möchte ich das spezifische Fenster für diese Datei freilich nicht verwenden, sondern nur die Suchfunktion der Datei-Task der Eigenschaften. Ideal für solche Fälle ist eine **Combo Box**, sie lässt neue Eingaben zu und listet die vorhandenen auf. Wenn dann die nächste Aktivität durchgeführt wird, schließt die **Combo Box** automatisch wieder ihre Übersicht. Das ist wunderbar im Normalfall, doch ich möchte eben zulassen, dass die Übersicht auch dann noch gezeigt wird, wenn die Anwender schnell etwas nachsehen wollen. Also arbeite ich nur mit einem Eingabefeld. Ich muss die Auswahl schließlich prüfen und wenn ich keine Eigenschaft dieses Namens finde, dann begebe ich mich mit dem gewünschten Text auf die Suche. Das Ergebnis soll in einer Liste angezeigt werden, da die Anzahl der gefundenen Sätze erneut zu groß für **POPUP**-Menüs oder für **POPUP**-Listen sein können, die letztendlich durch die Größe des Bildschirms begrenzt werden. Somit kommen diese hübschen Feldobjekte leider wieder nicht in Frage. Neben meinem Eingabefeld pflanze ich also eine **List Box** ein, die nicht editierbar, „enabled„ = **kFalse**, und außerdem unsichtbar ist, „visible„ = **kFalse**, denn die Sichtbarkeit will ich nur bei Bedarf haben. Die Anpassung an die Fenstergröße setze ich auf **kEFleftRight**.

Das Nächste Feld ist „Werte“, ein einfaches Editierfeld, auch mit einer Beschriftung. „Redundanz“ verlangt ebenso einen Klartext, ich muss aber keine freien Eingaben ermöglichen. Hier verwende ich eine **Dropdown List**, obwohl mir eine **Popup List** wesentlich besser gefällt, die sich sehr ähnlich verhält. Doch ähnlich wie schon in vorherigen Überlegungen ist die Anzahl der Fälle, die hier angezeigt werden sollen, möglicherweise größer als es eine **POPUP**-Anzeige erlaubt. Wir haben schließlich ca. 20 Felder pro Datei angenommen. Sie könnten natürlich auch abhängig von der anzeigenden Anzahl einmal eine **Popup List** und einmal eine **Dropdown List** für genau denselben Job benutzen. Das ist gar kein Problem in Studio, doch für unsere kleine Anwendung ist dies zuviel Aufhebens. Eine solche Lösung funktioniert im Prinzip wie die Ergebnisliste meiner Eigenschaftssuche. Man arbeitet mit einem Event-Empfänger irgendwelcher Art, lässt die anzuzeigende Liste ermitteln, prüft ihre Anzahl und lässt je nachdem entweder die vorher unsichtbare **Popup List** oder die **Dropdown List** sichtbar werden.

Die letzten beiden Felder, bei denen wir Aktivitäten unserer Anwender benötigen, sind die Vorläufer bzw. Nachfolger. Hierarchische Listen oder Popup Menüs, die hier wieder einmal sehr schön wären, sind wegen der Masse der möglichen Suchwerte wieder einmal ausgeschlossen, wir gehen also genauso vor wie bei den Eigenschaften. Die Anwender sollen uns die Datei eingeben, wenn der eingegebene Text kein korrekter Dateiname ist, wird er zur Suche verwendet. Es gibt jedoch einen einschneidenden Unterschied zwischen Eigenschaft und Datei - eine Eigenschaft muss nicht bereits bekannt sein, sie kann hier neu eingegeben werden, eine Datei jedoch muss existieren. Um das Verfahren für beide Varianten gleich zu halten, werde ich zu Beginn der Eigenschaftsliste eine Zeile „neue Eigenschaft“ einfügen, sodass die Anwender immer eine Auswahl treffen müssen, bevor die Ergebnisliste wieder verschwindet und die Verarbeitung fortgesetzt wird.

Zurück zu unseren Vorgängern bzw. Nachfolgern. Wenn ich eine bestimmte Datei habe, kann ich auch ihre Felder anbieten, deshalb ist das Feld für den Feldnamen wieder eine **Dropdown List**. Damit ich das alles nicht zweimal tun muss, füge ich bei diesen aktiven Feldern noch eine **Check Box** ein, die mir bestimmt, ob das ermittelte Dateifeld ein Vorgänger bzw. Nachfolger ist.

Damit ich auch diese Sorte Feldobjekt betexten kann, ergänze ich meine Filterung in der **$SetWindowText** wie folgt, damit meine Klartext-Prozedur auch diese Sorte Feld bearbeitet:

Startup_Task,
$SetWindowText

```
Calculate #F as
```

```
l_LST_Fields.$filter((pos('button',$ref.C2)>0)|(pos('pane',$ref.C2)>0)|($ref.C2='text')|($ref.C2='dat
agrid')|($ref.C2='checkbox')|($ref.C2='entry')|(pos('list',$ref.C2)>1))    ;; buttons, panes and text
```

Da ich in diesem Teil des Fensters von meinen Anwendern Eingaben anzufordern muss, bei denen ich in einem erheblichen Erklärungsnotstand bin wegen der unbekannten Methode, möchte ich auch die Eingabefelder mit **Tooltips** versehen und habe sie deshalb gleich mitaufgenommen, ebenso wie die **Popup Lists** und die **Dropdown List**.

Mit diesen vier Angaben pro Dateifeld kann ich den Rest alleine erledigen, alle weiteren Feldinhalte sind damit vollständig berechenbar. Wie die „Header"-Felder „Datei" und „Dateifeld" sind diese Bestandteile unseres Datensatzes somit rein informativ, deshalb unterlege ich sie mit einem grauen Rechteck, einem **background object**, um trotz spartanischer Einfachheit doch eine Hilfestellung für die Augen zu bieten.

Wie sehen nun diese informativen Angaben aus? Aus der „Eigenschaft" erhalte ich die Tentakelzahl, aus der Tentakelzahl das Profilgewicht. Aus den „Werten" erhalte ich das Eigengewicht, aus Profil- und Eigengewicht das Relativgewicht und damit den feldbezogenen Typ. Aus den Vorläufern und Nachfolgern werden die Portaldistanzen und ihre „Länge" und Gewichte berechnet, daraus der Reibungsfaktor und der auftragsbezogene Typ.

Das ist viel Zeug zum Zeigen! Wie geschickt unterbringen auf einen Blick? Mein Fenster möchte ich nicht zu groß werden lassen, also hatte ich beschlossen, nicht für jeden Einzelfall eine eigene Beschriftung zu wählen, sondern **Tooltips** zu verwenden.

In den „Header" nehme ich „Datei" und „Dateifeld" auf sowie die charakterisierenden Kennzeichnungen für Typ und Typgewicht, jeweils für „Datei" und für „Dateifeld". Da es sich hier um reine Texte bzw. Zahlen handelt, könnte ich Eingabefelder verwenden, die ich auf nicht eingabefähig setze, „enabled" = **kFalse**. Damit sie als echte Textfelder erscheinen, wäre die Einstellung „effect" = **kFlat** zu wählen. Wenn dann die Einstellung „backcolor" wie der Hintergrund des Fensters bestimmt würde, hätte ich also hundsnormale Texte mit **Tooltips** zur Verfügung. Ich könnte aber auch Buttons verwenden, sie deaktivieren und nur ihre Fähigkeit benutzen, Icons und **Tooltips** zu haben. Doch wie Eingabefelder und Listen können sie nicht durchsichtig gemacht werden. Ohne Durchsichtigkeit sehen aber flächige Hintergrundfarben wie diese **Wash Controls** schlicht unmöglich aus. Hier sind mit einfachen Mitteln nur Textfelder einsetzbar, die auf „backpattern" = 15, also durchscheinend, eingestellt sind, wenn ich solche durchsichtigen Beschriftungen verwenden möchte. Die Distanz-Listen repräsentiere ich über **Popup Lists**. Da habe ich wenig Probleme, da in gängigen Dateisystemen die „Länge" solcher Datenwege, in denen also Dateninhalte ohne Anwenderbeeinflussung oder sonstige Schnittstellenkontakte durchgereicht werden, meist nicht sonderlich ausgeprägt ist. Zwar kann theoretisch die Anzahl der Vorläufer bzw. Nachfolger oder deren Verwendung in **UsedInList** höher als 20 werden, doch das Risiko ist so gering, dass ich hier die optisch attraktiven und gleichzeitig raumsparenden **Popup Lists** verwende.

Aus Bequemlichkeitsgründen verzichte ich auf hübsche Hintergrundoptik und darf mich bei den Textfeldern deshalb für Buttons entscheiden, sie besitzen sowohl Icons als auch **Tooltips**. Nun beginnt die Kleinarbeit. Jeder Button und jedes Eingabefeld muss mit einer passenden Größenänderungsauswahl versehen werden, sodass die Felder sich nicht übelst durcheinandermischen, wenn die Anwender unser Fenster verändern. Ich habe es bei diesem vollgestopften Window so gehalten, dass ich die mittleren Felder überwiegend auf **kEFright**, die rechtsstehenden auf **kEFleftRight** gestellt habe, die linken Felder bleiben auf **kEFnone**. Richten Sie es sich einfach nach Gutdünken ein und öffnen Sie das Window dann. Bei den geringsten Größenänderungen stellen Sie sofort fest, welche Felder sich merkwürdig verhalten. Wenn ich nun die Felder für Texte und Gewichte bestimme und berücksichtige, dass **kEFright** seine Größe verändert, **kEFleftRight** aber nicht, so wähle ich die Felder mit **kEFright** für Texte und die mit **kEFleftRight** für Gewichte aus.

Die Icons für Typen wählte ich zu Nr. 1913 der Gruppe „Icon Editor“, die Gewichte zu Nr. 2018 aus „Enviroment 1“, Tentakel und die Distanzen zu Nr. 2019, wiederum aus „Icon Editor“. Für die Verarbeitungsbuttons „Bestätigen“ und „Abbrechen“ nehme ich Nr. 1681 und 1681, die rote und grüne Ampel finde ich recht aussagekräftig, aber klar ist das Geschmackssache. Suchen Sie sich ruhig selbst welche aus oder erstellen Sie doch sogar geeignetere mit dem **ICON EDITOR**, wenn Sie das Talent dazu haben! Ich jedenfalls habe keines und nehme deshalb mit dem Omnis-Angebot vorlieb.

Die Namen der Buttons im **Property Manager** halten sich wie bei den anderen Buttons an die Funktion, auch wenn die Funktion hier nichts weiter als „Anzeigen“ eines bestimmten Inhaltes ist. Für die Eingabefelder nehme ich gerne als Objekt-Name auch den Namen der Variablen, der dieses Feldobjekt im **Property Manager** mit der Einstellung „data name“ zugeordnet ist. Der Datentransfer ist bisher sehr einfach gehalten, dies möchte ich auch nicht ändern. Also füge ich in mein Fenster eine Liste **i_LST_MyList** ein, deren ausgewählte Zeile in einer Variablen **i_L** gespeichert wird und kann damit über die Spaltennummer mein Feld bestimmen, wobei die Spaltennummer direkt aus der Feldnummer des **SCHEMA**s stammt. Wenn ich mich nicht verzähle, verkünden Spalte und Zeile mir dann, welchen Inhalt ich ansehen oder verarbeiten möchte. Um mir die Übersetzung zu sparen, greife ich jedoch für die mit Klartexten aufbereiteten Typenfelder auf **i_LST_Show** zurück. Da die Listennotation zwar sehr bequem ist, aber eine einfache Variable noch leichter geschrieben und gelesen werden kann, nutze ich als Feldobjekt die Listennotation nur für die anzuzeigenden Felder, nicht jedoch für die zu verarbeitenden. Dies hat zudem den Vorteil, dass ungeprüfte Eingaben meiner Anwender noch hübsch separat von meinem Datenbestand in **i_LST_MyList** abgekoppelt sind und es mir problemlos erlauben, ihnen bei Fehlern auf die Finger zu klopfen, weil ich den alten, korrekten Stand noch ganz genau kenne.

Auch für die **Popup**-Listen benötige ich eigene Instanzvariable, da ich ihre gesamten Informationen erst aufbereiten muss. Für die Datei-Angaben, die aus meiner **Files**-Datei stammen, verwende ich eine Variable **i_Row_Info**, die ich über die Methode **$RetrieveFileInfo** meiner Datei-Task erhalte.

Nach den Objekt-Namen, der Positionierung und der Bestimmung des Verhaltens bei Größenänderung müssen jetzt alle neuen Felder mit einem **Tooltip** versehen werden: **[i_LST_Text(3,x)]**. Vergessen Sie bitte auch nicht den **Tooltip** von **Pane** selbst. Die exakte Zeilennummer „**x**“ besorge ich mir dabei mühsam beim Erstellen der Texte in der **Startup_Task**-Methode **$w_4fF**. Das muss ich sowieso tun, da ich meine Texte eben nur auf die primitive Art „selber schreiben“ meinem System zur Kenntnis bringen kann. Am besten tue ich das, indem ich einfach mein Fenster öffne und mit einem **BREAKPOINT** abwarte, bis mir Omnis meine benötigten Daten serviert. Alles, was ich dann in meinen Code als Text hineinschreibe, zeigt mir auch anschließend mein Fenster brav an. Doch es bleibt dabei - jetzt kopieren Sie sich entweder die ganze Methode **$w_4fF** und das Fenster aus der Ihnen über Internet angebotenen Applikation (Vorsicht mit der Fehlersyntax, die wurde dort geändert) oder Sie gehen die Ochsentour, das müssen alle Programmierer immer mal wieder tun. Was Sie genau tun müssen? Die Omnis-Identifikationen für Ihre Feldobjekte herausfinden, diesen die passenden Texte zuordnen und ihre Reihenfolge in der Textliste ermitteln. Diese Reihenfolge muss dann den **Tooltips** bzw. den Texten korrekt zugeordnet werden und leider auch die bereits bestehenden entsprechend den Verschiebungen angepasst werden. Denken Sie dabei auch an die **$ToTop**, die **$DoJobs** und die **$SetLanguage** wegen der Listenüberschrift von **i_LST_Show**.

Die Aktionen werden wie gehabt über die Tooltips gesteuert.

Feldmethode

```
Do $cinst.$DoJobs(,l_Button)    ;; 2. parm
```

Dies kommt im Augenblick jedoch nur für **Pane** und die beiden Buttons „ok“ und „cancel“ in Frage.

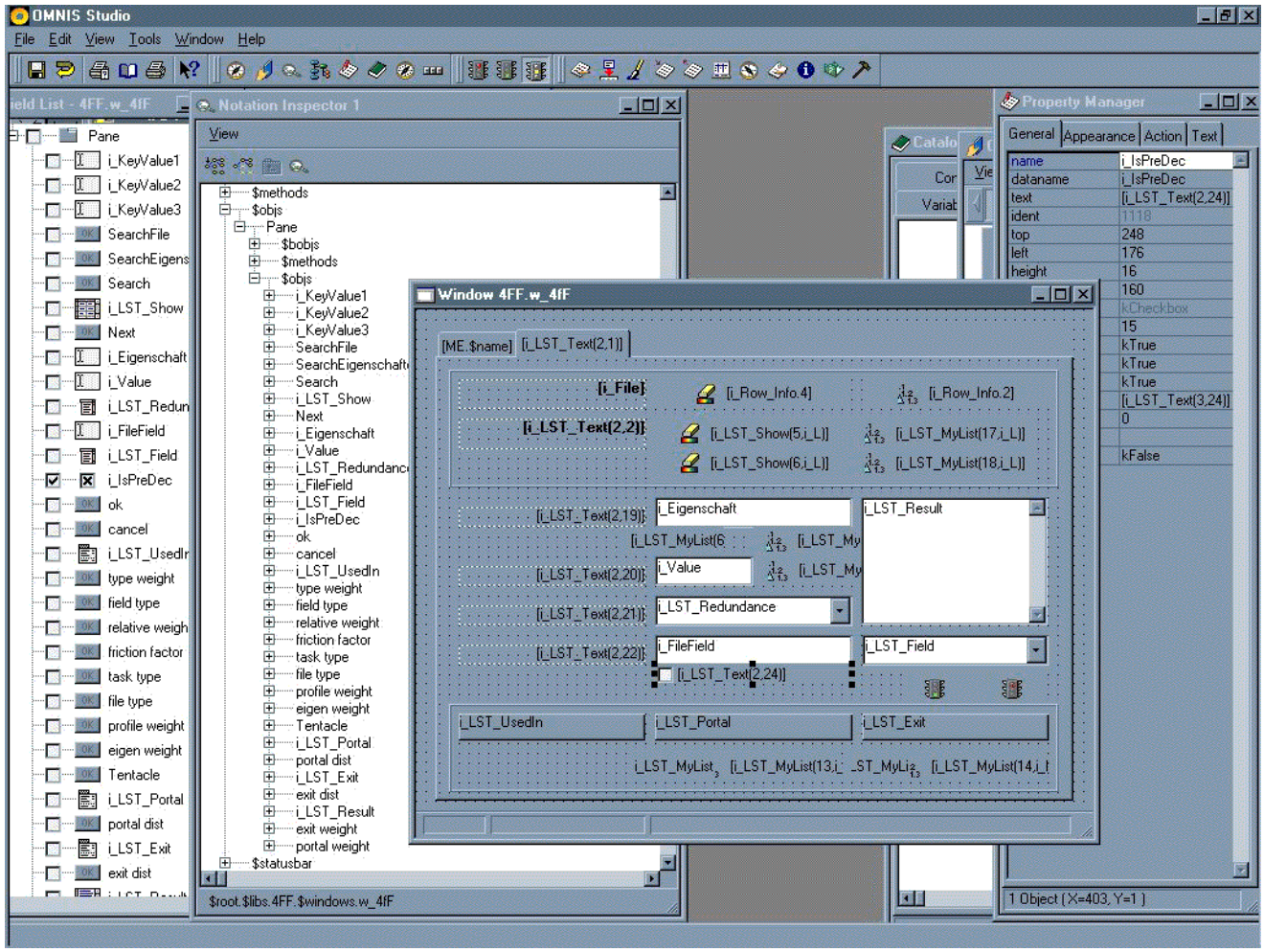

Abbildung 148 Verwaltungsfenster mit Feldliste und Notation Inspector

Übrigens - die Reihenfolge der Feldobjekte hängt nicht nur von der Anordnung auf dem Fenster ab, sondern auch von Reihenfolge, die bei Nutzung der Tabulator-Taste maßgeblich ist. Bestimmen können Sie dies über die Einstellung „order“ im **Property Manager**, Auswahl „General“ . Ich habe sie ganz einfach voreingestellt, sodass man über Tabulator die editierbaren Felder der optischen Anordnung nach aktiviert. Da die Reihenfolge der Feldobjekte meiner bisherigen **$SetWindowText** jedoch nicht nur von der Fensteranordnung und der Aktivierungsreihenfolge abhängt, sondern auch davon, auf welcher Ebene im Vordergrund oder Hintergrund das Element angeordnet ist, habe ich diese Ermittlung jetzt doch nach Identnummern der Feldobjekte sortiert. Dazu muss die Liste **l_LST_Fields** in der **$SetWindowText** mit drei Spalten versehen werden, was in der anfänglichen Schleife eine Änderung erforderte:

Startup_Task,
$SetWindowText

```
For l_L from 1 to 3 step 1
```

Auch musste jede Ermittlung der Feldobjektsattribute wie folgt mit **$ref.$ident** ergänzt werden

```
Calculate #F as $windows.[p_Window].$bobjs.$appendlist(l_LST_Fields,$ref.$name,$ref.$obj-
type,$ref.$ident)
```

Wenn Sie nun vor der Ermittlung der eigentlichen Textliste **l_LST_Text** noch die Sortierung der so erweiterten Liste **l_LST_Fields** durchführen,

Calculate #F as l_LST_Fields.$sort($ref.C3)

sollten wir in Zukunft von weiteren Umorganisierereien auch ohne Funktionentabelle befreit sein, denn Identitäten ändern sich nicht. Sie meinen, das hätte ich auch schon früher tun können? Sie haben völlig recht. Andererseits sagte ich Ihnen, begleiten Sie mich bei der Arbeit - und so perfekt bin ich nicht, dass alles auf Anhieb gleich klappt.

Zuletzt eine Anmerkung: die versprochene nähere Beschreibung der ausgewählten Eigenschaft aus der **Fields**-Datei wird nicht permanent in irgendwelchen Feldern auf dieses Fenster geliefert, sondern soll über die Rechte-Maus-Funktionalität zur Verfügung gestellt werden, deshalb können Sie im gezeigten Fenster-Entwurf natürlich kein repräsentierendes Objekt finden. Erläuternde Klartexte sind also nicht in der **Startup_Task**-Methode **$w_4fF** unterzubringen.

Was geschieht jetzt? Es gibt viel zu tun...

Wie immer fängt man am besten beim Anfang an. Wo liegt der für unser Verwaltungsfenster hier? Bei dem Wechsel von **Pane**, das heißt also, wenn die Anwender den einzelnen Datensatz auch im Einzelnen ansehen oder sogar verarbeiten wollen. Also fügen wir in die **$DoJobs** einen neuen **Case**-Zweig ein:

Case 1

w_4fF,
$DoJobs

Do $cinst.$InitPage2

Wenn Sie diesen Aufruf in die Feldmethoden der Ergebnisliste **i_LST_Show** mit der Option **On evClick** stellen, so können Ihre Anwender sogar direkt über die Auswahl ihres Wunschsatzes auf diese zweite Seite verzweigen.

Diese neue Methode **$InitPage2** soll nun all das tun, was wir für den ausgewählten Datensatz benötigen. Das Erste, was unsere zweite Seite anzeigen soll, sind die Dateiangaben zu **i_File**. Als nächstes müssen wir uns die Suchliste **MyList** aus der Datei-Task besorgen, da wir möglicherweise ihre Datensätze verändern wollen. Wir haben aber nicht vorgesehen, unsere Benutzer zu beschränken, es könnte es ja sein, dass unsere Datei-Task bereits von anderer Stelle zur Suche aufgerufen worden war, sodass die in der Datei-Task vorliegende Version von **MyList** mit unserer gar nicht mehr übereinstimmt. Der Befehl

w_4fF,
$DoJobs

Calculate i_LST_MyList as ME.i_MyList

Calculate #F as i_LST_MyList.$smartlist.$assign(kTrue)

gehört also nicht in unsere jetzt behandelte Methode **$InitPage2**, sondern in die **$DoJobs**, direkt nach der Suche und dem Aufbereiten unserer Ergebnisliste. Zu diesem Zeitpunkt ist die Instanzvariable der Datei-Task genau das, was wir brauchen. Als **SMARTLIST** will ich diese Liste haben, weil ich dann mit einem ganz einfachen **$dowork** die tatsächliche Datei anpassen kann, was meine **$StoreList** tut, und mit einem **$revertlistwork** meinen alten Zustand wiederkriege, wenn meine Anwender feststellen, dass „Abbrechen“ wohl im Augenblick das Beste ist, was sie sich und ihren Daten antun können.

Auch die nächste Aktion gehört eigentlich an eine andere Stelle, sie bestimmt uns nämlich die aktuelle Wunschzeile. Damit sollte sie direkt an dem Ort zu finden sein, wo der Wunsch der Meister entgegengenommen und verarbeitet wird, bei den Events der Ergebnisliste **i_LST_Show**. Doch den Wert **i_L** prüfen, ob er eine vernünftige Zeile meiner Liste **i_LST_MyList** darstellt oder nur die initiale Null ist, muss ich auf jeden Fall tun. Sonst sehen meine Anwender nur Leere und das muss ja nicht sein. Also berechne ich mir auch erst hier meine Zeilennummer, über die ich an den Datensatz und seine Aufbereitung herankomme.

Das nächste Feld in meinem Layout ist **i_Eigenschaft**. Es erfordert ein wenig mehr Arbeit als nur die Übernahme

des Feldinhaltes aus der Liste. Die Eigenschaft muss kontrolliert werden und die Liste für die Rechte Maus möchte ich erstellen lassen. Bei der Prüfung muss ich jedoch den Fall unterscheiden, ob ich hier einen Wert aus einem existierenden und deshalb notwendig korrekten Datensatz vorliegen habe oder ob es eine neue Eingabe der Anwender ist, die erst einmal auf Plausibilität abgeklopft werden muss. Also brauche ich eine neue Methode **$CheckEigenschaft**, die mit einem Parameter aufgerufen wird. Wie immer wähle ich mir meine Namen unabhängig von der Länge des Wortes, da ich ja nichts selber schreiben muss. Dem INTERFACE MANAGER ist es recht egal, wie lange ein Prozedurname ist. Um keine der Aufgaben zu vergessen, erstelle ich sofort diese neue Methode **$CheckEigenschaft** und notiere ich mir in den dokumentierenden Kommentarzeilen alles Wissenswerte, bevor ich mit der **$InitPage2** weitermache. Auch der Inhalt des Dateifelds „Werte“ muss vom Datensatz entgegengenommen werden, es gibt jedoch keine Möglichkeit für uns zu prüfen, ob die Zahlen, die uns die Benutzer geben, tatsächlich die möglichen Werte in diesem Datenfeld sind. Solche Angaben sind problembezogen und wenn wir sie bestimmen könnten, bräuchten wir sie nicht von unseren Anwendern zu verlangen.

Unser nächstes Feld ist die Liste **i_LST_Redundance**. Sie enthält alle unsere eigenen Dateifelder, denn unser Dateifeld **d_4fF_Redundance** soll uns verkünden, ob unser Dateifeld redundant ist. Wie zur Genüge bekannt, bedeutet Redundanz aber, dass es irgendwo ein Feld oder eine Feldkombination geben muss, an dem wir kleben wie eine Klette. Was wir also brauchen, ist die Liste der Dateifelder, denn wir beschränken uns auf den einfachsten Fall. Diese Liste können wir über unsere eigene Datei-Task **4fF** bekommen, die die Dateifelder versorgt. Als Methode einer Task ist der Zugang über den hilfreichen INTERFACE MANAGER das bequemste Vorgehen, da ich den genauen Rückkehrwert der Prozedur **$ListRecords** nicht mehr kenne. Also doppelklicke ich im INTERFACE MANAGER auf die Methode, werde umgehend in die Methodendarstellung meiner Task genau an den richtigen Fleck transportiert und kann direkt vor Ort nachprüfen, was ich zu wissen begehre. Dort sehe ich auch, dass genau die Liste **MyList**, die ich in meinem Window anzeige, der Rückkehrwert dieser Prozedur ist.

Damit komme ich auf den Gedanken, die gerade vorliegende Suchliste zu verwenden, also die Liste, die mein eigenes Fenster momentan anzeigt und zur Bearbeitung anbietet. Da aber die Anwender nicht nur dateigetreu ihre Datenfelder suchen können, sondern ganz beliebige Kombinationen von Dateiname, Feldname und Eigenschaft mein Suchergebnis erzielt haben können, kann ich nicht ohne weiteres davon ausgehen, dass ich in meiner vorhandenen Suchliste genau die Dateifelder meiner gewünschten Datei vorliegen. Im Extremfall kann ich schließlich sogar alle Dateifelder aller Dateien in meiner Liste vorfinden oder eben keines, wenn kein Suchkriterium passt. Deshalb muss ich die Feldliste meiner aktuellen Datei **i_File** schon mit den korrekten Kriterium frisch ermitteln, jedesmal wieder und wieder, wenn meine Anwender mit ihren Bearbeitungen zwischen ihren Dateien hin- und herspringen. Wieder und wieder?

Oder nicht?

Meine Liste **i_LST_Redundance** stellt alle Dateifelder meiner Datei dar, als Instanzvariable muss ich sie aber wenigstens einmal aufrufen, wenn ich mein Fenster geöffnet habe und jedesmal neu, wenn ich eine andere Datei bearbeiten möchte. Schließe ich mein Fenster, fängt alles noch mal von vorne an - dabei ändert sich meine Datei selbst doch gar nicht! Im Gegenteil - Dateien gehören zu den stabilsten Elementen in jeder Anwendung, allein deshalb, weil sie physikalische Behälter für kostbare Daten sind und deshalb mit höchster Vorsicht behandelt werden. Also gehört diese Listenermittlung nicht hierher ins Fenster.

Die Feldliste beschreibt originäre Eigenschaften der Datei selbst, also liegt die Vermutung nahe, sie gehöre in die Datei-Task. Dort würde sie nur einmal benutzt, genau dann, wenn die Datei-Task gestartet wird. Zwar müsste die Liste der Dateifelder angepasst werden, wenn die Datei verändert wird, doch das ist ein Ausnahmefall, den ich in meinem Fenster kaum mit vernünftigen Mitteln berücksichtigen könnte. Also stehen hier zig Ermittlungen gegen eine einzige und damit wäre die Sache klar.

Infinity kills information oder „zuviel ist ungesund“, das ist das sogenannte IKI-Prinzip. Das IKI-Prinzip ist Grundlage der Optimierung von Datenbanken, um die gesammelten Daten überschaubar zu halten. Denn überschaubare Daten sind wiederauffindbar. Genau deshalb berechnen wir mithilfe der **4fF-METHODE** auch Gewichte von Dateifeldern, um nämlich ihre Wiederauffindbarkeit abzuschätzen. Das IKI-Prinzip resultierte seinerseits aus der Definition der Information, die deren drei zentrale Eigenschaften klarstellte: Unterscheidbarkeit, Wiederholbarkeit und Änderung. Diese Eigenschaften sind ganz handfeste Anweisungen zum Abzählen. Unterscheidbarkeit ist eine Sache von Attributen, von Umständen, die erkennbar sein müssen, um die Unterscheidbarkeit überhaupt zu erlauben. In der Sprache der Programmierer sind das nichts weiter als Variable von Objekten. Unterscheidbarkeit ist dann die Frage, wie sich diese Variable verändern und welche konstant bleiben, das heißt, dass es drei Fälle gibt: die Variablen sind konstant für alle betrachteten Klassen, die Variablen sind konstant für alle Objekte einer Klasse oder sie unterscheiden sich für die verschiedenen Objekte einer Klasse. Wegen der fehlenden Unterscheidbarkeit ist der erste Fall nicht informativ innerhalb der betrachteten Klassen, solche Variable, beispielsweise Pfade von Dateien, werden meist nur als Systemvariable oder sogar nur im Programmcode gespeichert, wenn sie sogar für alle Klassen der Applikation konstant sind. Der zweite Fall ist die Grundlage der Bestimmung von Klassen, der dritte die Grundlage der Bestimmung von Objekten.

Wiederholbarkeit ihrerseits ist eine Sache von Ereignissen, von Abläufen, die mehrfach auftreten müssen, da sie sonst eben einfach nur einfach sind und nicht wiederholbar. Beide Fälle, Unterscheidbarkeit und Wiederholbarkeit, sind somit wenigstens zum Teil schlichte Zählarbeit.

Die dritte zentrale Eigenschaft ist die Änderung. Um ein Datum richtig einzustufen, betrachtet man also die verschiedenen Varianten, wie dieser aktuelle Wert des Datums hergestellt werden kann, auch hier ist die Gelegenheit gegeben zur Zählerei. Da Gehirntätigkeit schon auf sehr niedriger biologischer Ebene stattfand, ist die Effektivität dieser Zählerei jedoch keinesfalls zu unterschätzen, denn die Mustererkennung in den Gehirnen von Fischen kann kaum auf abstrakten Fantasien beruhen, nicht wahr?

Eine Möglichkeit zu Zählen bietet die sogenannte Messleine-Methode, die nach dem Prinzip der kleinsten Wirkung oder des geringsten Aufwandes funktioniert: man zähle die diversen Ereignis-Ketten, die eine gewünschte Wertveränderung herbeiführen und dann wähle man die kürzeste Strecke. Diese Methode ist schon so alt wie die praktizierte Geometrie selbst, es ist immerhin die Bestimmung des Abstandes zwischen zwei Punkten, das, was man eine „Linie“ nennt. Wie sagte das berühmte Langohr der Kult-SF-Serie denn auch? „Einfachste Logik ist die beste Logik“.

Aus solchen hochfliegenden Gründen heraus muss also meine Listenermittlung aus dem Fenster herausbefördert werden, doch wohin soll ich sie sonst bringen? Muss sie in die Datei-Task **t_File** wandern, wo sie übrigens schon längst zumindest teilweise vorhanden ist? Erinnern Sie sich noch an die **i_LST_Fields**? Sie tut genau das, ihre eigenen Dateifelder zur Verfügung stellen. Jeder Start meiner Applikation stellt für alle vorhandenen **SCHEMAS** die Task zur Verfügung. Doch was ist mit Dateien, die ich nur begutachten will, für die ich also Datensätze habe, die ich darüberhinaus jedoch nicht in meinem System kenne, weil ihre Verarbeitung andernorts erledigt wird?

Nun, wenn ich also nicht für jede beliebige Datei eine Task eröffnen möchte, deshalb kein Schema dafür anlege und deshalb beim Start der Applikation keine Task instanziiert bekomme, dann bleibt nur die Stelle übrig, wo meine Angaben über die Datei schlussendlich in Stein gemeißelt werden zum ewigen Gedenken - in der **Files**-Datei selbst. Also füge ich dort als neues, letztes Feld **d_FIL_Fields** ein, dessen Typ ich diesmal zu **BINARY** bestimme, was für Listen genauso gut funktioniert. Da ich meine Methoden bereits dauerhaft in dieser Datei untergebracht habe, lösche ich die Datensätze nicht mehr, sondern füge das neue Feld über die „Alter Table“-Funktionalität des **SQL OBJECT BROWSERS** ein, per Doppelklick zu erreichen. Neue Felder lassen sich dann

in diesem Fenster über die Rechte Maus einstellen, solange die SQL-Syntax zu ihrer Erstellung korrekt ist. Dieses neue Feld werde ich über meine Datei-Informationen für **i_File** erhalten, doch solange mein System noch nicht vollständig etabliert ist, prüfe ich in der **$InitPage2**, ob das Feld **d_FIL_Fields** leer ist und nehme in diesem Fall die Instanzvariable der betreffenden Datei-Task. Um zu bestimmen, welches Feld in meiner Liste sichtbar sein soll, benötige ich noch eine Instanzvariable **i_Text**, mit der ich alle meine Fensterlisten überdefiniere. Omnis zeigt bei dem Feldobjekt der Sorte „Liste“ nur eine einzige Spalte an, die ich natürlich frei vergeben darf, und zwar wie? Im PROPERTY MANAGER, Auswahl „General“, in der Einstellung „calculation“ wird das sichtbare Feld jeder Liste angegeben. Damit meine Fensterlisten auch eine spezielle Listenzeile anzeigt, bestimmte ich darüberhinaus für jede meiner Listen das Attribut **$line**, das die aktuelle Zeilennummer bezeichnet.

Die Feldnamen sehen mir ein wenig dürftig aus, deshalb füge ich in mein Fenster noch eine dritte Statuszeile ein, da ich das Window nicht mit noch mehr Feldern bevölkern möchte. Um dieses dritte Statuselement auch im Falle der Suchseite sinnvoll zu nutzen, versehe ich es in der Feld-Methode von **i_LST_Show**, dort, wo ich die Sortierung vornehme, mit dem Feldnamen aus meiner Liste, ebenso wie in der Methode **$DoJobs**, nachdem die Liste eingelesen wurde.

w_4fF, $DoJobs

```
Calculate #F as $cinst.$status-
bar.$panes.3.$text.$assign(i_LST_Show(2,$cinst.$objs.i_LST_Show.$gridvcell-1))
```

In der Methode **$DoJobs** wird die Suchliste neu bestimmt, dort reicht statt **$cinst.$objs.i_LST_Show.$gridvcell-1** jedoch einfach eine „1“. Bei einer neu eingelesenen Liste können meine Anwender ja kaum eine eigene Auswahl getroffen haben, also wähle ich für sie die allererste gefundene Listenzeile aus.

Dies trifft jedoch nur in diesem Falle zu, also dass die Anwender sich die Suchseite vorgenommen haben, um die Liste der gefundenen Datensätze zu beäugen. Auf der zweiten Seite der Detailangaben soll die Statusanzeige jedoch einen Klartext zu dem anzeigen, was gerade aktives Feld ist. Deshalb will ich beim Einlesen meines aktuellen Datensatzes den Inhalt des Feldes „Eigenschaft“ in der Statusanzeige offenlegen, einfach weil es das erste eingebbare Feld in diesem Teil des Fensters ist. Für die Eigenschaft habe ich schon eine eigene Verarbeitungsroutine, also bringe ich den entsprechenden Aufruf

w_4fF, $CheckEigenschaft

```
Calculate #F as $cinst.$statusbar.$panes.3.$text.$assign(i_Eigenschaft)
```

auch dort unter.

Die nächsten drei Felder handeln von den Distanzen, sie zeigen entweder die Vorgänger oder die Nachfolger auf, die den Datenfeld-Inhalt zu unserem Feld hin- oder von unserem Feld wegtragen. Auch ihre Bearbeitung dreht sich nicht nur um das Einlesen von Feldern, sondern unter Umständen um ihre Prüfung. Also führe ich auch hier eine neue Routine **$CheckDistance** ein, die mir das alles erledigen soll und erschlage damit auch die beiden Listen **i_LST_Portal** und **i_LST_Exit**.

Zuletzt bleibt nur noch die Verwendungsliste offen. Weil alle drei Listen den gleichen Aufbau haben, sie bestimmen schließlich Dateifelder über Datei- und Feldname, füge ich noch eine interne Methode ein, die solche Listen aufbereitet: **PrepareFieldList**.

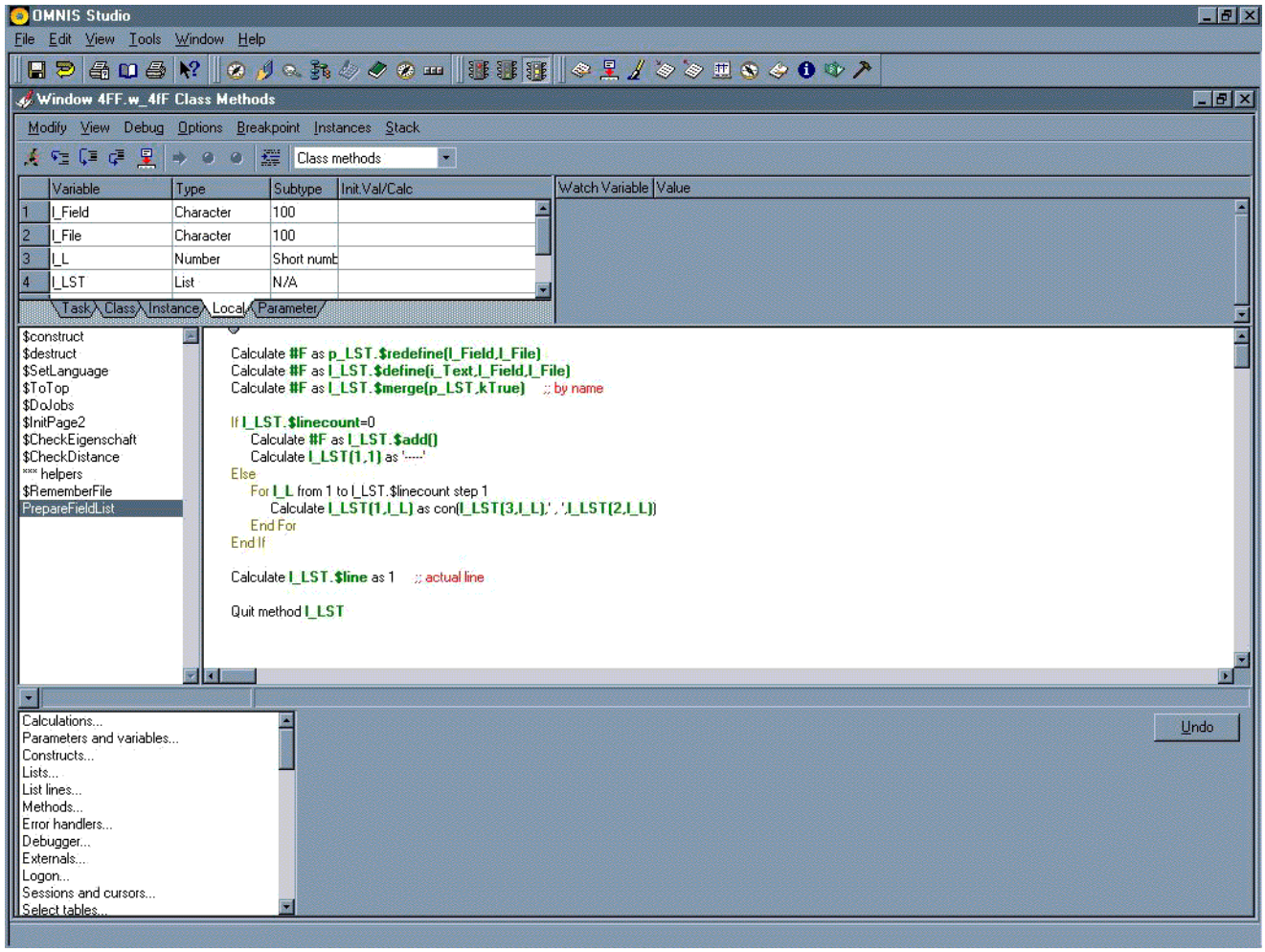

Abbildung 149 Aufbereitung der Feldlisten

Damit sieht unsere Routine **$InitPage2** wie folgt aus:

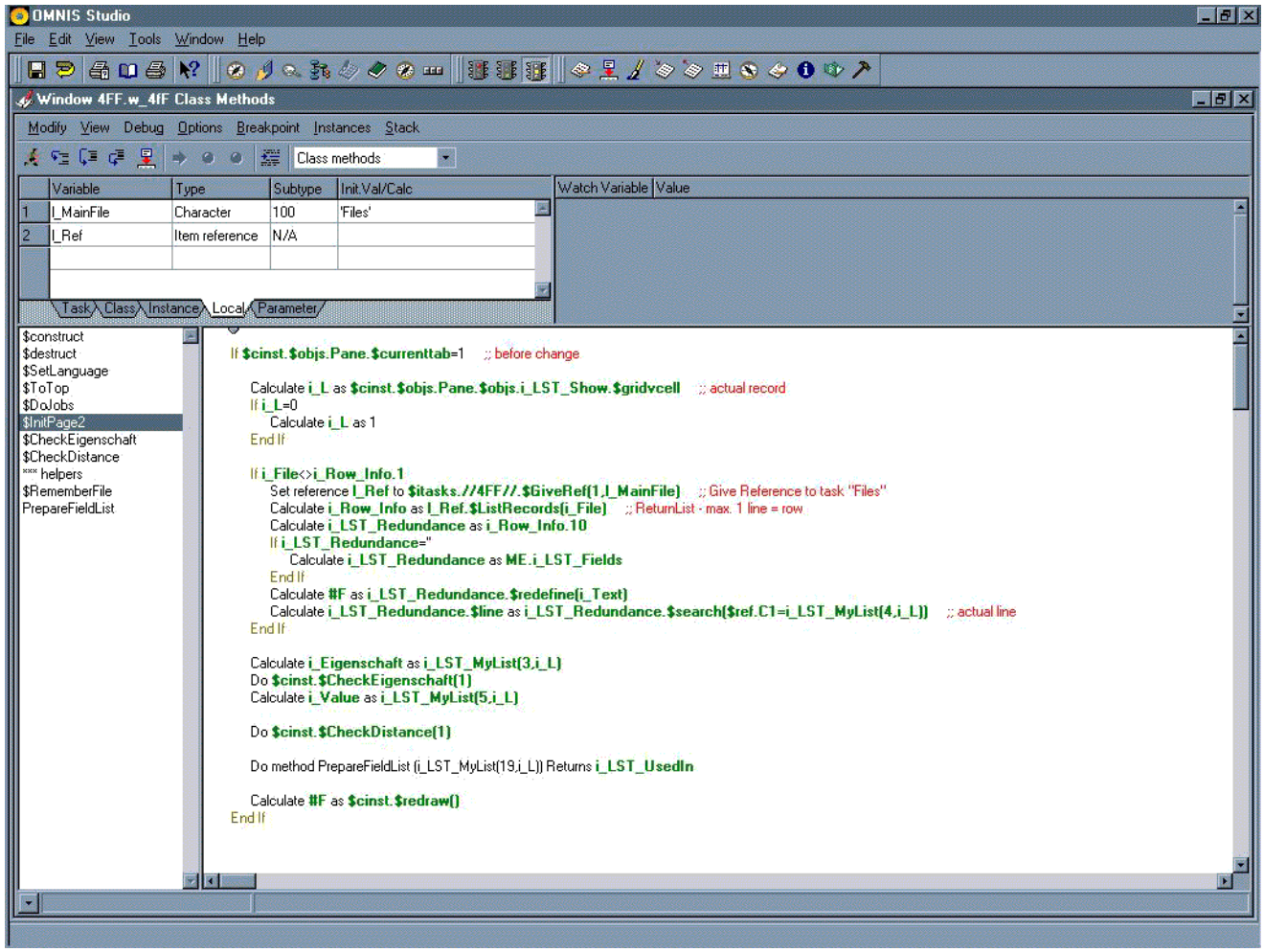

Abbildung 150 Vorbereitung der Verwaltungs-Seite

Prüfen Sie sie nach! Rufen Sie einfach per Rechte Maus das Fenster auf und kontrollieren Sie **Tooltips**, Beschriftungen und Dateninhalte, soweit sie überhaupt vorliegen können. Nach einigen diversen Kleinkorrekturen, Felder hin- und herzuschieben oder die **Tooltip**-Reihenfolgen zu bereinigen, sieht mein Fenster im Augenblick so aus, wenn ich es öffnete und über „Tab-Button“ meine Verwaltungs-Seite heranhole:

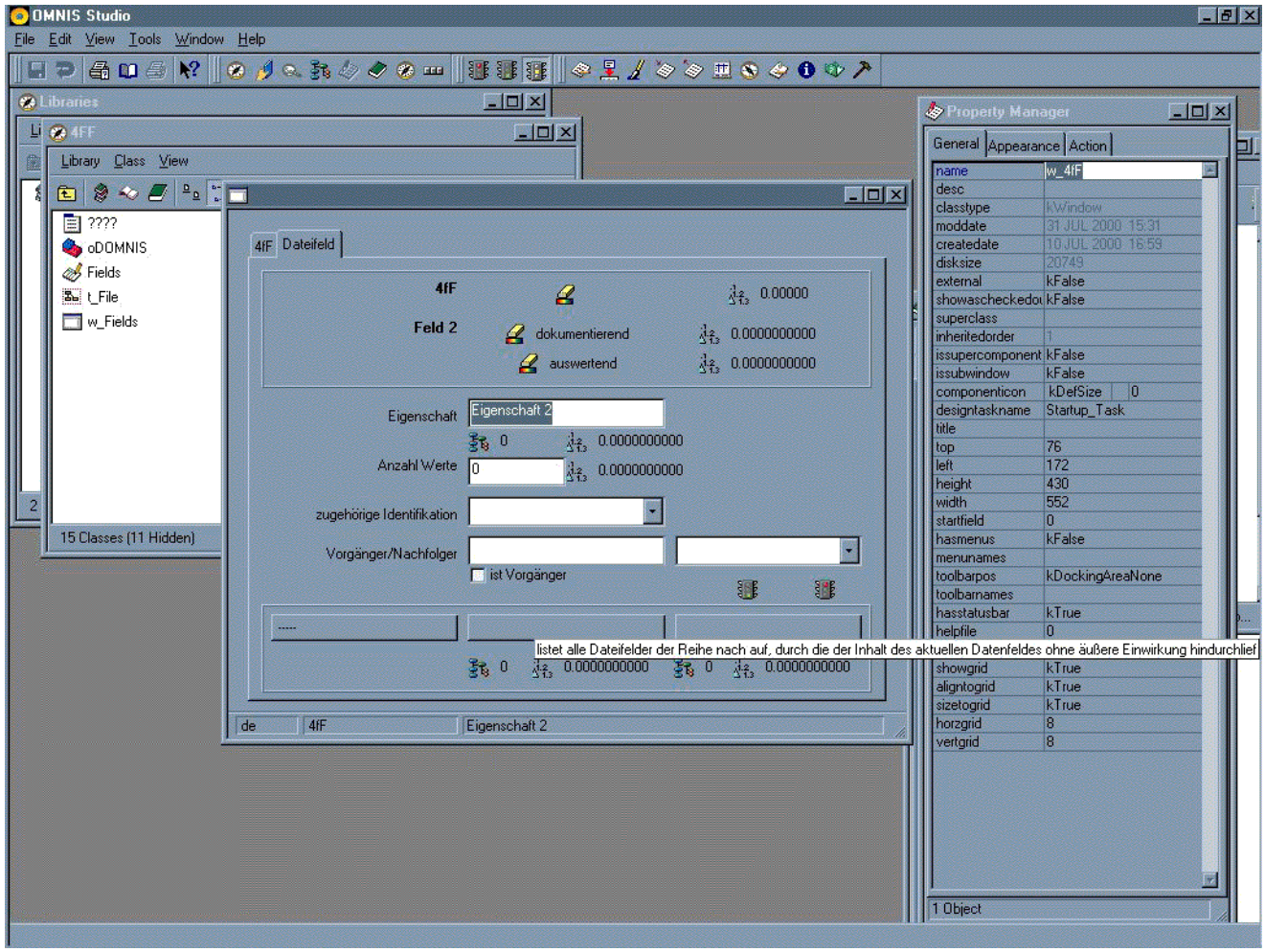

Abbildung 151 Verwaltungs-Seite in Aktion

Der in der Abbildung gezeigte **Tooltip** stammt von der Portal-Liste **i_LST_Portal**.

Als nächstes gehe ich an die **$CheckEigenschaft**, doch da mir diese Prozedur Fehlermeldungen ausgeben soll, füge ich in der **Startup_Task** folgende einfache Prozedur ein, um Fehler und Hinweise über die Omnis-Möglichkeiten **OK message** und **Yes/No message** auszugeben:

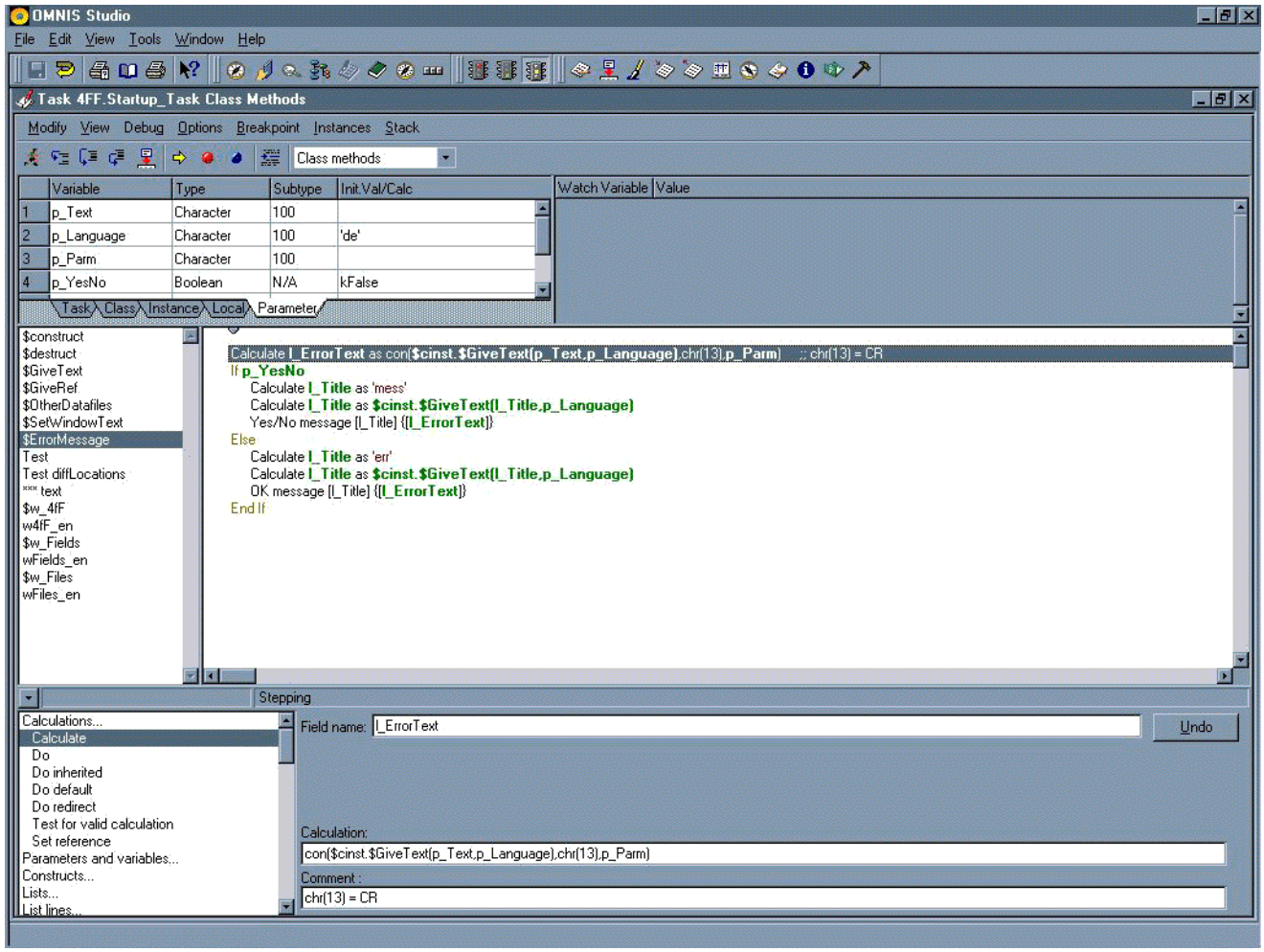

Abbildung 152 Fehlerausgabe

3.3 Die Integration individueller Dateimethoden

Das Erste, was die **$CheckEigenschaft** zu tun hat, ist nachzusehen, ob diese Methode nur aufgerufen wird, um die Werte einer Eigenschaft einzulesen, oder sie zu prüfen. Geprüft muss sie aber nur werden, wenn eine Änderung gegenüber dem Bestehenden erfolgt ist, dann muss zuerst nachgesehen werden, ob diese Eigenschaft überhaupt existert. Wenn nein, sollen Eigenschaften ähnlichen Namens gesucht werden und die Verantwortung an die Anwender zurückgegeben werden, wenn ja, wird nachgesehen, ob diese Eigenschaft bereits in unserer aktuellen Datei verwendet wird, was zu einem Fehler führen würde, ansonsten wird der **Fields**-Datensatz eingelesen und aufbereitet.

Prüfen möchte ich mit der existierenden Methode **$CheckValues** der individuellen Fields-Methoden, die mir auch die Liste aller Felder **i_LST_Connex** zusammenträgt, die meine Eigenschaft bisher benutzt haben. Doch diese Methode ermittelt noch ein wenig mehr, was ich sehr gut brauchen könnte, ohne mir das jedoch mitzuteilen - den Datensatz der Fields-Datei. Also füge ich in diese Prozedur noch einen weiteren Parameter ein: p_LST, der als Feldreferenz angelegt ist. Wenn ich jetzt, ganz zum Schluss der Routine **$CheckValues**

Individuelle Fields-Methode, $CheckValues

Calculate p_LST as l_LST_Eigenschaften