

Inhaltsverzeichnis

Datenbank-Bedienung	2
Wünsche	2
Aktion, Information, Selektion	2
Datenbank-Fähigkeiten	3
Anmeldung	3
Rechtesystem	4
Meine Daten	5
Favoriten	5
Papierkorb	6
Datenvergleich	6
Zeiten	7
Informationen sind Daten	7
Dateien	7
Zentrale Filter/Optionen	8
Dashboard/Inspektor	8
Umgang mit Datensätzen	9
Datensätze neu einfügen	9
Datensätze löschen	10
Datensätze aktualisieren	11
Datensatz-Duplikate zusammenfassen	11
Technik	12
Datenstruktur	12
Filter	12
Kopf-Formular	13
[Aktualisieren]-Button	14
[Menü]-Button	14
Änderungen am Eltern-Formular	15
System-Formulare	15
Dialoge	15
Treeview	16
Programmierung	16
Ereignisse	16
ImageList zum Treeview	17
Icons vorbereiten	17
Enumeration für die ImageList	17
JPGs	17
Ribbon	18
Fazit	19

Datenbank-Bedienung

Benutzer:innen haben erstaunlich einheitliche Wünsche an eine Datenbank: Sie soll beispielsweise generelle Filter erlauben, meine oft benutzten Objekte prominent anbieten, Datensätze schnell auffindbar machen, intuitive Bedienung enthalten und manches mehr.

Dafür habe ich in den letzten 30 Jahren immer wieder neue Lösungen entwickelt oder bei Kolleg:innen gesehen. Die waren nicht schlecht, aber allen gemeinsam war, dass sie nicht aus einem Guss, sondern an unterschiedlichen Stellen oder mit unterschiedlichen Methoden „drangebastelt“ waren. Mal wurden sie per Ribbon aufgerufen, mal waren sie in einem PopUp-Menü enthalten, sie konnten per Tastatur ausgelöst werden oder sie wurden besonders gerne an viele, viele Buttons geknüpft.

Deswegen habe ich (in Anlehnung an tatsächliche Datenbanken) einen Prototypen eines Datenbank-FrontEnds erstellt, der zeigt, wie sich alle die Wünsche der Benutzer:innen erstaunlich leicht unter einer einheitlichen Oberfläche zusammenfassen lassen.

Dabei nutze ich ein Treeview-Element, daher das Wortspiel *UsabiliTree* als Zusammensetzung aus *Usability* und *TreeView*. Aber es geht dabei gar nicht um die Programmierung des Treeviews, sondern um dessen Inhalte und seine Bedienung. Ich gucke also aus der Perspektive derjenigen, die diese Datenbank benutzen. Und damit ich sehen kann, ob deren Wünsche so überhaupt erfüllt werden, muss ich zuerst mal einen Wunschzettel erstellen:

Wünsche

Relativ unabhängig davon, was die konkrete Datenbank inhaltlich abbildet, lassen sich Forderungen aufstellen, die eigentlich überall gewünscht werden:

- **Anmeldung** an der Datenbank, inklusive Krankheits- und Urlaubsvertretung
- **Generelle Filter**, beispielsweise für den aktuellen Monat oder nur aktive Firmen
- **Papierkorb** für gelöschte Datensätze
- **Vergleich** mehrerer Datensätze des gleichen Typs
- **Hierarchischer Zugriff** auf Jahre Monate und Tage
- Verwaltung von externen **Dateien**
- Anzeige und Änderung von Datenbank-**Optionen**
- Reduktion auf nur **meine eigenen Datensätze** (was im Grunde auch nur ein Filter ist)
- **Favoriten** für häufig benutzte Elemente
- Eingeschränkte **Rechte** für Aktionen oder Anzeigen
- **Assistenz oder Analyse** für komplexe Aufgaben

Manche können wegfallen (z.B. eine Anmeldung bei einer SingleUser-Datenbank) oder bedingen sich gegenseitig (die Filterung nur meiner eigenen Datensätze benötigt eine Anmeldung), manche sind vielleicht nicht notwendig (etwa ein Papierkorb, falls die Datensatzlöschung verboten ist). Meistens werden jedoch ziemlich alle dieser Wünsche auftauchen.

Früher oder später braucht es ein Rechte-System, wenn zu viele Beteiligte die Datenbank benutzen. Weil auch dies eine Art Filter (für Aktionen oder Daten-Sichtbarkeit) ist, werde ich es hier beispielhaft berücksichtigen, obwohl gerade die Umsetzung von Rechten sehr viel zusätzliche Planung und Aufwand bedeuten.

Aktion, Information, Selektion

Die Umsetzung dieser Wünsche ist keineswegs an einen Treeview gebunden, aber dieser löst eine Grenze auf, die sonst einen Teil der Probleme verursacht. Es geht dabei um den Wechsel zwischen Information, Aktion und (Daten-)Selektion.

- **Informationen** tauchen in mehreren Zusammenhängen auf:
 - Als Datenfelder innerhalb eines Datensatzes
 - Als Hinweise in Meldungen
 - Als Texte in Ribbon-Labels
- **Aktionen** gibt es als
 - Ribbon-Buttons (oder -Menüs oder -Galerien ...)
 - Formular-Buttons
 - Tastatur-Makro-Aufrufe
 - PopUp-Menüs

- **Selektion** von Daten ist möglich über
 - Suchen/Filtern in Formularen
 - Auswahl-Tabelle in geteilten Formularen
 - Haupt- und Unterformularen
 - Verknüpfte/synchronisierte Formulare
 - Filter-Comboboxen oder Suchfelder
 - Suche/Gehezu-Makros

Je nach Gestaltung lassen sich Informationen, Aktionen und Selektionen in einer Datenbank noch ganz anders integrieren. Ein erster Schritt besteht daher darin, wenigstens innerhalb jeder einzigen Datenbank die Oberfläche zu vereinheitlichen.

Leider sehe ich viel zu viele Datenbanken mit inkonsistenten Bedienungen. Da wird z.B. eine bestehende Rechnung aus einem Endlosformular per Doppelklick ausgewählt, eine neue Rechnung in einem Ribbon-Menü *Neu / Rechnung* erzeugt und das Löschen findet per Rechtsklick mit einem PopUp-Menü statt. Das sind drei verschiedene Methoden für den Umgang mit einem einzigen Objekt: Maus-Doppelklick zum Bearbeiten, Ribbon-Befehl zum Erzeugen und PopUp-Menü zum Löschen.

Datenbank-Fähigkeiten

Aus Sicht der Benutzer:innen erfüllt diese Beispieldatenbank sehr viele Fähigkeiten, die oft ineinander greifen und entsprechend gegenseitig voneinander abhängig sind. Diese lassen sich grob in zehn Themen-Bereichen zusammenfassen.

Anmeldung

Um zu wissen, welche:r Benutzer:in derzeit mit der Datenbank arbeitet, greift **UsabiliTree** auf das Windows-Login zurück.

Wichtiger Hinweis: Hast Du Dein Login überhaupt eingetragen? Sonst darfst Du die Beispiel-Datenbank natürlich nicht benutzen! Dazu musst Du die Datenbank mit gedrückter <Umschalt>-Taste starten, mit <F11> den Navigationsbereich anzeigen und dann *tblPersonenBenutzer* öffnen.

Das ist allerdings nur der erste Schritt, denn für diese eine Person (=Windows-Login) sind optional mehrere Benutzer:innen auswählbar.

Hinweis: Weil es für das Verständnis wichtig ist, die Begriffe genau zu benutzen, möchte ich hier auf den Unterschied zwischen *Personen* und *Benutzer:innen* hinweisen.

Personen enthalten praktisch beliebige Namen ohne weitere Besonderheiten.

Benutzer:innen sind eine Untermenge davon und beschreiben nur diejenigen, welche diese Datenbank benutzen und daher sowohl ein Windows-Login als auch eine Rolle (mit den Rechten) enthalten müssen. Eine Person kann zu vielen Benutzer:innen gehören, es ist also eine 1:n-Beziehung.

Dadurch kann sich jemand mit Admin-Rechten für jede andere Person anmelden, was sehr praktisch für Tests und Fehlersuche ist. Oder in einem Sekretariat arbeitet jemand regelmäßig im Namen verschiedener Personen. Auch die Urlaubs- und Krankheitsvertretung basiert auf dem Wechsel der eigentlichen Benutzer:in-Identität.

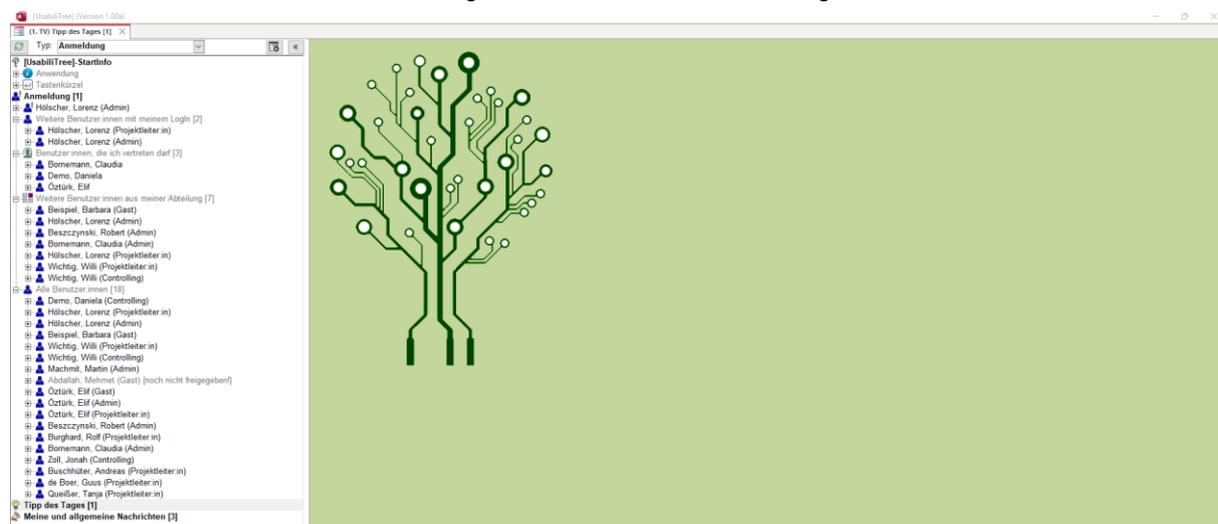


Abbildung 1: Verschiedene Möglichkeiten, unter wechselnden Identitäten zu arbeiten

Weil es diverse Anlässe gibt, für verschiedene Benutzer:innen zu arbeiten, sind hier beispielhaft mehrere Varianten implementiert:

- Eine **Anmeldung**, die keine explizite Bestätigung benötigt. Mit dem automatischen Öffnen des Treeviews im „Anmeldung“-Modus ist die Anmeldung bereits erfolgt und wird dort angezeigt. Falls es überhaupt mehrere mögliche Benutzer:innen zu meinem Login gibt, kann ich über eine „Favoriten“-Auswahl festlegen, welche: Benutzer:in dann standardmäßig sofort ausgewählt wird. Ein nachträglicher Wechsel ist problemlos möglich, aber im Normalfall ist keine weitere Aktion nötig.
- Eine **Auswahl** aus allen Benutzer:innen, die genau **zu meinem Windows-Login** hinterlegt sind. Bei den meisten wird es genau eine:n Benutzer:in geben und vor allem bei Admins eher mehrere. Im bereits erwähnten Sekretariat mit Datenbank-Pflege für mehrere Beteiligte sind dann einfach mehrere Benutzer:innen hinterlegt.
- Eine explizite **Vereinbarung mit Vertretungsrechten**. Dazu ist in einer Tabelle gespeichert, wer wen vertreten darf. Nur zu den Benutzer:innen, die ich vertreten darf, kann ich dann außerhalb meiner Login-Benutzer:innen wechseln. Lediglich mit Admin-Rechten ist der Wechsel zu beliebigen Benutzer:innen möglich.
- Wenn das inhaltlich sinnvoll ist, kann wie hier gezeigt auch der Wechsel zu **allen Benutzer:innen meiner Abteilung** oder einer ähnlichen Gruppe möglich sein. Diese erweiterte Auswahl benötigt keine explizite Angabe der zulässigen Benutzer:innen, sondern ergibt sich aus deren Zuordnung über einen Fremdschlüssel zur gleichen Abteilung.
- Nur mit Admin-Rechten wird überhaupt der Knoten für **alle Benutzer:innen** sichtbar. Da abhängig von den jeweiligen Rechten einige Knoten oder Buttons der Oberfläche ja gar nicht sichtbar sind, ist es zu Testzwecken wichtig, dann in die Rolle der jeweils betroffenen Benutzer:innen schlüpfen zu können.

Die tatsächliche Ummeldung geschieht via PopUp-Menü auf dem Namen:

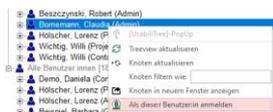


Abbildung 2: Ummeldung für eine:n andere:n Benutzer:in

Nicht zwingend, aber doch recht sinnvoll ist eine Nachverfolgung der Aktivitäten, wenn jemand unter fremdem Namen in der Datenbank arbeitet. Daher findet hier erstens beim Anmelden ein Hinweis auf den:die originale:n Benutzer:in anhand des Nachrichtensystems der Datenbank statt. Zweitens sind bei wichtigen Tabellen immer beide Werte in zwei Fremdschlüsseln gespeichert, nämlich zum korrekten Zuordnen und Filtern die offizielle Benutzer:in-ID und zum Nachverfolgen die tatsächliche Benutzer:in-ID.

Hier in der Datenbank werden zwar beide IDs für die angemeldete Person mitgeführt, aber nicht wirklich in den Datensätzen gespeichert, um deren Felder übersichtlich zu halten. In „echten“ Datenbanken gäbe es in den relevanten Datensätzen dann wie beispielsweise in *tblFirmen* zwei Felder:

- `firmaperbeIDRef_zustaendig` mit der gewählten (und manchmal eben vertretungsweise fremden) Benutzer:in-ID sowie
- `firmaperbeIDRef_geaendert` mit meiner echten (aus dem LogIn ermittelten) Benutzer:in-ID.

Normalerweise hätten beide Feldinhalte die gleiche ID, wenn ich nur meine eigenen Datensätze verändere. Wenn diese beiden IDs allerdings nicht identisch sind, lässt sich nachweisen, wer die Aktion vertretungsweise wirklich durchgeführt hat. Trotzdem greift immer der richtige Filter für die gewählte Person (ich selber oder als deren Vertretung ich mich angemeldet habe), weil hier nach `firmaperbeIDRef_zustaendig` gefiltert würde.

Rechtesystem

Das Rechtesystem basiert auf einer Zugehörigkeit aller Benutzer:innen zu jeweils genau einer Rolle. Es gibt keine Vererbung oder zusätzliche individuellen Rechte.

Hinweis: Wie schon erwähnt, gibt es allgemeine *Personen* und spezielle *Benutzer:innen*. Nur *Benutzer:innen* (nämlich der Datenbank) brauchen Rechte.

Der Einfachheit halber sind alle Rechte in der Rolle in eigenen Ja/Nein-Datenfeldern gespeichert:

rolleId	rolleName	rolleDarfNeuProjekte	rolleDarfFreigebenProjekte	rolleDarfAendernProjekte	rolleDarfLoeschenProjekte	rolleDarfNeuAlle	rolleDarfFreigebenAlle	rolleDarfAendernAlle	rolleDarfLoeschenAlle	rolleIstAdmin	rolleIstAktiv
0	Gast	<input type="checkbox"/>									
1	Admin	<input checked="" type="checkbox"/>									
2	Projektleiter:in	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	Controlling	<input type="checkbox"/>									
0		<input type="checkbox"/>									

Abbildung 3: Die Tabelle USys_tblRollen speichert die Rechte

Bei der Anmeldung sammelt der Code anhand der Rolle die Rechte des angemeldeten Benutzers bzw. der angemeldeten Benutzerin in einem dateiöffentlichen Type-Datentyp `p_benBenutzer`. Für die Anzeige von Knoten im Treeview oder von Einträgen im PopUp-Menü beispielsweise sind diese Rechte dann schnell und sprechend abrufbar:

```
If p_benBenutzer.RechtIstAdmin Then
    KnotenEinzeln trvDieser, p_nodExpandiert, icnBenutzer, zltXXX, "Alle Benutzer:innen"
End If
oder
If p_benBenutzer.RechtDarffreigebenAlle Then
    KnotenEinzeln trvDieser, p_nodExpandiert, icnFrei, zltYYY, "Freizugebende Objekte"
End If
```

Hinweis: Beim Öffnen der Datenbank bzw. beim Anzeigen des *Anmelden*-Treeviews erfolgt automatisch eine Anmeldung und damit auch die Zuweisung dieser Rechte. Diese Anmeldung ist nur im ersten von mehreren parallelen Treeview-Registern sichtbar. Die einzige andere Möglichkeit, die Rechte zu ändern, besteht im PopUp-Menü *Als diese:r Mitarbeiter:in anmelden*.

Meine Daten

Viele Datensätze innerhalb einer Datenbank sind für mich vor allem dann wichtig, wenn sie einen Bezug zu mir haben: *Meine Standardwerte, meine Projekte, meine Stunden etc.*

Daher gibt es hier einen eigenen Treeview-Typ, welcher genau diese für mich gefilterten Objekte zusammenfasst. Dabei ist es durchaus gewollt, dass manche dieser Informationen an anderer Stelle ebenfalls zu finden sind. Aber hier können diese Daten noch umfangreicher herausgefiltert werden, in großen Bau-Datenbanken beispielsweise:

- Projekte, die ich erstellt habe
- Projekte, für die ich verantwortlich bin
- Projekte, deren Pläne ich geprüft habe
- Pläne, die ich erstellt habe
- Pläne, die ich versendet habe
- Pläne, die ich geprüft habe
- Pläne, die ich geprüft und abgelehnt habe
- Pläne, die zu meinen Projekten gehören

Diese sind sicherlich auch irgendwo innerhalb der „normalen“ Treeview-Strukturen möglich, verbrauchen aber dort sehr viel Platz und ist hier zum Thema „*Meine ...*“ viel besser untergebracht.

Favoriten

Bestimmte Objekte werden innerhalb einer Datenbank häufiger gesucht, beispielsweise das Projekt, an dem ich gerade arbeite, oder Personen, mit denen ich regelmäßig zusammenarbeite. Diese lassen sich individuell als Favoriten speichern. Dazu reicht ein Rechtsklick auf deren Namen:

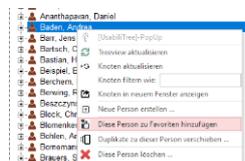


Abbildung 4: Aufnahme einer Person in die Favoriten

Damit ist diese Person dauerhaft für diese:n Benutzer:in als Favorit in die Liste aufgenommen. Ebendort lässt sich diese Person auch wieder entfernen:

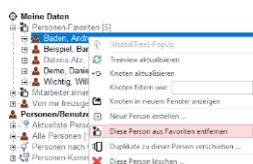


Abbildung 5: Per Rechtsklick lässt sich eine Person wieder aus den Favoriten entfernen

Entsprechend funktioniert es für alle anderen Objekte. Dabei prüft das PopUp-Menü immer, ob dieses Objekt bereits als Favorit gekennzeichnet ist. Dadurch ist es nicht möglich, es mehrfach als Favorit einzutragen:

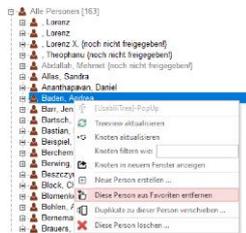


Abbildung 6: Das PopUp-Menü hat vorher automatisch geprüft, ob der Eintrag bereits ein Favorit ist

Eigentlich wäre in `tblFavoriten` außer dem Fremdschlüssel `favorperbeIDRef` (auf den:die Benutzer:in) jeweils ein eigener Fremdschlüssel für Personen (`favorpersoIDRef`) oder Firmen (`favorfirmaIDRef`) oder Projekte (`favorprojeIDRef`) usw. nötig. Das ist aber ziemlich unflexibel und bei vielen verschiedenen Favoritentypen auch recht platzverschwendend. Daher gibt es ein neutrales Fremdschlüsselfeld `favorxxxxxIDRef` und ein zweites Feld für den Typ. Dann ist zwar keine Referentielle Integrität möglich, aber das scheint mir hier unproblematisch.

Hinweis: Es gibt einen Favoriten-Sonderfall, nämlich die eigene Anmeldung. Während normalerweise beliebig viele Objekte einer Kategorie zu Favoriten erklärt werden können, ist aus den möglichen Anmeldungen mit meinem Login nur ein einziger Datensatz als Favorit auswählbar. Dieser dient dann der sofortigen, automatischen Anmeldung beim Programmstart.

Papierkorb

Das Löschen in einer Datenbank ist ein sehr heikler Vorgang. Einerseits erwarten alle Benutzer:innen, dass es wie bei Dateien immer eine Möglichkeit der Wiederherstellung gibt. Andererseits müssen oft viele abhängige Datensätze mitgelöscht werden, was zu immensen Folgeproblemen führen kann.

Dazu habe ich hier die drei wesentlichen Aspekte umgesetzt:

- Löschen darf nicht jede:r, es braucht explizite Löschrechte.
- Datensätze können nur gelöscht werden, wenn es keine abhängigen Daten dazu gibt (die also vorher gelöscht werden müssten). Dabei reicht es, die erste Kind-Ebene zu betrachten, weil es ohne Kinder keine Enkel geben kann.
- Es gibt gar keine echte Löschung, sondern lediglich eine Kennzeichnung des Datensatzes als „inaktiv“. Er verhält sich allerdings wegen grundlegender Datenbankfilter (siehe „Zentrale Filter“ auf Seite 8) so, als ob er gelöscht sei.

Das hier implementierte „allgemeine“ Löschrecht gilt für Rollen, nicht für einzelne Datensätze. Je nach thematischer Anforderung könnte es auch noch drastisch verschärft werden, wenn jede:r nur die eigenen Datensätze löschen darf. Dazu müsste dann natürlich in jedem betroffenen Datensatz der:die Ersteller:in gespeichert sein.

Beim Versuch, einen Datensatz zu löschen, erscheint in den meisten Fällen ein Dialog mit Treeview, der detailliert erläutert, aufgrund welcher Regeln das Löschen zulässig ist (siehe dazu „Datensätze löschen“ auf Seite 10).

Datenvergleich

Zu Vergleichszwecken beispielsweise zweier ähnlicher Adressen ist es hilfreich, diese Daten parallel ansehen zu können. Technisch ist es kein Problem, ein beliebiges Formular mehrfach und mit unterschiedlichen Inhalten anzuzeigen. Allerdings muss dafür zu jedem Formular eine globale Klassen-Variable bereitgestellt werden. Schon bei dieser wirklich kleinen Beispieldatenbank wären das 26 Formulare, eine ernsthafte Datenbank hat eher über 100 Formulare, deren Daten parallel vergleichbar sein sollten.

Das ist hier mit einer einzigen Klassen-Variablen möglich. Das Treeview-Formular selber hat die Datenformulare im rechten Teil als Unterformular eingebettet. Wenn also das Eltern-Formular mit dem Treeview parallel angezeigt wird, gilt das für alle eingebetteten Unterformulare ohne weiteres.

Anhand der -Schaltfläche wird jederzeit eine neue Instanz des Treeview-Formulars angelegt:



Abbildung 7: Beispielhaft drei Register, nämlich zwei Treeviews mit ausgewählten Projekten und einer mit ausgewähltem Ticket

Der Anzahl ist künstlich auf 10 Instanzen begrenzt, damit Benutzer:innen ein wenig zum Aufräumen gezwungen werden, aber es ist mit Änderung der Array-Obergrenze leicht änderbar:

```
Public p_frmTVparallel(9) As Form_frm_Treeview
```

Hinweis: Nur in der ersten Instanz ist die *Anmeldung* im Typ enthalten, damit nicht mehrere parallele Anmeldungen möglich sind.

Zeiten

Das Filtern von Zeiten ist oft mühsam, weil dann manuell auszufüllende Datumsfelder viel Eingabearbeit und Code-Prüfung verursachen. In den meisten Fällen geht es jedoch ohnehin um klare Zeiträume, nämlich Jahre, Monate oder Tage. Da ist es viel einfacher, diese direkt hierarchisch im Treeview anzubieten und nebenbei auch nur die vorkommenden Werte herauszufiltern.

Hinweis: Auch Quartale gehören eigentlich in diese Hierarchie und sind nur aus Gründen der Vereinfachung hier weggelassen. Wochen hingegen passen sowieso nicht in die Struktur, weil sie über die Monats- oder Jahresgrenzen hinausgehen.

Die hier im Beispiel gezeigten Daten mit gearbeiteten Stunden werden in einem eigenen Treeview-Typ angeboten, weil es bei den jeweils ausgewählten Objekten wie Projekten nicht um deren inhaltliche Daten geht (z.B. Details, zugehörige Tickets, etc.), sondern nur um die hierarchische Stunden-Darstellung.

Informationen sind Daten

Informationen und Dateninhalte werden im Treeview ausdrücklich gemischt. Die Angabe, dass ein Projekt bereits geschlossen (und damit vielleicht gegen Bearbeitung gesperrt) ist, gilt damit als ebenso wichtig wie die Namen seiner Unterprojekte. Während das eine ein Knoten mit einem freien Text ist, entstammen die anderen Knoten den zugehörigen Datensätzen. Für die Benutzung ist das gleichwertig und gleich wichtig.

Knoten entstehen übrigens nicht nur aus Texten und Datensätzen, sondern beispielsweise auch aus Dateinamen in Verzeichnissen oder Mails des Outlook-Kontos.

Dateien

Selbst wenn es in OLE-Objekten oder Attachments technisch möglich ist, gehören Dateien nicht wirklich als gespeicherte Inhalte in eine Datenbank. Die Größe der Datenbank wird explosionsartig zunehmen und die Pflege/Änderung solcher Dateien ist aufwändig. Außerdem besteht die Gefahr von unklaren Duplikaten außerhalb der Datenbank.

Trotzdem sind beispielsweise für Projekte oft Dateien wie Angebote, Fotos oder Rechnung wichtig und sollen schnell von der Datenbank aus erreichbar sein.

Wenn keine eigene Ordnerstruktur zwingend vorgegeben ist, werden diese einfach unterhalb eines Basis-Verzeichnisses in jeweils einem Pfad (hier in *<Pfad des BackEnds>\DateienFuerProjekte\Projekt<nnnn>*) gespeichert, wobei *<nnnn>* die fünfstellige ID des Projekts ist. Entsprechend gibt es hier solche Pfade für alle anderen passenden Objekte wie Firmen oder Tickets.

Die Dateien liegen also nicht in der Datenbank, aber der Treeview zeigt die Dateinamen als Unterelemente des entsprechenden Knotens genau wie Datensätze an:



Abbildung 8: Beispiel einer jpg-Datei im Ordner

Das „Daten“-Formular im rechten Teil zeigt also in Wirklichkeit keine Datenbankinhalte an, sondern Informationen zur extern gespeicherten Datei. Bei bekannten Dateiformaten wie JPG oder BMP wird außerdem auch der Dateiinhalt schon geladen. Für andere wie XLSX oder DOCX bleibt dieser Teil des Formulars leer.

Alle Dateien können per Rechtsklick auf den Dateinamen-Knoten direkt geöffnet werden:

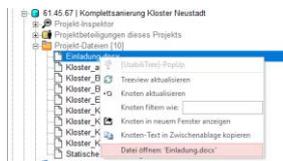


Abbildung 9: PopUp-Menü zum Bearbeiten der Datei

Dabei übernimmt Windows die Aufgabe, das passende Programm zur Dateiendung herauszufinden und zu starten. Es braucht dazu also keine Programmierung in Access.

Zentrale Filter/Optionen

Anstatt in jedem Formular/Bericht immer wieder einzustellen, ob nun beispielsweise alle Firmen oder nur die aktiven berücksichtigt werden sollen, ist es sinnvoller, dies zentral zu filtern. Eine der durchaus bequemen Möglichkeiten besteht in einem PopUp-Dialog, der alle solche Filter zum Beispiel in Checkboxes anbietet. Abfragen können dann auf deren Einstellung zurückgreifen. Das verbraucht allerdings relativ viel Platz durch den Dialog.

Diese zentralen Filter sind in **UsabiliTree** daher ebenfalls im Treeview integriert, so dass sie normalerweise platzsparend eingeklappt sind, aber jederzeit sichtbar gemacht werden können. Zudem wird bei deren jeweiliger Auswahl ein Formular zur Änderung angezeigt:



Abbildung 10: Zentrale Filter mit Anzeige und Möglichkeit der Änderung

Hier sind (derzeit zufällig) nur Ja/Nein-Filter enthalten, die ungefiltert einen schwarzen Text und gefiltert einen roten Text anzeigen. Auch wenn die Filter im jeweiligen (parallel angezeigten) Treeview-Formular erscheinen, gelten sie immer für die gesamte Datenbank.

Unabhängig davon gibt es noch Einstellungen für die Datenbank, die üblicherweise als Optionen bezeichnet werden. Dazu gehört beispielsweise, ob Aktualisierungshinweise eingeblendet sind oder wie viele aktuellste Elemente angezeigt werden sollen. Technisch handelt es sich um die gleichen (individuellen) Daten wie bei den Filtern, sie sind lediglich aus Gründen der Übersichtlichkeit von diesen getrennt.

Hier ist auch für die Anzahl der aktuellsten Elemente zu sehen, dass außer Ja/Nein-Werten auch Zahlen oder Texte oder Datumswerte speicherbar sind:



Abbildung 11: Beispiel eine Option mit Zahlenwert

In einer echten Datenbank blendet das „Standardwerte zentral“-Formular je nach Datentyp natürlich die übrigen Eingabemöglichkeiten aus, was ich hier aus Gründen der Übersichtlichkeit des Codes weggelassen habe.

Dashboard/Inspektor

Viele Objekte in großen Datenbanken sind so kompliziert, dass eine Prüfung auf korrekte Dateninhalte oft nicht in einem einzigen Datensatz möglich ist. Oder es ist notwendig, zwar „falsche“ Datensätze zuzulassen, aber war-

nend darauf hinzuweisen. Das gilt etwa für Firmen, zu denen (in 1:n-Datensätzen!) gar keine Adressen hinterlegt sind, oder für Projekte, zu denen noch offene Fehler-Tickets vorliegen.

Hier in **UsabiliTree** sind zwei Lösungsvarianten umgesetzt:

- Ein zentrales **Dashboard** mit Daten, die anhand verschiedener Kriterien gesammelt werden. Dabei wurden sie manuell bereits in *Fehler*, *Warnungen*, *Informationen* oder hier auch mal beispielhaft *Freizugebende Objekte* kategorisiert.
- Objektbezogene **Inspektor**-Knoten, welche alle Hinweise oder Warnungen zu diesem Objekt sammeln.

Das zentrale Dashboard zeigt seine Ergebnisse kategorisiert jeweils in beschreibenden Knoten, welche gegebenenfalls die eigentlichen Datenknoten anzeigen. Dadurch sind die gefundenen Objekte selber direkt bearbeitungsfähig und enthalten auch ihre Unterknoten:



Abbildung 12: Dashboard mit zentralen Ergebnissen

Das Dashboard ist vor allem geeignet, um einen generellen Überblick über problematische Daten zu erhalten.

Der Inspektor hingegen zeigt die Ergebnisse für die Prüfungen eines einzelnen Objekts an. Dabei muss es sich nicht nur um Fehler handeln, sondern es können auch allgemeine Hinweise und Informationen sein:



Abbildung 13: Projekt-Inspektor mit verschiedenen Hinweisen

Die jeweiligen Kategorien sind im Inspektor nur am Design mit Farbe und Icon zu erkennen. Im Gegensatz zum Dashboard sind es keine Objektknoten, sondern nur textliche Hinweise.

Umgang mit Datensätzen

Es gibt ein paar typische Aktionen im Zusammenhang mit Datensätzen, die in jeder Datenbank unabhängig von deren Inhalten gelten. Dabei existieren im Umgang mit Objekten (in der Programmierung) bzw. Datensätzen (in Datenbanken) ganz generelle Anforderungen, die als **CRUDL** abgekürzt werden:

- **C = Create:** Datensatz neu erstellen
- **R = Read:** Datensatz lesen
- **U = Update:** Datensatz aktualisieren/schreiben
- **D = Delete:** Datensatz löschen
- **L = List:** Mehrere Datensätze des gleichen Typs auflisten (evtl. gefiltert!)

Für jedes Element sollten also alle fünf Anforderungen erfüllbar¹ sein und es wäre viel intuitiver, das für alle Objekte einheitlich zu machen. Es darf nicht sein, dass eine neue Rechnung per Ribbon-Befehl *Neu | Rechnung* erzeugt wird, aber ein neuer Empfänger durch eine leere Zeile unter dem Endlosformular und ein neuer Veranstaltungsort mit dem Tastenkürzel *Strg+N*.

Datensätze neu einfügen

Neue Datensätze können an mehreren Stellen erzeugt werden. Sowohl auf Gruppenknoten (z.B. *Alle Personen*) als auch auf konkreten Datensatzknoten (z.B. *Hölscher, Lorenz*) bietet das Treeview-PopUp-Menü den jeweiligen Menübefehl (z.B. *Neue Person erstellen*) an.

Das jeweilige PopUp-Menü ruft eine allgemeine Prozedur auf, die im obigen Beispiel *PersonNeu* heißt und meistens nur das jeweilige Datenformular modal und mit neuem Datensatz aufruft:

¹ Wenn eine Anforderung wie beispielsweise das Löschen mal explizit verboten ist, wird sie natürlich nicht erfüllt. Aber auch das sollte nach gleichem Muster erkennbar sein, also beispielsweise ebenfalls im PopUp-Menü, aber dann deaktiviert. Oder das Löschen ist als Recht erlaubt, aber wird für diesen konkreten Datensatz dann im Löschanfrage-Dialog abgelehnt.

```
Function PersonNeu()  
    DoCmd.OpenForm "frmPersonenDetails", , , , acFormAdd, acDialog  
End Function
```

Hinweis: Mit Absicht handelt es sich bei dem modal angezeigten Dialog für einen neuen Datensatz um exakt das gleiche Formular wie neben dem Treeview für bestehende Datensätze angezeigt, lediglich die [Abbrechen]- und [OK]-Buttons sind unsichtbar. Dadurch gibt es keinen doppelten Code für alte und neue Datensätze.

Alle diese Prozeduren zum Neuerstellen eines Datensatzes stehen im Modul *modObjekte_Neu*.

Datensätze löschen

Das Löschen von Datensätzen ist hier absichtlich dann nicht möglich, wenn es noch davon abhängige Kind-Datensätze gibt. Daher zeigen die *Löschen*-PopUp-Menüs zuerst den Löschanfrage-Dialog:

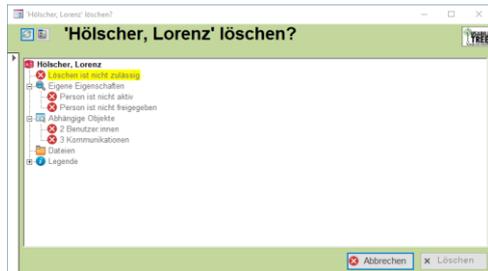


Abbildung 14: Beispiel für den Löschanfrage-Dialog

Dieser Löschanfrage-Dialog zeigt alle abhängigen Kind-Daten (keine Enkel-Daten, weil schon das Vorhandensein von Kindern als „Lösch-Stopper“ ausreicht) und eine hier gelb markierte Entscheidung, ob der Datensatz gelöscht werden darf. Datensätze mit vorhandenen Kind-Datensätzen (oder auch eigenen Datenfeld-Inhalten, welche eine Löschung verhindern sollen) werden dann als nicht löschar angezeigt.

Dazu übergibt der Befehl die ID des zu löschenden Datensatzes an die passende Funktion (z.B. *LoeschAnfragePerson(7)*²) und diese schreibt einen SQL-Aufruf in *qryLoeschAnfrage*, welche wiederum einen Recordset mit allen benötigten Informationen zurückgibt:

FeldID	FeldAnzeigen	Typ
7	Person	0
6	Hölscher, Lorenz	1
2	Benutzer:innen	2
3	Kommunikationen	2
-1	Person ist nicht aktiv	3
-1	Person ist nicht freigegeben	3

Hinweis: Diese Herumtrickserei mit VBA-Funktionen und wechselnden SQL-Anweisungen in einer Abfrage entspricht nicht der eigentlich geplanten sorgfältigen Trennung von Oberfläche und Daten. Mit einem SQL-Server als BackEnd ist das anders, dort übernimmt dann eine StoredProcedure diese Aufgabe und liefert direkt ein Recordset mit diesen Daten zurück. Die mittels VBA-Prozedur erzeugte *qryLoeschAnfrage* ist nur der Umweg, um in einer DAO-Umgebung ein Recordset mit virtuellen Daten zu erzeugen.

Anhand des Typs der Datenzeile (aufgrund der Enumeration *enmLoeschTyp* und deren *ltp...-Werten*) enthält diese verschiedene Informationen, wobei die Reihenfolge unerheblich ist:

- Typ 0 = *ltpZusammenfassung*: Der komplette Name und die zusammenfassende Beurteilung des zu löschenden Datensatzes.
- Typ 1 = *ltpAnzahl*: Die Anzahl der Minus-Punkte, also die Anzahl der Eigenschaften oder Kind-Datensätze, die gegen eine Löschung sprechen.
- Typ 2 = *ltpJaNein*: Die einzelne Eigenschaft mit der Ja/Nein-Angabe in *FeldID*, ob diese Einstellung gegen eine Löschung spricht.
- Typ 3 = *ltpInfo*: Allgemeine Informationen ohne Einfluss auf die Löscharkeit.
- Typ 4 = *ltpID*: Die ID des zu löschenden Datensatzes.

Beim Öffnen des Löschanfrage-Dialogs werden diese Datensätze je nach ihrem Typ in unterschiedlichen Knoten des Treeviews angezeigt. Für *Eigene Eigenschaften* und *Abhängige Objekte* wird ein angezeigt, falls der *FeldID*-Wert 0 ist, ansonsten bei jedem anderen Wert ein .

² Alle Prozeduren zum Umgang mit dem Daten-Löschen stehen im Modul *modLoeschAnfrage*.

Die *[Löschen]*-Schaltfläche wird nur dann aktiv, wenn im `ltpAnzahl`-Datensatz der `FeldID`-Wert 0 ist. Sie ruft beim Anklicken ebenfalls eine VBA-Funktion (z.B. `LoeschenPerson 7`) bzw. im Falle des SQL-Servers eine Stored Procedure auf, welche die eigentliche Löschung des Datensatzes übernimmt.

Hinweis: Auch diese VBA-Funktion ist den eingeschränkten Möglichkeiten von Access geschuldet. Mit einem SQL-Server im Hintergrund wird das von einer Stored Procedure übernommen, damit der Umgang mit Daten möglichst vollständig gekapselt ist.

Sollte es Dateien zu einem Datensatz geben, wird deren Anzahl genannt. Ebenfalls enthält der Dialog dann den Hinweis, dass diese Dateien nicht gelöscht werden. Das ginge technisch zwar, ist hier aber nicht vorgesehen, um den Code kürzer zu halten.

Datensätze aktualisieren

Die -Schaltfläche im jeweiligen Kopf-Formular aktualisiert den Datensatz des Elternformulars. Dieses Elternformular entspricht typischerweise einem `frm...Details`-Formular und zeigt seine Daten als Einzelformular an. Dabei wird jeweils die *Aktualisieren*-Prozedur des Elternformulars aufgerufen, welche meistens nur diesen banalen Code enthält:

```
Sub Aktualisieren()  
    Me.Requery  
End Sub
```

Als Nebeneffekt des Aktualisierens wird dabei das Speichern des angezeigten Datensatzes erzwungen. Das ist hilfreich, wenn der Datensatzmarkierer nicht sichtbar ist, der sonst zum Speichern genutzt werden kann.

Datensatz-Duplikate zusammenfassen

In einer relationalen Datenbank ist es grundsätzlich keine sinnvolle Lösung, identische Datensätze mit einer neuen ID zu duplizieren. Da wäre ein Fremdschlüssel besser. Allerdings gibt es oft historische Datenduplikate und manchmal den ausdrücklichen Bedarf für duplizierte Datensätze (z.B. bei allen Übernahmen von Adressen), damit diese nachträglich vielleicht doch verändert werden können.

Die meisten Duplikate lassen sich zwar per Augenschein als „gleich“ identifizieren, aber nicht automatisiert, beispielsweise beim Import. Das betrifft insbesondere Namensabweichungen wie *L. Hölscher* statt *Lorenz Hölscher* oder einfach dem fehlenden Vornamen.

Daher gibt es per PopUp-Menü *Duplikate zu diesem Objekt verschieben* die Möglichkeit, solche Duplikate eines Datensatzes zu erkennen und deren Unter-Datensätze einzeln im Original zusammenzufassen:

Analog zur Löschanfrage gibt es eine VBA-Funktion (z.B. `DuplikatsanfragePerson`³), welche die ermittelten Informationen via `qryDuplikatsanfrage` in einem Recordset zurückgeben. Dabei sieht das Ergebnis beispielsweise so aus:

FeldID	FeldAnzeigen	Typ
0	Hölscher, Lorenz	0
0	"Nachname WIE ""*Hölscher*" oder leer"	1
0	"Vorname WIE ""*Lorenz*" oder leer"	1
160	, Lorenz	2
159	, Lorenz	2
163	, Lorenz X.	2
162	Hölscher,	2
161	Hölscher,	2
7	Person	4

Anhand des Typs der Datenzeile (aufgrund der Enumeration `enmVerschiebeTyp` und deren `vtp...`-Werten) enthält diese verschiedene Informationen, wobei die Reihenfolge unerheblich ist:

- Typ 0 = `vtpZusammenfassung`: Der komplette Name und die zusammenfassende Beurteilung des zu verschiebenden Datensatzes.
- Typ 1 = `vtpVergleich`: Die Suchfilter zum Vergleich mit den Duplikaten.
- Typ 2 = `vtpDuplikate`: Die Namen und IDs der Duplikats-Datensätze.
- Typ 3 = `vtpInfo`: Allgemeine Informationen.
- Typ 4 = `vtpID`: Die ID des originalen Datensatzes.

Beim Öffnen des *DuplikateVerschieben*-Dialogs werden diese Datensätze je nach ihrem Typ in unterschiedlichen Knoten des Treeviews angezeigt.

³ Alle Prozeduren zum Umgang mit Duplikaten stehen im Modul `modDuplikate`.

Die *[Verschieben]*-Schaltfläche wird nur dann aktiv, wenn ein Duplikat-Datensatz markiert ist. Sie ruft beim Anklicken ebenfalls eine VBA-Funktion (z.B. `VerschiebePerson(7)`) auf, welche die eigentliche Verschiebung der Kind-Datensätze übernimmt. Die folgende Abbildung zeigt den DuplikateVerschieben-Dialog mit bereits expandiertem *Vergleichsfelder*-Zweig:

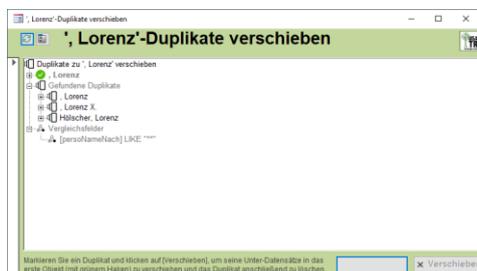


Abbildung 15: Dialog zum Verschieben von Duplikaten

Zu der ausgewählten Person *Lorenz* ohne Nachname (mit  gekennzeichnet) werden die gefundenen Duplikate (mit ) angezeigt.

Erst nach Markierung eines der Duplikate wird die *[Verschieben]*-Schaltfläche aktiv. Durch deren Bestätigung werden alle Kind-Datensätze des markierten Duplikats zum mit  gekennzeichneten Original verschoben.

Hinweis: Die Feldinhalte des Datensatzes selber (hier also die Person) hingegen werden nicht übertragen, weil sie ansonsten den Originaldatensatz überschreiben würden.

Da häufig in duplizierten Datensätzen auch gleiche Kind-Datensätze angehängt wurden, ist es sehr wahrscheinlich, dass sich beim Originaldatensatz in dessen Kind-Datensätzen wiederum Duplikate ansammeln. Das lässt sich nicht vermeiden, aber ich habe lieber ein paar Dateninhalte zu viel als zu wenig.

Anders als beim Löschen sind hier nacheinander mehrere sinnvolle Aufrufe möglich. Daher bleibt der Dialog geöffnet und wird nur inhaltlich aktualisiert. Erst mit der *[Schließen]*-Schaltfläche wird er geschlossen.

Technik

Damit die Datenbank einfach und übersichtlich funktioniert, sind viele Stellen standardisiert. Dazu gehört eine angepasste Reddick-Konvention (3-buchstabile Präfixe *tbl* für Tabellen, *qry* für Abfragen etc., aber die Feldnamen enthalten ein 5-buchstabiges Präfix wie *perso* mit Bezug zur Tabelle).

Datenstruktur

Diese Beispieldatenbank greift auf ein Access-BackEnd zurück. Ich bevorzuge aber SQL-Server-BackEnds mit Views und StoredProcedures. Das ist im Grunde zwar unerheblich, denn die Daten werden im FrontEnd immer durch standardisierte Abfragen gekapselt, aber StoredProcedures bieten technisch viel mehr Möglichkeiten als Access-Abfragen.

Zu jeder Tabelle (oder verknüpften SQL-Server-View) gibt es eine fast gleichnamige Abfrage, auf die alle Zugriffe stattfinden. Diese Abfrage stellt immer die beiden Felder *FeldID* und *FeldAnzeigen* bereit, auch wenn intern alle Tabellenfelder eindeutige Präfixe besitzen. Zu einer fiktiven Tabelle *tblBeispiele* gibt es also diese Abfrage *qryBeispiele*:

```
SELECT beispID AS FeldID, beispName AS FeldAnzeigen, tblBeispiele.* FROM tblBeispiele
```

Dabei kann das *FeldAnzeigen* auch aus mehreren Feldern zusammengesetzt werden wie hier:

```
SELECT persoID AS FeldID, persoNameNach & ", " & persoNameVor AS FeldAnzeigen, tblPersonen.*  
FROM tblPersonen
```

Durch diese zentralen Abfragen für jede Tabelle kann ich mich darauf verlassen, dass die beiden Felder *FeldID* und *FeldAnzeigen* in jeder Datenquelle vorhanden sind. Damit kann ich „blind“ eine Abfrage schreiben, um beispielsweise beliebige Treeview-Knoten zu erzeugen.

Filter

Der zweite Vorteil dieser standardisierten Abfragen besteht in deren Nutzung für allgemeine Filter, die überall in der Datenbank wirken. In der *UsabiliTree*-Datenbank gibt es solche Filter, welche beispielsweise nur aktive Projekte oder alle Projekte anzeigen lassen. Damit das durchgängig verwirklicht wird, ist so ein Filter direkt in dieser grundlegenden Abfrage (z.B. *qryProjekte*) eingebaut.

Diese zentralen Filter sind je Benutzer:in in der Datenbank in der Tabelle *USys_tblStandardwerteZentral* gespeichert und bleiben daher dauerhaft erhalten. Die meisten Inhalte sind Boole'sche Werte und liefern also 0 oder -1 zurück, was wegen der Nutzung in Abfragen über eine Funktion ausgelesen wird:

```
Function ZeigeProjekteNurAktive() As Boolean
    ZeigeProjekteNurAktive = StandardwertJaNeinLesen("ProjekteNurAktive")
End Function
```

Dadurch kann die Abfrage über den <=>-Operator⁴ entweder nur aktive oder alle Datensätze zulassen:

```
SELECT tblProjekte.projeID AS FeldID, [projeCode] & " | " & [projeName] AS FeldAnzeigen,
    tblProjekte.*
FROM tblProjekte
WHERE tblProjekte.projeIstAktiv <= ZeigeProjekteNurAktive();
```

Hinweis: Eine Anzeige nur der inaktiven Datensätze ist mit dieser Konstruktion nicht möglich, wurde bisher aber auch von meinen Kund:innen nicht benötigt. Dafür gäbe es frei definierbare spezielle Filterabfragen, die in der Datenbank hinterlegt werden können.

Tatsächlich können in einer Tabelle mehrere solcher generellen Filter greifen. Im Falle der Projekte gibt es noch die Unterscheidung in schon freigegebene oder noch freizugebende bzw. in noch offene oder schon geschlossene Datensätze:

```
SELECT tblProjekte.projeID AS FeldID, [projeCode] & " | " & [projeName] AS FeldAnzeigen,
    tblProjekte.*
FROM tblProjekte
WHERE tblProjekte.projeIstAktiv <= ZeigeProjekteNurAktive()
    AND tblProjekte.projeIstFreigegeben <= ZeigeProjekteNurFreigegebene()
    AND tblProjekte.projeIstOffen <= ZeigeProjekteNurOffene();
```

Alle jemals angezeigten Projekt-Daten basieren auf dieser Abfrage und sind damit automatisch korrekt gefiltert.

Kopf-Formular

Jedes Datenformular in *UsabiliTree* zeigt als Formularkopf ein eingebettetes Unterformular *sfmKopf* (*sfm = subform*). Dadurch sind sowohl eine einheitliche Gestaltung als auch vor allem der Einsatz standardisierter Bedienungselemente möglich, die nicht in allen Formularen einzeln hinterlegt werden müssen.



Abbildung 16: Entwurfsansicht des Formulars *sfmKopf*

Das Kopfformular wird jeweils vom einbettenden Hauptformular über Parameter gesteuert bzw. gibt seine Aktion an dieses Hauptformular weiter. Es enthält ...

- ... den Button *btnAktualisieren*, mit welcher im Hauptformular eine Datenaktualisierung ausgelöst wird. Technisch ruft der Code dabei die Sub-Prozedur *Aktualisieren* des Hauptformulars auf. Fehlt diese, wird der Fehler unterdrückt.
- ... den Button *btnMenue*, welcher dazu dient, ein (vom Hauptformular gesteuertes) PopUp-Menü anzuzeigen.

Hinweis: Eigentlich sollte das PopUp-Menü an der Unterkante des Buttons andocken, aber der technische Aufwand, um diese Position korrekt zu ermitteln, ist so hoch, dass stattdessen wie für PopUp-Menüs üblich die Mausposition benutzt wird. Auch wenn PopUp-Menüs normalerweise eigentlich per Maus-Rechtsklick geöffnet werden, gilt hier der Maus-Linksklick, weil dieser ja auf einen Button erfolgt.

- ... das Image-Objekt *imgSchloss*, welches erst dann sichtbar wird, wenn das Formular (z.B. bei gesperrten Projekten) schreibgeschützt wird.
- ... das Label *lblTitelHaupt* mit der (vom Hauptformular gesteuerten) Beschriftung. Dessen Breite wird durch Anker automatisch auf die Breite des Formulars vergrößert.
- ... das Image-Objekt *imgLogo*, welches das Datenbank-Logo enthält und durch Anker immer am rechten Rand des Formulars positioniert ist.

Eigentlich besteht die wesentliche Aufgabe dieses eingebetteten Kopf-Formulars darin, seine Controls passend zum Elternformular zu organisieren und die Aktionen weiterzureichen. Da jedoch deren *Form_Open*-Code auch in solchen eingebetteten Formularen ausgeführt wird, ist die Gelegenheit günstig, um darin weitere Aktionen durchzuführen. Dazu gehört beispielsweise das Umfärben der Formularhintergründe (basierend auf der personengebunden Farbauswahl), die Anzeige des Datensatzmarkierers nur für Admins oder die Farbänderung des Titeltex-tes auf Rot, falls die Daten des Formulars schreibgeschützt sind.

⁴ Danke an Bernd Jungbluth für diesen Tipp. 😊

Hinweis: Auch bei den eingebetteten Kopf-Unterformularen handelt es sich (welch' Überraschung!) um Klassenobjekte. Nur deshalb ist es möglich, das gleiche *sfmKopf*-Formular parallel in mehreren Formularen einzubetten und mit jeweils eigenen Titeln zu versehen oder nur auf die Parameter des eigenen Elternformulars zu reagieren.

[Aktualisieren]-Button

Der *[Aktualisieren]*-Button in beiden Kopf-Formularen befindet sich technisch gesehen ja in einem Unterformular (des jeweils sichtbaren Datenformulars). Daher „weiß“ er nicht, ob es in diesem Elternformular überhaupt etwas zu aktualisieren gäbe und wie das aufzurufen ist. Es gibt deswegen einfach nacheinander zwei Versuche, die mit ignoriertem Fehlerbehandlung durchlaufen werden:

```
Private Sub btnAktualisieren_Click()
    On Error Resume Next
    Me.Parent.Requery
    Me.Parent.Aktualisieren
    On Error GoTo 0

    DoCmd.Hourglass False
    DoCmd.Echo True
End Sub
```

Falls das Elternformular die Daten eines konkreten Datensatzes anzeigt, wird durch die *Requery*-Aktion dieser Datensatz gespeichert und erneut geladen. Dabei werden auch eventuell vorhandene Berechnungen auf dem Formular neu durchgeführt.

Die *Aktualisieren*-Aktion hingegen ruft eine im Elternformular möglicherweise vorhandene (und deswegen einheitlich so benannte) Prozedur auf. Deren Inhalt ist dabei beliebig, an vielen Stellen wird dort sogar wiederum nur das *Requery* selber aufgerufen⁵:

```
Sub Aktualisieren()
    Me.Requery
End Sub
```

Wegen des Abschaltens der Fehlerbehandlung beim Aufruf der *Aktualisieren*-Prozedur ist es auch kein Problem, falls das Elternformular gar keine solche Prozedur enthält. Auch (optionale!) Parameter mit Standardwerten sind für diese Prozedur möglich.

[Menü]-Button

Der *[Menü]*-Button in den beiden Kopf-Formularen dient dazu, ein PopUp-Menü anzuzeigen, welches flexible Inhalte auf möglichst kleinem Raum anbietet.

Die Position des PopUp-Menüs ist leider nicht exakt an die Position des Buttons gebunden (was optisch sicherlich schöner wäre), sondern beginnt immer an der Maus. Der Unterschied wird den meisten Benutzer:innen wahrscheinlich gar nicht auffallen, da sich die Maus beim Klicken zwangsläufig innerhalb des Buttons befindet. Da Access aber anders als die übrigen Office-Programme für seine Controls keine *POINT(x, y)*-Koordinaten bereitstellt, die für eine eigene PopUp-Menü-Position notwendig sind, würde das einen erheblichen Aufwand für die Umrechnung verschiedener Koordinatensysteme und Einheiten verursachen.

Hinweis: Da es sich um einen Button handelt, folge ich weiterhin der Konvention, dass dieser mit einem Links-Klick bedient wird und so das PopUp-Menü anzeigt. Auch das wird den meisten vermutlich gar nicht auffallen, dass PopUp-Menüs sonst per Rechts-Klick angezeigt werden.

Das PopUp-Menü organisiert sich beim Aufruf selber und besteht aus drei möglichen Teilen:

- Grundsätzlich bietet es anhand einer Codierung immer die Standard-Bearbeitungen für Datensätze an, also Neuerstellen, Bearbeiten und Löschen.
- Optional können wechselnde Filter für konkrete Formulare eingefügt werden.
- Für weitere Sonderwünsche sind optional noch beliebige Menübefehle möglich.

Die Angaben, ob ein Datensatz neu erstellt, bearbeitet oder gelöscht werden darf, sind in der *Tag*-Eigenschaft des Elternformulars codiert. Die Prozedur *ZeigeKopfPopUp* prüft deren Inhalte, nachdem vorher eine beliebige andere Prozedur diese möglicherweise verändert hat.

Hinweis: Bestimmte Formulare sind von dieser Codierungsprüfung grundsätzlich ausgenommen, beispielsweise die Standardwerte- und Filter-Formulare. Bei diesen geht es um die Einstellung individueller Werte, für die das allgemeine Ändern-Recht keine Einschränkung ist.

Diese Codierung besteht aus einer Zeichenkette nach diesem Muster *C1R1U1D1L1F0*. Die Werte haben folgende Bedeutungen:

⁵ Natürlich ist das doppelte Aufrufen der *Requery*-Aktion ein Performance-Verlust, aber tatsächlich ist das bei einem einzigen Datensatz unerheblich und stellt sicher, dass auf jeden Fall eine Aktualisierung durchgeführt wird, egal von wo aus das aufgerufen wird.

- **C = Create:** C0 deaktiviert und C1 aktiviert den Menübefehl zum Neuerstellen des Datensatzes
- **R = Read:** R0 deaktiviert den Menübefehl zum Bearbeiten des Datensatzes, R1 aktiviert ihn und R2 weist auf die Bearbeitung im aktuellen Dokument hin.
- **U = Update:** U0 deaktiviert den Befehl zur Aktualisierung des Formulars, U1 aktiviert ihn.
- **D = Delete:** D0 deaktiviert den Befehl zum Löschen des Datensatzes, D1 aktiviert ihn⁶.
- **L = List:** L0 deaktiviert den Befehl zum Aufruf des vorgesehenen Listen-Formulars, L1 aktiviert ihn.
- **F = Filter:** F0 zeigt kein Filter-Untermenü an, F1 ruft die Prozedur `ZeigeKopfMenueFiltern` auf, um die lokalen Filter anzuzeigen.

Wie hier schon zu sehen ist, werden die im Formular passenden Untermenü-Einträge zum Filtern über die Prozedur `ZeigeKopfMenueFiltern` erzeugt, die jeweils im Elternformular vorhanden sein muss.

Eine weitere Prozedur `ZeigeKopfMenuePlus` im Elternformular wird automatisch (mit Fehlerunterdrückung) ausgeführt, um zusätzliche Menübefehle anfügen zu können. Das geschieht beispielsweise in `frmDokumenteDetails`, um den Befehl *Dokument duplizieren* anzuzeigen.

Zusätzlich prüft der Aufruf dieses PopUp-Menüs, ob dabei die Shift-Taste gedrückt war. Dann werden einige zusätzliche, deaktivierte Menübefehle als Informationen für Admins angefügt, beispielsweise der Name des Eltern-Formulars und dessen `Tag-Inhalte`.

Änderungen am Eltern-Formular

Beim Öffnen des Kopf-Formulars nimmt dieses noch weitreichende Änderungen an seinem Elternformular vor, welches ja die eigentlichen Daten anzeigt. Diese Änderungen betreffen unter anderem die Übernahme der individuell gewählten Farbstile, die Editierbarkeit oder die Anzeige von *[Abbrechen]*- und *[OK]*-Buttons. Dadurch muss nicht jedes einzelne Datenformular diese Aufgaben selber aufrufen, was den Code deutlich vereinfacht.

System-Formulare

Die Access-eigenen Objekte sind mit `MSys...` gekennzeichnet und lassen sich dadurch mit der nicht angekreuzten Navigationsoption *Systemobjekte anzeigen* bei Bedarf verstecken. Entsprechendes gilt auch für die `USys...` (`USys` = User System) benannten Objekte.

Hinweis: Dieses Verhalten für `USys...`-Namen gilt übrigens nicht nur für Formulare, sondern für alle Access-Objekte im Navigationsbereich. Davon ist beispielsweise auch das VBA-Modul `USys_modEntwickler` betroffen.

Die beiden Formulare `USys_frmIcons` und `USys_frmJPGs` lassen sich dadurch im Normalfall ausblenden. Sie dienen weder der Datenpflege noch überhaupt der Anzeige, sondern enthalten lediglich jeweils ein `ImageList-OCX-Control`. Dieses dient der internen Speicherung der Icons (für den Treeview) bzw. der JPG-Bilder (für Menüs).

Dialoge

Grundsätzlich gibt es in Access keinen technischen Unterschied im Entwurf von (nicht-modalen) Formularen und (modalen) Dialogen. Im Gegenteil, in *UsabiliTree* ist es der Normalfall, dass ein Formular als eingebettetes Unterformular die Daten beispielsweise einer vorhandenen Firma anzeigt und das gleiche Formular als (modales) PopUp-Formular für die Eingabe einer neuen Firma erscheint. Wegen der Klassen-Eigenschaft von Formularen geht das sogar gleichzeitig.

Trotzdem sind einige Formulare mit dem Präfix `dlg` statt `frm` aus zwei Gründen ausdrücklich als Dialoge gekennzeichnet:

- Sie werden ausschließlich als PopUp aufgerufen, sind also nie eingebettet. Daher enthalten sie auch andere Schaltflächen wie *[Schließen]* statt *[Abbrechen]* oder *[OK]*.
- Wegen eines Fehlverhaltens von Access können sie nicht einfach mit dem `DoCmd.OpenForm`-Befehl plus dem Parameter `acDialog` aufgerufen werden, weil dann die Größenveränderung am Dialogrand nicht funktioniert (was bei deren oft Treeview-gesteuerten Inhalten wünschenswert ist). Stattdessen müssen deren `PopUp`- und `Gebunden`-Eigenschaften auf `Ja` stehen, was im Ergebnis ebenfalls eine modale Anzeige ist, aber die Größenänderung funktionsfähig lässt.

Hinweis: Leider führt nur der Aufruf mit `acDialog`-Parameter zu einem wirklich modalen Verhalten. Die zweite Variante behält zwar die Größenveränderlichkeit bei, führt aber den anschließenden Code sofort aus!!

⁶ Diese Option ist hier derzeit uninteressant, weil nur wenige Objekte gelöscht werden dürfen und dieses Löschen dann auch nur über das PopUp-Menü des Treeviews aufgerufen wird.

Eine Veränderung dieser beiden Eigenschaften per VBA zur Laufzeit ist nicht möglich, weil diese nur im Entwurf geändert werden können.

Treeview

Der Treeview basiert auf dem Microsoft-OCX-Control, welches eingebunden werden muss und dann den Verweis auf *Microsoft Windows Common Controls 6.0 (SP6)* erzeugt.

Programmierung

Der Treeview lässt sich nur per VBA mit Inhalten füllen, eine direkte Anbindung an eine Datenquelle ist nicht möglich. Nur seine Darstellungseigenschaften ließen sich im Eigenschaftfenster während des Formularentwurfs bestimmen, aber auch da ist eine zentrale VBA-Prozedur praktischer, weil diese für mehrere Treeviews gleiche Ergebnisse erzeugen kann.

Grundsätzlich wird ein Treeview mit einer Auflistung gefüllt, welche aus *node*-Objekten besteht, die später als Knoten angezeigt werden. Hat ein Knoten kein Elternobjekt, steht er in der obersten Ebene, ansonsten ist er automatisch Kind eines anderen Knotens.

Damit diese Aufgabe möglichst effizient erledigt wird, gibt es eine Prozedur *KnotenAusQuery*, welche anhand eines SQL-Codes die Knoten anlegt. Diese Prozedur erwartet immer das Vorhandensein der Felder *FeldID* und *FeldAnzeigen*, daher sind alle Abfragen so vorbereitet. Andernfalls müsste ich immer noch mitgeben, welche beiden Feldnamen hier benutzt werden sollen.

FeldAnzeigen entspricht dem sichtbaren Knotentext und *FeldID* wird in der *Tag*-Eigenschaft gespeichert, damit bei jedem Klick auf den Knoten dessen eindeutige ID bekannt ist.

Hinweis: Nebenbei zählt diese Prozedur *KnotenAusQuery* auch noch die Anzahl der hinzugefügten Knoten-Elemente und ergänzt am Elternknoten diesen Wert in eckigen Klammern.

Außerdem hat jeder Knoten einen sogenannten Zeilentyp, der angibt, welche Art Daten hier angezeigt werden. Da die ID und dieser Zeilentyp (und noch einige Inhalte mehr) zusammen in der *Tag*-Eigenschaft gespeichert werden, trennt die dortige Zeichenkette die verschiedenen Inhalte mit `|`-Zeichen.

Für die Anzeige einzelner Informationen, die nicht aus SQL-Code stammen, gibt es zusätzlich noch eine Prozedur *KnotenEinzeln* sowie eine Prozedur *KnotenEinzelnMeldung* für farblich gekennzeichnete Meldungen.

Ereignisse

Der Treeview löst verschiedene Ereignisse aus, von denen eigentlich nur zwei interessant sind: das Ausklappen/Expandieren und das Markieren/Anklicken eines Knotens.

Expandieren

Aus Gründen der Geschwindigkeitsoptimierung werden immer nur die obersten sichtbaren Knoten erzeugt. Erst wenn ein Knoten aufgeklappt ist, prüft der Code in diesem Ereignis, ob da noch ein Knoten mit *IstLeer*-Kennung enthalten ist und erzeugt in diesem Fall die tatsächlichen Knoten.

Dieser *IstLeer*-Knoten ist notwendig, weil sonst das `[+]`-Zeichen zum Ausklappen fehlen würde. Das Erzeugen der echten Unterknoten geht normalerweise so schnell, dass das nicht auffällt. Wird ein bereits mit echten Unterknoten ausgestatteter Knoten erneut ausgeklappt, fehlt der *IstLeer*-Knoten und keine Aktion wird durchgeführt.

Daher gibt es das *Knoten aktualisieren*-PopUp-Menü, um ein erneutes Erzeugen der Unterknoten zu erzwingen, die dann eventuell andere Inhalte anzeigen. Auch die Filterung der Unterknoten per PopUp-Menü nutzt diese Aktion, in dem dabei eine Filterzeichenkette übergeben und die Aktualisierung ausgelöst wird.

Damit der VBA-Code weiß, welche Unterknoten angezeigt werden sollen, ist (fast) jeder Knoten durch einen Zeilentyp gekennzeichnet. In einer recht umfangreichen `SELECT CASE`-Struktur ist hinterlegt, welcher *KnotenAusQuery*-Befehl mit welchen Parametern aufgerufen wird.

Hinweis: Durch diese (in anderen Datenbanken) sehr umfangreiche Prozedur bin ich erstmalig auf die Grenze gestoßen, dass Prozeduren maximal rund 2.100 Zeilen lang sein dürfen. Woanders gäbe es für so extrem lange Prozeduren selbstverständlich zehn Hiebe mit der Neunschwänzigen Katze. Mindestens.

Markieren

Ein Klick auf einen Knoten liest dessen *Tag*-Eigenschaft aus, in welcher sein Zeilentyp, seine ID und vor allem das darzustellende Formular hinterlegt sind. Falls das genannte Formular vorhanden ist und ein „normales“ Access-Formular zur Anzeige eines Datensatzes ist, wird es im rechten Teil angezeigt und passend gefiltert. Spezielle Formulare (Listen oder evtl. sogar Grids) erfahren über einen Parameter, welche Daten sie darstellen sollen.

ImageList zum Treeview

Der Treeview kann nur Grafiken im *ICO*-Format nutzen, die dazu in einer *ImageList* gespeichert und verbunden werden müssen. Entgegen üblicher Umsetzungen befindet sich dieses *ImageList*-Control nicht auf dem gleichen Formular wie der Treeview. Da die gleiche *ImageList* von mehreren Formularen mit dortigen Treeviews verwendet wird, wäre das technisch unsauber und würde zu unnötigen Ladezeiten führen.

Stattdessen gibt es ein Formular namens *USys_frmIcons*, welches völlig codefrei nur das *ImageList*-Control enthält und dadurch von mehreren anderen Formularen parallel und schnell zu laden ist. Wegen des Präfixes *USys* ist es zudem normalerweise ausgeblendet.

Icons vorbereiten

Da das *ICO*-Format nicht unbedingt zum Standardspeicherformat einer (Pixelgrafik-)Bildbearbeitung gehört, lassen sich andere Bildformate via Internet umwandeln. Ich nutze dazu die kostenlose Software mit dem Link www.ImageConverter.net. Dort lassen sich einzeln hochgeladene Grafikdateien umwandeln und wieder herunterladen.

Standardmäßig wird dort allerdings das Windows-Icon-Format genutzt, welches in der gleichen Datei die 16x16- bis 48x48-Pixel-Bilder enthält und dafür 32 kB Dateigröße braucht. Da die Bilder im Treeview ausschließlich 16x16-Größe haben, ist das eine nicht zu unterschätzende Platzverschwendung. Die rund 50 Icons in dieser Datenbank brauchen alleine dadurch schon 3,5 MB Dateigröße.

Nehmt stattdessen besser das *FavIcon*-Format als Ziel, denn das enthält nur ein 16x16-Bild und ist 1 kB klein. Auch wenn in Access in der *ImageList* noch ein paar Verwaltungsbits hinzukommen und nicht wirklich auf 1/32 der vorherigen Größe reduziert wird, bleiben für die 50 Icons immerhin nur 0,84 MB statt der 3,5 MB übrig.

Hinweis: Als Ausgangsgrafiken nehme ich PNG-Dateien mit Alpha-Kanal zum Maskieren, so dass deren transparente Teile im *FavIcon* automatisch berücksichtigt werden.

Enumeration für die ImageList

Da die Icons über ihre Position innerhalb der *ImageList* angesprochen werden und darin schnell mal über 50 Grafiken gespeichert sind, ist ein papierener Merktzettel für die korrekte Nummer indiskutabel. Eine Zuweisung anhand ihres eindeutigen Key-Wertes ist ebenso unbrauchbar, weil im VBA-Editor leider keine Liste dieser möglichen Werte angeboten wird.

Also führe ich parallel zur *ImageList* eine Enumeration, die sich wegen ihres erheblichen Umfangs sogar in einem eigenen Modul befindet. Alle Icons haben damit einen sprechenden Namen, der bei jeder Auswahl als IntelliSense-Liste ausklappt und sogar vom Compiler auf Korrektheit überprüft wird.

```
Enum enmIcons 'Achtung, zwingend gleiche Reihenfolge wie in USys_frmIcons!
    icnNONE = 0 'ja, ich weiß, dass es sowieso 0 wäre ...
    icnPerson
    icnBenutzerAngemeldet
    icnBenutzer
    icnLogo
    ' und noch viele mehr ...
End Enum
```

Da die Enumeration mit 0 und die *ImageList* mit der Position 1 beginnt, gibt es explizit ein Element *icnNONE*, welche ich dann einsetze, wenn doch mal kein Icon erscheinen soll.

JPGs

Auch die *PopUp*-Menüs haben optional kleine Icons vor den jeweiligen Menübeschriftungen. Diese sind ebenfalls 16x16 Pixel groß und — funktionieren nicht mit dem *ICO*-Format! Das ist mehr als ärgerlich, weil viele Grafiken im Treeview und im *PopUp*-Menü optisch gleich sind.

Eines der im *PopUp*-Menü möglichen Formate für die Icons ist *JPG*, so dass ich gezwungen bin, eine zweite *ImageList* für diese Icons zu pflegen. Bei ja identischer Größe wäre es technisch möglich, beide Dateiformate in der gleichen *ImageList* zu speichern, aber da sowieso nur ein Bild im richtigen Dateiformat funktionieren würde, ist es viel übersichtlicher, diese zu trennen. Einem bereits importierten Bild ist außerdem nicht mehr anzusehen, ob es mal *ICO*- oder *JPG*-Format war.

Daher gibt es ein zweites Formular *USys_frmJPGs* nach dem gleichen Muster, in dessen *ImageList* nur die *JPG*-Grafiken für die *PopUp*-Menüs enthalten sind. *JPGs* können grundsätzlich keine Transparenz oder Alphakanäle enthalten, daher hat sich das Thema hier erledigt.

Hinweis: Theoretisch bieten die *PopUp*-Menü-Icons eine Eigenschaft *.Mask*, welche aus einer zweiten Datei eine Maske nachladen könnte. Das wären dann mal eben doppelt so viele Dateien, daher spare ich mir das.

Auch für die JPG-ImageList gibt es eine Enumeration in einem eigenen Modul. Zur Unterscheidung beginnen deren Namen mit *jpg*.

Ribbon

Diese Oberfläche verzichtet komplett auf das Ribbon, denn Aktionen werden entweder per PopUp-Menü oder im Formular per Button ausgelöst. Anstatt jedoch die originalen Ribbon-Befehle nur weitgehend mit der Tabelle *USysRibbons* und *StartFromScratch=true* zu entfernen, wird das Ribbon gleich komplett unsichtbar gemacht. Dadurch gewinnt die Oberfläche erheblich Platz in der Höhe und hat (soweit auch der Navigationsbereich unsichtbar ist) praktisch die gesamte Fensterfläche zur Verfügung:



Das geht erfreulich einfach mit zwei Funktionen:

```
Function ZeigeRibbonAn()
    DoCmd.ShowToolbar "Ribbon", acToolbarYes
End Function
```

```
Function ZeigeRibbonAus()
    DoCmd.ShowToolbar "Ribbon", acToolbarNo
End Function
```

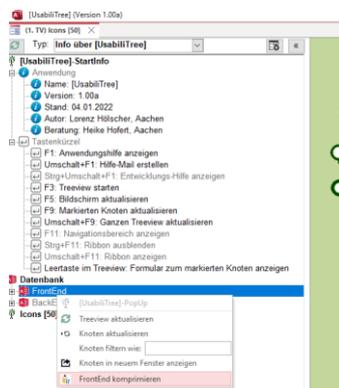
Diese beiden Funktionen wiederum werden (daher sind es Funktionen!) per Tastenkürzel aufgerufen, in diesem Fall mit *Strg+F11* und *Umschalt+F11*:



Hinweis: Leider gibt es offenbar keine wirklich funktionsfähige Prüfung, ob das Ribbon gerade sichtbar ist. Daher lässt sich das nicht mit einem einzigen Tastenkürzel umschalten.

Das Ausblenden des Ribbons wird hier direkt beim Anmelden vorgenommen. Parallel geöffnete Datenbanken sind davon nicht betroffen. Allerdings führt das zu einem neuen Problem, denn da das Tastenkürzel zum Einblenden nur mit Administrator:innen-Recht zugelassen ist, sind die originalen Access-Ribbon-Befehle nun für andere Benutzer:innen überhaupt nicht mehr erreichbar. Dazu zählt insbesondere das Komprimieren des FrontEnds.

Daher musste ich wenigstens diesen Befehl in der *UsabiliTree*-Oberfläche zugänglich machen. Da alle Befehle als PopUp-Menü mit Klick auf das zugehörige Objekt bzw. dessen Knoten verwirklicht sind, gilt das auch hier. Per Rechtsklick auf den Knoten *FrontEnd* (oder darunter den Namen der Datenbank) erscheint das passende PopUp-Menü:



Die zweite Herausforderung bestand darin, diesen Befehl funktionsfähig zu machen. Der VBA-Code zum Komprimieren

```
DoCmd.RunCommand acCmdCompactDatabase
```

funktioniert nämlich nicht für die geöffnete Datenbank, sondern zeigt lediglich den Hinweis, dass das nicht möglich ist. Aber der Trick über die `SendKeys`-Anweisung funktioniert weiterhin, jedenfalls wenn vorher das Ribbon überhaupt sichtbar ist:

```
ZeigeRibbonAn  
SendKeys "%die"
```

Hinweis: Diese `SendKeys`-Anweisung lässt sich nicht im VBA-Editor testen, sondern muss von der Access-Oberfläche gestartet werden. Ansonsten werden die Tastenanschläge ja im VBA-Editor aufgerufen und führen zum Menü-Befehl *Datei | Importieren*.

Entgegen häufiger Behauptung lässt sich das Komprimieren des gerade laufenden FrontEnds also doch per VBA auslösen.

Fazit

So machen große Datenbanken wieder Spaß! Und ich kann Euch versichern, dass jede kleine Datenbank mal groß wird, und wenn Ihr strukturell nicht vorbereitet seid, wird immer wieder drangeflickt, bis alles zusammenbricht.

Ja, es ist ein bisschen Anfangsaufwand und vielleicht müsst Ihr nicht sofort alles bei der nächsten Datenbank einbauen. Aber Ihr habt hoffentlich gesehen, wie benutzungsfreundlich sich alle Wünsche einheitlich in einer Oberfläche einbauen lassen. Und damit ist es auch programmierfreundlich, weil nicht für jeden Wunsch eine Sonderlösung notwendig ist.

Ich hoffe, dieses **UsabiliTree**-Beispiel regt Euch an, übersichtliche und leicht bedienbare Datenbanken zu erstellen. Das verbessert nebenbei auch die *accessability* und damit den Ruf von Access.

Lorenz