LIMUX

Landeshauptstadt München

OpenOffice.org Makros - „Kochbuch“

Anwendungsprogrammierung mit OpenOffice.org

„Best Practice“ - Erfahrungen aus der Umstellung von Makros im Münchener Projekt.

Autor:

Thomas Krumbein

DBI Privates EDV-Bildungsinstitut

Klarl & Schuler GmbH

Kabastastraße 5

81243 München

Juni 2012

Inhaltsverzeichnis

Lizenzvermerk:

Vorwort

Dieses Werk entstand quasi als „Abschlussarbeit“ im Rahmen der Migration der Arbeitsplätze der Stadt München von einem Microsoft geprägten Arbeitsplatz (Windows Betriebssystem und Microsoft Office) zu einem Linux-Arbeitsplatz (mit dem eigenen Betriebssystem LiMux auf der Basis von Debian-Linux sowie der freien Office-Suite OpenOffice.org).

Da die Makro-Migration von externen Dienstleistern durchgeführt wurde, stellt dieses Werk somit den „Know-How-Transfer“ dar, also die Übertragung der Erkenntnisse und des Wissens, die gesammelten Erfahrungen und die daraus resultierenden „Best-Practice“- Vorgehensweisen zur Umsetzung von (Alt-) Makros, aber auch zur Bewältigung neuer Anforderungen und Aufgaben.

Allerdings kann und soll dieses Buch keine umfassende Anleitung zum Programmieren von OpenOffice.org (oder Derivaten)-Makros sein noch ersetzt es ein Grundlagenwerk des Application Programming Interfaces (API) oder eine Anleitung wie dieses genutzt werden kann. Auch eine Einführung in die Basic-Sprache (die als Basis der vorgestellten Makroprogrammierung dient) ersetzt es nicht. Das „Kochbuch“ beschränkt sich eher auf Lösungen zu bestimmten in der Praxis vorgekommenen (Teil-) Aufgabenstellungen – und zeigt einfach nachvollziehbar die Vorgehensweise und die einzelnen Schritte. Um das Werk sinnvoll nutzen zu können, muss der Nutzer / die Nutzerin zumindest die folgenden Kenntnisse mitbringen:

- Grundlagen der Basic-Programmierung, Kenntnisse der Variablen, der Variablentypen sowie der typischen Basic-Funktionen
- Grundlagen des UNO-Frameworks – auf diesem basiert die API von OpenOffice.org
- Allgemeine Kenntnisse im Umgang mit Editoren, Dateiverarbeitungen und Dateien

Da im Rahmen des Projektes auch gemeinsam mit der Stadt München Richtlinien für die Programmierung von (Basic-)Makros erarbeitet und fixiert wurden, gehört das intensive Studium dieser Richtlinien und die Verinnerlichung ebenfalls zu den Grundlagen. Erst dann sind die Ausführungen wirklich zu verstehen und können sinnvoll eingesetzt werden.

Noch ein Wort zur Zukunft:

Während der Laufzeit des Projektes (siehe auch nächsten Abschnitt) wurden bei der Landeshauptstadt München diverse OpenOffice.org-Versionen eingesetzt – entsprechend dem zeitlichen Fortgang. Mit neuen Versionen kommen auch immer neue Features, Ergänzungen und Erweiterungen – so sind die hier folgenden Ausführungen eigentlich nur gültig für die zuletzt eingesetzte Hauptversion – OpenOffice.org Version 3.2.1. Seit der Veröffentlichung dieser Version hat sich allerdings auch viel im Projekt von OpenOffice.org selbst getan: Zunächst herrschte Stillstand, dann erfolgte eine Abspaltung (LibreOffice) und in diesem Zweig eine aktive Weiterentwicklung, schließlich ging der OpenOffice.org-Quellcode mit allen Rechten an

die Apache Foundation über, die sich nun bemüht, wieder eine lauffähige Version zu erstellen (aufgrund des Übergangs mussten die Lizenzen geklärt und Teile des Code entfernt bzw. erneuert werden, da diese nicht mit der neu zu verwendenden Apache Lizenz vereinbar waren). Dann gibt es noch einzelne Personengruppen, die sich ebenfalls bemühen, lauffähige Office-Derivate zu erzeugen.

So ist der Markt heute mit unterschiedlichen Versionen freier Office-Suiten bestückt und es ist nicht sicher, alle Aussagen hier auch auf alle Projekte übertragen zu können. Sicher funktionieren sie aber in der OOo-Version 3.2.1 sowie (meist getestet) in den LibreOffice Versionen 3.3 - 3.5.

Im Einzelfall sollte aber dennoch jede/r seine Distribution zunächst testen und eventuell Anpassungen am (Makro-) Code vornehmen.

Und jetzt viel Spaß beim „Schmökern“

Wiesbaden/München im Mai 2012 Thomas Krumbein

1 Ein Makro-Kochbuch

Die Landeshauptstadt München hat sich 2003 dazu entschlossen, auf ihren ca. 14.000 Arbeitsplatzrechnern zukünftig freie Software einzusetzen. Bis zum Sommer 2004 wurden in einem Feinkonzept die technische Umsetzung und die organisatorischen Voraussetzungen betrachtet, bevor der Stadtrat grünes Licht für die konkreten Konzepte geben konnte. Mitte 2005 wurde die Entwicklung des Basisclients begonnen und seit Frühjahr 2006 befand sich der Basisclient im Test- und Pilotbetrieb in einzelnen Bereichen der Stadtverwaltung. Mit dem Projektende Anfang 2012 sind nun nahezu alle Arbeitsplätze auf LiMUX umgestellt, dem Basisclient mit seinem Linux-Unterbau. Mit dem Wechsel auf Linux wurde ebenfalls das Office-Paket mit ausgetauscht – zum Einsatz kam OpenOffice.org, dessen Funktion und Bedienung dem bisher verwendeten Office 2000 von Microsoft ziemlich ähnlich ist. Allerdings mussten im Rahmen der Umstellung natürlich alle Vorlagen und Dokumente (soweit sie regelmäßig benötigt wurden) ebenfalls umgestellt werden – vom bisherigen proprietären MS-Format auf das jetzt verwendete offene OASIS-Format (Standard-Dateiformat von OpenOffice.org). Ebenfalls wurden die Dokumente vorbereitet und optimiert für den Einsatz des in München entwickelten „WollMux“-Systems, ein eigenes Vorlagen-Verwaltungs- und Bedienkonzept. Dadurch wurde im Rahmen der Migration ein Mehrwert für die Nutzer/innen geschaffen, die CI vereinheitlicht und Administration sowie Support vereinfacht und gestrafft.

Ein Teilprojekt im Rahmen dieser Migrationsaufgabe war die Umsetzung der verschiedenen Makros, die in den einzelnen Referaten unterschiedliche Dienste leisteten. Diese Makros waren realisiert im Microsoft eigenen VBA-Code, eine Basic-basierte Sprache. Auch OpenOffice.org lässt sich „scripten“ und bietet über seine eigene API mindestens die gleichen Möglichkeiten, die auch in MS-Office zur Verfügung stehen, und auch OOo bringt eine auf Basic basierende Rahmenscript-Sprache mit, doch sind die Objekte der Anwendungen völlig verschieden und auch unterschiedlich anzusprechen. Während die VBA-Scriptsprache noch viel stärker an ehemalige typische „Makro-Applikationen“ – also einfache Aufzeichnungen durchgeführter Schritte – angelehnt ist, bietet die OOo-API Zugriffe, wie sie in „Hochsprachen“ üblich sind – und volle Objektorientierung.

Ziel der „Makro-Migration“ war allerdings auch eine Bereinigung und Konsolidierung des aktuellen Bestandes und in jedem Einzelfall zu prüfen, ob die verwendete Makro-Lösung tatsächlich optimal für den Zweck ausgelegt ist oder ob andere Verfahren (WollMux-Vorlage, Webapplikation, Fachapplikation, Kaufprogramme) nicht effizienter und kostengünstiger einzusetzen wären. Erwartet wurde in diesem Zusammenhang, dass viele Makro-Lösungen wegfallen könnten.

Bevor die eigentliche Arbeit jedoch beginnen konnte, musste zunächst festgestellt werden, welche Makros überhaupt im städtischen Umfeld vorhanden sind und wofür diese eingesetzt werden.

Makros wurden typischerweise von qualifizierten Mitarbeitern/Mitarbeiterinnen an den jeweiligen Arbeitsplätzen zur Vereinfachung der dort anfallenden Arbeitsaufgaben geschrieben und dort auch gepflegt. Die Mitarbeiter/innen waren typischerweise keine EDV-Fachleute und haben sich das „Makro“-Wissen teilweise in Eigenregie oder in Weiterbildungskursen selbst beigebracht. In der Regel saßen die „Makro-Experten“ auf normalen Sachbearbeiter-Posten und erledigten ihre Programmierungen entweder in ihrer Freizeit oder im Rahmen ihrer täglichen Arbeit. Viele Makros vereinfachten dann die Arbeitsplatz-typischen Organisationsabläufe und wurden nach und nach zu einem integralen Bestandteil des Arbeitsplatzes – und so auch genutzt von weiteren Mitarbeitern/Mitarbeiterinnen mit gleichen oder ähnlichen Aufgabenstellungen. Über die Jahre hinweg entwickelte sich dadurch eine EDV-Subkultur, deren Funktionstüchtigkeit aber teilweise erheblich in den organisatorischen Gesamtablauf eingriffen und für viele Fachverfahren als unabdingbar galten.

Solch eine sich entwickelnde Struktur besitzt Vor- und Nachteile:

Die Vorteile liegen auf der Hand und werden schnell als „selbstverständlich“ angesehen: Die Produktivität steigt bei Vereinfachung der Arbeitsabläufe und Verbesserung der Organisation. Die Ideen der Sachbearbeiter/innen zur Verbesserung eigener Abläufe konnten direkt und unbürokratisch umgesetzt werden (man hat es einfach gemacht), die optimierten Ergebnisse wurden zum „Standard“.

Dem stehen allerdings Nachteile gegenüber, die sich erst im Laufe der Zeit zeigen: Die Dokumentation der Verfahren (Makros) ist oft dürftig beziehungsweise nicht vorhanden, von einer Standardisierung oder Normalisierung kann schon gar nicht gesprochen werden. Die Makros sind stark abhängig von ihren Entwicklern/Entwicklerinnen und dem dort vorhandenen Know-How – fällt diese Person weg (sei es durch Krankheit, Wechsel der Position oder auch altersbedingt) versiegt auch schlagartig die Fachkompetenz hinsichtlich der Makros und das Wissen über die verbesserten Abläufe. Solange die Makros weiterhin funktionieren, ist das zunächst kein Problem – Anpassungen und Änderungen können dann jedoch ein Verfahren blockieren oder sogar ganz zum Erliegen bringen. Die Produktivität sinkt in dem Fall schlagartig, der „Wiederanlauf“ ist deutlich teurer und arbeitsaufwendiger. Auch einen weiteren Nachteil sollte man nie vergessen: Die Fachverfahren (Makros) wurden zumindest teilweise während der Arbeitszeit entwickelt, gepflegt und gewartet. Hierfür wurden die Mitarbeiter/innen aber in der Regel weder eingestellt noch sind diese Aufgaben Teil ihrer Stellenbeschreibung – es wurde also Arbeitszeit für artfremde Tätigkeiten verbraucht und die mögliche Gesamtproduktivität der Stelle somit nicht erreicht. Dies führt im Extremfall zur Notwendigkeit, eine zusätzliche Stelle zu schaffen, um die Aufgaben des Arbeitsplatzes zu bewältigen, und damit zu einer Umkehr des Produktivitätsgewinns.

Zusätzlich bleiben die Risiken des Geschäftsbetriebs: Gerade bei komplexeren Applikationen, die den Arbeitsablauf nachhaltig beeinflussen, ist es zwingend notwendig, nicht nur eine allgemeingültige und komplette Dokumentation der Abläufe und Programme zu besitzen, sondern auch eine entsprechende Sicherungsstrategie (Backup) zu fahren, um Störungen des

Ablaufes so gering wie möglich zu halten und im Falle des Falles schnell Ersatz schaffen oder Störungen beheben zu können. Bei personenzentrierten Makroapplikationen ist dies aber nie der Fall gewesen – hier hängt das Wohl des Ablaufes an einigen oder wenigen „Wissenden“, also den Erstellern/Erstellerinnen bzw. Programmierern/Programmiererinnen.

Im Rahmen der Migration ergab sich nun für die LHM die Möglichkeit, die Abläufe zu straffen, Strukturen zu schaffen (wie zentrale Ablage und gemeinsame Programmierrichtlinien), Dokumentationen zu erzeugen und einheitliche Regelungen vorzugeben. Durch die Verlagerung des „Wissens“ aus den Fachabteilungen hin zu einer zentralen „IT_Dienstleistungsabteilung“ gelingt auch die Möglichkeit, Wissen zusammenzufassen, unabhängige Pflege und Support zu gewährleisten sowie die zentrale Sicherung des Wissens. Die Sachbearbeiter/innen vor Ort werden entlastet und haben wieder freie Kapazitäten, sich auf Ihre Aufgaben zu konzentrieren. Doch auch dieses System hat natürlich Nachteile: Der Verlust des „Wissens“ der Sachbearbeiter/innen am Arbeitsplatz. Dort hatte sich teilweise ein hohes Maß an EDV-Wissen angesammelt (VBA, MS Office), das mit der Umstellung komplett auf der Strecke bleibt. „Wissen“ wird nun zentralisiert, die Programmierung vor Ort untersagt. Dadurch steigen die bürokratischen Wege (bei Anpassung, Neuerstellung etc.) und es sinkt im gleichen Maße die Flexibilität (der Abteilung, der Mitarbeiter/innen). Unter Abwägung aller Vor- und Nachteile muss jedoch in größeren organisatorischen Einheiten (wie beispielsweise der LHM) der Zentralisierung Vorrang eingeräumt werden – dieser Weg ist für das Gesamtgebilde der bessere.

Nun wäre die beste Ausgangsbasis für das Migrationsprojekt gewesen, dass man sich zunächst auf Standards und (Programmier-)Richtlinien einigt und sich dann dem Projekt nähert. Doch auch ein Migrationsprozess ist eben fließend – mit Lerneffekten und neuen Herausforderungen während des Prozesses. Doch genau dies ermöglicht auch ein gutes Projekt: Anpassungen während der Projektlaufzeit an neue Erkenntnisse und somit dynamische Änderungen der Standards / Richtlinien. Jetzt – am Ende des Prozesses der Migration – lassen sich (natürlich rückblickend) eigentlich erst die besten Richtlinien für die Umsetzung definieren – und doch ist auch diese Gültigkeit begrenzt: Die Programme und die Abläufe in den Abteilungen entwickeln sich weiter und das, was heute als optimal angesehen wird, kann morgen bereits überholt sein.

Und doch wurden in den vergangenen sechs Jahren des Makro-Migrationsprojektes so viele Erfahrungen und Erkenntnisse gesammelt, dass es sich lohnt, diese als „Kochbuch“ zusammen zu stellen und als Basis für zukünftige Entwicklungen oder Anpassungen zu nutzen. Die hier dargestellten Artikel stellen typische Migrationsaufgaben dar und bieten Lösungen der Art „Best Practice“. Jedoch sollte man auch diese Lösungen nicht als „gottgegeben“ ansehen, das, was heute als machbar und „perfekt“ gilt, kann schon morgen überholt sein und als „schlechter Programmierstil“ angesehen werden. Nehmen Sie das Buch als Anregung, als Grundlage, und entwickeln Sie daraus Ihre eigenen Lösungen. Das Rad muss nicht neu erfunden werden, aber vielleicht können Sie den Rollwiderstand signifikant reduzieren.

2 Einführung in die Applikationsentwicklung

Blicken Sie zurück auf die Anfänge der „großen“ Module wie Textverarbeitung und Kalkulation, so erkennen Sie schnell die Wurzeln der Automatisierung: In Zeiten, in denen es keine grafischen Oberflächen, keine Maussteuerung und intuitive Benutzerführung gab, war man darauf angewiesen, häufig benutzte Tastaturbefehle in Gruppen zusammenzufassen und für eine spätere Verwendung aufzuheben, um sich die Arbeit und die Arbeitsabläufe zu erleichtern. Das war die Geburtsstunde der (Tastatur-)Makros, einer Ansammlung hintereinander ablaufender Tastatureingaben, die abgespeichert und sehr einfach wieder aufgerufen werden konnten.

Diese Möglichkeiten wurden in der Folgezeit immer weiter ausgebaut, und es entstanden so genannte Makrosprachen, die schon eigenständige Befehle verstanden, applikationsabhängig selbstverständlich, aber immer mehr Möglichkeiten boten. Mit den Sprachen entwickelte sich auch eine neue Art von Benutzern/Benutzerinnen: keine „echten“ Programmierer/innen, aber weit mehr als „einfache“ Anwender/innen.

Heute haben ausgewachsene Office-Programme auch ausgewachsene Möglichkeiten, die eigenen Funktionen zu manipulieren und zu gestalten – die Automatisierungssprachen sind objektorientiert, umfassend und bieten für jeden etwas. Auch die Benutzer/innen „wandeln“ sich wieder – je komplizierter die Makrosprache ausfällt und je mehr Möglichkeiten sie bietet, um so weniger „tief“ steigt der/die „normale“ Anwender/in dort ein – er/sie mutiert zurück zum/zur Benutzer/in. Und dann wird schon ein Aufruf eines Makros aus der IDE oder dem Makro-Selektor zu einem „echten“ Problem – zu kompliziert und zu fehlerträchtig.

„Makros“ bedeutet heute: eigenständige Programme, deren Bedienung und Aufruf sich an den Standard-Programmen zu orientieren haben – die mit gewohntem Bedienkomfort durch die zu bewältigende Aufgabe führen und deren Ergebnis dem erwarteten Arbeitsergebnis komplett entspricht.

Makroprogrammierung ist heute Anwendungsprogrammierung – lediglich begrenzt auf die darunterliegende Basisapplikation und auf deren Möglichkeiten (in dem Fall hier auf die freie Office-Suite OpenOffice.org und dort auf die entsprechenden Module Writer, Calc, Draw/Impress und Base).

Doch was bedeutet „Anwendungsprogrammierung“? Während sich in der Literatur hier nur sehr komplexe Definitionen zum Software-Design und ähnlichen Bereichen finden lassen, versuche ich eine eher nutzerorientierte Erklärung:

> *Anwendungsprogrammierung ist die Erzeugung eines (Software-)Programms, das den/die Nutzer/in im Bewältigen und Lösen einer Aufgabe zielorientiert unterstützt und zeitliche und/oder informative Vorteile bietet.*

Ich möchte das näher erläutern: Im Mittelpunkt steht zunächst die zu erledigende Arbeitsaufgabe sowie eine Person, die diese tatsächlich erledigt. Beendet ist die Aufgabe, wenn ein entsprechendes Arbeitsergebnis vorliegt. Dies kann zum Beispiel ein Schriftstück sein, aber eben auch ein Datensatz oder ähnliches. Die Person könnte dieses Arbeitsergebnis auch ohne Hilfe des Anwendungsprogramms erzeugen – das Programm muss nun mindestens einen zeitlichen und/oder informativen Vorteil bieten. „Informativ“ bedeutet in dem Fall, dass der/die Benutzer/in benötigte Informationen nicht selbst nachschlagen oder recherchieren muss, sondern sich auf das Programm verlassen kann, da die Informationen dort hinterlegt oder gezielt erfragt werden. Allerdings muss sich das Programm an den Fähigkeiten und Möglichkeiten des Nutzers / der Nutzerin orientieren und darf keine zusätzlichen Anforderungen stellen, die nicht zur Lösung der eigentlichen Aufgabe erforderlich wären.

Diese Definition setzt enge Grenzen der Programmiertechnik – und oft auch ein „Umdenken“ der Programmierer/innen voraus. Viele für Programmierer/innen selbstverständliche Dinge und Abläufe sind nämlich für den/die „Anwender/in“ keinesfalls üblich und verständlich – und werden von der zu lösenden Aufgabe auch nicht verlangt. Diese Prämisse muss immer oberstes Gebot einer sinnvollen Anwendungsentwicklung bleiben – egal, was „unter der Haube passiert“ - für den/die Benutzer/in müssen Abläufe und Oberflächen so gestaltet sein, wie sie seiner/ihrer Welt und Erfahrung entsprechen. Anwendungsprogrammierung bedeutet somit nicht nur die technische Lösung, sondern eben auch die organisatorische – die Schnittstelle zwischen Anwender/in und Programm klar zu beachten. Eine Überforderung des Anwenders / der Anwenderin durch für ihn/sie nicht verständliche Vorgehensweisen oder Systemmeldungen (auch Fehlermeldungen) sind unbedingt zu vermeiden, die Arbeitsergebnisse (und somit die Programmergebnisse) müssen für den/die Anwender/in so aufbereitet sein, dass sie nahtlos in den normalen Arbeitsalltag(-ablauf) des Nutzers / der Nutzerin integrierbar sind.

2.1 Voraussetzungen der Anwendungsentwicklung

Neben dem Fachwissen, das ein/e Programmierer/in sowieso mitbringen muss, also UNO, Programmierung, Kenntnisse der API etc., benötigt der/die Anwendungsentwickler/in ein „gutes Ohr am Markt“, sprich, er/sie muss wissen, wie der/die Anwender/in arbeitet und was er/sie erwartet beziehungsweise welche Aufgaben er/sie lösen kann. Dazu ist es eigentlich unerlässlich, regelmäßig vor Ort die Arbeitsabläufe kennen zu lernen und eingebunden zu sein in die organisatorische Entwicklung der Anwender-Abteilung. Ein steter Austausch von Informationen und Kontakte auch auf nicht fachlicher Ebene sind eigentlich unersetzlich. Leider sind solche Strukturen selten wirklich realisierbar, dann müssen aber Alternativen geschaffen werden (zum Beispiel eigene „Mittler-Manager“, die diese Aufgabe für mehrere Projekte übernehmen, ohne selbst dann physikalisch zu programmieren).

Unerlässlich für Anwendungsprogrammierer/innen sind also folgende Skills:

- Kommunikationsfähigkeit – Verstehen und Erläutern von Abläufen und Arbeitsprozessen.

- Einfühlungsvermögen – der/die Anwender/in (ein Mensch) ist Zielpunkt und Mittelpunkt der zu erzeugenden Anwendung – nicht die Applikation selbst.
- Schriftliche Ausdrucksweise – aus den beiden erst genannten Punkten ergibt sich, dass nicht nur die Erstellung des „perfekten“ Produktes wichtig ist, sondern eben auch die Nebenaufgaben, wie beispielsweise Schulung oder Bedienungsanleitung. Erstellt der/die Programmierer/in diese selbst, müssen sie verständlich (für den/die Anwender/in), vollständig und nicht langweilig sein.

Es reicht nicht, ein/e gute/r Programmierer/in (technisch gesehen) zu sein – es bedarf auch sozialer Kompetenz.

2.2 Der/Die „gemeine“ Benutzer/in

Anwendungsprogramme werden für den/die „gemeinen“ Nutzer/in geschrieben und dienen diesem/dieser als Arbeitsmittel zur Bewältigung seiner/ihrer Tagesaufgaben. Es macht also durchaus Sinn, sich ein wenig mit diesem Anwender / dieser Anwenderin auseinander zu setzen. Schließlich wird und muss er/sie mit dem erstellten Programm arbeiten.

Nun soll es aber keine „Typisierung“ des Nutzers / der Nutzerin geben, denn jede/r Anwender/in wird individuell verschieden und mit unterschiedlichen Voraussetzungen ausgestattet sein, sondern es soll mehr die Basislinie einer Anwendungsapplikation abgeleitet werden – vom / von der durchschnittlichen Benutzer/in.

Der 1. Grundsatz lautet dabei: Die EDV (und das Anwendungsprogramm) ist für den/die Benutzer/in ein Arbeitsmittel zur Erreichung seiner/ihrer individuellen Arbeitsaufgabe – ist also ein Hilfsmittel und weder Selbstzweck noch Mittelpunkt seiner/ihrer beruflichen Tätigkeit. Er/Sie kann seine/ihre Ziele und Aufgaben auch ohne dieses Programm erreichen (möglicherweise schwieriger, langwieriger oder umständlicher – aber eben lösbar) und damit tritt das Programm in Konkurrenz zu anderen Hilfsmitteln. Der/Die Benutzer/in wird sich immer für das entscheiden, was aus seiner/ihrer Sicht den größten Nutzen für ihn/sie selbst bringt – nicht für das absolut „beste“.

Die Aufgabe des Anwenders / der Anwenderin besteht nicht in der Bedienung des Programms – sein/ihr Arbeitsergebnis sieht anders aus. Dies kann ein zu erstellender Bescheid sein, ein Brief, ein Verwaltungsakt oder sonst irgendetwas. Nie aber ist es die Anwendungsapplikation selbst.

Durch Schulung und Erfahrung kann der/die Benutzer/in Kenntnisse erlangen, wie ein Arbeitsmittel sinnvoll für ihn/sie einsetzbar ist – dies gilt auch für EDV-Programme. Da es sich aber hierbei lediglich um Arbeitsmittel handelt, sind Lernprozesse entsprechend kurz und effektiv zu gestalten. Je schneller sich ein/e Benutzer/in zurecht findet (Optik, Benutzerführung, Dialoglayout und Hinweistexte) um so geringer ist die Einarbeitungszeit und die Akzeptanz steigt.

Der/Die „normale“ Nutzer/in ist weder in der Lage noch ist es seine/ihre Aufgabe, EDV-typische Probleme zu lösen oder entsprechende Gegenmaßnahmen zu ergreifen. Anwendungsprogrammierungen müssen dies berücksichtigen. Dies beginnt bei der (sinnvollen) Fehlerbehandlung mit verständlichen Fehlermeldungen und endet bei der möglichst einfachen Verwaltung von Optionen.

Hinweis: Viele Nutzer/innen haben sich im Laufe der Zeit durchaus fachspezifisches Wissen angeeignet – meist aus „Frustration“ oder anderen Beweggründen. Diese jedoch dürfen nicht zum Maßstab einer Anwendung werden – dafür ändert sich zu vieles zu schnell im IT-Bereich. Hier kommt der/die „normale“ Anwender/in gar nicht hinterher mit seiner/ihrer Fortbildung – und sollte das auch gar nicht.

Zu bedenken ist auch, dass der/die Nutzer/in eine einmal gefällte Entscheidung (zum Beispiel das Programm einzusetzen) durchaus wieder revidieren kann, falls er/sie andere Alternativen erkennt, die ihn/sie einfacher und effektiver zu seinem/ihrem Arbeitsergebnis bringen kann – und das ist nicht nur positiv zu sehen, auch „negative“ Ergebnisse des Anwendungsprogramms (übermäßig notwendige Administration, Fehlermeldungen, unerwartete Ergebnisse etc.) können dazu führen, dass für den/die Nutzer/in Alternativen interessanter erscheinen. Stehen dann aber Arbeitsanordnungen dem Wahlrecht des Nutzers / der Nutzerin entgegen, kommt es zum „Frust“ und zur Unzufriedenheit – und für all das sucht diese/r dann ein Ventil – eine/n „Schuldige/n“. Und das kann schnell das Anwendungsprogramm sein. Hier würde jetzt auch ein „Reparieren“ von Fehlern zunächst wenig bewirken – rein technische Lösungen beenden nicht automatisch den inzwischen emotionalen Konflikt.

2.3 Analyse und Briefing – der wichtige Einstieg

Start jeder Anwendungsprogrammierung muss die „Vor-Ort“-Analyse sein – dies gilt sowohl für neue Projekte als auch bei der Migration oder Neuausrichtung von bereits bestehenden Makros. Dieses zentrale Element legt den Grundstein der Anwendung und klärt wichtige Fragen, bevor auch nur eine Zeile Code geschrieben ist. Lassen Sie sich nicht von bestehenden Makros verleiten, auf die Analysephase zu verzichten Zwar kann man ja prinzipiell den bisherigen Code lesen und „nachbauen“ (und manchmal ist dies auch notwendig, um Details zu extrahieren), doch lässt ein bestehendes Makro keinesfalls den (organisatorischen) Arbeitsablauf erkennen und schon gar nicht die Einbindung des Prozesses in andere Aufgaben noch deren Schnittstellen. Gerade bestehende Makros haben oft eine „lange“ Geschichte und wurden Jahre vorher erzeugt – die Arbeitsprozesse haben sich aber seit der ersten Version oft geändert, Umgebungsbedingungen, Strukturen und Variablen sind heute ganz anders als „damals“. Da das Makro aber noch funktionierte, wurde es weiterhin genutzt – ohne technische Änderungen – und erfüllt dann oft zwar noch den eigentlichen Zweck, im Rahmen der aktuellen Arbeitsaufgabe aber ist dies eher suboptimal.

Das „einfache“ Umsetzen von bestehenden Makros macht beispielsweise bei benutzerdefinierten Funktionen Sinn, wie sie in Tabellenkalkulationsprogrammen gerne verwendet werden. Diese Kleinprogramme besitzen in der Regel keine UI, die Berechnungsformeln sind fix und eine Abweichung vom bisherigen Lösungsweg ist in der Regel nicht möglich. In diesen Ausnahmefällen reicht eine einfache Umsetzung des Codes auf die aktuellen API-Möglichkeiten und die programmeigene Syntax.

Bei allen größeren Makros und Anwendungsprogrammen jedoch ist eine ausführliche „Vor-Ort-Analyse“ und daraus resultierend ein ausführliches Pflichtenheft unumgänglich.

Die Analyse vor Ort umfasst dabei die folgenden Schritte (anwesend sein muss der/die Nutzer/in / Sachbearbeiter/in – bei mehrteiligen Arbeitsschritten auch alle Beteiligten – , zusätzlich sollte auch der/die zuständige IT-Betreuer/in und der/die Fachvorgesetzte am Termin mit teilnehmen). Im Rahmen der Analyse erklärt nun der/die Nutzer/in das bisherige Makro / die Anwendung und zeigt detailliert, wie er/sie das Programm nutzt. Das beginnt mit dem Aufruf, den im Rahmen der Laufzeit erfolgten Eingaben und Abläufen sowie der Präsentation des Ergebnisses (z.B. des fertigen Ausdrucks).

Der/Die Analyst/in erfragt nun darüber hinaus Details zum kompletten Arbeitsablauf, zu Rahmenbedingungen und Nebenarbeiten, zu angelieferten Datenformaten, deren Herkunft, zu Zielen des Arbeitsergebnisses und vieles mehr. Bei der LHM existiert hierzu ein entsprechendes Formblatt „Anforderungs- und Analyseprotokoll“.

Sinn der Analyse ist es, den kompletten Arbeitsablauf zu erkennen, nicht nur den Teil, der bisher durch das Makro oder die Applikation abgedeckt wurde. Erst mit dem Blick auf das Ganze lässt sich später eine optimale Lösung zur Unterstützung des Nutzers / der Nutzerin erarbeiten. Diese kann mehr oder auch weniger Features beinhalten als es die bisherige Lösung anbot. Möglicherweise sind die Aufgabenstellungen nun auch völlig anders als zu dem Zeitpunkt, zu dem das ursprüngliche Makro / die zugrundeliegende Applikation entstanden. In dem Fall macht es wenig Sinn, die neue Applikation 1:1 umzusetzen.

Mit dem Blick auf das „Ganze“ sowie mit der Erfahrung des Analysten / der Analystin wird nun gemeinsam ein (zunächst grober) Lösungsvorschlag erarbeitet. Dieser beinhaltet zunächst die Festlegung der Umsetzung (also zum Beispiel als Web-Applikation, als WollMux-Vorlage oder als Makro – dann natürlich auch Klärung der Sprache bzw. sonstiger Umgebungen wie zum Beispiel Datenbanken oder ähnlichem, auch existierende Programme, eigen oder fremd, sind mit zu berücksichtigen). In die Festlegung fließt auch ein, wieviele Nutzer/innen die zu erstellende Applikation voraussichtlich nutzen und in welchem Zeitrahmen. Eine wirtschaftliche Abwägung sollte bereits hier erfolgen. Die folgende Tabelle gibt zunächst eine „Priorisierung“ möglicher Lösungsvarianten vor; die „Entscheidungsmatrix“ hilft, den besten Weg zu finden.

Priorität	Umsetzung	Bemerkung
1	Keine Umsetzung	Die „wirtschaftlich günstigste“ Variante und immer dann anzusetzen, wenn der „Sinn“ der Makros /

Priorität	Umsetzung	Bemerkung
		Anwendung für den kompletten Arbeitsablauf nicht gesehen werden kann bzw. durch andere Arbeitsorganisation ersetzt werden kann.
2	Umsetzung mit „Bordmitteln“ des Office-Programms	Manche bestehenden Makros können problemlos durch bereits vorhandene Funktionen der Office-Suite ersetzt werden. Eventuell ist eine geringfügige „Umorganisation“ notwendig. Immer aber muss eine Einweisung/Schulung für die Nutzer/innen erfolgen.
3	Umsetzung durch Vorlagen ohne Makros – auch WollMux-Vorlagen	Gerade im Rahmen des WollMux können inzwischen viele Makros alleine durch entsprechende Vorlagen, Textbausteine und/oder Konfigurationen ersetzt werden. Für den/die Nutzer/in ergeben sich in der Regel sogar Mehrwerte.
4	Nutzung bereits vorhandener Programme, Extensions oder frei verfügbarer Erweiterungen.	Es ist zu prüfen, ob es nicht bereits intern oder extern Programme gibt, die die zu lösenden Aufgaben bereits erfüllen. Auch eine leichte Anpassung ist einer Neuprogrammierung immer vorzuziehen.
5	Browserbasierte Lösung	Eine browserbasierte Lösung (Web-Applikation) ist einer Programmlösung vorzuziehen, wenn die Parameter entsprechend stimmen. Browserlösungen sind typischerweise teurer als Makros.
6	Makro-Programmierung in Basic oder Python (Skriptsprachen)	Makro-Programmierung als Extension oder als Dokumentenmakro. Dies ist bei Neuerstellung die günstigste Methode. Das Makro ist applikationsabhängig (OOo oder Derivate), nicht aber plattformabhängig. Entsprechende Vorsorgen müssen getroffen werden.
7	plattformunabhängige Neuprogrammierung als Java-Applikation, auch Extension	Java-Applikationen sind deutlich teurer als Basic-Makros, dafür sind manche Betriebssystem-Zugriffe nur so zu realisieren. Statt Extension kann auch ein eigenes Programm entstehen.
8	Unabhängige Anwendungsprogrammierung – auch außerhalb des Office-Programms	Denkbar wären C++ oder andere Programme. Diese müssen allerdings für die jeweiligen Betriebssysteme kompiliert werden.
4-8	Kauflösung (Kauf eines vorhandenen proprietären Programms)	In den Stufen 4-8 sollte unabhängig von der internen Bewertung immer auch nach bestehenden „Kaufalternativen“ Ausschau gehalten werden und eine entsprechende Preisabschätzung erfolgen. Hierzu gehört auch die Möglichkeit, bestehende Systeme wie SAP etc. mit den gewünschten Funktionen aufzurüsten (ist jedoch in der Regel die teuerste Variante).

Die Priorisierungstabelle gibt Hinweise auf die Realisierung unter wirtschaftlichen und betriebsbedingten Gesichtspunkten und kommt zur Anwendung, wenn die technische Realisierung auf gleiche oder ähnliche Möglichkeiten stößt.

Die folgende technische Entscheidungstabelle gibt Hinweise, wann welche Lösung vorzusehen ist. Sie sollte nie absolut betrachtet werden, sondern als Hilfe zur Auswahl (z.B. Score-Tabelle o.ä.). Zur Vereinfachung:

Unter „Web-Applikation“ ist eine datenbankgestützte Anwendung zu verstehen – deren Frontend typischerweise im Browser realisiert wird (LHM Koi-System), die aber auch als lokales „Makro-Frontend“ vorstellbar ist (geringere Kosten, evtl. ausreichend). Immer aber erfolgt die Datenhaltung in einer zentralen Datenbank (inklusive standardisierter Sicherungen).

„Makro-Lösungen“ hingegen sind normalerweise lokal installiert und arbeiten auch auf dem Client-Rechner. Daten werden entweder in Calc-Tabellen oder in einfachen Textdateien gespeichert, für die Sicherung und Vorhaltung sind andere Stellen (z.B. der/die Nutzer/in, der/die Administrator/in etc.) zuständig.

Merkmal	spricht für Umsetzung als:	Bemerkung
Seltene Nutzung der Applikation (weniger als einmal pro Woche)	Makro (Dokument oder Extension)	Eine seltene Nutzung spricht eher für eine preiswerte Makrolösung.
Häufige Nutzung, täglich, evtl. mehrmals	Webapplikation, evtl. Makro	
Anzahl Nutzer/innen der Applikation klein (1-5)	Makro (Dokument oder Extension)	
Nutzer/innen arbeiten konkurrierend mit/an den Daten	Web-Applikation	
Anzahl der Nutzer/innen sehr hoch	Makro-Extension oder Web-Applikation	
Rollenkonzept notwendig	Web-Applikation	Bedeutet in dem Fall, dass nicht jede/r Benutzer/in zu allen Daten Zugang erhält.
Nutzer/innen arbeiten nicht immer am städtischen Netz	Makro-Applikation	Hier sind insbesondere mobile Arbeitsgeräte gemeint (Laptops.etc.), die keinen Zugang zum LHM-Netz haben.
Anzahl Daten(-sätze) ist sehr hoch	Web-Applikation (evtl. Makro-Extension)	Datensätze sind keine Einzelwerte, sonder eine Gruppe zusammengehöriger Daten.
Änderungen müssen als Historie erfasst werden (Log-Funktion)	Web-Applikation (evtl. Makro-Extension)	

Datenstruktur ändert sich häufig	Makro-Applikation	Datenstrukturen in Datenbanken sind eher statisch und insofern dafür weniger geeignet.
Programmfunktionen ändern sich häufig	Makro-Applikation	Makros lassen sich in der Regel einfacher anpassen und ändern.

Entscheidet man sich für eine neu zu erstellende Web- oder Makrolösung, so erfolgt bereits zu diesem Zeitpunkt gemeinsam die Festlegung des UI-Layouts, also der voraussichtlich benötigten Schnittstellen zum Benutzer / zur Benutzerin hin (Menüs, Symbolleisten, Dialoge mit den entsprechenden Eingabefeldern etc.). Hilfreich ist es, bereits jetzt einfache Skizzen der späteren Dialog- oder Formular-Oberflächen anzufertigen und gemeinsam diese zu beschließen. Typischerweise orientiert sich das neue Layout gerne am bisherigen – wurden jedoch die Aufgaben geändert und die Umsetzung optimiert, ist es oft besser, völlig neue Lösungen zu entwickeln. Dabei muss aber auch damit gerechnet werden, dass sich der/die Nutzerin ungern in neue Dinge eindenkt und gerne am bisherigen festhalten möchte. Entsprechendes diplomatisches und kommunikatives Geschick sind hier vom Analysten / von der Analystin gefragt.

Auf der Basis der Analyse schließlich wird ein ausführliches Pflichtenheft erzeugt, das alle Absprachen beinhaltet und – jetzt auch detaillierter auf die Programmierung bezogen – alle erforderlichen Funktionen, die Ein- und Ausgabeschnittstellen (hierzu zählen auch die Dialoge) und das die Besonderheiten des Projektes beschreibt. Dieses Pflichtenheft dient letztendlich dem/der Programmierer/in als Basis für die Umsetzung und gegenüber dem/der Auftraggeber/in (dem/der Nutzer/in bzw. dem Referat) auch als Abgrenzung des beauftragten Projektes. Funktionen, die im Pflichtenheft nicht enthalten sind, sind Zusatzaufwendungen und sollten unabhängig vom Projekt selbst betrachtet werden. Sie sind auch nicht abnahmerelevant und dürfen vor allem das Projekt nicht verzögern.

Das Pflichtenheft kann nach Fertigstellung nochmals vom Nutzer / von der Nutzerin (Referat) geprüft und abgesegnet werden – an sich aber wurden alle Details ja bereits während der Analyse besprochen.

Erst jetzt kann mit der Programmierung begonnen werden.

2.4 Benutzerinterfaces

Alle Anwendungsprogramme besitzen Benutzerschnittstellen. Über diese werden die Programme gestartet, Daten und Informationen eingegeben sowie Ergebnisse präsentiert. Die Benutzerschnittstellen bilden die Transferschicht zwischen dem/der (menschlichen) Benutzer/in und dem technischen Programm. An ihr entscheiden sich die Akzeptanz und die Ergebnisse der Applikation. Es ist also eine entsprechende Sorgfalt in die Gestaltung der Schnittstelle zu

investieren. Auch hier gilt wieder grundsätzlich: Maßstab ist und bleibt der/die (durchschnittliche) Benutzer/in.

2.4.1 Programmstart

Der Start des Programmes muss für den/die Benutzer/in intuitiv, kontextabhängig und passend zu seinem Wissensstand erfolgen. Dies kann zum Beispiel erfolgen durch eigene Schaltflächen in den Symbolleisten (siehe Abb.2.1), durch eigene Symbolleisten mit Schaltflächen (Abb. 2.2), durch zusätzliche Menüeinträge sowie als Hauptmenü-Punkte, aber auch als eingebundene Menü-Optionen (siehe Abb. 2.3) oder durch Schaltflächen innerhalb von Dokumenten. Oft ist es auch möglich, den Programmstart so zu automatisieren, dass der/die Benutzer/in keine eigene Aktion mehr durchführen muss (so kann der Start eines Programms zum Beispiel an das Ereignis „Öffnen eines Dokuments“ gebunden werden). Allerdings ist in diesem Fall immer ein „Plan B“ vorzusehen – das heißt, der/die Benutzer/in muss die Möglichkeit erhalten, einen „Restart“ des Programms anzustoßen, falls er/sie den normalen Ablauf aus welchen Gründen auch immer unterbrochen hatte.

Beispiele von Programmstart-Möglichkeiten

Abbildung 2.1: eingebundene Icons als Startfunktionen

Abbildung 2.2: Eigene Symbolleisten mit (Text-) Startbuttons bzw. mit Icons

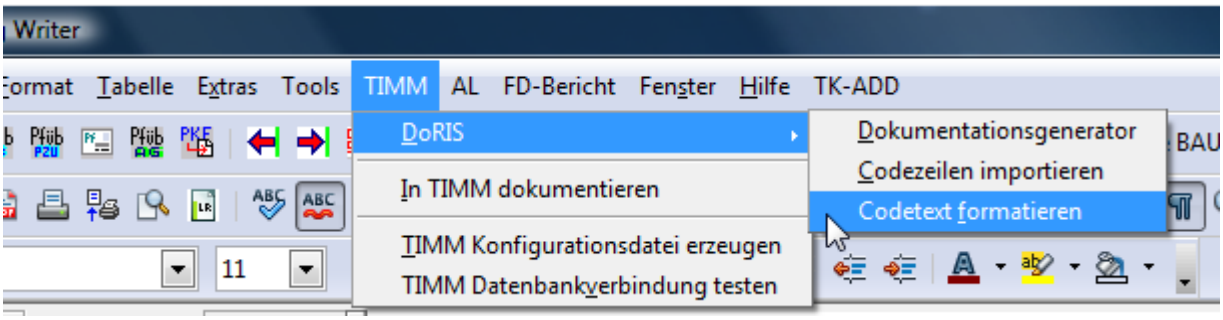


Abbildung 2.3: Eigener Menü-Eintrag mit Untereinträgen

Startmöglichkeiten sollten dem/der Benutzer/in allerdings auch nur in dem Umfeld angeboten werden, in dem das Programm sinnvollerweise gestartet werden kann. So sind Textverarbeitungsfunktionen in einem Calc-Umfeld nicht sinnvoll und umgekehrt – dies führt nur zu Verwirrungen und zu unnötigen Fehlermeldungen.

2.4.2 Dateneingabe und Benutzerinteraktion

Viele Programme erfordern Daten beziehungsweise Benutzerinteraktionen. Auch diese müssen für den/die Nutzer/in im Rahmen seines/ihres erlernten Wissens erwartbar auftreten. Typischerweise werden diese durch Dialoge erledigt. Ein Dialog entspricht dabei dem Windows-Manager des vorhandenen EDV-Systems und wird in seiner Basisfunktion allen anderen „Fenstern“ ähneln. Diese nahtlose Integration verspricht einen hohen Wiedererkennungsgrad und somit eine hohe Akzeptanz beim Benutzer / bei der Benutzerin. Andererseits müssen Dialoge aber so gestaltet werden, dass der/die Benutzer/in problemlos damit zurecht kommt und ihm/ihr die erwartete Eingabe so einfach wie möglich gemacht wird. Dazu gehören Klartext-Beschreibungen in einer Art, wie sie der/die Nutzer/in korrekt interpretieren kann, eventuell Hilfestellungen per „Tooltipp“ sowie geeignete Kontrollelemente, um Fehler von Anfang an zu unterbinden.

Beispiel: Der/Die Nutzer/in soll den Preis pro Stunde eingeben. Abb. 2.4 zeigt eine einfache Basic-Input-Box (ungeeignet), Abbildung 2.5 einen eigenen Dialog (geeignet).

Abbildung 2.4: Ungeeigneter Eingabedialog für den Stundensatz (Basic-InputBox)

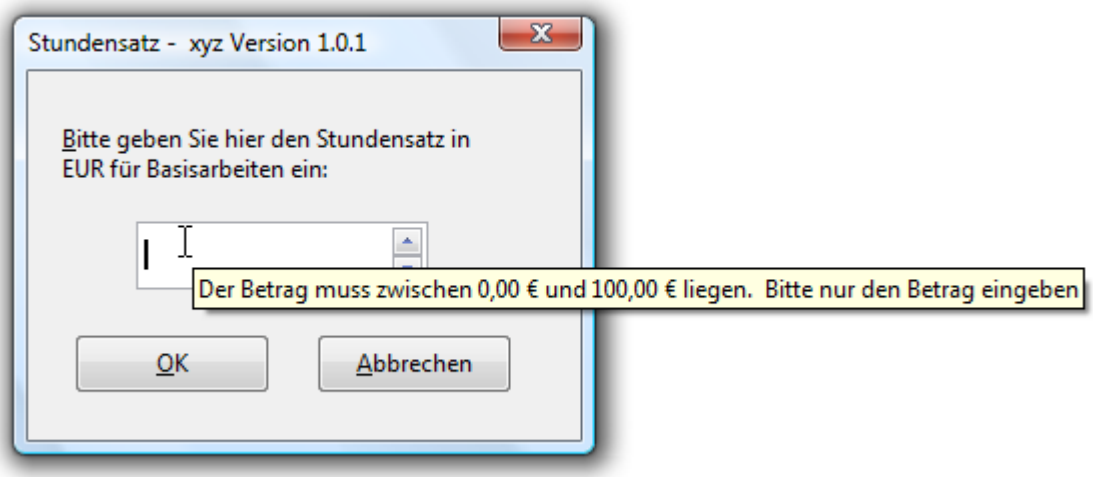


Abbildung 2.5: Eigener Dialog mit Tooltipp, Erklärungstext und Währungseingabefeld

Die Aufgabe des Programmierers / der Programmiererin ist es im Gegenzug, immer mit dem „schlechtesten“ zu rechnen und entsprechend Vorkehrungen (und Überprüfungen) vorzusehen. Das heißt, es muss damit gerechnet werden, dass der/die Nutzer/in zu viel / zu wenig eingibt oder falsche Angaben macht. Eine entsprechende Überprüfung und Rückmeldung ist immer zwingend erforderlich.

Beispiel: In einem Dialog wurde die Eingabe einer Stellplatznummer (Campingplatz) vom Nutzer / von der Nutzerin erwartet. Diese wurde dann direkt in eine Datenbank geschrieben. Der Campingplatz nutzte maximal dreistellige Nummern, wobei es nicht zwingend Ziffern sein mussten. Als Eingabefeld wurde ein Textfeld gewählt, der Hinweis lautete: „Bitte Stellplatznummer eingeben:“, die Datenbank benutzte ebenfalls Textfelder mit einer Länge von 5 Zeichen. Eine Überprüfung der Eingabe erfolgte nicht. Ergebnis: es kam zu einem Datenbankfehler. Der/Die Nutzer/in hatte tatsächlich „Platz 123“ in das Feld eingegeben – es kam dadurch zu einem Datenüberlauf.

Bei allen Eingaben muss immer mit Fehleingaben gerechnet werden, auch wenn eine eindeutige Beschreibung vorhanden ist. Nutzer/innen sind keine Programmierer/innen!

Detailliert wird auf die Gestaltung von Dialogen noch in Kapitel 6 eingegangen.

2.4.3 Ausgabemethoden

Jedes Anwendungsprogramm und jedes Makro hat auch eine Programm-Ausgabe, also ein Ergebnis. Dieses kann unterschiedlich ausfallen – muss aber immer dem/der Nutzer/in mitgeteilt werden. Wird das Ergebnis in ein vorhandenes und geöffnetes Dokument integriert so ist das Ende des Programms in der Regel erkennbar und bedarf keiner weiteren Aktion (beispielsweise ein Texteintrag in einem Writer-Dokument oder der Inhalt einer Zelle in einem Tabellendokument). Aber auch hier sollte man bedenken, dass der/die Nutzer/in zwischenzeitlich möglicherweise ein anderes Programm gestartet hat und der Fokus plötzlich

auf einem anderen Fenster liegt – und er/sie somit das Ende des Makros gar nicht mitbekommt. Hier sind also entsprechende (aussagekräftige) Hinweise vorzusehen. Dies kann durch eine kleine Mitteilungsbox erfolgen (siehe Abb. 2.6) aber auch durch Ausgabedialoge oder ähnliches.

Abbildung 2.6: Aussagekräftige "Fertigmeldung"

Wichtig: Besteht die Ausgabe im Druck eines Dokumentes, sollte vorher unbedingt dem/der Nutzer/in die Chance gegeben werden, benötigte technische Geräte zu aktivieren und betriebsbereit zu gestalten – zum Beispiel mit einer Hinweisbox: Bitte Drucker vorbereiten, einschalten, Papier einlegen etc. Zu diesem Zeitpunkt muss der/die Benutzer/in auch die Möglichkeit bekommen, den Vorgang abzubrechen ohne dass die Anwendung kollabiert.

Werden Informationen lediglich in Dialogform ausgegeben, achten Sie unbedingt darauf, dass diese sinnvoll auf den Bildschirm passen (also bei großen Textausgaben Bildlaufleisten vorsehen) und sorgen sie eventuell auch für eine Möglichkeit, die Daten per Copy&Paste aus dem Dialog auslesen zu können.

2.5 Fehlerbehandlung

Auch Fehlermeldungen sind Teil der Interaktion mit dem/der Nutzer/in. Diese müssen für ihn/sie verständlich sein und Lösungsansätze aufzeigen. Die Meldung „Ein Fehler ist aufgetreten“ ist völlig nichtssagend und für niemanden hilfreich. Der/Die Nutzer/in weiß weder, ob der Fehler bei ihm/ihr lag (eventuell durch Falscheingaben), ob es sich um ein Konfigurations- oder Einstellungsproblem handelt noch ob das Programm dennoch weiterarbeitet oder ob es seine Aktivitäten eingestellt hat. Auch bekommt er/sie keine Hinweise, wie der Fehler vermeidbar wäre oder was weiter zu tun ist.

Bevor jedoch eine Fehlermeldung ausgegeben werden kann, muss der/die Programmierer/in sich Gedanken machen, wann es zu Fehlern kommen könnte und wie darauf zu reagieren ist.

Grundsätzlich ist zu vermeiden, dass der/die Benutzer/in durch den Stopp des Programms sich plötzlich im Quellcode wiederfindet mit einer für ihn/sie unverständlichen Meldung. Dies tritt immer dann auf, wenn in Basic-Makros ein „Basic-Laufzeit-Fehler (BLF)“ auftritt und die aktive Bibliothek nicht verschlüsselt ist[1]. In diesem Fall „poppt“ die Basic-IDE auf, die Fehlerzeile wird markiert und die Meldung (BLF) wird ausgegeben. Nicht nur, dass dieses Verhalten ungünstig

1 also zum Beispiel immer bei Extensions!

für den/die Nutzer/in ist, er/sie kann jetzt auch irreparablen Schaden am Code durch unbeabsichtigte Tastatur-Anschläge oder ähnliches erzeugen. Wird die Meldungsbox geschlossen, so bleibt ja noch die Fehlerzeile markiert – und würde überschrieben werden durch jedes nun eingegebe Zeichen. Da der/die Nutzer/in kaum die Tragweite abschätzen kann noch überhaupt weiß, was gerade passiert, ist dieses Szenario unbedingt zu vermeiden.

Allerdings:Eine komplett fehlerfreie Software herzustellen ist nahezu unmöglich; der Versuch, alle möglichen Fehlerquellen im Vorfeld zu erkennen und entsprechende Routinen zum Abfangen zu schreiben, würde den Aufwand nahezu ins Unermessliche steigern. Insofern besitzen Programmiersprachen typischerweise eine sogenannte Generalklausel („Catch all-Fehlerfunktion"). Diese verhindern ein Auftreten der unverständlichen Systemmeldungen und einen Wechsel in den Debug-Mode.

2.5.1 Basic-Generalklauseln

Auch OOo-Basic besitzt an sich eine solche „Klausel“. Diese muss jedoch quasi in jeder eigenständigen Funktion neu aktiviert werden. Beispiel:

```
sub EineWichtigefunktion
  Dim sVariable as string  'Variablendeklarationen
  On Error goto Fehler:
..REM  hier folgt jetzt der Code
  exit Sub    'geplantes normales Ende der Funktion
Fehler:
  msgbox „Ein Fehler ist aufgetreten..."   'Fehlerbehandlung Bsp!
end sub
```

Jeder Fehler im Code führt nun zur Fehlerbehandlung, in der es zwar möglich ist, eine interne Basic-Fehlercode-Nummer mit auszugeben, aber eben nicht mehr einen individuellen Text. Zwar verhindert man dadurch den BLF, die Verständlichkeit des Programms sowohl für den/die Administrator/in als auch für den/die Programmierer/in sinkt so dramatisch. Bei komplizierten Codes ist eine Fehlersuche dann sehr schwierig. Insofern schränken die LHM-Makrorichtlinien schon die Verwendung dieser Generalklausel stark ein; sie sollte grundsätzlich vermieden und durch individuellere Fehlerbehandlungen ersetzt werden. Im Einzelfall jedoch ist dies auch eine Abwägung der Wirtschaftlichkeit und der Wahrscheinlichkeit des Auftretens eines Fehlers.

In Basic gibt es noch eine weitere „Generalklausel“, die in ihrer Wirkung noch radikaler ist als oben genannte:

```
on Error resume next
```

In diesem Fall wird der Fehler einfach ignoriert und die Code-Verarbeitung in der Folgezeile fortgesetzt. So bekommt weder der/die Nutzer/in zur Laufzeit noch der/die Programmierer/in während der Testphasen irgendeine Rückmeldung, dass irgendetwas nicht stimmt. Das ignorieren von Fehlern und das darauffolgende „weiterlaufenlassen“ des Programms provoziert

typischerweise Folgefehler, die allerdings durch die Generalklausel wiederum „übergangen“ würden, sodass das Programmergebnis selten nutzbar wäre.

Diese Klausel ist daher generell nicht einzusetzen!

Natürlich gibt es Ausnahmen, bei denen diese Klausel erforderlich ist; der/die Programmierer/in muss dann jedoch die Benutzung in engen Grenzen halten und durch geeignete Maßnahmen dafür Sorge tragen, dass keine Folgefehler auftreten können und der Programmfluss nicht gestört wird.

2.5.2 Aufbau von Fehlermeldungen

Schon während der Programmierung muss man sich bei jedem Schritt darüber klar sein, welche Randbedingungen erfüllt sein müssen, damit eine Programmzeile ausgeführt werden kann. Typischerweise handelt es sich bei den Randbedingungen um Variablen, die in einer bestimmten Form vorliegen müssen – aber auch Umgebungsvariablen (Objekte), die vorhanden und von einem bestimmten Typ sein müssen.

All dies muss der/die Programmierer/in entsprechend abfragen, prüfen und nur bei positivem Ergebnis den Code fortsetzen – ansonsten mit einer entsprechenden Fehlermeldung entweder verzweigen oder den Code abbrechen.

Eine „gute“ Fehlermeldung soll dabei mindestens die folgenden Inhalte ausgeben:

- Rückmeldung, dass ein „unerwartetes“ Ereignis eingetreten ist (Fehler). Dies kann auch durch entsprechende Symbole oder Titelleisten geschehen.
- Information, welcher Fehler oder welches Fehlverhalten der Auslöser des Hinweises war. Dies muss in einer dem/der Nutzer/in verständlichen Form geschehen, sowohl sprachlich als auch seinem/ihrem Erfahrungshorizont entsprechend.
- Hinweis, wie er/sie oder andere den Fehler vermeiden bzw. korrigieren können – als klare und verständliche Handlungsanweisung. Kann der/die Nutzer/in den Fehler nicht selbst beheben, so muss dieser Sachverhalt sowie die Person(engruppe), die dies kann, benannt werden.
- eventuell Hinweis darauf, ob das Programm dennoch ein Ergebnis liefert oder ob ein Abbruch hier erfolgt. Dies ist fehler- und kontextabhängig.

Ein paar Beispiele akzeptabler Fehlermeldungen (inklusive einer kurzen Beschreibung des Umfeldes und der Lösung):

In einer Dialogmaske werden zu einem Standort (Referat, Name, Straße, Ort) die dort installierten Feuerlöscher ausgegeben. Die Daten kommen aus einer Datenbank, die Standorte sind über die Indexnummer mit den Feuerlöscherwartungsdatensätzen verbunden. Es herrscht eine 1:n-Beziehung. Die Auswahl des Standortes kann bei Neuanlegen eines Datensatzes über einen eigenständigen Dialog erfolgen, der über einen Button aktiviert wird. Beim Ändern der

Datensätze darf aber der Standort nicht geändert werden – insofern darf die Auswahl nicht angezeigt werden. Wird dennoch auf den Button geklickt, so erscheint die folgende Fehlermeldung:

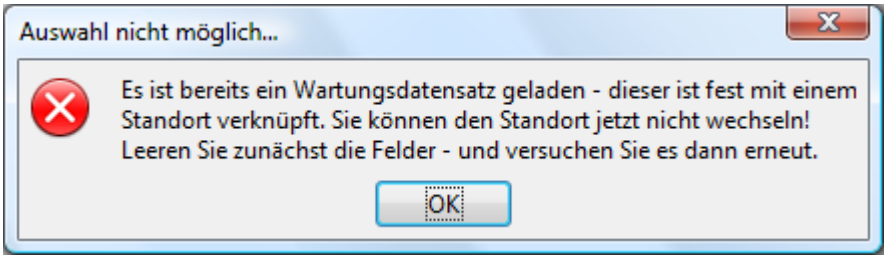


Abbildung 2.7: Beispiel Fehlertext - mit Handlungshinweis

Das Makro selbst läuft weiter, der/die Nutzer/in kann weiterhin mit dem Programm arbeiten.

Benötigte Variablen werden über die Extension „SimpleConfig“ eingelesen. Die folgende Fehlermeldung informiert über das Fehlen einer Variablen:

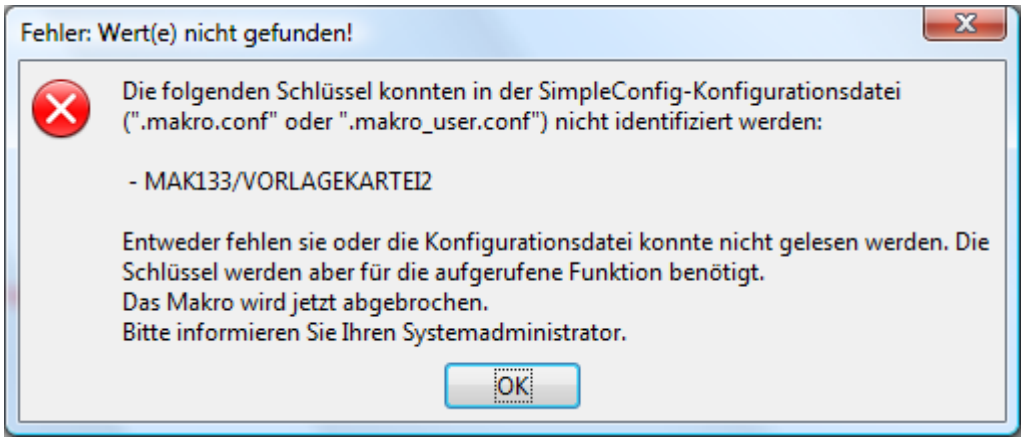


Abbildung 2.8: Beispiel Fehlertext - mit Abbruch und Handlungshinweisen

Der/Die Nutzer/in weiß hier, dass das Makro nicht weiter ausgeführt wird. Er/Sie weiß jetzt auch, dass er/sie selbst den Fehler nicht beheben kann und dass sein/ihr Systemadministrator dafür herangezogen werden muss. Der/Die Administrator/in wiederum kann aus der Meldung den Fehler leicht identifizieren und korrigieren und sich auf den fehlenden Schlüssel konzentrieren – ohne die möglicherweise weiteren 30 Variablen zu überprüfen.

Fehlermeldungen besitzen typischerweise eine „One-Way“-Aussage. Der Fehler tritt auf – das Programm geht hier nicht weiter. Es muss etwas korrigiert werden. Dies kann eine Eingabe in einem Dialog sein (fehlt oder ist fehlerhaft), dann kann es der/die Benutzer/in direkt korrigieren, oder es trat im Rahmen des Programmes ein Fehler auf – dann endet das Programm hier.

In seltenen Fällen kann der/die Benutzer/in auch beim Auftritt eines Fehlers noch eine Handlungsalternative wählen, dann muss diese klar beschrieben und auch die Buttons entsprechend mit Handlungsalternativen belegt sein. Das folgende Beispiel zeigt einen solchen

Fehler: (Im Rahmen der Verarbeitung sollte das Ergebnis auf einem bestimmten Drucker ausgegeben werden):

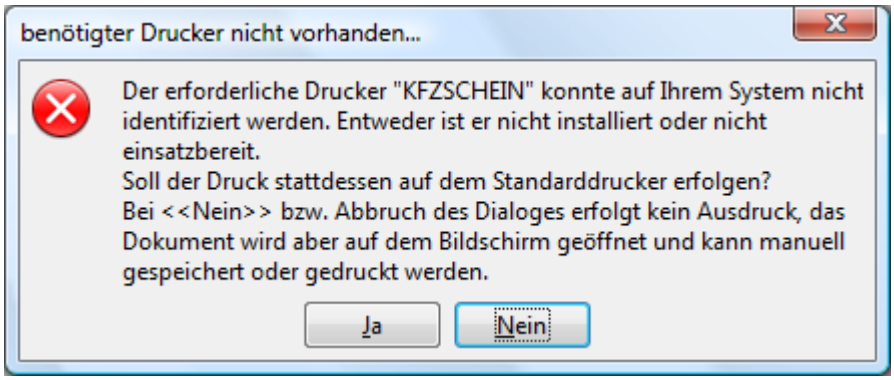


Abbildung 2.9: Fehlermeldung mit Entscheidungsabfrage des Benutzers / der Benutzerin

Dem/Der Nutzer/in werden auch die Folgen seiner/ihrer Wahl entsprechend erläutert, das Makro läuft anschließend weiter.

Achten Sie bei solchen Meldungen unbedingt darauf, dass die Meldung auch über das „Schließen“-Kreuz beendet werden kann – und berücksichtigen Sie das entsprechende Ereignis![2]

2.5.3 Fehlermeldungen bei ungültigen Eingaben

Neben den Programmfehlern gibt es durchaus auch „geplante“ Fehler – beispielsweise bei Eingabedialogen. Auch hier müssen Fehlermeldungen für den/die Nutzer/in klar ersichtlich und aufklärend sein und er/sie muss die Chance erhalten, die Eingabe zu wiederholen.

Dialoge erhalten also eine Eingabeprüfung, sind „Mussfelder“ nicht oder nicht korrekt ausgefüllt, so erhält der/die Nutzer/in einen entsprechenden Hinweis und einen Dialog zur Korrektur – eventuell noch mit den bereits eingegebenen und korrekten Daten und einer farbigen Hervorhebung der zu korrigierenden Daten. Aber auch hier darf nicht vergessen werden: Der/Die Nutzer/in muss auch die Möglichkeit erhalten, den Vorgang dennoch zu beenden – eventuell dann eben ohne das gewünschte Programmergebnis. Eine „Endlosschleife“ und das „Erraten“ der korrekten Eingabe ist immer zu vermeiden!

Typische Vertreter dieser Art sind Passwort-Abfragedialoge. Hier werden allerdings normalerweise schon vom Programmierer / von der Programmiererin die maximale Schleifenzahl vorgegeben – aber die Methodik ist übertragbar:

2 immer als Abbruch zu bewerten

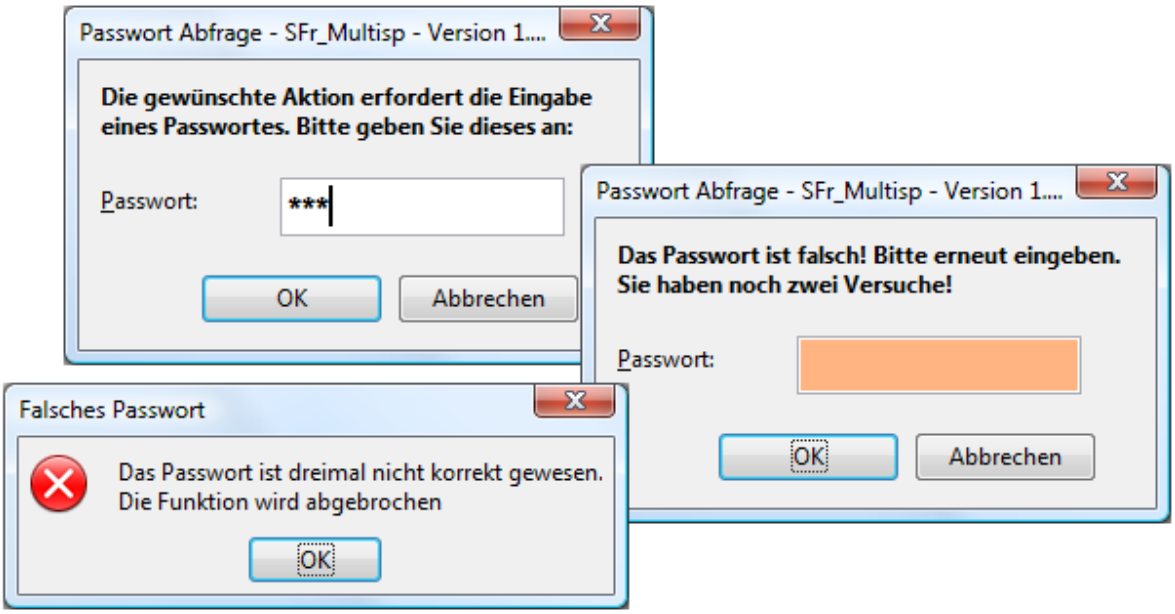


Abbildung 2.10: Eingabefehler mit Hinweisen und Abbruchmöglichkeit

Falsche oder fehlerhafte Eingaben werden entfernt und die Felder markiert, Hinweise werden ausgegeben. Zusätzlich besteht zu jedem Zeitpunkt für den/die Nutzer/in die Möglichkeit, den Vorgang abzubrechen – mit entsprechenden Konsequenzen natürlich (normalerweise Ende des Programms).

2.5.4 „Normalisierte“ Fehlermeldung

Auch wenn es nicht immer möglich ist, so hat es sich doch bewährt, eine „standardisierte“ Fehlerfunktion zu nutzen und sie dann entsprechend aufzurufen. Bitte bedenken Sie, dass diese Meldung nie alles abdecken kann und auf die Ausgabe von Standard-Fehlern mit typischerweise automatischem Programmabbruch spezialisiert ist. Eine solche Fehlerfunktion könnte lauten:

```
'/**  FehlerMeldung
'*************************************************************************.
'* @kurztext Diese Funktion zeigt eine Fehlermeldung an.
'* Fehlermeldung mit übergebenen Text.
'*
'* @param   string   Die Fehlermeldung im Klartext
'*************************************************************************
'*/
sub FehlerMeldung(sFehlertext as string)
   dim FText as string
   fText = "Bei der Ausführung des Programms ist ein Fehler aufgetreten!" & chr(13) & chr(13)
   fText = fText & "Bitte informieren Sie Ihren Systemadministrator und geben Sie" & chr(13)
   fText = fText & "folgenden Fehlertext weiter:" & chr(13) & chr(13)
   fText = fText & sFehlertext & chr(13) & chr(13)
```

```
    fText = fText & "Das Makro wird jetzt beendet."
    msgbox (FText, 16, "Programmfehler")
end sub
```

In den einzelnen Routinen wird dann der individuelle Fehlertext erzeugt und die Fehlerfunktion aufgerufen. Nach dem Aufruf der Fehlerfunktion muss die Programmroutine beendet werden – sei es als Ende der Hauptfunktion oder als Ende der Sub-Routine; dann aber muss auch dafür gesorgt werden, dass die Hauptfunktion ebenfalls beendet wird.

Aufrufbeispiel:

```
…
  sFehlerText = "Die benötigte Datei ""MeineWichtigeDatei.txt"" konnte leider nicht gefunden
werden."
  FehlerMeldung(sFehlertext)
  exit sub
…
```

Sinnvoll werden solche „normalisierten“ Fehlermeldungen immer dann, wenn zu erwarten steht, dass es viele Fehlermöglichkeiten im Programm gibt, die einen Abbruch des Gesamtprogramms notwendig machen und deren Meldetexte teilweise automatisiert werden können (und somit gleich aussehen). Kein Programm wird aber nur mit dieser Meldungsfunktion auskommen können. Zwar könnte man die Übergabeparameter flexibilisieren – so in etwa: <Titeltext>, <Fehlertext>, Anzeigeverhalten Buttons inklusive Grafik – und einen Rückgabewert liefern lassen, der den gedrückten Button (also das Ereignis) repräsentiert, jedoch steigt der Aufwand dann pro Fehlermeldung stark an und könnte bei gleichem Aufwand auch direkt gelöst werden. So werden also durch eine eigene Fehlerfunktion keine echten Vorteile gewonnen, eher Flexibilität eingebüßt.

2.5.5 Zusammengefasste Fehlermeldungen

Bei Fehlermeldungen ist ebenfalls immer zu prüfen, ob „Einzelfehlermeldungen“ tatsächlich sinnvoll sind oder ob es nicht besser zusammengefasste Meldungen geben sollte. Ein Beispiel:

In einem Dialog werden Daten erfasst – darunter sind 10 Felder mit Mussdaten (also Daten, die ausgefüllt werden müssen), wobei im letzten Muss-Datenfeld auch noch ein bestimmtes Format (z.B. Zahl zwischen 10 und 20) vorgeschrieben ist.

Füllt der/die Nutzer/in nun nicht alle Daten aus, so erfolgt eine Fehlermeldung. Je exakter diese dem/der Nutzer/in beschreibt, was genau fehlt, um so besser ist die Reparatur. Jedes Feld einzeln zu prüfen und bei Abweichungen zum Soll-Wert sofort eine Fehlermeldung auszugeben führt im Extremfall zu 11 einzelnen Meldungen (10 Felder + eine zusätzliche Formatprüfung) und zu mindestens elf Rekursionen für die korrekte Eingabe – dies ist nicht wirklich zumutbar für den/die Benutzer/in.

Auch eine „allgemeine“ Fehlermeldung wie „Nicht alle Mussdaten wurden eingegeben“ ist nicht wirklich zielführend. In diesem einfachen Beispiel (9 Textfelder + 1 Zahlenfeld) könnte eine Fehlerprüfung wie folgt aufgebaut sein (dabei werden die erklärenden Bezeichner der einzelnen Textfelder in einem Array vorgehalten, die Textfelder selbst besitzen durchnummerierte Namen der Art txt_eing1):

```
…
dim aFliste()     'Fehlerliste
dim aFelder()   'Bezeichner der Felder
aFelder = array("Name", "Vorname", "Namenskürzel", "Abteilung", "Telefonnummer", "E-Mail",
"Adresse", "Straße", "Postleitzahl", "Ort")
REM Eingabekontrolle
n = 0          'Vorgabe Fehlerzähler
for i = 0 to uBound(aFelder())
  oCtl = oDlg.getControl("txt_eing" & i+1)
  if trim(oCtl.getText()) = "" then     'keine Eingabe - Fehler!
    redim preserve aFListe(n)      'Fehlerliste um einen Eintrag erweitern
    aFListe(n) = " - das Feld """ & aFelder(i) & """ muss ausgefüllt werden"
'Feldbezeichnung
    n = n + 1                 'Fehlerzähler um eins erhöhen
  end if
next i    'nächstes Feld
REM jetzt Zahlenfeld auswerten
oCtl = oDlg.getControl("num_anzahl")
if oCtl.value = 0  then         'keine Eingabe
    redim preserve aFListe(n)      'Fehlerliste um einen Eintrag erweitern
    aFListe(n) = " - die Anzahl der gewünschten Exemplare muss eingegeben werden"
    n = n + 1                 'Fehlerzähler um eins erhöhen
elseif oCtl.value < 10 OR oCtl.value > 20 then
    redim preserve aFListe(n)      'Fehlerliste um einen Eintrag erweitern
    aFListe(n) = " - die Anzahl der gewünschten Exemplare muss zwischen 10 und 20 liegen!"
    n = n + 1                 'Fehlerzähler um eins erhöhen
end if
REM jetzt Fehlerliste auswerten
if uBound(aFListe) > -1 then    'Fehler vorhanden
  sTxt = "Die erforderlichen Eingaben des Dialoges wurden nicht vollständig oder nicht " &
chr(13) & _
"korrekt ausgefüllt. Folgende Fehler konnten identifiziert werden:" & chr(13) & chr(13)
  sTxt = sTxt & join(aFListe(), chr(13)) & chr(13) & chr(13)
  sTxt = "Bitte ergänzen bzw. korrigieren Sie zunächst die Einträge."
  msgbox (sTxt, 64, "Fehlende oder inkorrekte Einträge...")
  exit sub
end if
… 'hier würde jetzt die Fortsetzung des Programms starten bei fehlerfreier Eingabe
```

Die Fehlermeldung wird hier also über einen Fehlersammel-Array aufgebaut. Solange dieser nicht leer ist, wird die Meldung angezeigt – der/die Nutzer/in wird aber im Klartext darüber informiert, welche Felder noch auszufüllen sind. Unterstreichen kann man das noch durch entsprechende Hintergrundfarben des Kontrollfeldes. Aber nicht vergessen: Diese müssen auch wieder zurückgesetzt werden vor einem neuen Schleifendurchlauf (Prüfung)!

2.5.6 Einzelfehler ohne Meldung

Eine letzte Gruppe von Fehlerbehandlungen soll auch noch erwähnt werden: Nicht immer muss es zu einer entsprechenden Fehlermeldung kommen – es gibt auch durchaus Fälle, in denen mit einem Fehler gerechnet werden muss, ohne dass dieser erheblich für den Programmablauf ist oder diesen nachhaltig stört. Solche Fehler müssen dennoch abgefangen und „neutralisiert“ werden, denn sonst würden sie trotzdem eine Systemfehlermeldung produzieren und das Makro abstoppen.

Ein Beispiel wäre die folgende Situation: Sie möchten erkunden, ob eine Datei bereits geöffnet ist und somit ein Fenster auf dem Bildschirm repräsentiert. Dies ist wichtig für das „Filehandling“ (siehe auch Kapitel 3). Zum Check prüfen Sie nun jedes bereits geöffnete Fenster hinsichtlich des Titels (Titelzeile). Dabei kann es nun vorkommen, dass ein Fenster gar keinen „Titel“ besitzt oder dieser gar nicht ausgelesen werden kann. Geprüft wird jetzt, ob das Fensterobjekt ein Interface "com.sun.star.frame.XModel" unterstützt – nur dann kann ein Titel überhaupt ausgelesen werden. Leider gibt es aber auch Komponenten, die diese Eigenschaften überhaupt nicht besitzen – und schon die Prüfmethode „HasUnoInterface“ schief läuft und einen Fehler provoziert. Nun sind diese Elemente völlig irrelevant für die eigentliche Prüfung und für den Programmfortschritt – der Fehler ist also ebenfalls nicht wichtig – er muss aber abgefangen werden, um einen Programmstopp zu verhindern. In diesem Fall kommt dann das „on Error resume next“ zum Einsatz, d.h. ein Fehler wird ignoriert und das Programm läuft einfach weiter. Da die Funktion gekapselt ist und weitere Auswirkungen nicht zu erwarten sind, kann diese Generalklausel hier verwendet werden – ja, sie muss eigentlich. Als Programmierer/in muss einem aber klar sein, welche anderen Fehler auftreten könnten, die für den Programmverlauf entscheidend wären, und man muss entsprechend die Funktionen so kleinteilig gestalten, dass die Verwendung der Generalklausel keine anderen Auswirkungen hat als die gewünschte. Hier das Code-Beispiel – detailliert später nochmals in Kapitel 3.2.2:

```
Function CheckFensterOffen(sFensterTitel As String)
  Dim oComp as variant   'Alle Komponenten
  Dim oElement as variant  'ein einzelnes Element
  dim sFrameTitle as string, oLeer as object

  oComp = StarDesktop.getComponents.CreateEnumeration
  on Error resume next
  Do While oComp.HasMoreElements
    oElement = oComp.NextElement()
    If HasUnoInterfaces(oElement, "com.sun.star.frame.XModel") Then
      sFrameTitle = oElement.currentController.Frame.title

      If left(lcase(sFrameTitle), len(sFenstertitel)) = lCase(sFensterTitel) then 'Fenster
schon offen
         CheckFensterOffen = oElement
         exit function
      end if

    End If
  Loop
  CheckFensterOffen = oLeer
```

```
end function
```

Typische Fehlerquellen:

Eine Basic-Programmierung von OOo bietet vielfältige Möglichkeiten einer bewussten oder unbewussten Fehlerquelle. In den Einzelkapiteln wird noch regelmäßig auf Besonderheiten hingewiesen, hier also zunächst nur eine grobe Tabelle der möglichen Fehler – und Ihrer Prüfungen:

Fehler	Beschreibung	Lösungsgedanken
ThisComponent	interne Variable – verweist auf das aktuelle Dokument. Fehlerquelle: Der/Die Nutzer/in wechselt während der Laufzeit des Makros das Dokument – Fehler!	Tritt typischerweise bei Extensions auf. ThisComponent nur einmal nutzen und das Dokument dann einer Variablen zuordnen – dann damit weiterarbeiten.
Dateien	falscher oder fehlender Pfad, falscher oder fehlender Dateiname	Immer testen! Sowohl Pfad als auch Datei. Bei zusammengesetzten Werten immer auf Trenner hin überprüfen. Darf weder fehlen noch doppelt vorkommen.
Dateien vorhanden	werden vorhandene Dateien überschrieben, kann es zu Fehlern kommen, da das Betriebssystem die Aktion verweigert.	Immer prüfen, ob die Datei evtl. geöffnet oder sonst wie gesperrt ist – dann zunächst dieses Hindernis beseitigen. Rechteprobleme lassen sich in Basic nicht lösen!
Datei speichern/öffnen Fehler bei Http-Adressen	bei Verwendung einer Http-URL treten teilweise Fehler auf	Die internen Methoden „convertToURL“ bzw. „ConvertFromURL“ funktionieren nur mit dem Protokoll „File“, nicht mit „http“ oder anderen Protokollen. Dadurch werden die Pfade und Dateinamen nicht codiert, Leer- und Sonderzeichen führen zu Fehlern! Prüfen und ggf. manuelle Ersetzung.
Position in Textdokumenten	bei Benutzung von Cursern kommt es zu Fehlern	Position im Dokument prüfen. Durch die Hierachie im Dokument kommen unterschiedliche Objekte in Frage. Position muss vorher eindeutig sein!
falscher Dokumenttyp	klassischer BLF – Objekt nicht vorhanden	Unbedingt zuerst prüfen, ob das erwartete Dokument überhaupt vorliegt – gerade bei Extensions können diese oft in allen Modulen gestartet werden.
Nicht vorhandene Objekte	tritt typischerweise bei Extensions auf – ab und zu auch bei Dokumentenmakros	Bei allen Zugriffen auf ein (Dokumenten-) Objekt zunächst dessen Existenz prüfen. Dabei berücksichtigen, welche Möglichkeiten der User insgesamt so hat. Beispiel: Einen View-Cursor gibt es in der Seitenansicht des Writerdokumentes

Fehler	Beschreibung	Lösungsgedanken
		nicht – könnte aber gerade vom Nutzer / von der Nutzerin aktiviert worden sein.
Überlauf von Arrays oder Variablen	BLF – einem Array wird ein 11. Element zugeordnet, obwohl die Definition nur 10 vorsieht, eine Zählvariable für Zeilen in Calc wurde als Integer definiert, der Bereich umfasst aber mehr als 40000 Zeilen	Vermeiden von statischer Fixierung bei Arrays, korrekten Typ der Variablen wählen.
Falscher Datentyp	klassischer Fehler bei Datumsfeldern in Dialogen und Eintragungen in Datenbanken.Wurde „nichts“ in ein Datumsfeld eingetragen, liefert dieses den Wert 0 (Null) zurück. Dies entspricht dem Datum 30.12.1899 – ein Datum, das in den meisten Datenbanken nicht akzeptiert wird (dort beginnt die Zählung typischerweise beim 01.01.1900 oder später!	Prüfung des Wertes. Ersetzen des „Nullwertes“ durch einen eigens definierten „Nichts“-Wert – also z.B. 1900-01-01 – und dieser Wert wird wieder durch Null bzw. Nichts ersetzt beim Auslesen des DS und Darstellen der Werte.

2.6 Applikationen verteilen

Für die Verteilung der fertigen Applikation gibt es grundsätzlich zwei verschiedene Möglichkeiten:

- als Dokumentenmakro – dann sind die Funktionen in einer eigenen Bibliothek im Dokument vorhanden. Das Dokument muss nur dem/der Nutzer/in bereitgestellt werden.
- als OpenOffice.org(oder Derivate)-Extension. Dabei handelt es sich um eigene Archiv-Dateien (Zip-Archive) einer bestimmten Struktur, die über den Extension-Manager des Programms direkt eingelesen und integriert werden. Typischerweise werden diese Bibliotheken im User-Verzeichnis gespeichert, Extensions können aber auch vom Administrator / von der Admistratorin „- shared“ installiert werden, dann werden sie im Programmverzeichnis gespeichert.

Folgend einige Vor- und Nachteile der Verteilungsmöglichkeit sowie einige Hinweise zum optimalen Einsatz.

2.6.1 Dokumentenmakros

Im Fall eines Dokumentenmakros ist der komplette Code und alle Dialoge im Dokument gespeichert. Typischerweise bedarf es keiner weiteren Bibliothek (außer eventuell der Bibliotheken, die das Programm OpenOffice.org standardmäßig immer mitbringt – z.B. die Bibliothek „Tools“).

Der Vorteil liegt auf der Hand: Wo immer die Datei geöffnet wird stehen die Makros und somit die programmierten Zusatzfunktionen stets zur Verfügung.

Den Vorteilen sind jedoch die nicht zu unterschätzenden Nachteile entgegen zu halten: Das Dokument kann beliebig kopiert und reproduziert werden – der Code wird dadurch redundant mehrfach vorgehalten (in jedem Dokument). Auch wenn heute Speicherplatz kein großes Thema mehr ist, ist dies doch ein gravierender Nachteil, der insbesondere offensichtlich wird, wenn es eine neue Version des Makros gibt. In diesem Fall erfolgt ja kein „Update“ der bestehenden Dateien, sondern es werden neue Dateien erzeugt und verteilt – so dass nach mehreren Versionswechseln viele unterschiedliche Dateien mit unterschiedlichen Programm-Versionen im Umlauf sind – ohne echte Kontrolle, wer gerade mit welcher Version arbeitet.

Wann sollten Dokumentenmakros genutzt werden:

Bereits im Vorfeld muss entschieden werden, ob man ein Dokumentenmakro erstellt oder besser eine Extension. Ein Dokumentenmakro macht Sinn, wenn möglichst viele der folgenden Punkte zutreffen:

- Die Anzahl der Personen, die mit dem Makro arbeiten ist sehr klein (ideal eine Person). Es besteht keine Notwendigkeit, das Dokument weiterzugeben oder anderen Personen zur Verfügung zu stellen.
- Die Aufgaben des Makros liegen darin, einen bestimmten „Output“ zu erzeugen, der lediglich gedruckt, nicht aber gespeichert wird. Mehrmaliges Aufrufen und Durchlaufen des Makros überschreibt die vorherigen Daten.
- Die Datei wird gleichzeitig als „Datenbank“ genutzt, es werden also Daten „archiviert“. In dem Zusammenhang muss allerdings die Prüfung erfolgen, ob nicht eine „echte“ Datenbank die bessere Alternative wäre (siehe auch Kapitel 2.3).
- Es handelt sich um benutzerdefinierte Funktionen (nur Calc), die allerdings auch nur in der aktuellen Datei benötigt werden.
- Das Dokument wird an unterschiedliche Stellen weitergegeben, die alle Ergänzungen oder Eingaben vornehmen. Es kann aber nicht sichergestellt werden, dass diese Personen auch eine entsprechende Extension installiert haben bzw. installieren können/wollen. In dem Fall wären Dokumentenmakros die bessere Wahl.
- Die Funktionen sind eher „klein“ (im Zeilenumfang) und ihre Bedeutung ist nicht strategisch. Eine Extension wäre „überdimensioniert“.

Es gibt sicher noch ein paar weitere Argumente – in der Summe jedoch sollten Dokumentenmakros eher die Ausnahme bleiben.

Es ist auch davon abzusehen, Makros in Vorlagen unterzubringen. In dem Fall nämlich befindet sich der Code auch komplett in jedem auf der Basis der Vorlage erzeugten Dokument. Diese Redundanz ist normalerweise nicht wünschenswert und wenig „zweckmäßig“. Es bleibt auch zu bedenken, dass Dokumentenmakros (wie alle anderen Makros im übrigen auch) potentielle Gefahrenquellen darstellen und somit entweder gar nicht ausgeführt werden (Voreinstellung von OpenOffice.org und Derivaten) oder den/die Benutzer/in zwingen, eine Sicherheitslücke auf seinem/ihrem Arbeitsplatz zu öffnen.

Natürlich aber gibt es Anwendungsfälle und sinnvolle Dokumentenmakros.

Werden Dokumentenmakros eingesetzt, so sind die Startfunktionen ebenfalls normalerweise nur auf das Dokument begrenzt – also zum Beispiel die individuelle Symbolleiste, der zusätzliche Menü-Eintrag oder der „Startbutton“. Die Installation dieser Möglichkeiten ist einfach und kann direkt über die UI (Extras – Anpassen) erledigt werden.

2.6.2 Extensions

Extensions (Erweiterungen) „klinken“ sich in das Hauptprogramm von OpenOffice.org ein und stehen dann wie eingebaute Funktionen zur Verfügung. Ein eigener Programmteil (der Extensionmanager) übernimmt die Verwaltung der Erweiterungen und steuert auch einen möglichen Update-Prozess. Dieser ist problemlos jederzeit möglich.

Eine Extension wird als Datei verteilt (Dateierweiterung *.oxt bzw. teilweise auch noch *.zip oder *.uno.pkg) und kann direkt bzw. auch vom Administrator /von der Administratorin via Software-Verteilung installiert werden. Der Code ist damit sofort nutzbar – um ihn aber zu starten, sind auch bei Extensions entsprechende Startmöglichkeiten für den/die Nutzer/in vorzusehen (typischerweise eigene Symbolleisten oder Menü-Einträge). Diese müssen der Extension beigefügt werden (siehe hierzu auch ausführlich Kapitel 9 – Applikationen verteilen), sind dann aber mindestens modulweit sichtbar – also auch für jedes Dokument. Das kann gewünscht sein, manchmal aber ist es nicht möglich, spezielle Funktionen in allen Dokumenten auszuführen – dann müssen schon während der Programmierung entsprechende Merkmale zur Unterscheidung der (berechtigten) Dokumente mit berücksichtigt werden.

Analog zu den Dokumentenmakros hier einige Punkte, die dafür sprechen, eine Extension zu verwenden:

- Die Anzahl der Personen, die mit dem Makro arbeiten ist eher groß. Die Installation der Extension muss auf vielen Rechnern erfolgen (Softwareverteilung). Man erhält dadurch eine einheitliche Basis und gleiche Versionen.
- Die Aufgaben des Makros liegen darin, einen bestimmten „Output“ zu erzeugen, der immer auch gespeichert wird. Mehrmaliges Aufrufen und Durchlaufen des Makros

erzeugt neue Daten; die gespeicherten Daten können erneut bearbeitet oder verändert werden.

- Die Applikation greift auf mehrere Dateien zu – auch unterschiedlichen Typs (Writer, Calc, etc). Der Start der Applikation ist nicht dateiabhängig.
- Es handelt sich um Funktionen, die generell nutzbar sind und den Funktionsumfang des Programms erweitern.
- Daten werden unabhängig von Dokumenten bearbeitet (bzw. als Frontend einer Datenbank) – aber auch die Aufbereitung von externen Textdateien oder CSV-Dateien gehört hier dazu.
- Die Funktionen sind eher „groß“ (im Zeilenumfang), ihre Bedeutung ist hoch. Stabilität und Pflege sind wichtig.

Auch hier gibt es sicher noch viele andere Argumente, die man anführen könnte. In der Praxis zeigt sich, dass eine Extension immer die erste Wahl sein sollte – und man nur im Einzelfall auf Dokumentenmakros zurückgreifen sollte.

Weitere Details zu Extensions siehe auch Kapitel 9.

2.7 Pflege, Archivierung und QS

Programmierte Anwendungen müssen auch gepflegt und geprüft werden. Diese Aufgaben verschlingen mindestens genauso viel Zeit wie die eigentliche Erstellung. Es gibt daher einige Grundsätze, die von Anfang an beachtet werden sollten, um Doppelarbeiten zu vermeiden.

Zu jedem Programm gehört eine entsprechende Dokumentation – sowohl für den/die Benutzer/in des Programms als auch für zukünftige Entwickler/innen. Gerade wenn der Code von unterschiedlichen Personen später gewartet werden soll, ist es für diese immens wichtig, zu verstehen, wie der/die ursprüngliche Autor/in gedacht und welche Schritte er/sie wie umgesetzt hat. Das erleichtert das Eindenken in den Code und in die Problematik und wird dann auch zu schnelleren Ergebnissen führen.

2.7.1 Kommentare

Unabdingbar sind somit Kommentare im Code. Davon kann es gar nicht genug geben. Bereits während der Erstellung sollte der/die Programmierer/in jeden einzelnen Schritt dokumentieren – nachträglich macht man das nicht mehr. Zum Ende des Projektes hin wird die Zeit normalerweise knapp, die Schwerpunkte liegen dann auf anderen Aktivitäten – und der Code bleibt „unvollständig“.

So ist generell zu empfehlen, immer zuerst die formalen Teile einer Prozedur zu schreiben (also Kommentarkopf etc.) und erst dann mit dem eigentlichen Code zu beginnen. Und auch dieser

sollte von Anfang an „korrekt“ entsprechend den Makrorichtlinien ausgestaltet sein – nicht erst mal „provisorisch“ mit Kurzbezeichnern und Hilfsvariablen.

Als „best practice“ hat sich bewährt, Code-Strukturen zunächst mit Kommentaren zu definieren und später dann Stück für Stück abzuarbeiten und mit Leben (Code) zu füllen. Eine solche „Hülle“ könnte zum Beispiel wie folgt aussehen:

```
'/** MAK133_StartApplikation
'*************************************************************************.
'* @kurztext startet die Applikation
'* Dies ist die Hauptstart-Funktion. Es startet den Hauptdialog.
'*
'*************************************************************************
'*/
Sub MAK133_StartApplikation

  REM Prüffunktion, ob Makro läuft

  REM Basisinitialisierungen
  if NOT MAK133_HP1.MAK133_InitParameter then exit sub   'SimpleConfig lesen - bei Fehler Ende

  REM Startbildschirm

  REM  Erzeugen und Initialisierungen Hauptdialog

  REM Linux/Windows Optimierung Dialog

  REM Initialisierungen abgeschlossen - Startdialog ausschalten, Start Hauptdialog

  REM Aktuelle Daten speichern

End Sub
```

Dies strukturiert den Ablauf, gibt die Gedanken des Programmierers / der Programmiererin wieder, die er/sie sich gemacht hat, bevor der Code begonnen wurde und liefert gleichzeitig das Gerüst für die eigentliche Programmierung. Die Kommentierung verbleibt und liefert dem/der späteren Leser/in gleichzeitig einen einfachen Einstieg in die Struktur des Programmes.

Details zur Programmierung sind definiert in den Makrorichtlinien, die unbedingt beachtet werden sollten. Dann ist es auch möglich, den Code später automatisiert dokumentieren zu lassen (in ein Writer-Dokument) und ihn so lesbar zu sichern.

2.7.2 Archivierung

Sowohl während der Entwicklung als auch später ist es sinnvoll, regelmäßige Sicherungen des Codes durchzuführen und Zwischenstände zu fixieren. Während dies bei Dokumentenmakros noch relativ einfach ist (hier wird einfach das Dokument unter einem anderen Namen abgespeichert), scheitert es bei Extensions. Hier müssen regelmäßige manuelle Sicherungen (Export der Bibliothek in vorher erstellte Unterverzeichnisse einer Speichereinheit) erfolgen. Es empfiehlt sich, tatsächlich die Exportfunktion „Als Basic-Bibliothek exportieren“ im Verwaltungsdialog, Reiter „Bibliotheken“, zu nutzen und nicht „als Extension exportieren“ zu

wählen. Dadurch lassen sich später die einzelnen Dateien auch direkt mit einem Editor bearbeiten und es gibt auch eine klare Trennung zwischen „Entwicklung“ und „Releases“.

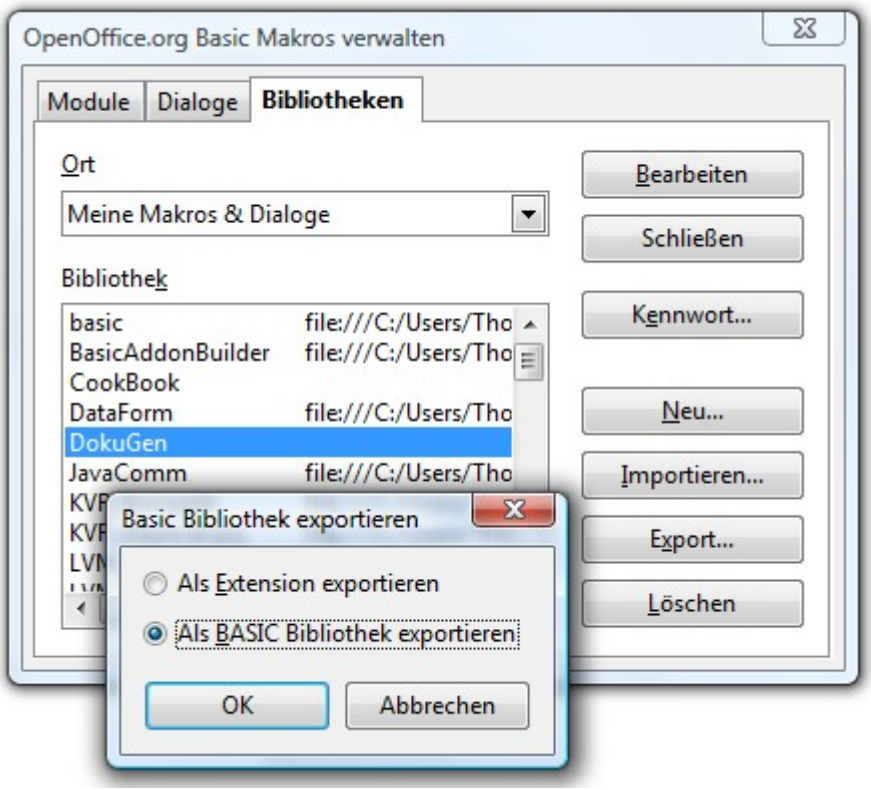


Abbildung 2.11: Sicherungsmöglichkeit bei Extension-Entwicklung

Theoretisch ist es sogar möglich, mit Hilfe eines Repositorys (zum Beispiel Git) die exportierten Dateien zu versionieren und somit automatisch zu archivieren – inklusive der Zwischenstände.

Unabhängig davon sollte der/die Programmierer/in auch für sich mit einer Versionierung arbeiten. Spätestens ab dem Release-Candidaten (RC) ist sie sowieso notwendig – aber man kann sie intern auch schon von Beginn an mitführen und entsprechend hochzählen. Alle Versionsstände werden archiviert.

Es hat sich auch bewährt, ein ausführliches Protokoll der Änderungen von Version zu Version zu führen, das erleichtert das Einlesen in den Code und – im Programmzyklus nicht unüblich – die Rückname einer Änderung/Erweiterung. Während eine Kurzform der Versionsänderungen direkt im Code gepflegt wird, sollte die ausführliche Beschreibung in einem separaten Dokument fortgeschrieben werden.

Zur Archivierung gehören die folgenden Unterlagen:

Code-Stand, Übungs- und Hilfsdateien, die für Testzwecke genutzt wurden, Dokumentationen soweit schon vorhanden, Konfigurationsdateien. Nur so lässt sich ein Teststand erneut reproduzieren.

2.7.3 Qualitätssicherung (QS)

Die Qualitätssicherung ist ein wesentlicher Prozess sowohl während der Programmierung als auch vor der Auslieferung. Allerdings unterscheiden sich beide Prozesse voneinander:

2.7.3.1 QS-Test vor der Auslieferung

Vor jeder Auslieferung des Produktes an den Kunden steht eine (interne) Qualitätsprüfung an. Diese kann nicht vom Entwickler / von der Entwicklerin selbst durchgeführt werden, sondern muss von einer dritten Person absolviert werden.

Neben formalen Prüfungen (entsprechend den Makrorichtlinien) und Vollständigkeitsprüfungen (Dokumentationen, Dateien, Konfigurationen) sowie den Installationstests entsprechend der Dokumentation werden auch Funktionstests durchgeführt. Dabei gelten als abnahmerelevant die im Pflichtenheft (siehe auch Kapitel 2.3) definierten Testfälle und beschriebenen Funktionen, aber auch die in der Benutzeranleitung beschriebenen Vorgehensweisen. Zusätzlich wird auf „Robustheit“ getestet.

QS-Tests erfolgen nach standardisierten Vorgehensweisen (Checkliste) und werden dokumentiert.

2.7.3.2 OS während der Programmierung

In diesem Fall hier geht es nicht um den QS-Check vor der Auslieferung des Programms, der natürlich nicht vom Programmierer / von der Programmiererin selbst durchgeführt wird, sondern um die laufende QS Prüfung des Programmierers / der Programmiererin selbst. Diese sollte häufig und intensiv durchgeführt werden – verbessert sie doch den Code und lässt Fehler früh erkennen. Allerdings kann die laufende Qualitätskontrolle den Schlusstest nicht ersetzen, denn das Funktionieren vieler Einzelteile bedingt nicht automatisch des fehlerfreie Funktionieren des Ganzen.

Grundlage der internen laufenden QS ist die korrekte Anwendung der Programmierrichtlinien von der ersten Code-Zeile an. Wird als erstes in einem Modul „Option explicit“ gesetzt, müssen alle Variablen bereits korrekt definiert werden – eine spätere Suche nach nicht deklarierten entfällt.

Weitere Praxistipps zur laufenden QS:

- Die Struktur des Programmes wird so angelegt, dass kleinere Teileinheiten (idealerweise einzelne Prozeduren) mit geringem Aufwand unabhängig vom Rest getestet werden können. Dafür lassen sich dann einfache Testroutinen schreiben, die zunächst die benötigten Parameter-Umgebungen schaffen und dann die Funktion aufrufen. Diese Debug-Stukturen können auch im Code verbleiben – auskommentiert und mit Info-Texten versehen.

Beispiele:

```
'/** Debug_OptionaleBestandteile
'*************************************************************************.
'* @kurztext Einzeltest Optionale Bestandteile
'* Diese Funktion prüft den Code optionalen Bestandteile
'* Das aktuelle Dokument muss Teststrukturen enthalten. Die Start-Funktion
'* "StartHauptprogramm" muss in Zeile 45 eine "exit sub" Anweisung erhalten
'*
'*************************************************************************
'*/
sub Debug_OptionaleBestandteile
  'StartHauptprogramm
  'oDoc = thiscomponent
  'OptionaleBestandteileEinfuegen
end sub


'/** OptionaleBestandteileEinfuegen ….
function OptionaleBestandteileEinfuegen
```

Hier gibt es eine eigene Testfunktion am Start des Moduls – es wird aber nur dieser Code-Teil (sonst Bestandteil eines Gesamtablaufes) getestet.

```
function PlatzhalterErsetzen
  dim oSuchErg as object
  dim oSuche as object
  dim i%, n%
'##### nur Entwicklung ###
'oDoc = thisComponent
'#########

  oSuche = oDoc.createSearchDescriptor()
  REM suche über alle Elemente der gelieferten Platzhalternamen
  for i = 0 to uBound(aPlatzhalter())
```

In diesem Fall wird im Code ein Debug-Code integriert, der einen Bezug zum aktuellen Dokument herstellt und somit den Test dieser Funktion mit einem geöffneten und aktivierten geeigneten Testdokument ermöglicht. Im Gesamtumfeld wird das Dokument per Code erzeugt und diese Funktion hier ist eine nachgeschaltete Aufgabe. Um sie zu testen, ist es aber gerade während der Entwicklung unmöglich, immer zuerst den kompletten Code ablaufen zu lassen – zu viele andere Fehlerquellen und Einflussmöglichkeiten. Auch hier kann der Debug-Code (auskommentiert) im Code verbleiben – das erleichtert späteren Entwicklern/Entwicklerinnen ebenfalls, die Funktion separat zu testen.

- Einzelne Code-Sequenzen können separat getestet werden – in einem eigenen Modul, nur für die Entwicklung gedacht. Liefern sie korrekte und erwartete Ergebnisse, werden sie mit Copy&Paste in das Hauptprogramm übernommen. Dafür ist es notwendig, die gleiche Syntax und die gleichen Variablenbezeichnungen beizubehalten, die auch im Hauptcode verwendet werden – sonst erzeugt man zusätzliche Fehlerquellen.
- Dialoge erhalten zu Entwicklungszwecken eigene „Testbuttons“, die dialogspezifische Dinge zur Laufzeit des Dialoges testen. Hierzu zählen insbesondere Ergebnisse der

Eingaben (und wie sie intern verarbeitet werden) sowie der gezielte Aufruf einzelner Funktionen, die Dialogfelder ausfüllen oder auslesen. Nur so lassen sich diese einzeln testen.

Je detaillierter und strukturierter die QS bereits während der Entwicklung durchgeführt wird, umso stabiler ist später das Gesamtprogramm.

Natürlich gehört auch ein „Gesamttest“ vor der Übergabe zu den abschließenden Aufgaben des Programmierers / der Programmiererin. Auch er/sie kann sich an den Anforderungen des Pflichtenheftes orientieren.

3 Filehandling

Dieses Kapitel behandelt viele Aufgabenstellungen rund um das Thema „Dateihandling“. Dabei werden sowohl die OpenDokument -Dateien (also Writer-, Calc- oder Base-Dateien) als auch einfache Textdateien (Konfigurationsdateien, Datendateien) besprochen.

Vorüberlegung: Alle Pfad- und Dateiangaben werden intern in einer URL-Schreibweise verwaltet und genutzt. Diese sind nicht geeignet, um sie dem/der Nutzer/in anzuzeigen bzw. von ihm/ihr zu erfragen. Die URL-Schreibweise codiert beispielsweise Leerzeichen und Sonderzeichen und nutzt als Pfadtrenner immer den Slash („/“).

Werden also Pfadangaben und Dateinamen abgefragt (sei es direkt durch Eingabe in ein Textfeld oder über die OOo-eigenen Dialoge), so müssen diese immer erst umgewandelt werden – sowohl beim Input als auch beim Output. Das gleiche gilt bei der Übergabe per Parameter – zum Beispiel durch Konfigurationsdateien. Auch dort werden die Pfad- und Dateinamen in Klartext stehen.

Dabei existieren diverse „Fallen“: Die Methode converttoURL() bzw. convertFromURL() helfen zwar, einen Pfad umzuwandeln, können aber auch nur so gut sein, wie der Input, der vorgefunden wird. So ergänzt die Methode convertToURL() immer den Protokolltyp „File:///“; der folgende Aufruf würde also zu einem Fehler führen:

sURL = convertToURL(„meinPfad/mitVerzeichnis“) & convertToURL(„meine Datei.odt“)

das Ergebnis: file:///meinPfad/mitVerzeichnisfile:///meine%20Datei.odt

Das gleiche Ergebnis würde auch entstehen, wenn der erste Teil (also der Pfad) aus einer URL ausgelesen und dann mit einem neuen Dateinamen kombiniert würde. Hier sollten unbedingt ausreichende Tests durchgeführt und die unterschiedlichen Möglichkeiten ausprobiert werden (Tests mit Dokument- und Verzeichnisnamen durchführen, die Umlaute und Leerzeichen beinhalten!).

Im Übrigen fehlt zwischen Pfad und Dateiname noch der Pfadtrenner – auch hier muss eine interne Prüfung erfolgen und gegebenenfalls der Trenner ergänzt werden. Dies ist auch wichtig, wenn Pfad und Dateiname über Konfigurationsdateien (z.B. SimpleConfig) übergeben werden –

es ist immer damit zu rechnen, dass Pfadnamen mit oder ohne abschließenden Pfadtrenner angegeben werden. Die folgende Prüffunktion kann dies erledigen und gegebenenfalls den Trenner ergänzen – das Ergebnis ist dann immer ein Pfad mit abschließendem Trenner:

```
'/** CheckPfadKomplett()
'*************************************************************************.
'* @kurztext ergänzt gegebenenfalls eine Pfadangabe um den Trenner
'* Die Funktion ergänzt gegebenenfalls eine übergebene Pfadangabe um den Trenner
'* Zurückgeliefert wird der Pfad mit abschließenden Trenner. Der Pfad ist noch keine
'* URL Schreibweise, sondern systemspezifisch.
'*
'* @param1  sPfad as string   der bisherige Pfad
'*
'* @return  sPfad as string   der Pfad mit abschließenden Pfadtrenner
'*
'*************************************************************************
'*/
function CheckPfadKomplett(sPfad as string)
  if NOT (right(convertToURL(sPfad), 1) = "/") then  'ergänzen
    REM Prüfen, ob protokoll "file" oder "http" - dann slash
    if (lcase(left(trim(spfad), 5)) = "file:") OR (lcase(left(trim(spfad), 5)) = "http:") then
      sPfad = sPfad & "/"
    else  'sonst systemspezifisch
      sPfad = sPfad & GetPathSeparator()
    end if
  end if
  CheckPfadKomplett = sPfad
end function
```

Bei diesem Code zeigt sich gleich noch eine zweite „Falle“: Neben dem „File-Protokoll“ können Dateien auch direkt über einen „http://“-Pfad angesprochen werden; in diesem Fall wird das Internet-Protokoll verwendet. Die Methoden „ConvertToURL()“ und „ConvertFromURL()“ können aber mit dem Http-Protokoll nichts anfangen und schleifen den übergebenen Parameter 1:1 durch – es erfolgt demnach keine Codierung! Wird also ein Dateipfad wie folgt aufgebaut:

```
sURL = ConvertToURL("http://123.123.12.24/" & "Meine Datei.ods")
```

dann führt dies zu einem Fehler, da der Pfad nach wie vor lautet: "http://123.123.12.24/Meine Datei.ods", das heißt, das Leerzeichen wurde nicht codiert. Zwar wurde das Verhalten in den neueren LibreOffice-Versionen korrigiert, der Fall sollte aber immer geprüft werden.

Wichtig:

Werden Pfade oder Dateinamen über die internen Services "com.sun.star.ui.dialogs....“ abgefragt (Dialoge), so werden die Ergebnisse immer in URL-Schreibweise zurückgeliefert!

3.1 Erzeugen von OOo-Dateien

Das Erzeugen von OOo-Dateien ist unkritisch und bedarf wenig Beschreibungen. Ein Dokument wird mit der Funktion loadComponentFromURL() erzeugt, wobei es für jeden Dokumententyp

spezifische Namen gibt. Die Funktion erwartet vier Parameter, wobei lediglich der letzte (eine Liste von Property-Values) interessant ist, jedoch selten bei der Ersterzeugung eingesetzt wird (Ausnahme siehe Kapitel 3.2.5 – unsichtbares Dokument). Alle theoretisch zu übergebenden Parameter können auch nachträglich gesetzt werden – das ist der Vorteil eines neuen Dokumentes.

Beispiel der Erzeugung eines Writer-Dokumentes:

```
oDoc = StarDesktop.loadComponentFromURL("private:factory/swriter", "_blank", 0, array())
```

Man kann – muss aber nicht – das erzeugte Dokument gleich einer Variablen zuordnen. In der Regel macht dies natürlich Sinn, nur so kann direkt mit dem Dokument weitergearbeitet werden.

Noch ein Hinweis: Typischerweise erhält das neu erzeugte Dokument sofort den Fokus und liegt somit „oben auf“. Auch die vordefinierte Variable „ThisComponent“ verweist jetzt auf das neue Dokument.

Im Einzelfall kann es Sinn machen, eine „Wait“-Funktion nach der Erzeugung anzugeben, um dem Dokument die komplette Erstellung zu ermöglichen und erst dann Manipulationen im Dokument durchzuführen. Zwar bleibt das Makro an der Erzeugungszeile so lange stehen, bis OOo meldet, dass das Dokument erzeugt ist, die Praxis zeigte jedoch, dass manche interne Objekte noch nicht unmittelbar zur Verfügung stehen. In der Regel reicht eine Wartezeit von 100 ms (wait(100)).

Alle neuen Dokumente werden auf der Basis der hinterlegten Standard-Vorlagen erzeugt.

3.2 Öffnen von bestehenden Dokumenten

Das Öffnen von Dokumenten benutzt die selbe Methode wie das Erzeugen – dennoch sind hierbei viele Punkte zunächst zu beachten:

- Ein Dokument kann nur geöffnet werden, wenn es überhaupt existiert – es muss also unbedingt vorher eine Prüfung sowohl des Pfades als auch des Dateinamen erfolgen.
- Ein Dokument könnte bereits auf dem aktuellen Bildschirm geöffnet sein – dann führt die Methode ebenfalls zu einem Fehler. Abhilfe: siehe Abschnitt „Prüfung, ob Datei schon geöffnet ist“.
- Auch wenn ein Dokument existiert, ist es möglicherweise nicht direkt zu öffnen. Es könnte sein, dass dieses Dokument derzeit von einer anderen Person bearbeitet wird. Dies ist insbesondere bei Netzwerk-Dokumenten zu beachten. Leider gibt es keine direkte Prüfung, ob dies der Fall ist. Allerdings legt OOo eine versteckte Lock-Datei an, die den/die Benutzer/in und die Uhrzeit der Benutzung enthält. Eine Möglichkeit wäre die Prüfung auf das Vorhandensein der Lock-Datei (siehe Abschnitt Lock-Datei prüfen). Dieser Weg ist jedoch nicht „wasserdicht“, da nicht immer gewährleistet ist, dass eine Lock-Datei auch wieder gelöscht wird.

- Enthält das Dokument Makros, die benötigt werden, so muss das Öffnen den Makro-Excecution-Mode explizit setzen – standardmäßig werden Makros nie ausgeführt!
- Das zu öffnende Dokument sollte immer einer Variablen zugeordnet werden, da nicht sicher ist, ob der/die Benutzer/in nicht kurzfristig ein anderes geöffnetes Dokument aktiviert – und somit „ThisComponent“ fehlläuft!

Die meisten Prüfungen lassen sich „auslagern“ und nacheinander abarbeiten:

```
sUrl = "/users/MeinName/test/testdatei.odt"    'URL der zu öffnenden Datei

if NOT checkExistDatei(sURL) then exit sub    'Prüfen, ob die Datei überhaupt existiert

REM Prüfen, ob die Datei bereits auf dem Rechner geöffnet ist - dann übernehmen - sonst öffnen

oDoc = CheckDateiOffen(sURL)

if isEmpty(oDoc) then

 if checkDateiUsed3Partie(sURL) then exit sub   'Prüfen, ob die Datei bereits geöffnet ist (3. Person)

  oDoc = StarDesktop.loadComponentFromURL(sURL, "_blank", 0, args())

end if
```

Die Prüfungen sehen dabei wie folgt aus:

3.2.1 Prüfung, ob Datei existiert

```
'/** CheckExistDatei()
'*************************************************************************.
'* @kurztext prüft, ob eine Datei existiert
'* Die Funktion prüft, ob eine Datei existiert (URL)
'*
'* @param1  sURL as string   der Pfad und Dateiname
'*
'* @return  Flag as Boolean   True, wenn Datei existiert, sonst false
'*
'*************************************************************************
'*/
function CheckPfadKomplett(sURL as string)
  dim oSFA as variant

  CheckPfadKomplett = false   'Vorgabe
  oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  if oSFA.exists(sURL) then  CheckPfadKomplett = true
end function
```

3.2.2 Prüfung, ob Datei schon geöffnet ist

Die Prüfung, ob die Datei bereits auf dem aktuellen Rechner vom Benutzer / von der Benutzerin geöffnet ist, erfolgt durch Vergleich der URL aller bereits geöffneten Komponenten. Ist die Datei

bereits geöffnet, so kann sie nicht erneut geöffnet werden (höchstens als Kopie bzw. schreibgeschützt – aber das ist in der Regel nicht gewünscht), die Funktion liefert dann das geöffnete Dokument als Referenzobjekt zurück.

```
'/** CheckDateiOffen
'*************************************************************************.
'* @kurztext prüft, ob eine bestimmte Datei bereits offen ist
'* Diese Funktion prüft, ob eine per sDatURL übergebene Datei (Pfad und
'* Dateinamen) bereits geöffnet ist.
'*
'* @param1  sDatURL as string   Pfad und Dateiname (in URL-Schreibweise!)
'*
'* @return  oObjekt as object   Das Objekt des geöffneten Dokumentes (falls vorhanden),
'*                              sonst ein leeres Objekt
'*
'*************************************************************************
'*/
Function CheckDateiOffen(sDatURL As String)
 Dim oComp as variant  'Alle Komponenten
 Dim oElement as variant 'ein einzelnes Element

 oComp = StarDesktop.getComponents.CreateEnumeration
 Do While oComp.HasMoreElements
  oElement = oComp.NextElement()
  If HasUnoInterfaces(oElement, "com.sun.star.frame.XModel") Then
   If oElement.hasLocation() Then
    If oElement.getLocation = sDatURL Then
     'die Datei ist geöffnet
     CheckDateiOffen = oElement
     Exit Function
    End If
   End If
  End If
 Loop
end function
```

Die Funktion liefert das Objekt des Dokumentes zurück – falls es bereits geöffnet ist, ansonsten eben ein leeres Objekt.

3.2.3 Prüfung, ob Datei schon geöffnet ist (3. Person)

Diese Prüfung ist etwas schwieriger durchzuführen. OOo legt einen Lockfile im selben Verzeichnis an, in dem sich auch die Datei befindet. Dieser Lockfile ist wie folgt aufgebaut:

Name der Datei: .~lock.<Name der Datei inkl. Dateiendung>#

Inhalt: Benutzername (aus OOo), Rechner / Benutzer/in, Rechnername der Benutzung, Datum Uhrzeit, Pfad zum von der Datei verwendeten OOo-Profil;

Ein Beispieleintrag könnte wie folgt aussehen (eine Zeile):

Thomas Krumbein,P3000/Thomas,P3000,30.01.2012 13:10,file:///C:/Users/Thomas/AppData/Roaming/OpenOffice.org/3;

Um absolut sicher zu stellen, dass die Datei nicht bereits in Benutzung ist, müsste man nun sowohl das Vorhandensein der Lock-Datei als auch die Relevanz des Inhaltes prüfen. Die Lock-Datei wird normalerweise gelöscht, wenn die Datei wieder geschlossen wird. Dies funktioniert jedoch nicht zuverlässig in folgenden Fällen:

- Absturz des Systems – also bei nicht normalem Beenden von OOo

- in manchen Netzwerkkonfigurationen mit Dateien, die auf Netzwerkordnern liegen

In diesen Fällen können die Lock-Dateien übrig bleiben – und müssten dann detaillierter ausgewertet werden (Datum, Uhrzeit etc.).

```
'/** checkDateiUsed3Partie()
'*************************************************************************.
'* @kurztext prüft, ob eine Datei bereits genutzt wird
'* Die Funktion prüft, ob eine Datei bereits genutzt wird (URL)
'*
'* @param1  sURL as string   der Pfad und Dateiname
'*
'* @return  Flag as Boolean   True, wenn Datei benutzt wird, sonst false
'*
'*************************************************************************
'*/
function checkDateiUsed3Partie(sURL as string)
  dim a()                 'Hilfsarray
  dim sDatName as string  'Dateiname
  dim oSFA as variant     'Simple File Access Service
  dim n%

   checkDateiUsed3Partie = false    'Vorgabe
  oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  a = split(sURL, "/")
  n = ubound(a)
  sDatName = a(n)
  redim preserve a(n-1)
  sLockURL = join(a, "/") & "/.~lock." & sDatName & "#"
  if oSFA.exists(sLockURL) then  checkDateiUsed3Partie = true
end function
```

Diese Funktion prüft nur das Vorhandensein der Lock-Datei, nicht ob sie noch gültig ist oder ob sie auf den aktuellen Rechner / Benutzer/in zutrifft. Diese Prüfungen könnte man zusätzlich noch mit einbauen, der Aufwand sollte jedoch im Einzelfall geprüft werden.

3.2.4 Öffnen mit Parametern

Beim Öffnen von OOo-Dateien können diverse Parameter mit angegeben werden. Diese werden als PropertyValue-Paar dem letzten Parameter als Liste übergeben.

Der Aufbau ist dann beispielsweise wie folgt:

```
dim args1(0) as new com.sun.star.beans.PropertyValue
args1(0).name = "MakroExecutionMode"
args1(0).value = 3
```

```
oDoc = StarDesktop.loadComponentFromURL(sURL, "_blank", 0, args1())
```

Nutzen Sie in einem Makro häufiger dieses Konstrukt, achten Sie auf die Bezeichner der PropertyValues. Es gab zeitweise Fehler durch „nicht definierte Variablen“ bzw. durch „Variable schon anderweitig definiert“ bei Verwendung gleicher Bezeichner (typischerweise args()) in unterschiedlichen lokalen Umgebungen (Funktionen). Umgehen Sie das, indem Sie unterschiedliche Bezeichner wählen – zum Beispiel durchnummerieren).

Hier die wichtigsten Parameter in der Übersicht:

MakroExecutionMode

Dateien werden oft benötigt in der Form, dass die dort enthaltenen Makros ausgeführt werden können. Dazu zählen auch schon Schaltflächen, die auf Makros (jetzt auch Extensions) verweisen. Diese Funktionalität ist typischerweise nicht aktiviert.

Wird der Mode nicht explizit definiert, so gilt er als „nicht übergeben“ bzw. „0“ und somit werden keine Makros ausgeführt. Dies ist lediglich immer dann sinnvoll, wenn das zu öffnende Dokument lediglich gelesen oder gedruckt werden soll.

In allen anderen Fällen sollte der MakroExecutionMode definiert und mit übergeben werden, auch wenn nicht gewünscht wird, dass Makros im Dokument ausgeführt werden sollen.

Wird ohne Mode gearbeitet (oder Mode „0“), dann werden leider auch alle Extensions und Extension-Symbolleisten blockiert, es können dann also auch keine Markos generell in diesem Frame ausgeführt werden. Dies ist in der Regel ein unerwünschtes Verhalten.

Der Mode muss also angegeben werden: Statt der Konstanten "ALWAYS_EXECUTE_NO_WARN" (Mode 4) wäre es sinnvoller, die Dokumente mit "USE_CONFIG" (Mode 3) zu öffnen, dann wird das Verhalten übernommen, wie es in der Konfiguration eingestellt ist, oder (eventuell noch besser) mit der Konstanten "USE_CONFIG_APPROVE_CONFIRMATION" (Mode 6) , jetzt wird unabhängig von der Konfiguration immer gefragt, ob Makros ausgeführt werden sollen (wenn welche im Dokument vorhanden sind). Klickt der/die Benutzer/in jetzt auf nein, werden keine Makros ausgeführt. Klickt er/sie auf ja, wird das Verhalten ausgeführt, das in der Konfiguration festgelegt ist (wenn also dort steht: keine Makros ausführen, dann wird trotz positiver Bestätigung dennoch kein Makro ausgeführt).

Die Empfehlung lautet somit: Wenn möglich sollte beim Öffnen von Dokumenten mit dem MakroExecutionMode "USE_CONFIG" (Mode 3) oder "USE_CONFIG_APPROVE_CONFIRMATION" (Mode 6) gearbeitet werden, im Einzelfall sind aber auch andere Varianten möglich.

Wird das Dokument mit "ALWAYS_EXECUTE_NO_WARN" (Mode 4) geöffnet, weil es im Arbeitsablauf so notwendig ist, dann muss sichergestellt sein, dass das zu öffnende Dokument

„sicher“ ist. Es darf also beispielsweise nicht über einen Dateibrowser ausgewählt werden können!

AsTemplate

Mit dieser Eigenschaft kann jedes Dokument als Vorlage geöffnet werden, das bedeutet, das Dokument besitzt nun keine URL mehr (und keine Eigenschaft Location), alle Daten, Formate etc. sind aber vorhanden. Sinnvoll ist dies in der Regel nur dann, wenn der/die Benutzer/in das Dokument selbst speichert und dabei das Ursprungsdokument nicht „zerstören“ soll (durch versehentliches Überschreiben beim Drücken des „Speichern“-Button). Ansonsten lässt sich natürlich immer auch direkt nach dem Öffnen des Dokumentes auch ein „Speichern unter“ per Makro ausführen – auch dadurch wird das versehentliche Speichern vermieden.

Ein anderer typischer Anwendungsfall: Das Dokument soll nur gedruckt werden, der/die Nutzer/in muss aber noch ein paar Daten eingeben (die aber nicht gespeichert werden sollen). Auch hier würde man das Dokument „als Vorlage“ öffnen – jetzt können alle Eingaben erfolgen und gedruckt werden – das Ausgangsdokument bleibt unversehrt.

Filter

Wird ein Dokument im Fremdformat geöffnet, sind unbedingt entsprechende Filter anzugeben. Die Filter sind in exakter Schreibweise zu übergeben – die Liste der möglichen Filter muss eventuell zur Version passend extrahiert werden.

Der folgende Code kann die Filter in ein neues Writer-Dokument schreiben:

```
Sub FilterNamenExtrahieren
   oFF = createUnoService( "com.sun.star.document.FilterFactory" )
   oFilterNames = oFF.getElementNames()

   ' Create a Writer doc and save the filter names to it.
   oDoc = StarDesktop.loadComponentFromURL( "private:factory/swriter", "_blank", 0, Array() )
   oText = oDoc.getText()
   oCursor = oText.createTextCursor()
   oCursor.gotoEnd( False )

   ' Print the filter names into a Writer document.
   For i = LBound( oFilterNames ) To UBound( oFilterNames )
      oText.insertString( oCursor, oFilterNames(i), False )
      oText.insertControlCharacter( oCursor,
com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, False )
   Next
End Sub
```

3.2.5 Dokument nicht sichtbar öffnen

Ein weiterer häufiger Anwendungsfall ist das Öffnen eines Dokumentes zur internen Weiterverarbeitung, ohne dass es auf dem Bildschirm und damit für den User sichtbar ist. Diese Methode wird immer dann gewählt, wenn eine Benutzereingabe nicht oder noch nicht im

Dokument benötigt wird, per (Makro-)Code aber dennoch schon auf das Dokument zugegriffen werden muss.

Vorteil der Methode: Sie ist fehlerresistent gegenüber dem/der Benutzer/in (er/sie kann nicht versehentlich Daten ändern) und „unsichtbar“.

Nachteil: Das Dokument wird im Arbeitsspeicher geladen – wird das Makro abgebrochen (durch Benutzereinwirkung oder Fehler) so verbleibt das Dokument dort. Es kann manuell nicht geschlossen werden – außer „brutal“ über den Taskmanager und durch Beenden des soffice-Prozesses. Dies ist nicht immer wirklich wünschenswert.

Das Laden des Dokumentes erfolgt wie gewohnt – mit entsprechenden Parametern:

```
dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).name = "Hidden"
args2(0).value = True
oDoc = StarDesktop.loadComponentFromURL(sURL, "_blank", 0, args2())
```

Ein so geöffnetes Dokument kann nun auch zu jedem Zeitpunkt per Code sichtbar geschaltet werden – also zum Beispiel nachdem es per Code vorbereitet wurde und nun eine Benutzereingabe oder eine Freigabe erwartet.

```
oDoc.getCurrentController().getFrame().getContainerWindow().setVisible(true)
```

Zum Hintergrund: Sichtbar oder unsichtbar ist immer das Fenster, der Rahmen. Das Dokument selbst ist Teil des Frames (Container Window) und eben nur dann sichtbar, wenn dieser sichtbar ist.

In gleicher Weise lässt sich ein aktuelles Dokument auch „verstecken“, also dem/der Benutzer/in entziehen, ohne es physisch zu schließen.

Achten Sie immer darauf, dass versteckte Dokumente rechtzeitig entweder wieder sichtbar geschaltet oder geschlossen werden, bevor das Makro beendet wird. Das muss insbesondere bei der Verwendung von Dialogen berücksichtigt werden, deren Ende auch durch das Schließen-Kreuz vom Benutzer / von der Benutzerin erzwungen werden können – das Makro darf nicht damit beendet werden – sondern muss unbedingt nun noch die „Restarbeiten“ wie das Schließen der Hidden-Dokumente erledigen. Siehe dazu auch Kapitel 6 – Dialoge.

3.2.6 WollMux Dokumente

Handelt es sich bei dem zu öffnenden Dokument um ein WollMux-Dokument, so müssen Sie damit rechnen, dass dieses Dokument direkt nach dem Öffnen zunächst vom WollMux „belegt“ und manipuliert wird. In dieser Zeit können Sie keine sinnvollen Makro-Zugriffe darauf starten – teils, da manche Objekte noch nicht existieren, teils, da das Dokument gerade per API bearbeitet wird (und die API keine Multitask-Fähigkeit besitzt).

Zwar bietet der WollMux diverse Möglichkeiten Ereignisse zu senden und so zu kommunizieren, wann er fertig ist – in Java gibt es sogar eine eigene Methode, die dieses Problem auf elegante Weise löst (loadComponentFromURLandWaitForWollMux() – siehe auch Wollmux-Dokumentation), in Basic aber erwiesen sich die Versuche über einen entsprechenden Listener als sehr instabil und nicht brauchbar.

Möglicherweise ändert sich das in Zukunft. Aktuell verbleibt hier nur, die Zeit entsprechend zu schätzen und eine Wait()-Anweisung einzubauen. Eine Wait(2000) war meist ausreichend – müsste aber im Einzelfall getestet werden.

3.3 OOo Dokumente speichern und schließen

Ein geöffnetes Dokument wird verarbeitet, dann gespeichert und wieder geschlossen. Verwendet werden die Methoden:

```
oDoc.store()    'speichern des Dokumentes (Location muss vorhanden sein!)

oDoc.close(true)    'Schließen des Dokumentes (Parameter sollte immer true sein!)
```

In der Praxis erweisen sich die beiden Zeilen als jedoch gar nicht so trivial. Das Speichern des Dokumentes geht nur, wenn eine gültige URL im Dokument hinterlegt ist – wenn nicht, dann gibt es einen Basic-Laufzeitfehler. In der Regel reicht es, die Eigenschaft hasLocation zu überprüfen – es könnte jedoch theoretisch sein, dass der hinterlegte Pfad gar nicht mehr verfügbar ist (z.B. bei einem Netzlaufwerk). Dann müsste zusätzlich auch noch die hinterlegte URL auf Gültigkeit geprüft werden – nur dann ist das Speichern „sicher“:

```
if oDoc.hasLocation AND oSFA.exists(oDoc.url) then  'alles OK - speichern
   oDoc.store()
else
  'Prozedur zum Speichern ohne URL - also Pfad und Name erfragen oder per Script erzeugen
end if
```

Ein Dokument, das noch keine Location besitzt (eigene Speicheradresse) kann dann nur durch die Prozeduren StoreAsURL() (entspricht Datei / Speichern unter) bzw. StoreToURL() entspricht Datei / Exportieren) gespeichert werden. In beiden Fällen ist mindestens die URL (Pfad und Dateiname in qualifizierter Schreibweise) zu übergeben.

Es empfiehlt sich immer, vor dem Speichern zu überprüfen, ob der Pfad / Dateiname gültig ist, existiert und beschreibbar ist. Insbesondere bei Pfad- und Dateinamen, die vom Benutzer / von der Benutzerin eingegeben werden (über Dialoge) ist dies dringend erforderlich!

Existiert die Datei bereits an der angegebenen Stelle, so würde ein Speichern dort die existierende Datei überschreiben – auch hier muss der/die Programmierer/in Vorsorge treffen, dass dies zulässig ist. Ein Überschreiben einer noch geöffneten Datei ist nicht möglich! Das Ergebnis wäre ein Basic-Laufzeitfehler. Prüfen Sie auch das, falls die Möglichkeit besteht.

Jeder erfolgreiche Speicherprozess setzt immer das Modified-Flag des Dokumentes zurück!

3.3.1 Dokument schließen

Mit der Methode close() wird ein Dokument geschlossen. Ist das Modified-Flag gesetzt, so erfolgt von OOo aus die Benutzer-Abfrage, ob die geänderten Daten gespeichert werden sollen. In der Regel ist diese Abfrage im Makro-Verlauf weder sinnvoll noch erwünscht. Ein eventueller Speicherzyklus muss vom Entwickler / von der Entwicklerin vorher abgefangen und bearbeitet werden.

Sorgen Sie also dafür, dass vor der close-Anweisung das Modified-Flag auf „False“ steht. Dies können Sie zum Beispiel erreichen durch die Store()-Prozedur in der Zeile vorher oder durch manuelles Überschreiben des Modified-Flags. Beispiel:

```
if oDoc.isModified then odoc.setModified = false

odoc.close(true)
```

Das Zurücksetzen des Modified-Flags verwirft allerdings auch alle Änderungen seit dem letzten Speichern – ein manchmal durchaus erwünschtes Verhalten. Ansonsten sollte vor dem Zurücksetzen unbedingt ein Speicherprozess stattfinden.

Die Methode close() erwartet einen Boolschen Wert als Parameter. In Basic-Programmierungen sollte dieser immer „True“ sein – das bedeutet, dass der Schließprozess an eventuell andere Prozesse abgegeben wird, die ein Schließen noch verhindern. Diese Prozesse übernehmen dann die Verantwortung für den erfolgreichen Abschluss des Schließens.

Mit der Übergabe „False“ verbleibt die Verantwortung im aktuellen Prozess – der/die Entwickler/in muss jetzt selbst dafür sorgen, dass es keine CloseVetoExeptions mehr gibt – das heißt er/sie müsste sich alle Listener ansehen und quasi abfragen, ob es noch Einsprüche gegen das Schließen gibt – und das Dokument erst dann schließen, wenn alle eventuellen Einsprüche aufgehoben wären. Das ist in Basic unüblich. Dieser Weg wird nicht empfohlen.

Ein „hartes“ Schließen ist auch durch das Zerstören des Dokumentenobjektes möglich:

oDoc.dispose()

Da dadurch aber auch alle anderen Prozesse „zwangsbeendet“ werden, kann es unter Umständen zu Verlusten kommen. Die Methode sollte nicht verwendet werden. Lediglich, falls Kompatibilität zu sehr frühen OOo-Versionen (1.x) gewahrt werden muss, kann es sein, die Methode einsetzen zu müssen. Es sollte dann die folgende If-Bedingung verwendet werden:

```
if HasUnoInterfaces(oDoc, "com.sun.star.util.XCloseable") then
  oDoc.close(true)
else
  oDoc.dispose()
end if
```

Wichtig bei Dokumentenmakros:

Die Close()-Methode muss eigentlich die letzte Anweisung im Basic-Block sein. Mit dem Schließen des Dokumentes werden auch die Makros geschlossen und die Bibliotheken/Module etc. aus dem Speicher entfernt. Nach dem Schließen-Befehl können also keine weiteren Makroschritte bearbeitet werden bzw. dies kann dann zu einem Fehler führen. Sorgen Sie dafür, vorher alle nötigen Arbeiten abzuschließen.

3.4 Öffnen, Erzeugen und Speichern von Textdateien

Neben den „normalen“ OpenDocument-Dateien spielen „einfache“ Textdateien immer dann eine Rolle, wenn Konfigurationsdaten zwischengespeichert werden müssen oder Daten als CSV-Dateien eingelesen oder ausgegeben werden sollen.

Grundsätzlich bietet sowohl die UNO-API als auch die Script-Sprache Basic Methoden zum Erzeugen, Lesen und Schreiben von (Text-)Dateien. Für welche Sie sich letztendlich entscheiden, hängt von den Umständen ab und von persönlichen Preferenzen.

Ich empfehle, als erste Wahl immer die UNO-API-Klasse (com.sun.star.ucb.SimpleFileAccess) zu verwenden und nur dann auf Basic auszuweichen, wenn die Verwendung der UNO-Klasse zu erheblichem „Mehraufwand“ führen würde.

Ein solcher Fall ist „das Anhängen“ von Daten an eine bestehende Textdatei, also den klassischen Logfile. Nur die Basic-Variante erlaubt das „Append“, also das Anhängen von Informationen an vorhandene Textdateien.

3.4.1 Logfile Schreiben 1

Beispiel Logfile schreiben mit neuen Informationen am Ende, Verwendung der Basic-Methoden zum Öffnen und Schreiben einer Textdatei:

```
…
  sTxt = "Diese Information wurde erzeugt am: " & format(now(), "dd.mm.yyyy hh:mm") & " Uhr"
  LogFileSchreiben1(sTxt)
…
sub LogFileSchreiben1(sTxt as string)
  dim sLogFileURL as string, fn as integer

  sLogFileURL = convertToURL("D:\Logs\Logtest.txt")
  fn = freeFile      'freien Datenkanal erhalten
  Open sLogFileURL FOR APPEND As #fn
  Print #fn, sTxt
  close #fn

end sub
```

Die Methode ist klein und schnell – die Datei wird erzeugt, wenn sie noch nicht vorhanden ist. Leider bietet die Methode keine Möglichkeit, den Zeichensatz einzustellen – sie ist also weniger geeignet, Informationen aus Calc- oder Writer-Dateien in einfache Textdateien zu schreiben. Für Debug-Logfiles ist sie aber hilfreich.

3.4.2 Logfile Schreiben 2

Die Alternative ist die Verwendung der UNO-Klasse. Genutzt wird hierzu der Service com.sun.star.ucb.SimpleFileAccess sowie die Services des Moduls com.sun.star.io.

Leider gibt es hierbei kein „Append“; eine (bestehende) Datei wird immer von vorne her überschrieben, das heißt, nach dem Öffnen steht der Zeiger vor der ersten Zeile vor dem ersten Zeichen. Ab hier wird der Text nun „überschrieben“ – entsprechend der übergebenen Zeichenanzahl.

Dieses Vorgehen birgt diverse „Gefahren“:

Möchte man die Datei tatsächlich überschreiben – also die bisherigen Inhalte entfernen und durch neue ersetzen (wie dies typischerweise beim Neuschreiben von Konfigurationsdateien der Fall ist) – könnte es passieren, dass der neue Text kürzer ist als der bisherige Text, dann würden Reste der alten Textdatei übrig bleiben – was wiederum beim späteren Verarbeiten der Datei zu unerwartetem Verhalten und dementsprechenden Ergebnissen führen kann.

Möchte man die bisherigen Inhalte gar nicht überschreiben, sondern nur neue Informationen hinzufügen, so muss man die bisherigen Informationen erst sichern und dann neu schreiben.

Ein „Append“ kann man also manuell erreichen und simulieren: Zuerst den Dateizeiger hinter das letzte Zeichen setzen (EOF), dann erst den Schreibvorgang beginnen. Auch zu beachten ist, dass die Methode „WriteString()“ keinen Zeilenumbruch hinzufügt, der muss also entsprechend manuell hinzugefügt werden – und zwar passend zum Betriebssystem (für Windows chr(13) + chr(10) – für Linux nur chr(10)).

Beispiel zur Erzeugung des Logfiles wie in Kapitel 3.4.1:

```
sub LogFileSchreiben2(sTxt as string)
  dim oSFA as variant
  dim oInnputstream as variant, oOutputstream as variant, oDatei as variant
  dim sLogFileURL as string, sZeilenumbruch as string
  dim aDaten()
  dim n as long

  oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  oInputstream = createUnoService("com.sun.star.io.TextInputStream")
  oOutputstream = createUnoService("com.sun.star.io.TextOutputStream")
  sLogFileURL = convertToURL("D:\Logs\Logtest2.txt")
  sZeilenumbruch = chr(13) & chr(10)

  REM Datenzeiger an das Ende setzen
  oDatei = oSFA.OpenFileReadWrite(sLogFileURL)
  oOutputStream.SetOutputStream(oDatei.getOutputStream)
  oInputStream.SetInputStream(oDatei.getInputStream)
  REM Datenzeiger an das Ende setzen
  do while NOT oInputStream.isEOF
    oInputStream.readLine()
  loop
  REM Daten schreiben
  oOutputStream.writeString(sTxt & sZeilenumbruch )
  oOutputStream.closeOutput()
```

```
end sub
```

Über die Do...While-Schleife wird der Zeilenzeiger an das Ende gesetzt, und neue Informationen können nun dort hinzugefügt werden.

Möchte man hingegen die neuen Informationen immer am Anfang der Datei einfügen, so müssen zunächst alle (vorhandenen) Daten gesichert werden, dann wird die Datei gelöscht und komplett neu geschrieben, so dass also ältere Daten nach unten hin verschoben werden.

Das sähe dann wie folgt aus:

```
sub LogFileSchreiben2(sTxt as string)
  dim oSFA as variant
  dim oInnputstream as variant, oOutputstream as variant, oDatei as variant
  dim sLogFileURL as string, sZeilenumbruch as string
  dim aDaten()
  dim n as long

  oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  oInputstream = createUnoService("com.sun.star.io.TextInputStream")
  oOutputstream = createUnoService("com.sun.star.io.TextOutputStream")
  sLogFileURL = convertToURL("D:\Logs\Logtest2.txt")
  sZeilenumbruch = chr(13) & chr(10)


  REM einlesen der bisherigen Daten, wenn Datei schon vorhanden
  if oSFA.exists(sLogFileURL) then
    oDatei = oSFA.OpenFileReadWrite(sLogFileURL)
    oInputStream.SetInputStream(oDatei.getInputStream)
    REM alle Zeilen einlesen
    n = 0
    do while NOT oInputStream.isEOF
      Redim preserve aDaten(n)
      aDaten(n) = oInputStream.readLine()
      n = n + 1
    loop
    oInputStream.closeInput()
    oSFA.kill(sLogFileURL)
  end if

  REM Schreiben der Datei
  oDatei = oSFA.OpenFileReadWrite(sLogFileURL)
  with oOutputStream
    .SetOutputStream(oDatei.getOutputStream)
    .writeString(sTxt & sZeilenumbruch)
    for n = 0 to uBound(aDaten)
      .writeString(aDaten(n) & sZeilenumbruch)
    next
    .closeOutput()
  end with

end sub
```

Alle vorhandenen Daten werden zeilenweise in einen Array eingelesen und später wieder zurückgeschrieben.

Diese Methode eignet sich allerdings nur bedingt für Logfiles, da deren Größe sehr schnell zunimmt und die Performance der oben dargestellten Routine dann massiv sinkt (siehe dazu auch „Arrays (Listen)“ – Kapitel 4.2.3).

3.4.3 CSV-Dateien lesen und schreiben

Neben Log- und Konfigurationsfiles spielen CSV-Dateien eine große Rolle im praktischen Einsatz. Diese werden typischerweise in Calc-Dateien weiterverarbeitet oder aus diesen erzeugt. Zwar besitzt Calc eingebaute Möglichkeiten, CSV-Dateien einzulesen und zu erzeugen, diese reichen jedoch oft nicht aus beziehungsweise sind für den/die „normale/n“ Benutzer/in zu umständlich zu bedienen.

Im Grunde sind CSV-Dateien auch nur „einfache“ Textdateien, die aber einem speziellen Schema folgen.

Die erste Zeile enthält normalerweise die Spaltenbezeichner (Spaltennamen), die folgenden Zeilen beinhalten dann die Daten. Zwischen den Spalten gibt es einen spezifizierten Trenner (ein Komma, ein Semikolon oder ähnliches), die Gesamtzahl der Spalten pro Zeile ist identisch.

Wichtig sind neben den Informationen über den Spaltentrenner und eventuell einen „Textmarkierer“ insbesondere der verwendete Zeichensatz, der Dezimaltrenner und der Zeilenwechsel. Nur dann lassen sich CSV-Dateien korrekt erzeugen oder lesen!

3.4.3.1 Einlesen von CSV-Dateien und Daten in Calc-Tabelle schreiben

Die folgenden (Rumpf-)Codes zeigen beispielhaft das Einlesen einer CSV-Datei und das Schreiben eines Daten-Arrays in eine Calc-Tabelle.

Die benötigten Parameter werden typischerweise vorher mit Hilfe eines Dialoges erfragt. Im Beispiel werden sie einfach definiert.

```
'/** CSVDateiEinfuegen()
'*************************************************************************.
'* @kurztext fügt die Daten einer CSV-Datei an der aktuellen Zelle beginnend ein
'* Diese Funktion fügt die Daten einer CSV-Datei an der aktuellen
'* Zelle beginnend ein
'*************************************************************************
'*/
sub CSVDateiEinfuegen()
  dim oZelle as variant   'Objekt der aktuellen Zelle
  dim sZelle as string    'Zellname
  dim oTab as object      'aktuelles Tabellenblatt
  dim odoc as variant     'aktuelles Dokument
  REM Parameter für die Beschreibung der CSV-Datei
  dim sFt as string, sDt as string, sTt as string, sTk as string, sZs as string
  dim aOpt(6)    '7 Werte: FT, DT, TT, TK, ZS, ersteZeile, Feldeinstellung
  dim aDaten()            'einzutragenden Daten
```

```
  oDoc = thisComponent
  if NOT Helper.CheckCalcDokument(odoc) then exit sub
  oZelle = Helper.GetFirstSelectedCell(oDoc.getCurrentSelection())

  REM Vorgabedaten definieren (würcen üblicherweise über Dialog erfragt!)
  sCSVURL = convertToURL(„D:\Daten\meineCSVDatei.csv"))

  REM CSV-Parameter Liste -7 Werte: FT, DT, TT, TK, ZS, ersteZeile, Feldeinstellung
  aOpt(0) = ";"     'Feldtrenner
  aOpt(1) = "."     'Dezimaltrenner
  aOpt(2) = ""     'Tausendertrenner - keiner
  aOpt(3) = """"     'Textkennzeichen (Doppelte Hochzeichen - hier maskiert!
  aOpt(4) = "UTF-8"     'Zeichensatz CSV-Datei
  aOpt(5) = true      'Erste Zeile eintragen (true) oder weglassen (false)
  aOpt(6) = true      'Calc erkennt Feldtyp automatisch (true) oder nurText einfügen (false)
  REM Jetzt Daten einlesen
  aDaten = Helper.CSV_Liste_Einlesen(sCSVURL, aOpt())
  if isEmpty(aDaten) then exit sub   'leere Liste - Fehler aufgetreten
  aZeile = aDaten(0)
  REM Daten eintragen
  oTab = oDoc.getCurrentController.getActiveSheet()   'das aktuelle Tabellenblatt
  oBereich = oTab.getCellRangeByPosition(oZelle.CellAddress.Column, oZelle.CellAddress.row, _
          oZelle.CellAddress.Column + uBound(aZeile()), oZelle.CellAddress.row +
uBound(aDaten))

  if aOpt(6) then   'Calc wählt Typen
     oBereich.setFormulaArray(aDaten)
  else
    oBereich.setDataArray(aDaten)
  end if
end sub


'/** CSV_Liste_Einlesen
'*************************************************************************.
'* @kurztext  liest die CSV-Liste ein
'* Diese Funktion liefert die CSV Datei ein und liefert die Daten als Array zurück.
'*
'* @param1  sURL as string  Dateiname und Pfad
'* @param2  aOptionen as array  Liste der Optionen
'*                              '7 Werte: FT, DT, TT, TK, ZS, ersteZeile, Feldeinstellung
'*
'* @return  aDaten as array   die einzutragenden Daten
'************************************************************************
'*/
function CSV_Liste_Einlesen(sCSVURL as string, aOpt())
  dim oSFA as variant   'simplefileOption
  dim oInputSream as variant
  dim oDatei as variant
  dim aDaten()         'Datenarray
  dim sZeile as string          ' eine Zeile
  dim aZeile()
  dim n%, d%

  oSfa = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  if NOT oSFA.exists(sCSVURL) then
    msgbox ("Die Datendatei kann nicht gelesen werden - sie existiert nicht!", 16, "Fehler")
    exit function
  end if
```

```
  oInputStream = createUnoService("com.sun.star.io.TextInputStream")
  oInputStream.setEncoding(aOpt(4))  'Zeichensatz
  oDatei = oSFA.OpenFileReadWrite(sCSVURL)
  oInputStream.SetInputStream(oDatei.getInputStream)
  REM erste Zeile lesen
  n = 0
  sZeile = oInputStream.readLine
  if aOpt(3) <> "" then 'Texttrenner extrahieren
    aZeile = helper.extractWithTextTrenner(sZeile, aOpt(3), aOpt(0))
  else
    aZeile = split(sZeile, aOpt(0))
  end if
  if aOpt(6) then aZeile = helper.DatenNurText(aZeile)
  d = uBound(aZeile)   'Dimension
  if aOpt(5) then   'erste Zeile eintragen
    redim aDaten(n)
    aDaten(n) = aZeile
    n = n+1
  end if
  Rem Rest der Daten -
  do while NOT oInputStream.isEOF
    redim aZeile()
    redim preserve aDaten(n)
    sZeile = oInputStream.readLine
    if aOpt(3) <> "" then 'Texttrenner extrahieren
      aZeile = helper.extractWithTextTrenner(sZeile, aOpt(3), aOpt(0))
      aZeile = helper.DatenDezimalTausender(aZeile, aOpt(1), aOpt(2))
    else
      aZeile = split(sZeile, aOpt(0))
      aZeile = helper.DatenDezimalTausender(aZeile, aOpt(1), aOpt(2))
    end if
    REM evt. Einträge als Text formatieren
    if NOT aOpt(6) then aZeile = helper.DatenNurText(aZeile)
    if uBound(aZeile) <> d then
      redim preserve aZeile(d)
      if isEmpty(aZeile(d)) then aZeile(d) = ""
    end if
    aDaten(n) = aZeile
    n = n+1
  loop
  oInputStream.closeInput()

  CSV_Liste_Einlesen = aDaten
end function
```

Zur Aufbereitung der Daten wurden noch diverse Hilfsfunktionen genutzt, die im Modul „helper“ standen. Zur Vollständigkeit seien diese hier jetzt aufgeführt:

```
'/** DatenNurText
'*************************************************************************.
'* @kurztext  liefert einen Datenarray-Zeile als "nur Text" zurück.
'* Diese Funktion liefert eine Datenzeile als Array zurück, dabei werden
'* alle Felder als "Text" für das Eintragen in Calc vorbereitet.
'*
'* @param1  aZeile as array   Datensatz zum Aufarbeiten
'*
```

```
'* @return  aZeile as array   die Aufbereitete Datenzeile als Array
'************************************************************************
'*/
function DatenNurText(aZeile)
  dim i%
  for i = 0 to uBound(aZeile)
    select Case left(aZeile(i), 1)
      case "1","2","3","4","5","6","7","8","9","0","+", "-", "_"
         aZeile(i) = replaceString(aZeile(i), ",", ".")  'interne Punkte (Dezimaltrener) durch
Komma ersetzen
    end select
  next
  DatenNurText = aZeile
end function

'/** DatenDezimalTausender
'************************************************************************.
'* @kurztext  liefert einen Datenaray-Zeile als "nur Text" zurück.
'* Diese Funktion liefert eine Datenzeile als Array zurück, dabei werden
'* alle Felder als "Text" für das Eintragen in Calc vorbereitet.
'*
'* @param1  aZeile as array   Datensatz zum Aufarbeiten
'*
'* @return  aZeile as array   die Aufbereitete Datenzeile als Array
'************************************************************************
'*/
function DatenDezimalTausender(aZeile, sDz as string, sTs as string)
  dim i%
  for i = 0 to uBound(aZeile)
    select Case left(aZeile(i), 1)
      case "1","2","3","4","5","6","7","8","9","0","+", "-"
        if NOT (instr(aZeile(i), ".") > 1) then   'Datumswerte haben zwei Punkte - nicht
ersetzen
          if NOT (sTS = "") then aZeile(i) = replaceString(aZeile(i), "", sTs)
          if NOT (sDz = "") then aZeile(i) = replaceString(aZeile(i), ".", sDz)
        elseif isDate(aZeile(i)) then     'Wert kann in Datumsformat umgewandelt werden
          aZeile(i) = Format(cDate(aZeile(i)), "MM.DD.YYYY")
        end if
    end select
  next
  DatenDezimalTausender = aZeile
end function

'/** extractWithTextTrenner
'************************************************************************.
'* @kurztext  extrahiert die Datenzeilen, wenn ein Texttrenner angegeben ist
'* Diese Funktion liefert eine Datenzeile als Array zurück. Dabei werden
'* Texttrenner ausgewertet und die Daten entsprechend aufbereitet.
'*
'* @param1  sZeile as string   Datensatz aus CSV Datei
'* @param2  sTextKZ as string  Textkennzeichen
'* @param3  sFeldTr as string  Feldtrenner
'*
'* @return  aDaten as array   Datenzeile als Array
'************************************************************************
'*/
function extractWithTextTrenner(sZeile, sTextkz, sFeldTr)
  dim aZeile()
  dim zz as long  'Zeichenzähler
```

```
  dim sTxt as string  'Feldinhalt
  dim n as long   'Zeilenzähler

  zz = 1 : n = 0
  do until zz >= len(sZeile)
    if left(trim(mid(sZeile, zz)), 1) = sTextKZ then   'mit Texttrenner
      sTxt = trim(mid(sZeile, instr(zz, sZeile, sTextKz)+1, instr(zz, sZeile , sTextkz &
sFeldTr)-1-zz))

        if instr(zz, sZeile , sTextKz & sFeldTr) >0 then
          zz = instr(zz, sZeile , sTextKz & sFeldTr)+2
        else
          sTxt = left(stxt, len(stxt) -1)
          exit do
        end if

    else    'ohne Texttrenner
      sTxt = trim(mid(sZeile, zz, instr(zz, sZeile ,sFeldTr)-zz)
      if instr(zz, sZeile ,sFeldTr) > 0 then
        zz = instr(zz, sZeile ,sFeldTr)+1
      else
        exit do
      end if
    end if
    redim preserve aZeile(n)
    aZeile(n) = sTxt
    n = n + 1
  loop
  REM Letztes Feld einlesen
  redim preserve aZeile(n)
  aZeile(n) = sTxt
  extractWithTextTrenner = aZeile
end function
```

Es ist also durchaus einiges an Code notwendig, um eine „reine“ Textdatei (CSV) korrekt und flexibel einzulesen.

3.4.3.2 Schreiben einer CSV-Datei

Technisch sehr ähnlich ist das Schreiben einer CSV-Datei. Wie eine Textdatei generell geschrieben wird, wurde schon in Kapitel 3.4.2 ausführlich dargelegt, ich verzichte hier auf eine Wiederholung. Allerdings müssen die Daten zunächst vorbereitet werden – quasi umgekehrt wie beim Lesen.

Dabei sind folgende Hinweise zu beachten:

Beim Lesen der Daten aus Calc: Datumswerte sind intern Double-Zahlen und müssen lesbar umgewandelt werden (z.B. mit format(<Zellinhalt>, "yyyy-mm-dd hh:mm").

Auch Dezimalzahlen müssen oft auf amerikanische Schreibweise geändert werden (Dezimaltrenner ist dann der Punkt, Tausendertrenner das Komma). Auch hier empfiehlt sich die folgende Vorgehensweise: (Beispiel: zwei Stellen nach dem Komma):

```
sZahlNeu = format(<Zelle>.value, "#,##0.00")      'der Wert der Zelle als Text, aber deutsche
```

```
Schreibweise!
sZahlNeu = join(split(sZahlNeu, ","), "?")     'Tausendertrenner ersetzen
sZahlNeu = join(split(sZahlNeu, "."), ",")     'Dezimaltrenner austauschen
sZahlNeu = join(split(sZahlNeu, "?"), ".")     'Tausendertrenner neu ersetzen
```

Umwandeln in einen Text, Ersetzen des Kommas durch einen neuen Platzhalter (z.B. ?), Ersetzen des Kommas durch einen Punkt, Ersetzen des neuen Platzhalters durch das Komma. Als Ergebnis erhält man nun die passende Zahl (als Text). Aber Achtung: Dieser Text darf jetzt nicht „maskiert“ werden.

Absatzwechsel in Zellen(-Texten) müssen entfernt werden (zum Beispiel mit Leerzeichen ersetzen).

Werden im Text doppelte Hochzeichen verwendet, so müssen diese „maskiert“ werden (in Basic).

Werden im Text die gleichen Zeichen verwendet, die auch in der CSV-Datei als Text-Identifier eingesetzt werden, so müssen diese ebenfalls entfernt oder ersetzt werden.

Möglicherweise sind noch andere Vorbereitungen notwendig. Die Daten werden typischerweise in einem Array gesammelt, wobei jede Zeile wiederum einen Array darstellt. Diese Daten können dann in eine Textdatei geschrieben werden:

Beispiel-Code zum Schreiben der CSV-Datei. Die Aufbereitung der Daten ist dabei bereits abgeschlossen:

```
public const  Zeichensatz as string = "ANSI"    'Zeichensatz der CSV-Datei

'/** ErzeugeCSVDatei()
'*************************************************************************.
'* @kurztext Schreibt eine einzelne CSV Datei
'* Diese Funktion schreibt eine einzelne CSV-Datei mit den als Array übergebenen Daten
'* (Pro Element eine Zeile). Der Zeilenumbruch wird durch die Kombination CR/LF erzeugt -
'* chr(13) + chr(10) - also passend für Windows.
'* Der Name und Pfad der Datei wird ebenfalls übergeben. Existiert die Datei schon wird sie
'* ohne Rückmeldung gelöscht.
'*
'* @param1  sURL as string  Pfad und Name der CSV-Datei
'* @param2  aDaten() as array  die Daten als Zeilenarray
'* @param3  bNextIn as Boolean  (optional, Vorgabe false) true, wenn Daten an bestehende
'*                              Datei angehängt werden sollen, sonst false
'*
'* @return  bFlag as boolean  True, wenn alles OK, false, wenn Fehler
'*************************************************************************
'*/
function ErzeugeCSVDatei(sUrl as string, aDaten(), optional bNextIn as Boolean)
  dim oSFA as variant
  dim oOutputStream as variant
  dim oInputStream as variant
  dim oDatei as variant
  dim bNext as boolean
  dim i%

  on error goto Fehler
```

```
  if isMissing(bNextIn) then
    bNext = false
  else
    bNext = bNextIn
  end if

  oSFA = createUnoService("com.sun.star.ucb.SimpleFileAccess")
  oOutputStream = createUnoService("com.sun.star.io.TextOutputStream")
  oInputStream = createUnoService("com.sun.star.io.TextInputStream")

  REM Datei erzeugen, zunächst  evtl. eine vorhandene Datei löschen
  if not bNext then    'wenn nicht angehangen wird, löschen
    if oSFA.exists(sURL) then kill(sURL)
    oDatei = oSFA.OpenFileReadWrite(sURL)
  else
    oDatei = oSFA.OpenFileReadWrite(sURL)
    oInputStream.SetInputStream(oDatei.getInputStream)
    do until oInputStream.isEOF
      oInputStream.readLine()
    loop
  end if

  REM jetzt Daten schreiben, Titelzeile aber nur wenn neu
  with oOutputStream
    .setOutPutStream(oDatei.getOutputStream)
    .setEncoding(Zeichensatz)
  end with

  REM evtl. Kopfzeile schreiben oder Zeilenumbruch
  if bNext AND (uBound(aDaten()) > 0) then   'anhängen, Zeilenumbruch, aber nur, wenn auch
Daten kommen!
    oOutputStream.writeString(chr(13) & chr(10))
  elseif bNext then       'anhängen, aber keine Daten folgen
    'nix tun
  else    'neue Datei Kopfzeile schreiben
    oOutputStream.writeString(aDaten(0))
    if uBound(aDaten()) > 0 then oOutputStream.writeString(chr(13) & chr(10))
  end if
  REM Rest der Daten schreiben
  for i = 1 to uBound(aDaten)
     oOutputStream.writeString(aDaten(i))
     if not (i = uBound(aDaten())) then oOutputStream.writeString(chr(13) & chr(10))
  next
  ErzeugeCSVDatei = true
  exit function
Fehler:
  ErzeugeCSVDatei = false
end function
```

Neben CSV-Dateien spielen Konfigurationsdateien oft eine wichtige Rolle. Das Vorgehen ist jedoch identisch. Auch hier werden die Textdateien typischerweise in einen Array eingelesen (zeilenweise) bzw. aus einem Array heraus geschrieben (auch zeilenweise).

Die interne Verarbeitung der Inhalte hängt dann von der Spezifikation der Konfigurationsdatei ab – normalerweise besteht diese aber nur aus zwei Typen von Zeileninhalten: Kommentar-

oder Beschreibungszeile und Konfigurationszeile, wobei die letztere typischerweise der folgenden Notation entspricht: KonfigurationswertName <Trenner> Konfigurationswert – Wert.

Kommentarzeilen beginnen normalerweise mit einem festgelegten Zeichen und lassen sich somit leicht identifizieren. Alle anderen (nicht leeren) Zeilen sind Werte-Zeilen. Diese werden am Trenner aufgeteilt und eventuell auch daran identifiziert – schon hat man die benötigten Werte.

4 Applikationen in/mit OpenOffice.org

Alle Programme (Applikationen) und Makros, die in und mit OpenOffice.org geschrieben werden, greifen über die UNO-API direkt auf das Programm zu. Dieser Mechanismus bedingt unterschiedliche Herangehensweisen. Die beiden Extreme:

- Alle Funktionen werden im eigenen Programm-Code realisiert – die API dient lediglich zur Datenein- und -ausgabe (also zum Beispiel dem Lesen von Daten aus einer Calc-Zelle oder dem Schreiben von Daten dort hinein).
- Alle Funktionen sind bereits im Core-Code vorhanden und die Applikation dient lediglich dem „Anstoß“ von Aktionen auf der Basis von Ereignissen (Beispiel: Drucken der aktuellen Datei mit „uno:print“). Dies entspricht in etwa dem Klick auf einen vorhandenen Button bzw. auf ein Icon der UI. Funktionen nutzen dann die internen Dialoge und arbeiten diese vollständig ab – der/die Benutzer/in muss wie im normalen Programm interagieren.

In der Regel ist jede Applikation eine Mischung beider Möglichkeiten mit deutlichem Schwerpunkt auf der ersten Option; manche gewünschten Funktionen lassen sich jedoch über diese erste Option nicht oder nur sehr aufwendig erreichen. Dann nutzt man in der Regel die Möglichkeit des Dispatchers.

4.1 Der Dispatcher

Der Dispatcher (übersetzt am ehesten mit „Helferlein“[3]) ist ein eigener Service, der die im Core-Code definierten Funktionen aufrufen kann. Dies sind in erster Linie alle Funktionen hinter den Schaltflächen und Icons aller Symbolleisten und noch einige mehr. Ein Dispatcher wird sehr einfach deklariert und ausgeführt:

```
Sub dispatchURL(urlStr as String)
  Dim frame As Object
  Dim dispatch As Object
  Dim url As New com.sun.star.util.URL
  Dim args() As New com.sun.star.beans.PropertyValue

  url.Complete = urlStr
  frame = ThisComponent.currentController.Frame
```

3 dispatcher = Verteiler, Versender, Erlediger

```
  dispatch = frame.queryDispatch(url, "_self", com.sun.star.frame.FrameSearchFlag.SELF)
  dispatch.dispatch(url, args)
End Sub
```

Oder alternativ:

```
Sub dispatchURL(urlStr as String)
  Dim frame As Object
  Dim dispatch As Object
  Dim args() As New com.sun.star.beans.PropertyValue

  frame = ThisComponent.currentController.Frame
  dispatch = createUnoService("com.sun.star.frame.DispatchHelper"
  dispatch.executeDispatch(frame, urlStr,"_self",0, args)
End Sub
```

Die Schwierigkeiten bestehen nun darin, erstens die „urlStr“, also die internen Programmaufrufe, zu kennen und dann natürlich die entsprechenden Parameter, falls welche benötigt werden.

Man kann versuchen, den gewünschten Vorgang per UI durchzuführen und den Makro-Recorder mitlaufen zu lassen – dieser erzeugt nämlich lediglich Dispatcher-Aufrufe. Dann erhält man zumindest den URL-Code und eventuell auch die benötigten Parameter. Auch kann man die Definitionsdateien der Symbolleisten durchforsten – auch dort sind die URLs hinterlegt, die aufgerufen werden beim Klick auf das Icon. Bei Fremd-Applikationen gibt es zusätzlich in der Regel gute Dokumentationen, in denen die benötigten Parameter sowie Aufrufe beschrieben werden – damit sind die Quellen aber auch umfassend beschrieben.

Das grenzt dann den Einsatz des Dispatchers auch klar ein.

Andererseits gibt es gute Gründe, diesen manchmal einzusetzen. So ist zum Beispiel der Befehl „Rückgängig“ (also die letzte Aktion des Programms/Benutzers rückgängig machen) per Code nur sehr aufwendig bis gar nicht zu lösen, mit der URL „uno:Undo“ und dem Dispatcher ist es hingegen ganz einfach:

```
  dispatcher.executeDispatch(document, ".uno:Undo", "", 0, Array())
```

Ähnliches gilt auch immer dann, wenn man schon eingebaute Prozesse nicht selbst nachbauen will – wie zum Beispiel das Exportieren als PDF inklusive dem Optionen-Dialog, das Speichern mit Abfrage des Dateinamen und weitere Optionen.

Auch der WollMux bietet einige Dispatch-Kommandos zum direkten Aufruf interner Funktionen. Siehe hierzu auch Dokumentation des WollMux.

In allen anderen Fällen jedoch ist „native Code“ die korrekte Wahl und wohl auch die Regel.

4.2 Umgang mit Strings, Werten und Arrays

Das Kochbuch ersetzt kein Grundlagenwissen – die Variablentypen werden hier nicht weiter vorgestellt. Wohl aber „best practice“ und „Fallen“.

4.2.1 Strings

Strings – also Texte – sind sicher mit die am häufigsten benutzten Variablen-Typen. String-Variable können maximal 64K groß werden – das erscheint zunächst „groß genug“, birgt aber in der Praxis so manche „Falle“. So lässt sich zwar der Text eines Writer-Dokumentes komplett in eine String-Variable einlesen (oDoc.Text.String), doch eben nur die ersten 64K – das sind im Extremfall lediglich rund 20.000 Zeichen, also 4 normal gefüllte DIN A4-Seiten. Da es keine Fehlermeldung gibt, würde eine Überschreitung der Anzahl der Zeichen gar nicht auffallen, führte aber unweigerlich zu nicht erwünschten Ergebnissen.

Einen ähnlichen Effekt erlebt man, wenn die Module eines Makroprogramms eingelesen werden sollen. Auch hier können die Textinhalte ganz einfach über

```
sCode = BasicLibraries.getByName(sBibName).getByName(sModulName)  'der komplette Code
```

eingelesen werden – wird hier aber die 64K Grenze überschritten (in dem Fall sind das ca. 64-tausend Zeichen inkl. Leerzeichen und Sonderzeichen!), so bleibt die Variable einfach leer. Auch das führt dann zu unerwünschten Ergebnissen.

In all diesen Fällen muss der/die Programmierer/in unbedingt darauf achten, die maximale Größe nicht zu überschreiten – oder entsprechende Maßnahmen ergreifen.

Für den ersten Fall könnte man eine Absatz-Analyse vornehmen und den Gesamttext als Array von Absatz-Texten betrachten, im zweiten Fall müsste der Modultext über einen anderen Weg zeilenweise eingelesen werden.

Absatzzeichen im Text

Absatzzeichen werden in jedem Betriebssystem und oft auch innerhalb von Programmen unterschiedlich gehandhabt. Typischerweise gibt es folgende Alternativen: CR (Carriage Return (Chr(13)) und LF (Line Feed – Chr(10)). Während Windows eine Kombination aus beiden Zeichen nutzt (CRLF), kommen Linux/Unix Systeme mit lediglich einem Zeichen (LF) aus.

OOo/LibO verarbeitet intern das Zeichen CR (chr(13)) als „harten“ Zeilenumbruch (Absatzende) und LF (chr(10)) als „weichen“ Zeilenumbruch – also einen Zeilenwechsel ohne Absatzwechsel.

Das ist immer dann zu beachten, wenn man „einfache“ Textdateien einliest und diese in ein OpenOffice-Dokument überträgt (und natürlich auch umgekehrt!). Die normale String-Variable beinhaltet selbstverständlich ebenfalls Sonderzeichen – und sie können auch geschrieben werden:

```
sString = "dies ist die erste Zeile" & chr(13) & "dies ist die zweite Zeile"
```

liefert einen zweizeiligen Text mit Absatzumbruch – in Writer oder Calc. Umgekehrt geht dies aber genauso: Liest man zum Beispiel den Inhalt einer Calc-Zelle (oder auch eines Formular-/Dialog-Kontrollelementes) ein und wurde dort ein Zeilenumbruch verwendet, so beinhaltet der String diesen als ASCII-Zeichen (13) – und wird intern immer wieder als „Absatzumbruch“ oder „Zeilenende“ interpretiert. Braucht man nun die „Strings“, um daraus zum Beispiel einen SQL-Befehl zusammenzubauen, so führt dies mit Sicherheit zum Fehler.

Beispiel:

Inhalt einer Tabellenzelle (A1):

```
dies ist wichtig
zweite Zeile
```

Einlesen in eine Variable:

```
sInhalt = oTab.getCellRangeByName(A1).string
```

SQL-Befehl aufbauen:

```
sSQL = "INSERT INTO <tabellenname> (spaltenname) VALUES ('" & sInhalt & "')
```

Führt im Ergebnis zu einem SQL-Fehler, da der übermittelte Befehl später wie folgt aussehen würde:

```
INSERT INTO <tabellenname> (spaltennamen) VALUES ('dies ist wichtig
zweite Zeile')
```

Der Zeilenumbruch im SQL-Kommando kann nicht aufgelöst – die Zeilen nicht interpretiert werden.

Es ist also unbedingt darauf zu achten, wie Strings eingesetzt und verwendet werden.

Oft müssen diese zunächst „bereinigt“ werden – ein solches Beispiel wurde schon in Kapitel 3.4.3 dargestellt – hier wurden Dezimaltrenner ausgetauscht. Der schnellste Weg, in einem String Teile zu ersetzen, ist der folgende:

```
sString = join(split(sString, sAlterWert), sNeuerWert)
```

wobei der sAlterWert das/die Zeichen beinhaltet, der/die ersetzt werden soll/en, und sNeuerWert den Ersatzwert repräsentiert. So können zum Beispiel alle Leerzeichen des Strings mit der Codierung %20 (URL-Codierung) schnell ersetzt werden:

```
sString = join(split(sString, " ") , "%20")
```

Als neuer Wert kann auch „nichts“ stehen – also ein leerer String – dann wird das gesuchte Element einfach entfernt. Ist das Element nicht enthalten, so gibt es keinen Fehler – eine sehr performe und universelle Methode.

Kürzen von zu langen Strings (Pfadangaben)

Bei der Arbeit mit Dialogen und Formularen kommt es immer wieder vor, dass man einen Dateinamen inklusive des Pfades darstellen will, der zur Verfügung stehende Platz aber

eventuell nicht ausreicht, um alles zu zeigen. Dann kürzt man den Pfad, und zwar in der Mitte, das heisst, der Anfang ist sichtbar (und der komplette Dateiname), längere Pfade dazwischen sind aber „gekürzt“.

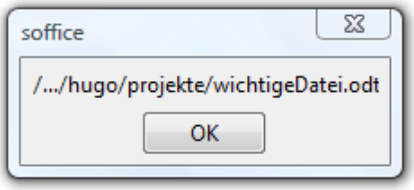
Wichtig: Wird der Pfad nochmals benötigt, so muss er unbedingt zwischengespeichert werden (z.B. in einer zusätzlichen Variablen oder auch einem anderen – nicht sichtbaren – Kontrollelement, so dass man später mit der kompletten Variante arbeiten kann).

```
'/** ShortenPathView
'*******************************************
'* @kurztext kürzt einen übergebenen Pfad auf eine entsprechende Zeichenanzahl
'* Die Prozedur kürzt einen Pfad/Datei string auf eine
'* ebenfalls übergebene Länge von Zeichen. Die Kürzungen erfolgen am Trenner,
'* wenn möglich.
'* Wichtig: Die Funktion liefert lediglich einen String des gekürzten Pfades!
'* Dieser kann nicht zurückgewandelt werden und dient nur der Darstellung!
'* Hiermit kann nicht weiter gearbeitet werden!
'*
'* @param1  sPfad as string  die "lange" Pfad/Dateibezeichnung
'* @param2  iZeichenMax as integer   die maximale Länge des Ergebnisstrings
'* @param3  sTrenner as string   Pfadtrenner
'* @param4  bStart as boolean   (optional) wenn true, wird der Pfad in der Mitte gekürzt
'*                               wenn nicht übergeben dann ab Start
'*
'* @return  sNeuPfad as string  der gekürzte Pfad
'*
'*******************************************
'*/
function ShortenPathView(sPfad as string, iZeichenMax as integer, _
              sTrenner as string, optional bStart as boolean)
 dim sNeuPfad as string  'neuer Pfad
 dim sPfadNeuStart as string 'Start des Neupfades
 dim bFlag as boolean

 if isMissing(bStart) then
  bFlag = false
 else
  bFlag = bStart
 end if
 REM Wenn Pfad schon kurz genug oder weniger als 10 Zeichen gefordert werden - Ende
 if (len(sPfad) <= iZeichenMax) OR (iZeichenMax < 10) then
  ShortenPathView = sPfad
  exit function
 end if

 sNeuPfad = sPfad
 REM ersten Trenner suchen - nur bei bFlag = true
 if bFlag then
  sPfadNeuStart = left(sPfad, inStr(sNeuPfad, sTrenner))
  sNeuPfad = right(sNeupfad, len(sNeuPfad) - len(sPfadNeuStart))
 end if

 REM Letzten Trenner suchen, dass der Pfad noch passt
 do until inStr(sNeuPfad, sTrenner) = 0
  sNeuPfad = Mid(sNeuPfad, instr(sNeuPfad, sTrenner)+1)
  if len(sNeuPfad) <= iZeichenMax-4 then exit do  'Länge passt - Ende
```

```
    if instr(sNeuPfad, strenner) = len(sNeuPfad) then exit do 'nur noch das letzte Zeichen ist der
Trenner
   loop
   REM Falls Pfad nicht über Trenner kürzbar, Zeichen kürzen (am Anfang)
   if len(sNeuPfad) <= iZeichenMax-4-len(sPfadNeuStart) then
    sNeuPfad = sPfadNeuStart & "..." & sTrenner & sNeuPfad
   else 'Pfad passt noch nicht, vorne kürzen
    sNeuPfad = sPfadNeuStart & "..." & right(sNeuPfad, iZeichenMax - 4 -len(sPfadNeuStart) )
   end if

   ShortenPathView = sNeuPfad

end function
```

ShortenPathView im Einsatz:

```
sPfad = "/home/mein1Verzeichnis/mein2Verzeichnis/irgendwo/daten/hugo/projekte/wichtigeDatei.odt"

msgbox ShortenPathView(sPfad , 40, "/", true)
```



Daten „auf Länge“ formatieren

Eine andere oft benötigte Variante ist die tabellenartige Darstellung von Daten in Listenfeldern (Dialoge oder Formulare). In Formularen gibt es hierzu zwar das Tabellen-Grid-Kontrollelement, das dies „automatisch“ organisiert, in Dialogen ist dies jedoch noch nicht implementierbar – und manchmal ist es einfach hilfreich (oder auch erforderlich), Informationen quasi spaltenweise zu formatieren.

Dazu wird das spätere Kontroll-Feld mit einem Schrifttyp fester Breite formatiert – dadurch sind alle Zeichen gleich breit und stehen exakt untereinander. Nun werden die Einträge der Listbox (jeder Eintrag ist ein String) exakt gleich ausgerichtet, das heisst, für jede Spalte gibt es eine bestimmte Anzahl von Zeichen und die dort einzufüllenden Daten werden linksbündig eingetragen – eventuell gekürzt, falls die Länge die vorgesehene Spaltenbreite übersteigt, oder, im umgekehrten Fall, Leerzeichen hinzugefügt. Das folgende Beispiel soll dies verdeutlichen (ich nutze in diesem Beispiel nur 3 Spalten, die Anzahl könnte aber auch beliebig sein):

ohne Formatierung	gewünschtes Ergebnis (erreicht mit Formatierung)

Die Datenzeilen sind normalerweise Teil eines Arrays und werden entsprechend behandelt.

Die folgende Funktion erledigt die Formatierung:

```
'/** ListeSymAufbereiten()
'*************************************************************************.
```

```
'* @kurztext bereitet einen Array als Liste mit festen Zeichenspalten auf
'* Diese Funktion bereitet einen Array als Liste mit festen Zeichenspalten auf
'* Sind die Spalteneinträge länger als die Spaltenbreite, so werden die überfälligen
'* Zeichen abgeschnitten und zwei Punkte ".." ergänzt. Sind die Spalteneinträge kürzer,
'* so erfolgt ein Auffüllen mit Leerzeichen.
'* Wichtig: Anzahl der Spalten (Arrayelemente) muss identisch sein dem übergebenen Längenarray!
'*
'* @param1  aListeAlt() as array   ein Array von (Zeilen-) Arrays (Werte der Spalten)
'* @param2  aFeldLen() as array  eine Liste der gewünschten Zeichenanzahlen pro Spalte (für
jede Spalte ein Eintrag!)
'* @param3  sSpTr as string   Spaltentrenner (kann leer sein)
'* @param4  iSpTrTyp as integer   Spaltentrenner Typ 0- keine, 1- nur erste Spalte, 2 - nur
letzte Spalte, 5 - alle
'*
'* @return  aListe() as array   die aufbereitete Liste
'************************************************************************
'*/
function ListeSymAufbereiten(aListalt, aFeldlen(), sSpTr as string, iSpTrTyp as integer )
  dim aListe(), aDSalt()
  dim sZeile as string, n%, j%, i%, sTxt as string

  if uBound(aListalt()) > -1 then  redim aListe(uBound(aListalt()))

  for i = 0 to uBound(aListe())
    aDSalt = aListalt(i)
    sZeile = ""
    for j = 0 to uBound(aFeldLen())
      n = aFeldLen(j)
      sTxt = aDSalt(j)
      if len(sTxt) > n-1 then
        if NOT (len(sTxt) <= 3) then sTxt = left(sTxt, n-3) & ".. "
      else
        sTxt = sTxt & Space(n-len(stxt))
      end if
      sZeile = sZeile & sTxt
      select case iSptrTyp
        case 1   'erste Spalte trennen
          if (j=0) then sZeile = sZeile & sSpTr
        case 2   'letze Spalte trenen
          if (j=uBound(aFeldLen())-1) then sZeile = sZeile & sSpTr
        case 5   'alle Spalten Trennen
          if NOT (j = uBound(aFeldLen())) then sZeile = sZeile & sSpTr
      end select
    next j
    aListe(i) = sZeile
  next i

   ListeSymAufbereiten = aListe()
end function
```

4.2.2 Werte

Wird von „Werten“ gesprochen, sind in der Regel Zahlen gemeint, mit denen man auch rechnen möchte. Zunächst gibt es wenig Besonderheiten zu beachten.

Basic akzeptiert als Dezimaltrenner nur den Punkt, ein Tausender-Trenner ist nicht vorgesehen. Eine Zahl wird allerdings in der lokalisierten Schreibweise dargestellt – also dann mit passendem Dezimaltrenner (in Deutschland das Komma).

Normalerweise kommen nur die folgenden Typen zum Einsatz: Integer, Long und Double. Auch wenn der Typ „Single“ meist ausreichen würde, wird dieser kaum genutzt. Ein Double-Wert wird auch intern mit höherer Genauigkeit berechnet – deshalb oft der „Double-Typ“.

Bei Ganzzahlen kommt häufig der Typ „Integer“ zum Einsatz, man sollte sich jedoch über seine Grenzen bewusst sein (ganze Zahl im Bereich -32.768 und +32.768). Während diese Grenzen in vielen Fällen völlig ausreichen, wäre beispielsweise ein Integer-Typ für das Zeilen-Zählen in Calc ungeeignet (mehr als 1 Million möglicher Zeilen!), für das Spaltenzählen hingegen würde er ausreichen.

Richtig kompliziert wird es bei der Berechnung von Währungszahlen und beim Rechnen damit – insbesondere wenn Zwischensummen ausgegeben und verändert werden können.

Während wir typischerweise mit zwei Nachkommastellen rechnen und somit immer gerundet wird, nutzt der Rechner beim Double-Typ den Wertebereich von 10E308 bis 10E-324 voll aus, also er rechnet viel genauer. Je mehr Rechenoperationen durchgeführt werden, um so eher wird es Rundungsabweichungen geben zu den (manuellen) 2-Stellen-Rechnungen.

Um das abzufangen, kann man generell mit Ganzzahlen rechnen. Dafür wird dann zunächst die Währungsdezimalzahl mit zwei Nachkommastellen mit 100 multipliziert und dann in eine Ganzzahl umgewandelt – jetzt erfolgen alle Rechnungen mit der Ganzzahl – zum Schluss wird die Ganzzahl dann wieder durch 100 dividiert – und so als Dezimalzahl mit 2 Stellen angezeigt. Dieses Verfahren kann die dargestellte Ungenauigkeit reduzieren – ist aber sicher kein „Allheilmittel“.

Wenig hilfreich ist in diesem Fall übrigens der Typ „Währung“, der grundsätzlich mit vier Nachkommastellen rechnet. Auch hier wird ja nur gerundet – eben da ab der vierten Stelle.

4.2.3 Arrays (Listen)

Arrays oder Listen sind ein sehr beliebter Variablentyp für die Programmierung von Makros – können sie doch Daten unterschiedlicher Art sinnvoll verwalten.

Zwar unterstützt Basic auch mehrdimensionale Arrays, diese haben in der Praxis jedoch nur einen geringen Nutzwert, da ein assoziativer Zugriff leider nicht so einfach realisiert werden kann.

Einen weitaus höheren praktischen Nutzwert hingegen haben Arrays von Arrays – also Listen, deren Elemente wiederum Arrays darstellen. Diese werden zum Beispiel genutzt beim Auslesen von Zellbereichen in Calc (der Bereich wird als Zeilenarray abgebildet – also jede Zeile spiegelt ein Element wieder, das Zeilenelement ist dann wiederum ein Array der Spaltenwerte) und

natürlich dann auch beim Schreiben der Werte in einen Calc-Bereich. Beide Methoden sind äußerst performant und unbedingt zu verwenden (siehe auch Kapitel 8, Calc).

Arrays werden auch benötigt für List- oder Combo-Boxen in Formularen oder Dialogen – für die Übergabe der Listeneinträge.

Eine Array-Variable muss in Basic deklariert werden, bevor sie benutzt werden kann – und sie wird bei der Deklaration bzgl. ihrer Größe bestimmt (meist). Beispiele:

```
dim aListe()     'deklariert die Variable aListe als Arrayvariable, aktuell ohne Elemente
(Länge -1)
aListe2 = array(1,2,3,4)      'deklariert die Variable aListe2 (ist zunächst unbestimmt) zu
einem Array mit 4 Elemente (Länge 3)
redim aListe(2)   'deklariert die Arrayvariable aListe() - die muss aber schon deklariert
gewesen sein! - als Arrayvariable mit 3 Elementen (Länge 2). Vorhandene Inhalte werden im
übrigen entfernt!
redim preserve aListe(2)   'deklariert die Arrayvariable aListe() - die muss aber schon
deklariert gewesen sein! - als Arrayvariable mit 3 Elementen (Länge 2). Vorhandene Inhalte
bleiben erhalten.
```

Doch die Arbeit mit Arrays hat auch Ihre Tücken. Ein paar typische Schwachpunkte:

Größen-Falle

Ein Array kann – laut OOo-Hilfe – maximal 16.368 Elemente indizieren. Tatsächlich ist die Obergrenze wohl höher, so lassen sich durchaus Elemente von -32.768 bis +32.767 indizieren, was insgesamt dann rund 65.000 Einträge ausmachen würde. Dennoch muss man sich der Grenzen bewusst sein – und bei sehr vielen Einträgen mit Fehlern rechnen.

Calc selbst unterstützt jetzt Tabellenblätter mit mehr als 65.000 Zeilen und 1.024 Spalten – ein solcher Bereich lässt sich in einem Array nicht mehr abbilden.

Und dass eine solche Grenze schnell erreicht wird, mag das folgende kleine Beispiel demonstrieren:

Ein Makro soll dazu dienen, alle Formeln einer Tabelle zu dokumentieren. Dazu werden zunächst alle Zellen mit Formeln aus dem Dokument (Tabelle) ausgelesen und anschließend wird aus jeder einzelnen Zelle die Formel in ein Element eines Arrays geschrieben.

```
'/** TIMM_getFormelListe()
'*************************************************************************.
'* @kurztext liefert eine Liste der Formeln
'* Diese Funktion liefert eine Liste der Formeln, und zwar als Liste von
'* Arrays mit zwei Werten:  Zelladresse, Formel
'* ### Achtung! falls mehr als 64000 Formelzellen vorhanden - Arrayüberlauf!
'*
'* @Param1 oQDoc as object  Objekt des Quelldokumentes
'* @param2 sTabName as string   Name der Tabelle
'*
'* @return aListe as array  die Liste der Formelzellen
'*************************************************************************
'*/
function TIMM_getFormelListe(oQDoc as object, sTabname as string)
  dim aElementNamen()
```

```
  dim aEinElement()
  dim oTab as object
  dim iSCol%, iECol%, iSRow as long, iERow as long
  dim sp as integer, ze as long
  dim oZelle as variant
  dim aZName()
  dim aFormel(1)      'Formelarray
  dim aFormelListe()
  dim i as long, n as long, n1 as long, j%

  n = 0
  n1 = 500
  redim aFormelliste(n1)

  oTab = oQDoc.getSheets().getByName(sTabname)
  aElementNamen = oTab.queryContentCells(16).ElementNames

  for i = 0 to uBound(aElementNamen)
    aEinElement = split(aElementNamen(i),".")  'Trennen Tabellennamen - Zellname (Bereich)
    j = uBound(aEinElement)     'nur das letzte Element des gesplitteten Bereiches enthält die
Adresse
    if instr(aEinElement(j), ":") > 0 then  'Bereich
      iSCol = oTab.getCellRangeByName(aEinElement(j)).RangeAddress.startColumn
      iECol = oTab.getCellRangeByName(aEinElement(j)).RangeAddress.EndColumn
      iSRow = oTab.getCellRangeByName(aEinElement(j)).RangeAddress.startRow
      iERow = oTab.getCellRangeByName(aEinElement(j)).RangeAddress.EndRow
      For sp = iSCol to iECol
        For ze = iSRow to iERow
          oZelle = oTab.getCellByPosition(sp, ze)
          aZName = split(oZelle.absoluteName,".")
          redim aFormel(1)
          if n > n1 then
            n1 = n1 + 500
            redim preserve aFormelliste(n1)
          end if
          aFormel(0) = DeleteStr(aZName(uBound(aZName)), "$")
          aFormel(1) = oZelle.FormulaLocal()
          aFormelliste(n) = aFormel()
          n = n+1
        next ze
      next sp

    else   'einzelne Zelle
      redim aFormel(1)
      if n > n1 then
         n1 = n1 + 500
         redim preserve aFormelliste(n1)
      end if
      aFormel(0) = DeleteStr(aEinElement(j), "$")
      aFormel(1) = oTab.getCellRangeByName(aEinElement(j)).FormulaLocal()
      aFormelliste(n) = aFormel()
      n = n+1

    end if
  next

  redim preserve aFormelliste(n-1)  'Liste auf maximale einträge kürzen

  TIMM_getFormelListe = aFormelliste()
```

```
end function
```

Die Formelliste kann hier theoretisch bis zu 65.000 Elemente enthalten. Wird nun aber versucht, diesen Array mit der Funktion

```
oTab.getCellRangeByPosition(0,1,1,uBound(aFormelliste())+1).setDataArray(aFormelliste())
```

in eine Writer-Tabelle zu schreiben, so führt dies zu einem Speicherüberlauf – und in dessen Folge zu einem Runtime-Error. In diesem Fall darf der Array maximal eine Größe von – durch Versuche getestet – rund 20.000 Elementen haben – es waren einige mehr, aber dies ist die Größenordnung.

Es empfiehlt sich also, bei der Arbeit mit Arrays auch die Anzahl der Elemente unbedingt im Auge zu behalten und entsprechend zu reagieren. Benötigt man mehr Elemente, so legt man eben einen Array von Arrays an – wobei dann jeder „Unterarray“ beispielsweise maximal 10.000 Elemente trägt.

Performance-Falle

Arrays nutzt man, um Daten zu speichern, also zum Beispiel um die Werte der Spalte A in Calc auszulesen. Die Problematik ist nun, dass man die Anzahl gar nicht kennt – der Array muss also entsprechend immer nachjustiert werden:

```
dim aListe()
n = 1     'Einlesen ab Zeile 2 (Index 1)
Do until oTab.getCellByPosittion(0,n).getType = 0   'einlesen, bis eine leere Zelle kommt
  redim preserve aListe(n-1)
  aListe(n-1) =  oTab.getCellByPosittion(0,n).getValue
loop
```

Das Prinzip ist klar – die Schleife läuft solange, bis die erste leere Zelle in Spalte A erreicht wird. Für jede Zeile wird der Array um ein Element erweitert und der aktuelle Zellwert darin gespeichert.

Dies ist ein System, das bis vielleicht 40-50 Zeilen schnell genug läuft, werden es aber zum Beispiel 1.000 Elemente, so sinkt die Performance erheblich und der/die Benutzer/in muss mehrere Minuten warten, bis die Daten eingelesen wurden. Als Performance-Killer erweist sich hier die „redim preserve“-Anweisung. Dies belastet den Prozessor und den Hauptspeicher unnötig lange. Steht also zu erwarten, dass größere Anzahlen von Daten eingelesen werden müssen, empfiehlt sich ein anderes Vorgehen: Geben Sie der Liste bei der ersten Deklaration eine bestimmte Grundgröße mit und passen Sie die Größe nur dann an, wenn die Grundgröße aufgebraucht ist – und auch dann wieder in großen Schritten.

Kürzen Sie am Ende den Array auf die tatsächlich benötigte Anzahl Elemente – dafür muss natürlich ein Zähler mitlaufen.

Beispiel: In den Array sollen alle Texte eingelesen werden, die mit „a“ beginnen. Die Schleife läuft ab Zeile 2 bis zur ersten leeren Zelle. Vorgabegröße für den Array ist 100.

```
private Const Lint as integer = 100     'Größenfaktor Array

...
  n2 = Lint    'Vorgabegröße Array
  redim aDaten(n2)
  j = 0
  REM jetzt für jede Zeile:
  Do until oTab.getCellByPosittion(0,n).getType = 0   'einlesen, bis eine leere Zelle kommt
    if lcase(left( oTab.getCellByPosittion(0,n).getString, 1) = "a" then    'passt
      aDaten(j) =  oTab.getCellByPosittion(0,n).getString
      j = j+1
      if j > n2 then
        n2 = n2 + Lint
        redim preserve aDaten(n2)
      end if
    end if
  loop
  REM Liste korrigieren
  if j > 0 then
    redim preserve aDaten(j-1)
  else
    redim aDaten()   'kein Datensatz vorhanden!
  end if
...
```

Werden größere Datenmengen erwartet (also so 2.000-5.000 Stück), passt man einfach die Konstante entsprechend an, z.B. auf 500 oder sogar 1.000. Dann bleiben so 5 bis 10 „redim“ Anweisungen – und die wird der/die Benutzer/in nicht „merken“.

Arrays sortieren

Oft ist es notwendig, Arrays zu sortieren, um eine passende Reihenfolge zu erhalten. Das „Sortieren“ ist immer zeitaufwendig – es ist also zu überlegen, ob diese Arbeit nicht in andere Teile zu verschieben ist (zum Beispiel bei der Arbeit mit Datenbanken – das Sortieren wird dem DBMS überlassen – Ergebnisse von SQL-Befehlen sind dann schon sortiert).

Kann man den Umweg über Calc gehen, so ist es performanter, den Array in einen Zellbereich zu schreiben, diesen dann mit dem eingebauten SortDescriptor zu sortieren und anschließend wieder auszulesen. Das alles macht natürlich nur Sinn bei großen Listen (> 100 Elemente).

Einen Array kann man auch über einen BubbleSort-Algorithmus relativ schnell sortieren lassen. Die Bibliothek „Tools“ bringt eine solche Funktion im Modul „Strings“ mit (bubbleSortList(byVal aListe(), optional bSort2ndValue as Boolean)). Man kann aber auch einen eigenen Code schreiben:

```
'/** ListeSortieren()
'*************************************************************************.
'* @kurztext sortiert eine Liste (Bubblesort)
'* Diese Funktion sortiert eine übergebene Liste (Array) entweder aufsteigend
'* oder absteigend. Verwendet wird ein BubbleSort Algorythmus.
```

```
'*
'* @param1  aListe() as array  die zu sortiernede Liste
'* @param2  bCaseSensitiv as boolean  Sortierflag: true = case-Sensitiv, false = caseinsensitiv
'* @param3  bSortFlag as boolean   Sortierflag - true = aufsteigend, false = absteigend
'*
'* @return  aSortListe() as array  die sortierte Liste
'*************************************************************************
'*/
function ListeSortieren(byVal aListe(), bCaseSensitiv as boolean, bSortFlag as boolean)
  dim i as integer   'Zählvariable
  dim n as integer   'Dimension der Liste
  dim t as integer   'Zähler
  dim vElement as variant

  n = uBound(aListe())

  For i = 1 to n-1
    For t = 0 to n-i
      if bSortFlag then   'aufsteigend
        if bCaseSensitiv then
          if aListe(t) > aListe(t+1) then
            vElement = aListe(t)
            aListe(t) = aListe(t+1)
            aListe(t+1) = vElement
          end if
        else
          if lCase(aListe(t)) > lcase(aListe(t+1)) then
            vElement = aListe(t)
            aListe(t) = aListe(t+1)
            aListe(t+1) = vElement
          end if
        end if
      else                'absteigend
        if bCaseSensitiv then  'Casesensitiv
          if aListe(t) < aListe(t+1) then
            vElement = aListe(t)
            aListe(t) = aListe(t+1)
            aListe(t+1) = vElement
          end if
        else
          if lcase(aListe(t)) < lcase(aListe(t+1)) then
            vElement = aListe(t)
            aListe(t) = aListe(t+1)
            aListe(t+1) = vElement
          end if
        end if
      end if

    next t
  next i

  ListeSortieren = aListe()
end function
```

Der Vorteil dieses Codes: es kann sowohl aufsteigend als auch absteigend sortiert werden und man kann auch noch angeben, wie Strings sortiert werden sollen (case-sensitiv oder nicht).

4.3 ThisComponent – eine vordefinierte Variable

In Basic kann man sehr einfach Zugang zum aktuellen Dokument erhalten. Die (vordefinierte) Variable beinhaltet immer das Objekt des aktuellen Dokumentes – oder genauer: des aktuellen Hauptmoduls und der davon abgeleiteten Komponente, die gerade den Fensterfokus besitzt. Es sind also nur die Hauptkomponenten (Writer, Calc, Draw, Impress oder Base, manchmal jedoch auch die Basic-IDE), deren aktuelles Fenster und das darin befindliche Dokument der Variablen zugewiesen wird.

Intern wechselt somit der Inhalt von „ThisComponent“, sobald man das aktive Fenster wechselt. Das grenzt den Einsatz von ThisComponent deutlich ein. Auf der einen Seite ist es eine bequeme Möglichkeit, Zugang zu dem aktuellen Dokument zu erhalten, auf der anderen Seite ist diese Variable trügerisch, ändert sie doch schnell den Inhalt. Benötigt man also den dauerhaften Zugang zu einem bestimmten Dokument, so muss der Inhalt von ThisComponent einer neuen Variablen zugewiesen und dann nur noch mit dieser gearbeitet werden. Dies geschieht typischerweise ganz am Anfang des Programms – nur so ist sichergestellt, dass das korrekte Dokument gewählt wurde!

Typische Einsatzgebiete von ThisComponent:

Ein Makro wird aus einem Dokument heraus gestartet – sei es mit Hilfe einer Symbolleiste, eines Menübefehls oder eines Schalters im Dokument, oder auch eines Ereignisses – und das Programm nimmt Manipulationen am aktiven Dokument vor: Jetzt wird mit Hilfe von ThisComponent eine direkte Beziehung zum Dokument hergestellt, dieses in eine eigene (globale) Variable abgelegt und nun kann das Dokument bearbeitet werden – auch wenn der/die Benutzer/in zwischenzeitlich ein anderes Dokument aktiviert.

Beispiel (korrekt):

```
public oDoc as variant

sub eineWichtigefunktion
  odoc = thisComponent
  oDoc.text.string = "Hallo, hallo"    'ein Text wird eingegeben
  '… hier folgen jetzt jede Menge Anweisungen (intern)
  oDoc.print(args())     'Dokument wird gedruckt
end sub
```

Würde diese Programm wie folgt geschrieben, so könnte es einige „Überraschungen“ geben:

```
sub eineWichtigefunktion
  thisComponent.text.string = "Hallo, hallo"    'ein Text wird eingegeben
  '… hier folgen jetzt jede Menge Anweisungen (intern)
  thisComponent.print(args())     'Dokument wird gedruckt
end sub
```

In der Zeitspanne zwischen dem Schreiben des Textes und dem Drucken könnte der/die Benutzer/in das Dokument auf dem Bildschirm gewechselt haben (durch Klick auf ein anderes

geöffnetes Dokument), in diesem Fall verweist „thisComponent“ nun auf das aktive Dokument und dieses würde gedruckt werden. ThisComponent muss also mit Vorsicht genutzt werden, insbesondere, wenn das Makro nicht direkt aus dem Dokument heraus gestartet wurde (in diesem Fall wäre es nämlich ein Tochterprozess des aktuellen Dokumentes – und der „Rest“ wäre erst einmal blockiert). Insbesondere aber bei Extensions wird der Makro-Prozess unabhängig vom aktuellen Dokument gestartet – und dann kommt es zu den oben beschriebenen Phänomenen.

4.4 Makrospeicherung und Orte

Makros (insbesondere Basic-Makros) können an drei verschiedenen Orten gespeichert werden. Jeder Ort hat Vor- und Nachteile, wichtig ist jedoch, sich vor dem Beginn des Projektes über den späteren Speicherort klar zu sein und die Erstellung dann schon passend dafür zu starten. Eine spätere „Verschiebung“ ist technisch zwar möglich, erzeugt aber unnötig hohen Aufwand und birgt versteckte Fehlerquellen!

Hauptort wählen:

4.4.1 Makros werden im Dokument gespeichert.

Dieser Weg ist immer (und eigentlich auch ausschließlich dann) sinnvoll, wenn die Makrofunktionen ausschließlich an ein (das aktuelle) Dokument gebunden sind und sie dort auch absehbar verbleiben. Das Dokument selbst ist kein Vorlagendokument, es wird also nicht „vermehrt“, sondern ist ein „Arbeitsdokument“. Der Prozess ist abgeschlossen, das Dokument selbst kann auf einem Netzlaufwerk oder auch lokal abgelegt sein, es muss weder synchronisiert noch geteilt bearbeitet werden.

Beispiele für typische Dokumenten-Makros:

MAK_110 – Bergheim: Ein Abrechnungsmakro für eine Berghütte. Das Dokument besitzt eine „kleine Datenbank'“ (eine Tabelle), ansonsten dient es zur Berechnung und zum Druck der Abrechnungsdaten für die Hüttenbelegung.

MAK_057 – Verjährungsberechnung: Ein Dokument zur Berechnung und zum Ausdruck der Verjährung von Bußgeldern. Eine Speicherung und Archivierung eines Vorganges ist nicht vorgesehen.

Dokumentenmakros sollten immer dann vermieden werden, wenn die Gefahr besteht, dass die Dokumente „vermehrt“ werden, also entweder als Vorlagendokumente dienen oder aber durch Kopie und Speichern unter neuem Namen beliebige Arbeitskopien entstehen. In all diesen Fällen sind Pflegearbeiten am Makro nahezu unmöglich, es werden also in kürzester Zeit unterschiedliche Versionen des Makros nebeneinander existieren und Versionsverwirrungen

stiften. Zudem wird durch die Redundanz des Codes in unterschiedlichen Dateien Speicherplatz unnötig verbraucht.

Werden Makros im Dokument gespeichert, so dürfen sie nicht in der „Standard“-Bibliothek gespeichert werden. Eine solche Bibliothek gibt es nämlich in allen Dokumenten und bei (gewünschten) Zusammenführungen würden diese regelmäßig überschrieben werden. Zudem lässt sich die Standard-Bibliothek nicht exportieren und manuell durch „importieren“ wieder einfügen. Die Standard-Bibliothek ist also per se tabu für das Speichern von Dokumentenmakros, lediglich für zwei fest definierte Ausnahmen ist sie zu verwenden:

1. Falls die Makrobibliothek im Dokument verschlüsselt werden sollte (entsprechend den LHM-Makrorichtlinien), so wird das Modul „_Info“ zusätzlich in die Standard-Bibliothek kopiert, jedoch ohne die Variablen-Deklaration. Somit verbleiben Lizenz- und Projektinformationen im Klartext erhalten, auch ohne Kenntnis des Passwortes.
2. Handelt es sich um ein Calc-Dokument und werden dort benutzerdefinierte Funktionen verwendet (auch diese sind ja „nur“ Makros), so müssen diese in der Standard-Bibliothek abgelegt werden. Nur die Standard-Bibliothek wird automatisch mit dem Dokument vorgeladen – und nur dann funktionieren die benutzerdefinierten Funktionen.

4.4.2 Sonstige Speicherorte (Meine Makros)

Werden Makros nicht in Dokumenten gespeichert, so können sie auch direkt im Programm abgelegt werden – und stehen dann allen Modulen und Dokumenten zur Verfügung. Der übliche Mechanismus zum Installieren und zum Verwalten ist in diesem Fall das Verwenden von Extensions.

Extensions sind ebenfalls Zip-Archive, heute in der Regel mit der Dateierweiterung „*.oxt“, die vom Programm intern gespeichert und verwaltet werden (Extension-Manager). Damit dies problemlos funktioniert, müssen Extensions eine Datei namens description.xml beinhalten, in der die wichtigsten Parameter definiert werden. Diese Datei wird vom Extension-Manager ausgewertet und in einer internen Datenbank entsprechend abgelegt. Alle weiteren Aktivitäten müssen dann allerdings auch über den Extension-Manager (oder über das interne Programm unopkg(.exe)) abgewickelt werden – sonst sind die Inhalte der Datenbank und die tatsächlichen Verhältnisse unterschiedlich und das Programm insgesamt fehlerhaft.

Extensions können an zwei verschiedenen Orten gespeichert werden:

„**Meine Makros**“ – das heißt im Benutzerprofil des/der aktiven Benutzers/Benutzerin. Dies ist der übliche Speicherort. Erweiterungen bleiben dort erhalten – auch wenn das Programm selbst upgedatet wird. Sie gehen jedoch verloren, wenn das Benutzerprofil manuell gelöscht wird.

Intern werden die Makros dabei an folgender Stelle gespeichert:

<Benutzerprofil>/OpenOffice/3/user/uno_packages/cache/uno_packages/<1234.tmp_>/<extensionname>/

Das <Benutzerprofil> bedeutet unter Linux das „home“-Verzeichnis, unter Windows ist dies (z.B. Windows 7) C:\Benutzer\<Benutzername>\AppData\Roaming\.

Unterhalb des „cache“-Verzeichnisses wird ein Verzeichnis mit einem eindeutigen Zufallsnamen angelegt – hier dargestellt als <1234.tmp_> – es ist immer eine Zahlen/Buchstabenkombination mit der Erweiterung *.tmp_“. Darunter liegt dann das Verzeichnis mit dem Namen des Extension-Files (also zum Beispiel „MeineExtension.oxt“) – und darin dann alle Dateien der Extension.

Die Verwaltung aller Einzeldateneinträge übernimmt die Datenbank unter

<Benutzerprofil>/OpenOffice/3/user/uno_packages/cache/registry/,

deren Einträge aber nicht manuell geändert werden können oder geändert werden sollten!

Alle Extensions werden später direkt im entsprechenden Makro-Bereich mit aufgelistet, also erscheinen zum Beispiel Extensions in Basic unter der Auswahl Extras/Makros/Makros verwalten/OpenOffice.org Basic... → unter Meine Makros.

OpenOffice.org Makros

Neben der Möglichkeit, die Extensions im Benutzerprofil zu installieren, bietet unopkg auch die Möglichkeit, Extensions im Programm-Verzeichnis zu platzieren. Dazu wird die Installation auf der Kommandozeile mit Aufruf des Programms unopkg sowie dem Schalter „--shared“ durchgeführt. Details dazu lassen sich mit dem Parameter „- help“ und dem Aufruf des Programms auf der Konsole ausgeben.

Mit „shared“ installierte Programme werden nicht im Benutzerprofil abgelegt, sondern in der Programminstallation (auch dort im Verzeichnis share/uno_packages/ – sie stehen dann allerdings allen Benutzern/Benutzerinnen direkt zur Verfügung.

Manuelle Makro-Installation

Neben der immer vorzuziehenden Möglichkeit, Makros per Extension zu verteilen und zu installieren, besteht auch die Möglichkeit der manuellen Installation. „Manuell“ bedeutet in diesem Fall das Kopieren der (Makro-)Verzeichnisse in die jeweiligen Positionen.

Im Grunde ist das nichts anderes, als das, was passiert, wenn man eine neue Bibliothek (und darin Module) im Verwaltungsdialog anlegt – nur, dass jetzt die interne Verwaltung manuell erfolgen muss.

Dieser Weg wird ganz bewusst nicht empfohlen, da die Fehlermöglichkeiten doch sehr hoch sind – der Vollständigkeit und des Verständnisses der Strukturen wegen seien dennoch diese Bemerkungen erlaubt:

Der Speicherort für direkt erzeugte Bibliotheken und Makros ist im User-Profil der Unterordner „Basic“. Dieser besitzt mindestens ein Unterverzeichnis „Standard“ (die Standard-Bibliothek). Jede weitere Bibliothek wird durch ein Verzeichnis mit dem Namen der Bibliothek repräsentiert. Innerhalb des Verzeichnisses befinden sich alle Module als eigenständige XML-Dateien, die Namen der Module entsprechen den Modulnamen bzw. den Dialognamen. Bei allen Dateien handelt es sich um einfache XML-Dateien (UTF-8), die zur Not mit einem entsprechenden Editor geöffnet und bearbeitet werden können. Sinn macht dies beispielsweise bei der Übernahme eines Dialog-Moduls, dessen Programm-Aufrufe nun aber auf andere Pfade gebunden werden sollen – dies geht am schnellsten mit Hilfe der „Suchen und Ersetzen“-Funktion eines normalen Editors.

Allerdings ist es mit dem Kopieren und Einfügen der Makro-Verzeichnisse nicht getan. Damit OpenOffice.org diese später auch findet und entsprechend anzeigen kann (und damit arbeiten kann), müssen sie auch korrekt in der Verwaltungsstruktur aller Makros eingebunden werden. Diese Aufgabe übernimmt im Fall der „Nicht-Extensions“ keine Datenbank, sondern XML-Steuerdateien.

4.4.3 Die wichtigen Skripte : script.xlb/xlc und dialog.xlb/xlc

In jedem Makro-Verzeichnis (nur Basic) finden sich zwei Verwaltungsscripte (XML-Dateien):

Auf der Ebene des einzelnen Makros (Bibliothek) im Root-Pfad der Bibliothek: script.xlb sowie dialog.xlb (gilt auch für Dokumentenmakros!)

Auf der Ebene der Position (also „Meine Makros“ - Benutzerprofil im Hauptordner Basic): script.xlc und dialog.xlc

Hierbei handelt es sich um die Verwaltungsdateien. In den script-Dateien sind alle Code-Module aufgeführt, in den dialog-Dateien alle Dialog-Module.

Eine **script.xlb** Datei (hier des Makros MAK_045 – Bibliotheksname) sieht beispielsweise wie folgt aus:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE library:library PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN" "library.dtd">
3  <library:library xmlns:library="http://openoffice.org/2000/library" library:name="MAK_045"
   library:readonly="false" library:passwordprotected="false">
4   <library:element library:name="MAK045_Tools1"/>
5   <library:element library:name="Module1"/>
6   <library:element library:name="MAK045_RPCTOOls"/>
7  </library:library>
```

Abbildung 4.1: Die Datei script.xlb

Das Makro beinhaltet drei (Script-)Module mit den Namen „MAK045_Tools1“, „Module1“ sowie „MAK045_RPCTOOls“.

Nur diese drei werden später gefunden und können angezeigt und ausgeführt werden – sollte es noch ein viertes geben (z.B. mit Namen „Test“), und wird dieses Modul hier nicht in der Liste

korrekt aufgeführt, so existiert das Modul für OpenOffice.org nicht. Es wird auch nicht angezeigt, noch sind Routinen daraus aufrufbar.

Alle Bibliotheken wiederum sind in der Datei **script.xlc** aufgelistet (im Root-Ordner!). Diese könnte wie folgt aussehen:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE library:libraries PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN" "libraries.dtd">
3  <library:libraries xmlns:library="http://openoffice.org/2000/library"
   xmlns:xlink="http://www.w3.org/1999/xlink">
4   <library:library library:name="DBI_QSTOOL" xlink:href="$(USER)/basic/DBI_QSTOOL/script.xlb/"
   xlink:type="simple" library:link="false"/>
5   <library:library library:name="XrayDyn" xlink:href="$(USER)/basic/XrayDyn/script.xlb/"
   xlink:type="simple" library:link="false"/>
6   <library:library library:name="MAK_045" xlink:href="$(USER)/basic/MAK_045/script.xlb/"
   xlink:type="simple" library:link="false"/>
7   <library:library library:name="Standard" xlink:href="$(USER)/basic/Standard/script.xlb/"
   xlink:type="simple" library:link="false"/>
8   <library:library library:name="XrayTool" xlink:href="$(USER)/basic/XrayTool/script.xlb/"
   xlink:type="simple" library:link="false"/>
9   <library:library library:name="MAK_064_FDB" xlink:href="$(USER)/basic/MAK_064_FDB/script.xlb/"
   xlink:type="simple" library:link="false"/>
10  <library:library library:name="FR_SJAB" xlink:href="$(USER)/basic/FR_SJAB/script.xlb/"
   xlink:type="simple" library:link="false"/>
11 </library:libraries>
```

Verwiesen wird auf die script.xlb-Datei der jeweiligen Bibliothek – und nur die hier aufgeführten Bibliotheken werden gefunden.

Ähnlich aufgebaut sind auch die dialog-Dateien – eben dann mit Verweisen auf die dialog.xlb bzw. auf die Dialog-Module.

Fehler in diesen Dateien führen zumeist zu Basic-Laufzeit-Fehlern (Bibliothek/Routine nicht gefunden). Im Einzelfall können diese Dateien manuell repariert und ergänzt werden.

Falls jemand manuell Bibliotheken integriert, müssen die entsprechenden Verwaltungsdateien unbedingt entsprechend ergänzt werden, sonst funktioniert das komplette System nicht.

Wie gesagt – ich rate davon ab, manuell Installationen vorzunehmen. Nutzen Sie Extensions – da müssen Sie keine manuellen Eingriffe vornehmen.

4.5 Module und Bibliotheken dynamisch erzeugen

Mit Makros kann man auch Bibliotheken und/oder Module direkt dynamisch erzeugen. Solche Möglichkeiten sind immer dann sinnvoll, wenn man individuelle Einstellungen zwischenspeichern möchte und und dafür nicht auf (Text-)Konfigurationsdateien zurückgreifen

möchte. Vorstellbare Anwendungen sind zum Beispiel die letzten Einträge in Listboxen, Umgebungsparameter wie Pfade etc, oder auch individuelle Farbeinstellungen oder ähnliches.

Dynamische Module oder Bibliotheken sind in der Regel nur für Extensions geeignet!

Dabei sind folgende Punkte zu unterscheiden:

Bibliotheken bleiben auch nach einem Update-Mechanismus erhalten (es wird ja nur die ursprüngliche Bibliothek ersetzt), allerdings verbleiben diese dann auch „als Leichen“ im Benutzerprofil, wenn eine dazugehörige Extension endgültig gelöscht wurde.

Module werden beim Update mit überschrieben, die Einstellungen gehen verloren. Module in verschlüsselten Bibliotheken lassen sich nicht erzeugen – wenn also die Bibliothek verschlüsselt werden soll, müssen andere Zwischenspeicher (zum Beispiel „SimpleConfig“) gewählt werden.

Der Vorteil, dynamische Module anzulegen, ist einmal der sehr einfache Weg (einfacher ASCII-Text) und zum zweiten die sehr einfache Aufrufmöglichkeit. Ein dynamisches Modul wird komplett geschrieben und enthält eine oder mehrere Prozeduren. Die können dann genauso aufgerufen werden wie alle anderen Prozeduren auch. Das Schreiben erfolgt als reiner ASCII-Text, zwar werden die Module später als XML-Files abgespeichert, den Umbau übernimmt aber OpenOffice intern. Auch die Anpassung der Skripte (script.xlb und script.xlc) übernimmt OOo intern, darüber muss man sich keine Gedanken machen.

4.5.1 Eine Bibliothek per Code erzeugen

Eine Bibliothek (im Speicherbereich „Meine Makros“) kann per Skript sehr einfach erzeugt werden. Normalerweise prüft man vorher, ob die Bibliothek bereits besteht – wenn ja, dann löscht man sie zunächst. Alle Module sind benannte Objekte der Bibliothek und können ebenfalls einfach erzeugt oder gelöscht werden. Ein Modul wiederum besitzt als Eigenschaft den Text, der somit als ASCII-Text übergeben werden kann. Wichtig: Das Übergeben eines Modulnamens und des Inhaltstextes überschreibt ohne Rückfrage einen bereits vorhandenen Text!

Das folgende Beispiel zeigt die Erzeugung eines dynamischen Moduls „MAK064_dyn“ in der Bibliothek „MAK064_FDBdyn“ – es wird also lediglich ein Modul erzeugt! Der zu übergebende Modultext wird ebenfalls vorher zusammengebaut, die benutzten Variablen im nicht dargestellten Code-Teil per Dialog erfragt und auf Plausibilität geprüft:

```
sub MAK064_OptionenSichern
  dim sTxt as string  'Textplatzhalter

<...>
  REM jetzt Daten eintragen
  sTxt = "REM Bitte hier nichts ändern."  & chr(10) &_
         "REM Modul wird dynamisch erzeugt und Bibliothek regelmäßig gelöscht!" & chr(10) & _
         "REM letzte Änderung: " & format(now(), "dd.mm.yyyy hh:mm") & chr(10) & chr(10) &_
                "Sub MAK064_init" & chr(10) &_
```

```
            "   MAK064_Musterdatei = """ & MAK064_Musterdatei & """"  & chr(10) &_
            "   MAK064_Importpfad = """ & MAK064_Importpfad & """"  & chr(10) &_
            "   MAK064_Exportpfad = """ & MAK064_Exportpfad & """"  & chr(10) &_
            "   MAK064_Passwort = """ & MAK064_Passwort & """"  & chr(10) &_
            "   MAK064_Plauspfad = """ & MAK064_Plauspfad & """"  & chr(10) &_
            "   MAK064_NeuDatPfad = """ & MAK064_NeuDatPfad & """"  & chr(10) &_
            "   MAK064_CSVPfad = """ & MAK064_CSVPfad & """"  & chr(10) &_
            "   MAK064_Zeichensatz = """ & MAK064_Zeichensatz & """"  & chr(10) &_
            "   MAK064_DatTrenner = """ & MAK064_DatTrenner & """"  & chr(10) &_
            "   MAK064_CSVDatErw = """ & MAK064_CSVDatErw & """"  & chr(10) &_
            "   MAK064_TABNAME = """ & MAK064_TABNAME & """" & chr(10) &_
            "   MAK064_TABNAMEBET = """ & MAK064_TABNAMEBET & """" & chr(10) &_
        "End Sub"

 if globalScope.BasicLibraries.hasByName("MAK064_FDBdyn") then _
    globalScope.BasicLibraries.removeLibrary("MAK064_FDBdyn")
  GlobalScope.BasicLibraries.storeLibraries()
  globalScope.BasicLibraries.createLibrary("MAK064_FDBdyn")
  GlobalScope.BasicLibraries.getByName("MAK064_FDBdyn").insertByName("MAK064_dyn", sTxt)
  GlobalScope.BasicLibraries.storeLibraries()
  GlobalScope.BasicLibraries.loadLibrary("MAK064_FDBdyn")
end sub
```

Wichtig sind die Aufrufe des Befehls storeLibraries() – nur die stellen sicher, dass der aktuelle Stand tatsächlich gesichert und aus dem Arbeitsspeicher auf die Festplatte zurückgeschrieben wird.

Im Prinzip lassen sich so natürlich auch weitere Module dynamisch erzeugen und ändern.

4.5.2 Ein Modul per Code erzeugen

So wie eine Bibliothek erzeugt werden kann, lässt sich auch ein Modul der aktuell genutzten Bibliothek neu schreiben. Auch hier ist zu beachten, dass immer das komplette Modul überschrieben wird – nicht nur der zu ändernde Part. Typischerweise besitzen dynamische Module lediglich eine Routine (sub oder function) – mehr wären allerdings auch denkbar.

Ein Beispiel, ein Modul dynamisch zu beschreiben, findet sich im folgenden Code:

```
'/** MAK017_SaveNewPref()
'*****************************************
'* @Kurztext: Sichern der Präferenzen
'* Das Makro schreibt die neuen Voreinstellungen dynamisch zurück
'* und sichert sie so.
'*
'*****************************************
'*/
sub MAK017_SaveNewPref()
  dim aBas(15)  '16 Zeilen
  dim sBas as string

  aBas(0) = "REM dieser Teil wird dynamisch geändert!"
  aBas(1) = "REM letzte Änderung:  " & format(now(),"dd.mm.yyyy hh:mm")
  aBas(2) = ""
  aBas(3) = "Sub MAK017_InitAL"
  aBas(4) = "  if NOT GlobalScope.BasicLibraries.isLibraryLoaded(""Tools"") then
```

```
GlobalScope.BasicLibraries.loadLibrary(""Tools"")"
  aBas(5) = "  oSFA = createUnoService(""com.sun.star.ucb.SimpleFileAccess"")"
  aBas(6) = "  DialogLibraries.loadLibrary(""MAK017_AL"")"
  aBas(7) = "  sALakt = """ & sALakt & """"
  aBas(8) = "  aALs = array(""" &  join(aAls, """,""") & """)"
  aBas(9) = "  aModTB = array(""com.sun.star.text.TextDocument"")"
  aBas(10) = "  Monate = array("""", ""Januar"",""Februar"", ""März"", ""April"", ""Mai"",
""Juni"",_"
  aBas(11) = "                 ""Juli"", ""August"", ""September"", ""Oktober"", ""November"",
""Dezember"")"
  aBas(12) = "  MonateKurz = array("""", ""Jan"", ""Feb"", ""Mar"", ""Apr"", ""Mai"", ""Jun"",
""Jul"", ""Aug"",_"
  aBas(13) = "                     ""Sep"", ""Okt"", ""Nov"", ""Dez"")"
  aBas(14) = "  bToolbarFlag  = " & bToolbarFlag
  aBas(15) = "End Sub"
  sBas = join(aBas(), chr(10))   'Gesamttext erzeugen
  REM Modultext zurückschreiben
  GlobalScope.BasicLibraries.getByName("MAK017_AL").replaceByName("MAK017_dyn", sBas)

end sub
```

Achtung!

Die Anzeige in der IDE wird durch das Erneuern eines Moduls nicht geändert! Hier verbleibt der ursprüngliche Code, auch wenn dieser gar nicht mehr existent ist. Lassen Sie sich also durch die Anzeige nicht täuschen. Lediglich das komplette Schließen der IDE und das erneute Öffnen bewirkt auch ein Neuladen des angezeigten Codes. Wird der Code jedoch ausgeführt, so greift OOo auf den gespeicherten Code zurück – nicht auf die in der IDE angezeigten Code-Teile.

Aufruf des Code:

Der Aufruf des dynamischen Code-Teils erfolgt dann einfach mit Hilfe des Funktionsnamens, hier also mit „ MAK017_InitAL“ – und das war es dann auch schon.

Noch eine Besonderheit: Der Inhalt eines Basic-Moduls besitzt die Eigenschaft „string“, diese repräsentiert den kompletten Text des Moduls. Allerdings kann diese Eigenschaft entsprechend der Basic-Variablen-Definition lediglich 64 K an Zeichen aufnehmen, das Modul selbst könnte aber weitaus mehr Zeichen besitzen. In diesem Fall ist die Eigenschaft leer – die entsprechenden einfachen Methoden versagen. Um nun an den Inhalt eines Moduls heranzukommen, ist ein Input-Stream zu erzeugen und dieser dann entsprechend auszuwerten.

4.6 Variable und Parameter extern speichern

Statt Parameter und Variable dynamisch in Modulen oder in eigenen Bibliotheken zu speichern, können diese auch als Textdateien ausgelagert werden. Dies ist im übrigen der bevorzugte Weg der LHM. Dafür wurde eigens die Extension „SimpleConfig“ erzeugt, welche die Aufgaben übernimmt, Parameter und Variable aus einer Datei („.makro.conf“) auszulesen und eventuell auch zu schreiben.

4.6.1 SimpleConfig

SimpleConfig ist als Extension verfügbar und Bestandteil des LiMux-Desktops. Dennoch muss vor der Verwendung der dort vorhandenen Funktionen zunächst das Vorhandensein der Extension geprüft werden. Anschließend können Parameter ausgelesen und auch geschrieben werden. Der folgende Code-Teil zeigt exemplarisch, wie Parameter mit Hilfe von SimpleConfig ausgelesen und weiterverarbeitet werden können. SimpleConfig ist die erste Wahl zum Speichern und Verwalten von änderbaren Variablen.

```
'/** MAK133_InitParameter
'*************************************************************************.
'* @kurztext liest die Parameter aus der Makro.conf Datei
'* liest die Parameter (Pfade und Kürzel) aus der makro.conf Datei und
'* übergibt sie an die globalen Variablen.
'*
'* @return  flag as boolean  true, wenn alles OK, sonst false
'*************************************************************************
'*/
function  MAK133_InitParameter
  dim ParListe()     'Liste der Parameter für SimpleConfig
  dim ParListeWerte()  'Liste der Werte für die Parameter
  dim sPfad as string  'Platzhalter Pfadnamen
  dim sFPfad as string  'Platzhalter Unterpfade
  dim aFilter()      'Filterpfade
  dim sBearb as string, sRef as string
  dim aBearb(), aBe(), aRef()
  dim i%

  REM prüfen, ob SimpleConfig installiert ist
  If GlobalScope.BasicLibraries.hasByName("SimpleConfig") Then
      GlobalScope.BasicLibraries.loadLibrary("SimpleConfig")
  Else
     msgbox ("Die Extension ""SimpleConfig"" ist offensichtlich auf diesem Arbeitsplatz" &
chr(13) & _
             "nicht oder nicht korrekt installiert. Sie wird aber für die aufgerufene" &
chr(13) & _
             "Funktion benötigt. Das Makro wird jetzt abgebrochen." & chr(13) & _
             "Bitte informieren Sie Ihren Systemadministrator.", 16, "SimpleConfig Fehler!")
     MAK133_InitParameter = false
     exit function  ' oder exit Function  - je nach Anwendungszweck
  End If

  REM Toolsbibliothek laden
  if NOT GlobalScope.BasicLibraries.isLibraryLoaded("Tools") then _
        GlobalScope.BasicLibraries.loadLibrary("Tools")

  REM SimpleConf - alle Variablen leeren
  sDBServer = ""          '0
  sVorlPfad = ""          '1   'Vorlagenpfad
  sStoBericht = ""        '2   'Vorlage Karteikarte

  REM Parameter zusammenstellen, die über SimpleConfig ausgelesen werden
  ParListe = array("MAK133/DBSERVER", "MAK133/VORLAGENPFAD", "MAK133/VORLAGEKARTEI")

  ParListeWerte = SimpleConfig.Helper.getSimpleConfigVar(ParListe, 2)
```

```
  REM Prüfen, ob überhaupt etwas zurück kommt
  if isEmpty( ParListeWerte()) then
    MAK133_InitParameter = false
    exit function  '
  Else   'ist etwas vorhanden!
    sDBServer = ParListeWerte(0)
    sVorlPfad = CheckPfadKomplett(ParListeWerte(1))
    sStoBericht = ParListeWerte(2)
  end if

  MAK133_InitParameter = true
end function
```

Fehlen einzelne Parameter oder sind diese nicht korrekt in der .makro.conf-Datei eingebunden, so liefert SimpleConfig bereits entsprechende Fehlermeldungen und keine Daten zurück (auch nicht die korrekten!) – es muss also keine weitere Aktion selbst programmiert werden (außer dem Programmende). Lediglich die Überprüfung, ob SimpleConfig überhaupt vorhanden ist, obliegt dem/der Programmierer/in.

4.6.2 Text-Steuerdateien

Wird SimpleConfig – aus welchen Gründen auch immer – nicht verwendet, und sollen Parameter in eigenen Steuerdateien ausgelagert werden, muss man sich immer zunächst bewusst werden, wo diese Dateien abgelegt werden können. Da die Möglichkeit der Installation vielfältig ist, genauso wie die Struktur des verwendeten Rechners und des Dateisystems, wäre es sinnvoll, die Dateien dort abzulegen, wo auch das Makro selbst liegt – also entweder in der Extension selbst oder im Dokument. Beides ist möglich – Dokumente sind XML-Zip-Archive und können ohne weiteres zusätzliche Dateien aufnehmen, auch Extensions werden auf dem System gespeichert und können zusätzliche Dateien aufnehmen.

Allerdings ist es nicht immer einfach, den internen Speicherpfad zu identifizieren. Hierzu kann man den Service „PackageInformationProvider“ verwenden. Dieser liefert den Pfad zu einer bekannten Extension – identifiziert durch den eindeutigen Identifier-String in der Description-XML.

Das Beispiel zeigt den Pfad zu der installierten Extension mit dem eindeutigen Identifier-String:

```
sub GetExtensionPath
  dim oPackageInfoProvider as variant
  dim sService as string, sExtensionIdentifier as string, sPackageLocation as string

  sService = "com.sun.star.deployment.PackageInformationProvider"
  sExtensionIdentifier = "de.muenchen.allg.dbi.mak_131_lrl"

  oPackageInfoProvider = GetDefaultContext.getValueByName("/singletons/" & sService)

  sPackageLocation = oPackageInfoProvider.getPackageLocation(sExtensionIdentifier)

  msgbox sPackageLocation
```

```
end sub
```

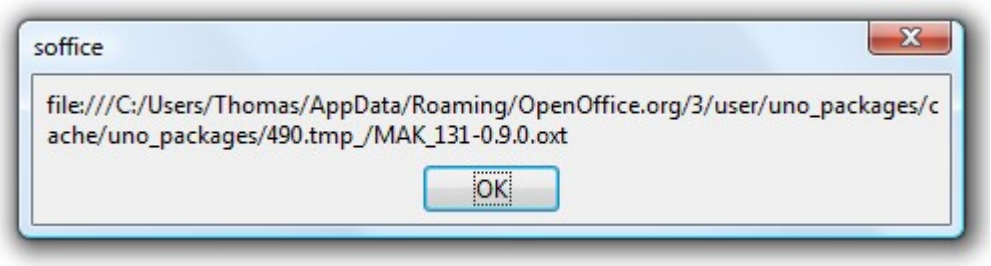


Abbildung 4.2: Pfad zu einer Extension - im Bereich Benutzerprofil

Das zurückgelieferte Verzeichnis kann nun direkt verwendet werden oder man erzeugt zunächst noch ein eigenes Unterverzeichnis und speichert dann dort die Konfigurationsdateien.

Zum Schreiben und Lesen von einfachen Textdateien siehe auch Kapitel 3.4.

5 Office-Arbeitsumgebung

OpenOffice.org ist ein großes Programm – und wird gestartet, sobald ein Programmmodul aufgerufen wird. Dies kann durch das Öffnen einer OOo-Datei erfolgen, aber auch durch den Start des „Schnellstarters“ oder des „Startzentrums“. In jedem Fall wird der Programm-Code in den Hauptspeicher geladen und der „StarDesktop“ gestartet. Für den/die Nutzer/in selbst ist dies zunächst nicht sichtbar – der StarDesktop besitzt keinen sichtbaren „Rahmen“ oder „Fenster“, lediglich der soffice-Prozess ist im Taskmanager sichtbar.

5.1 Der StarDesktop

Der StarDesktop ist ein Relikt aus den Anfängen des Programmpaketes (StarOffice 5.x), das ist ungefähr 12 bis 15 Jahre her. Damals besaß das Programm ein sogenanntes Rahmenfenster, den „StarDesktop“, also eine eigenständige Benutzeroberfläche, in der sich dann alle weiteren Module als Tochterprozesse öffneten. Die Struktur ist geblieben, nur die Fenster sind jetzt eigenständiger. Die Rahmenapplikation (der StarDesktop) ist zwar immer noch der Hauptprozess, doch besitzt dieser keine sichtbare Oberfläche mehr. Die Begrenzungen des StarDesktops sind jetzt identisch mit den Begrenzungen des physikalischen Desktops – gesteuert durch das X-System des benutzten Rechners.

Für das Verständnis im Makro-Alltag ist das Hintergrundwissen dennoch unverzichtbar.

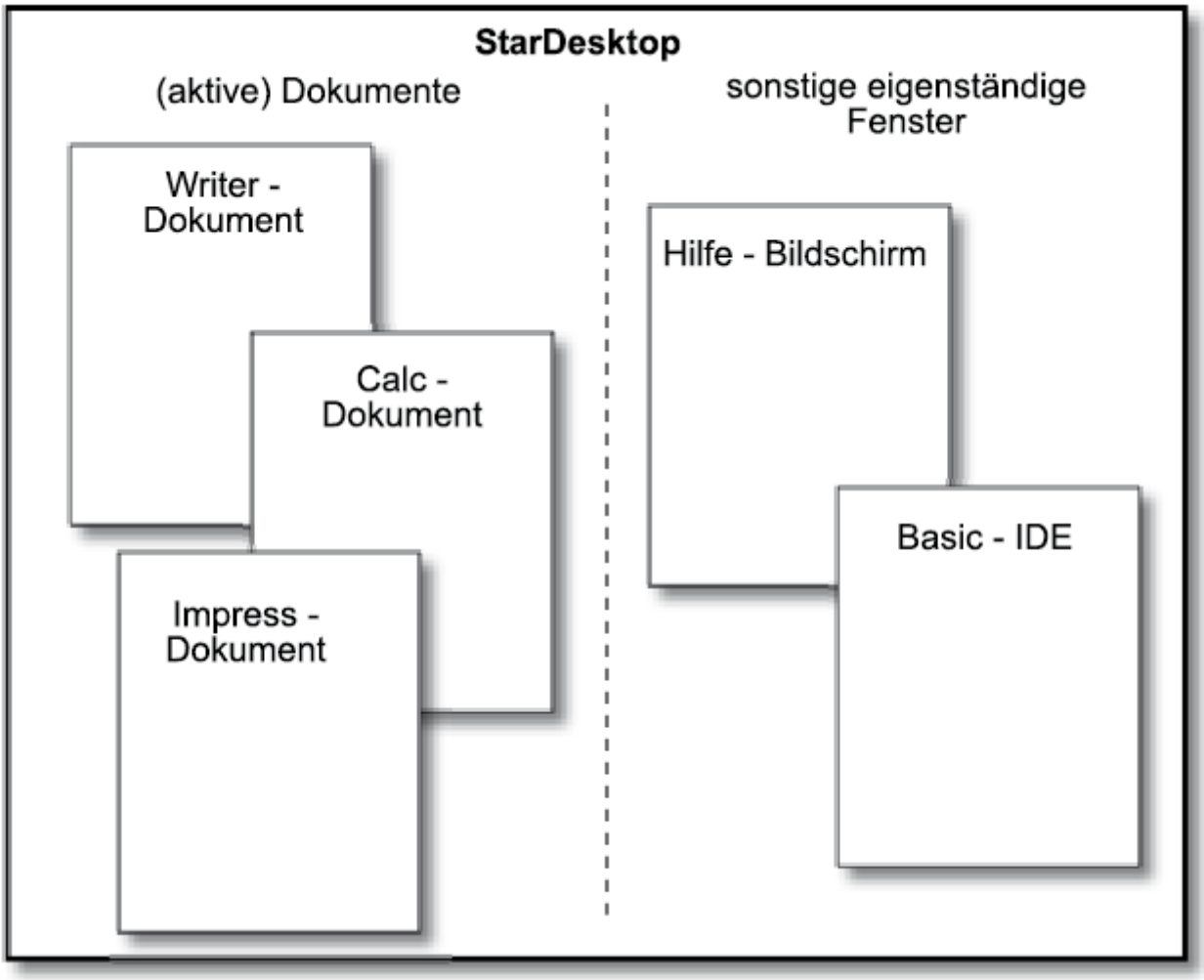


Abbildung 5.1: Der StarDesktop

Der StarDesktop bildet also den Rahmen, die oberste Instanz aller OOo-Bildschirmapplikationen.

In diesem Zusammenhang sind folgende Services wichtig:

- Der `com.sun.star.frame.Desktop-Service` liefert die Funktionen des übergeordneten Desktops. Hierzu zählen zum Beispiel die Möglichkeiten, Dokumente zu erzeugen, zu öffnen oder zu importieren.
- Möchte man auf die einzelnen Dokumente direkt zugreifen, so liefert das Modul `com.sun.star.document` die entsprechenden Services; so sind zum Beispiel die Funktionen zum Speichern, Exportieren oder zum Drucken von Dokumenten im Service `com.sun.star.document.OfficeDocument` zu finden.

Es bleibt allerdings eine wichtige Einschränkung: Über die Frames und den Desktop erreichen Sie alle geöffneten Applikationen und Fenster, die ja alle eigenständige Frames (Fenster) bilden innerhalb des StarDesktops. Das ist aber oft gar nicht erwünscht. Beispiel: Die Basic-IDE stellt ein eigenständiges Fenster (und somit auch einen eigenständigen Frame) auf dem Desktop dar.

Wird in diesem gerade ein Programm entwickelt (und getestet), so soll der Zugriff auf die Objekte in dem aktiven Dokument (Writer, Calc etc.) erfolgen und nicht im aktuell aktiven Frame (der Basic-IDE).

Beispiel:

```
oDoc = StarDesktop.currentComponent
```

liefert das Objekt der aktuellen (aktiven) Anwendung zurück. Das kann ein Calc-Dokument sein, wenn das Makro aus eben diesem Calc-Dokument aus aufgerufen wurde, oder es ist eben die Basic-IDE, wenn das Makro dort getestet wird. Dies kann insbesondere bei Entwicklungen zu großen Verwirrungen führen.

Um diese Problematik zu umgehen beziehungsweise zu minimieren, wurde neben der eingebauten globalen Variablen „StarDesktop“, die das Desktop-Objekt repräsentiert, auch eine weitere globale Variable zur Verfügung gestellt, die das aktive Dokument beinhaltet – ThisComponent. Mit

```
oDoc = ThisComponent
```

erhalten Sie nun immer das aktive Dokument (also entweder ein Writer-, ein Calc-, Impress- oder Draw-Dokument); jetzt ist es auch möglich, aus der Basic-IDE heraus direkt auf das Dokument zuzugreifen. Seit der Programmversion 3.1 gibt es hier allerdings eine Einschränkung: Ab diesem Zeitpunkt können Makros auch direkt in Base-Dokumenten gespeichert werden, das „ThisComponent“ kann in diesem Fall entweder auf das Base-Dokument selbst oder auf ein davon abgeleitetes Formular (meist Writer Dokument) verweisen. Um diesen Widerspruch zu vermeiden, wurde zusätzlich die globale Variable „ThisDatabaseDocument“ eingeführt, die immer auf die Base-Datei verweist, während „ThisComponent“ dann immer auf das gerade aktive Dokument (Formular) deutet.

5.1.1 Dokumente identifizieren

Wenn man den StarDesktop als übergeordneten Frame ansieht, der alle anderen Applikationen (Frames) enthält, so ist es natürlich möglich, alle diese anzusprechen und zu analysieren. Hierbei helfen verschiedene Methoden.

Methode	Beschreibung
GetImplementationName()	Liefert einen String,der den internen Namen der Applikation darstellt
GetSupportedServiceNames()	Liefert ein Array (Strings), das die Namen der unterstützten Services repräsentiert
SupportsService(sService)	Liefert „True“, wenn das Objekt den als String übergebenen Service-Namen unterstützt

Mit Hilfe dieser Methoden kann eine Applikation (Frame) eindeutig identifiziert werden. Da jede Anwendung mindestens einen Service unterstützt und es meist einen gibt, der hinreichend eindeutig ist, lassen sich die Anwendungen über SupportsService() eindeutig identifizieren.

Der erste Schritt ist also zu überprüfen, welche Services überhaupt unterstützt werden. Das kann für das aktuelle Dokument beispielsweise wie folgt aussehen:

```
Sub GetServiceNames(optional oDoc)
  If isMissing(oDoc) then oDoc = thisComponent
  MsgBox Join(oDoc.getSupportedServiceNames())
End Sub
```

Ist das aktuelle Dokument ein Calc-Dokument, so erhält man folgende Ausgabe:

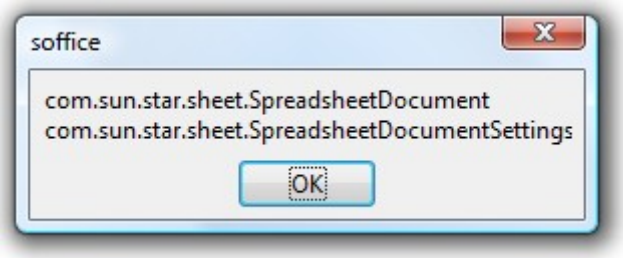


Abbildung 5.2: SupportedServices eines Kalkulationsdokumentes

Der Service `com.sun.star.sheet.SpreadsheetDocument` ist somit eindeutig und identifiziert ein Calc-Dokument. Ähnliches gilt für andere Anwendungen:

Dokumenttyp (Anwendung)	Eindeutiger Service
Textdokument Writer	com.sun.star.text.TextDocument
Tabellendokument Calc	com.sun.star.sheet.SpreadsheetDocument
Zeichnungsdokument Draw	com.sun.star.drawing.DrawDocument
Präsentation Impress	com.sun.star.presentation.PresentationDocument
Formeldokument Math	com.sun.star.formular.FormularProperties
Datenbank	com.sun.star.sdb.DatabaseDocument
Die Basic-IDE	com.sun.star.script.BasicIDE

Hiermit lässt sich eindeutig feststellen, um welches Dokument es sich handelt. Eine solche Überprüfung sollte immer dann erfolgen, wenn der Makro-Code nicht universell, sondern nur für einen speziellen Typ einsetzbar ist. Im folgenden wird überprüft, ob es sich um eine Präsentation handelt (Impress), anderenfalls erfolgt eine Fehlermeldung:

```
Sub EinWichtigesMakroFuerPraesentationen
  If NOT thisComponent.supportsService("com.sun.star.presentation.PresentationDocument") then
    MsgBox "Dieses Makro funktioniert nur mit einer Präsentation (Impress-Dokument)", 48,
"Fehler"
    Exit Sub
  End if
  REM hier folgt jetzt der restliche Code
End Sub
```

Mit Hilfe der Methode supportsService() lässt sich somit klar festlegen, für welches Modul (Dokument) das Makro überhaupt fehlerfrei läuft – und diese Prüfung muss immer am Anfang eines Makros erfolgen – insbesondere bei Extensions, die in der Regel nicht direkt aus dem Dokument aufgerufen werden!

Mit Hilfe dieses Wissens ist es selbstverständlich auch möglich, alle aktuell laufenden Applikationen zu erfassen. Hierzu nutzen Sie den StarDesktop, den übergeordneten Frame. Über die Methode getComponents() erhalten Sie ein Objekt, das eine Liste aller geöffneten Dokumente (Komponenten) enthält. Um Zugriff auf die jeweiligen Objekte zu erhalten, müssen Sie zunächst eine Nummerierung erzeugen und arbeiten diese dann in einer Schleife ab (Stichwort iterativer Zugriff).

```
Sub AlleAktivenApplikationen
  On Error resume Next
  Dim oComp as Object, oDocs as Object, oDoc as Object, s$
  oComp = StarDesktop.getComponents()
  oDocs = oComp.createEnumeration()
  Do While oDocs.hasMoreElements()
    oDoc = oDocs.nextElement()
    s = s & getDocType(oDoc) & CHR$(10)
  Loop
  MsgBox s, 0, "offene Anwendungen"
End Sub
```

Neben den in getDocType() identifizierten Applikationen gibt es immer noch zusätzliche Frames, die nicht eindeutig zuzuordnen sind, beispielsweise ein geöffnetes Hilfefenster, das zwei Frames erzeugt. Diese werden in der Funktion mit „unbekannt“ ausgegeben.

Diese Struktur ermöglicht die eindeutige Identifizierung der Dokumente; wenn aber jedes Dokument auch ein Frame ist, so haben diese auch bestimmte Eigenschaften, die ebenfalls ausgelesen werden können. Eine der Eigenschaften ist beispielsweise der Titel, also das, was in der Titelleiste jedes Fensters steht. Bei den meisten Dokumenten steht dort der Dateiname sowie die aktive Applikation. Ersatzweise kann aber auch der Titel (erreichbar über Datei/Eigenschaften) angezeigt werden – also Achtung: eventuell nicht immer eindeutig!

Abbildung 5.3: Beispiel eines Writer-Dokuments

Um genau diesen Titel zu erhalten, benötigen Sie nur eine Zeile Code:

```
MsgBox StarDesktop.getActiveFrame().title
```

Dieser Code gibt dabei immer nur den aktiven Frame-Titel aus, das heißt, wird er innerhalb der Basic-IDE ausgeführt, liefert er auch den Titel der Basic-IDE zurück, wird das Makro aber aus dem Dokument ausgeführt, dann zeigt er den Dokumententitel an.

Die Methode getFrames() liefert ein Objekt aller Frames, zu dem Sie entweder über einen iterativen Zugriff (createEnumeration()) Zugang (zu den einzelnen Frames) erhalten oder – da dieses Objekt das Interface com.sun.star.frame.XFrame unterstützt und dieses wiederum abgeleitet wird aus dem com.sun.star.container.XIndexAccess-Interface – Sie nutzen den indexbasierenden Zugriff.

Das folgende Beispiel zeigt diese Möglichkeit:

```
Sub AlleFrames
  Dim oFrames as object, oFrame as object, i%, s$
  oFrames = StarDesktop.getFrames()
  For i = 1 to oFrames.getcount()
    oFrame = oFrames.getByIndex(i-1)
    s = s & CStr(i) & " :  " & oFrame.Title & Chr$(10)
  Next
  MsgBox s
End Sub
```

Der Unterschied zu dem Code weiter oben und dem dort zugrunde liegenden Programm ist folgender: Der Titel eines Frames kann auch verändert werden und bestimmt daher nicht eindeutig die darunter liegende Anwendung. Der Check, ob ein bestimmter Service unterstützt wird, identifiziert die Anwendung (das Dokument) hingegen eindeutig.

Noch ein paar Hinweise zum Sprachgebrauch in OpenOffice.org-Basic: Mit Frames (Rahmen) sind quasi die Fenster gemeint, die auf dem Bildschirm sichtbar sind (auf der GUI-Oberfläche). Diese können eine Baumstruktur bilden (also ein Frame ist von einem anderen abhängig, von diesem abgeleitet), sie können auch unsichtbar oder verkleinert sein. Mit Komponenten (Components) sind die Anwendungen gemeint, die in dem Fenster ausgeführt werden. Zu jeder Komponente gehört auch ein Frame, Komponenten sind jedoch selten voneinander abgeleitet.

Abreißbare Menüleisten und andere eigenständige Fenster tauchen übrigens nicht in der Liste der Frames auf. Auch der Desktop an sich (der übergeordnete Frame) besitzt keine verwertbaren Eigenschaften und ist nicht direkt ansprechbar. Andererseits agiert er als alles umfassende Rahmenanwendung, so dass es durchaus Methoden gibt, die auf ihn anwendbar sind. Siehe hierzu auch das Interface com.sun.star.frame.XDesktop.

5.2 Größe und Platzierung der Module

Jedes einzelne Fenster (Frame) auf dem Desktop hat eigene Eigenschaften und kann somit auch per Makro manipuliert werden. Dies trifft insbesondere die folgenden, wichtigen Eigenschaften:

- **Visibility**: Ein Fenster kann „versteckt“ werden, das heißt, es ist zwar auf dem StarDesktop vorhanden und kann beliebig per Makro bearbeitet werden, es ist aber nicht sichtbar auf dem tatsächlichen Bildschirm. Der/Die Benutzer/in sieht es also nicht.
- **Position**: Die Position eines Fensters kann auf dem tatsächlichen Bildschirm definiert werden.
- **Größe**: Die Größe eines Fensters kann per Makro eingestellt werden.

Versteckete Fenster müssen unbedingt auch per Makro geschlossen werden, möchte man einen Datenverlust vermeiden!

Die Position und die Größe eines Moduls (also zum Beispiel von Writer, Calc oder Draw) wird intern immer global zwischengespeichert und dem nächsten Dokument des gleichen Moduls automatisch zugeordnet. Verkleinert man also ein z.B. Writer-Dokument und öffnet der/die Benutzer/in dann ein neues Writer-Dokument, so erscheint dies in der per Makro eingestellten Größe! Um das zu vermeiden, sollte die ursprüngliche Größe vor dem Schließen immer wieder hergestellt werden.

Doch nun zu den Details:

5.2.1 Visibility

Die Sichtbarkeit von Dokumenten spielt in der Praxis eine große Rolle. So sind typischerweise Dokumente nicht sichtbar, die nur zum Lesen oder Auslesen von Daten benötigt werden. Auch Dokumente, die erst zusammengebaut werden, sollten zunächst unsichtbar erzeugt werden und erst dann sichtbar geschaltet werden, wenn sie für den/die Benutzer/in fertig sind.

Wird ein Dokument geöffnet, kann die Sichtbarkeit direkt als Parameter mit übergeben werden:

```
…
sUrl = convertToURL("/home/daten/meinDatenDokument.ods")
dim arg(0) as new com.sun.star.beans.PropertyValue
arg(0).Name = "Hidden"
arg(0).value = true
oDoc = StarDesktop.loadComponentFromURL(sURL, "_blank", 0, arg())
...
```

In der Variablen oDoc ist nun das Dokumentenobjekt gespeichert und kann per API beliebig manipuliert werden – auf dem Bildschirm selbst ist jedoch nichts zu sehen. Dadurch fehlt natürlich auch die Möglichkeit für den/die Benutzer/in, dieses Dokument zu speichern oder zu schließen – das muss nun sicher durch den Code gewährleistet werden.

Bevor Sie ein solches Dokument neu öffnen, prüfen sie ebenfalls, ob das Dokument nicht bereits geöffnet ist – die Prüfung (siehe auch Kapitel 3.2.2 sowie 3.2.3) umfasst auch die nicht sichtbaren Dokumente!

Ein solches Dokument kann zu jedem Zeitpunkt wieder sichtbar geschaltet werden. Dazu benötigt man den Frame (das Fenster) des Dokumentes – und kann dann die Eigenschaft entsprechend einstellen. Genauso kann natürlich auch ein bereits sichtbares Dokument „verschwinden“ – also unsichtbar geschaltet werden:

Dokument unsichtbar schalten:

```
oDoc.getCurrentController().getFrame().getContainerWindow().setVisible(false)
```

Dokument sichtbar schalten:

```
oDoc.getCurrentController().getFrame().getContainerWindow().setVisible(true)
```

Praktische Anwendung:

Das unsichtbar geöffnete Dokument kommt immer dann zum Einsatz, wenn man Daten aus dem Dokument benötigt oder dort speichern möchte.

Es kommt auch dann zum Einsatz, wenn im Rahmen eines Workflows zum Beispiel aus Daten eines Calc-Dokumentes per Makro ein neues Dokument erzeugt wird, der/die Benutzer/in aber zunächst diverse Teile im Calc-Dokument auswählen muss. In diesem Fall wird das neue Dokument zunächst „hidden“ geöffnet, dort Stück für Stück aufgebaut und zum Schluss sichtbar geschaltet.

Ein anderer typischer Anwendungsfall ist die komplette Applikation – immer dann, wenn das Dokument zwar als „Trägerdokument“ dient und auch alle Daten speichert, der/die Benutzer/in aber lediglich in Dialogen seine/ihre Eingaben/Aktionen auswählt und das Ergebnis dann erzeugt wird. In diesem Fall benötigt der/die Benutzer/in kein Dokument (er/sie nimmt dort keine Eingaben vor), er/sie startet das Programm aber durch Öffnen des Dokumentes. Auf diese Weise verknüpft man das Ereignis „Dokument öffnen“ mit einem Makro, das das interne Programm startet und als erstes das Dokument unsichtbar schaltet:

```
sub AutoStartHauptProgramm
  dim StartDoc as variant   'das Startdokument

  REM Initialisierung
  StartDoc = thisComponent
  'Dokument unsichtbar
  StartDoc.getCurrentController().getFrame().getContainerWindow().setVisible(false)
  ' Hauptprogramm starten
  StartHauptprogramm
  ' Programmende - Startdokument schliessen
  if StartDoc.isModified() then StartDoc.setModified(false)
  StartDoc.close(true)
end sub
```

Auch hier wieder wichtig: Nach dem Ende des Hauptprogramms muss man dafür Sorge tragen, dass das Dokument wieder geschlossen wird! Die Überprüfung, ob das Dokument geändert wurde, sollte immer unmittelbar vor dem Schließen-Befehl erfolgen – dadurch verhindert man die Rückfrage für den/die Benutzer/in (aber nicht vergessen: Wurde das Dokument bewusst geändert – zum Beispiel durch Aufnahme neuer Daten – dann muss es auch im Makro selbst gespeichert werden!)

Noch ein Anwendungsfall: Das temporäre Ausblenden des aktiven Dokumentes – entweder, um die Aufmerksamkeit des Benutzers / der Benutzerin auf ein anderes Dokument zu lenken oder auch auf einen Dialog, oder ähnliches. Aber auch hier gilt: Das Dokument ist ja noch aktiv – im Hauptspeicher. Vergessen Sie es nicht, schließen Sie es oder schalten sie es wieder aktiv!

Ganz wichtig in den Fällen, in denen Sie mit mehreren Dokumenten arbeiten: Speichern Sie die Dokumente zunächst in globalen Variablen – arbeiten Sie später nie mit „ThisComponent“ – gerade wenn ein Dokument unsichtbar geschaltet wird, ändert sich die Zuweisung sofort!

5.2.2 Positionen

Die Position bestimmt die Lage des Dokumentes auf dem Desktop, wobei die obere linke Ecke den Bezugspunkt darstellt. Ein Fenster mit der Position 0, 0 würde also exakt oben links in der Ecke des Bildschirms beginnen. Aber: Jeder Bildschirm hat sein eigenes Bezugssystem, dies setzt die absoluten Grenzen – es reicht also, die Position exakt auf X = 0 und Y = 0 zu setzen – in diesem Fall ist das Fenster dann oben links, auch wenn die Kontrolle der Werte später zum Beispiel die Position 28, 10 ergibt. Weiter ließ sich das Fenster eben nicht in der Ecke platzieren.

Man kann ein Fenster also nicht aus dem Bildschirm herausschieben – mit der Maus im sichtbaren Bereich geht das – per Code aber nicht.

Zum Platzieren des Fensters wird die Methode setPosSize() verwendet, mit der allerdings auch die Größe geändert werden kann!

```
oDoc = thisComponent
oWin = oDoc.getCurrentController().getFrame().getContainerWindow()
oWin.setPosSize(0,0,0,0,3)
```

Details siehe Kapitel 5.2.3

Hinweis:

Nie Größe und Position gleichzeitig setzen wegen maximierter Fenster!

5.2.3 Größe

Ähnlich wie die Position kann auch die Größe eines Fensters per Makro geändert oder justiert werden. Dies ist immer dann sinnvoll, wenn beispielsweise zwei Komponenten oder Fenster

nebeneinander auf dem Bildschirm angezeigt werden sollen (der WollMux ist dafür ein schönes Beispiel – links die Formular-GUI, rechts das Dokument).

Auch hier werden die Angaben in Pixel gemacht. Der folgende Code ändert die Größe des Fensters auf 400 px Breite und 250 px Höhe:

```
oDoc = thisComponent
oWin = oDoc.getCurrentController().getFrame().getContainerWindow()
oWin.setPosSize(0,0,400,250,12)
```

Das Fenster wird dabei – ebenfalls vom oberen linken Bezugspunkt – entsprechend angepasst – nur die Größe, nicht die Skalierung der Inhalte. Ein so kleines Fenster wie hier eingestellt lässt allerdings keinen Arbeitsbereich mehr übrig – hier würde man später nur Teile der Symbolleisten etc. sehen.

Mit der Methode SetPosSize() können also die Position und Größe des Fensters gesetzt werden. Die Methode erwartet fünf Parameter, und zwar:

setPosSize(X-Koord., Y-Koord., Breite, Höhe, PosSize)

x- und y-Koordinaten sind die Werte in Bildpunkten (Pixeln; als Long) von der oberen linken Ecke des Rechtecks, Breite und Höhe bezeichnen die Größe des Fensters (als Long), ebenfalls in Bildpunkten, PosSize ist eine Konstante (als Short), die bestimmt, welche Werte wirklich geändert werden. Sie finden diese Werte im Modul com.sun.star.awt.POSSIZE:

Wert (als Text)	Wert (als Zahl)	Beschreibung
X	1	Ändert nur die x-Koordinate
Y	2	Ändert nur die y-Koordinate
WIDTH	4	Ändert nur die Breite
HEIGHT	8	Ändert nur die Höhe
POS	3	Ändert nur die Position (x- + y-Koordinate)
SIZE	12	Ändert nur die Größe (Höhe, Breite)
POSSIZE	15	Ändert sowohl die Position als auch die Größe

Ich erinnere nochmal daran, dass die Position und Größe des Moduls intern global zwischengespeichert werden – und alle weiteren Dokumente dieses Moduls mit genau diesen Werten erzeugt werden. Es ist also in der Regel sinnvoll, zunächst die aktuelle Größe auszulesen und zwischenzuspeichern und später diese wieder zurück zu schreiben:

```
...
  oDocPosSize = oDoc.currentController.frame.ContainerWindow.getPosSize()  'zwischenspeichern
der aktuelle Position/Größe - Parent Fenster Position
  oDoc.setPosSize(xx,xx,xx,xx, 15)  'neue Größe zuweisen
... 'hier folgt jetzt der weitere Code ...
  'Originalgröße wieder herstellen
```

```
    ds = oDocPosSize
    oDoc.currentController.frame.ContainerWindow.setPosSize(ds.X, ds.Y, ds.width, ds.height, 15)
...
```

5.2.4 Menü- und Symbolleisten anpassen

Alle Menü- und Symbolleisten in OOo lassen sich individuell anpassen. Insbesondere bei der Entwicklung von Extensions werden diese Möglichkeiten gerne genutzt – sei es, dass vorhandene Menü- oder Symbolleisteneinträge mit eigenen Makros belegt werden oder dass die vorhandene Struktur mit eigenen Begriffen und Icons erweitert wird. Auch das Erzeugen völlig eigener und neuer Symbolleisten (siehe hierzu auch Abschnitt 5.2.5) ist eine beliebte Technik, um eigene Startroutinen zu definieren.

Bevor nun die Möglichkeiten der Manipulation detailliert dargestellt werden, zunächst ein paar grundlegende Kenntnisse des Ladevorgangs von OpenOffice.org.

Alle Symbol- und Menüleisten sind in speziellen XML-Dateien definiert, in denen auch die dazugehörigen Programmaufrufe stehen sowie die angezeigten Icons und Texte. Unterschieden wird dabei zwischen den vom Programm vorgegebenen Strukturen sowie den vom Benutzer / von der Benutzerin selbst angepassten Darstellungen. Der Aufbau sieht nun wie folgt aus:

Mit dem Start von OOo werden zunächst die im Programmverzeichnis definierten Menüleisten und Strukturen sowie die dort festgelegten Symbolleisten geladen (programmweite). Anschließend werden die Strukturen aus dem User-Verzeichnis nachgeladen, die dort benutzerspezifisch definiert wurden. Hier handelt es sich um die vom Benutzer / von der Benutzerin änderbaren Darstellungen der Symbolleisten und der Menüs. Auch hierfür gibt es eine Voreinstellung, die jede/r Benutzer/in beim ersten Start (zum Zeitpunkt des Anlegens seines/ihres Profils) erbt. Diese kann er/sie nun jedoch individuell anpassen – die geänderten XML-Dateien werden nun im Benutzerprofil gespeichert.

Die benutzerdefinierten Einstellungen überschreiben die Basiseinstellungen des Programms, wenn definiert.

Als dritte Schicht schließlich werden Erweiterungen der Symbolleiste oder der Menüleiste nachgeladen, die zum Beispiel von installierten Extensions geliefert werden – und die überschreiben wiederum die bereits vorhandenen Einstellungen. Sind mehrere Extensions installiert, die wiederum die gleiche Funktion überschreiben, so „gewinnt“ immer die zuletzt installierte.

Beispiel:

Der Menüeintrag „Datei drucken“ ist im Programm verknüpft mit dem internen Programmaufruf „uno:print“ und mit dem Label „Datei drucken“.

Dies ist im Hauptprogramm so festgelegt. Der/Die Benutzer/in kann nun für sich (für sein/ihr Benutzerprofil) das Label ändern auf zum Beispiel „sofort drucken“ und er/sie kann auch das

Icon austauschen. Die Zuordnung dieses Befehls zu „uno:print“ kann er/sie über das Frontend nicht ändern.

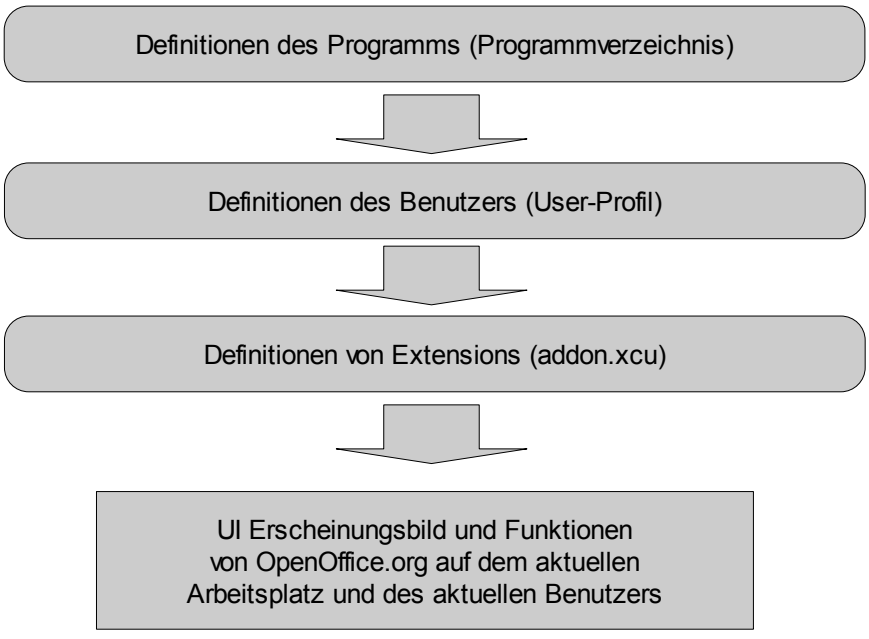
Jedenfalls würde nun in seiner/ihrer Oberfläche nur noch der Menü-Befehl „Datei – sofort drucken“ auftauchen.

Eine Extension (nennen wir sie „Komfortdruck“) bietet beispielsweise verbesserte Druckfunktionen und überschreibt jetzt während der Installation den Befehl „uno:print“ mit einem eigenen Programmaufruf – einem mit dem Start der Extension. Dies bedeutet nun: Klickt der/die Benutzer/in zukünftig auf das Druck-Icon oder auf den Menübefehl „sofort drucken“, so startet nicht mehr die interne Druckroutine („uno:print“) sondern die durch die Extension „Komfortdruck“ definierte.

Allerdings bleiben die „alten“ Zuordnungen natürlich bestehen – sie werden nur nacheinander geladen und die letzte überschreibt im Arbeitsspeicher immer den Wert der vorherigen. Dieser Ladevorgang ist wichtig zu verstehen – zeigt er doch auch, wo und wie Manipulationen sinnvoll und wirkungsvoll sind.

So ist es wenig hilfreich, die xml-Dateien im Programmverzeichnis direkt zu ändern – werden sie doch später sicher überschrieben, andererseits lassen sich mit Extensions völlig neue Zuordnungen erzeugen – im jeweiligen Client.

Nochmals der Ladeprozess:



Um über eine Extension Menü- oder Symbolleisten zu verändern, muss sich im Verzeichnis der Extension eine „addon.xcu“-Datei befinden. Diese wird während des Ladeprozesses ausgewertet und entsprechende Einstellungen werden überschrieben.

Das Beispiel zeigt eine addon.xcu-Datei, in der ein neues Symbol in die Standard-Symbolleiste eingefügt und dieses mit dem Makro „aktuelle Seite drucken“ verbunden wird:

```
<?xml version='1.0' encoding='UTF-8'?>
<oor:component-data xmlns:oor="http://openoffice.org/2001/registry"
xmlns:xs="http://www.w3.org/2001/XMLSchema" oor:name="Addons"
oor:package="org.openoffice.Office">
 <node oor:name="AddonUI">
  <node oor:name="OfficeToolbarMerging">
   <node oor:name="SeiteDrucken_MAK042.OfficeToolBar" oor:op="replace">
    <node oor:name="T1" oor:op="replace">
     <prop oor:name="MergeToolBar">
      <value>standardbar</value>
     </prop>
     <prop oor:name="MergePoint">
      <value>.uno:PrintDefault</value>
     </prop>
     <prop oor:name="MergeCommand">
      <value>AddAfter</value>
     </prop>
     <prop oor:name="MergeFallback">
      <value>AddLast</value>
     </prop>
     <prop oor:name="MergeContext">
      <value/>
     </prop>
     <node oor:name="ToolBarItems">
      <node oor:name="m1" oor:op="replace">
       <prop oor:name="Context" oor:type="xs:string">
         <value>com.sun.star.text.TextDocument</value>
       </prop>
       <prop oor:name="URL" oor:type="xs:string">
         <value>macro:///SeiteDrucken_MAK042.MAK042_Code.MAK042_SeiteDrucken</value>
       </prop>
        <prop oor:name="ImageIdentifier" oor:type="xs:string">
         <value/>
       </prop>
       <prop oor:name="Title" oor:type="xs:string">
         <value>Seite drucken</value>
       </prop>
       <prop oor:name="Target" oor:type="xs:string">
         <value>_self</value>
       </prop>
      </node>
    </node>
   </node>
  </node>
 </node>
 <node oor:name="Images">
  <node oor:name="mic.de.tk.o3.images" oor:op="replace">
   <prop oor:name="URL" oor:type="xs:string">
     <value>macro:///SeiteDrucken_MAK042.MAK042_Code.MAK042_SeiteDrucken</value>
   </prop>
```

```
    <node oor:name="UserDefinedImages">
      <prop oor:name="ImageSmallURL" oor:type="xs:string">
        <value>%origin%/images/SeiteDrucken_16.png</value>
      </prop>
      <prop oor:name="ImageSmallHCURL" oor:type="xs:string">
        <value>%origin%/images/SeiteDrucken_16.png</value>
      </prop>
    </node>
   </node>
  </node>
 </node>
</oor:component-data>
```

Erkennbar ist, dass neben dem Einfügeort (hier „MergePoint“ → „uno:PrintDefault“ – also das Icon „Datei direkt drucken“) auch die Position („MergeCommand“ → „addAfter“ – also nach dem Icon) sowie eine Ersatzmöglichkeit mit übergeben wird, falls das Icon überhaupt nicht vorhanden ist („MergeFallBack“ → „AddLast“ – wird ans Ende der vorhanden Icons/Befehle gestellt).

Über den Knoten „m1“ (ein frei wählbarer Name für das Item) wird nun der verknüpfte Programmbefehl (hier → `macro:///SeiteDrucken_MAK042.MAK042_Code.MAK042_SeiteDrucken`) sowie die Umgebung übergeben, in der der Befehl überhaupt nur zur Verfügung steht und auch nur angezeigt wird (hier → `com.sun.star.text.TextDocument` , also nur Textdokumente Modul Writer).

Details zu dem Aufbau und der Struktur von addon.xcu-Dateien lassen sich im Internet finden, zum Beispiel unter

http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/Extensions/Extensions

5.2.5 Eigene Menü- /Symbolleisten

Neben dem Ändern und Anpassen bereits bestehender Symbolleisten kann man auch über die addon.xcu völlig eigene Leisten erzeugen und anzeigen lassen.

Leider gibt es hierbei zwei unterschiedliche Wege mit unterschiedlichen Auswirkungen.

Die „normale“ und somit am häufigsten verwendete ist der Weg über die „addon.xcu“, also die Definition der eigenen Symbolleiste direkt in der Extension. Das folgende Beispiel zeigt eine solche „einfache“ Definition für eine eigene Symbolleiste mit genau einer Schaltfläche (ohne Icon):

```
<?xml version='1.0' encoding='UTF-8'?>
<oor:component-data
 xmlns:oor="http://openoffice.org/2001/registry"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 oor:name="Addons"
 oor:package="org.openoffice.Office">
  <node oor:name="AddonUI">
    <node oor:name="OfficeToolBar">
      <node oor:name="MAK_140.OfficeToolBar" oor:op="replace">
```

```
        <node oor:name="m1" oor:op="replace">
          <prop oor:name="Context" oor:type="xs:string">
            <value>com.sun.star.text.TextDocument,com.sun.star.sheet.SpreadsheetDocument,com.su
n.star.sdb.OfficeDatabaseDocument</value>
          </prop>
          <prop oor:name="URL" oor:type="xs:string">
            <value>vnd.sun.star.script:MAK_140.code_dlg_start.MAK140_StartApplikation?
language=Basic&amp;location=application</value>
          </prop>
          <prop oor:name="Title" oor:type="xs:string">
            <value>B&#252;cherliste BAU</value>
          </prop>
          <prop oor:name="Target" oor:type="xs:string">
            <value>_self</value>
          </prop>
        </node>
      </node>
    </node>
  </node>
</oor:component-data>
```

Zu der addon.xcu gehören dann noch diverse Dateien im Verzeichnis Office/UI – diese xcu-Dateien passen die Ansicht der jeweiligen Module an. Die Symbolleiste wird dort wie folgt aufgerufen:

```
<?xml version="1.0" encoding="UTF-8"?>
<oor:component-data xmlns:oor="http://openoffice.org/2001/registry"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 oor:name="WriterWindowState"
 oor:package="org.openoffice.Office.UI">
...<node oor:name="UIElements">
     <node oor:name="States">
       <node oor:name="private:resource/toolbar/addon_MAK_140.OfficeToolBar" oor:op="replace">
         <prop oor:name="UIName" oor:type="xs:string">
           <value>MAK_140_Buecherliste</value>
         </prop>
       </node>
     </node>
   </node>
</oor:component-data>
```

Zwar erhält auch diese Symbolleiste einen eindeutigen Namen („MAK_140_Buecherliste“) und wird korrekt geladen nach dem Start des User-Profils, versucht man jedoch später per (Basic-)Makro auf diese Symbolleiste zuzugreifen, so existiert sie nicht. Auch kann der/die Benutzer/in diese „Addon-“Symbolleiste nicht über den Menübefehl Extras/Anpassen → Symbolleisten verändern – sie existiert auch dort nicht. Er/Sie kann sie aber ein- oder ausblenden – im Menü Ansicht/Symbolleisten wird sie korrekt unter dem Namen „MAK_140_Buecherliste“ aufgeführt und gelistet.

Um eine solche Symbolleiste zu erzeugen, empfiehlt es sich, die Extension zunächst mit Hilfe des Tools „Basic-Addon-Builder“ (siehe auch Anhang) zu erzeugen – dann sind schon einmal

die korrekten Strukturen, Verzeichnisse und xml-Dateien erstellt. Diese können nun einfacher mit einem normalen Editor geändert und den eigenen Wünschen angepasst werden.

Der Vorteil einer solchen Symbolleiste: Wird die Extension wieder deinstalliert, so verschwindet auch die Symbolleiste – man muss sich also um die „Entsorgung“ keine Gedanken machen.

Der zweite Weg, eine Symbolleiste zu erzeugen, wäre direkt über die API und den Script-Code. In diesem Fall wird eine Leiste erzeugt ähnlich wie in der UI, wenn unter Extras/Anpassen → Symbolleisten → neu gewählt wird. Eine solche Symbolleiste erhält immer als Namensbestandteil den Begriff „custom_“ – als Vorsilbe. Und so kann sie später auch identifiziert werden.

Auf eine so erzeugte Symbolleiste kann per Code zugegriffen und es können zur Laufzeit Änderungen vorgenommen werden. Das bewährt sich immer dann, wenn zwischenzeitlich zusätzliche Schaltflächen benötigt werden oder das Label einer Schaltfläche kontextbezogen verändert werden soll. Eine so erzeugte Symbolleiste kann vom Benutzer / von der Benutzerin aber auch über die UI geändert werden und sie ist persistent – einmal erzeugt bleibt sie erhalten bis sie wieder entfernt wird (manuell über die UI oder per Code). Wird die Extension (das Makro) entfernt, so verbleibt die Symbolleiste an ihrer Stelle – und ein Klick auf die Buttons führt jetzt zu einem Runtime-Fehler. Erzeugt man eine solche Symbolleiste, so muss auch dafür Sorge getragen werden, dass sie wieder entfernt wird.

Bewährt haben sich „custom“-Symbolleisten als temporäre Leisten zur Laufzeit des Makros, die mit Beendigung des Makros auch wieder entfernt werden. Man kann mit diesen Symbolleisten auch reine Informationen übertragen – also Buttons ohne echte Funktion, aber mit wechselnden Label-Bezeichnungen.

Der folgende Code zwingt die Technik, eine solche „custom“-Symbolleiste per Code zu erzeugen, den Inhalt zu ändern und auch wieder zu löschen.

Wichtiger Hinweis:

Eine selbst erzeugte Symbolleiste **muss** die Vorsilbe **custom_** erhalten! Nur so wird sie intern erkannt und entsprechend angezeigt bzw. kann geändert werden.

```
public const AL_TOOLBAR = "private:resource/toolbar/custom_altoolbar"

'/** MAK017_Check_AL_ToolbarErzeugen
'*************************************************************************.
'* @kurztext prüft, ob der AL Toolbar vorhanden ist. Wenn nein, wird er erzeugt.
'* Die Funktion prüft, ob der AL Toolbar vorhanden ist. Wenn nein, wird er erzeugt.
'* Die AL Symbolleiste wird für viele Funktionen benötigt und muss vorhanden sein.
'*
'* @return  bFlag as boolean   true, wenn vorhanden, false bei Fehler oder wenn nicht vorhanden
'*************************************************************************
'*/
function MAK017_Check_AL_ToolbarErzeugen
  dim oModuleCfgMgrSupplier as variant
```

```
  dim oModuleCfgMgr as variant
  dim i%

  MAK017_Check_AL_ToolbarErzeugen = true   'Vorgabe

  For i = 0 to uBound(aModTB())
    oModuleCfgMgrSupplier =
createUnoService("com.sun.star.ui.ModuleUIConfigurationManagerSupplier")
    oModuleCfgMgr = oModuleCfgMgrSupplier.getUIConfigurationManager(aModTB(i))
    if ( oModuleCfgMgr.hasSettings( AL_TOOLBAR )) then
      'vorhanden, nichts tun...
      exit function
    else   'Toolbar erzeugen
      ToolbarErzeugen
      exit function
    end if
  next

  MAK017_Check_AL_ToolbarErzeugen = false  'bei Fehler
end function


'/** ToolbarErzeugen
'*************************************************************************.
'* @kurztext erzeugt eine Symbolleiste (AL) in den Modulen
'* Die Funktion erzeugt eine (Custom_) Symbolleiste in den Modulen.
'* Sie wird hier benutzt, um die AL Symbolleiste zu erzeugen, die später den
'* aktuellen AL-Plan anzeigt. Die Symbolleiste muss per Makro geändert
'* werden können, dies geht derzeit nur mit "Custom_" Symbolleisten, die von
'* Extensions nicht erzeugt werden können.
'*
'*************************************************************************
'*/
sub ToolbarErzeugen
  dim oModuleCfgMgrSupplier as variant
  dim oModuleCfgMgr as variant
  dim oToolbarSettings as variant
  dim oToolbarItem as variant
  dim i%

  MAK017_dyn.MAK017_InitAL
  'use this if you want to store the menu globally
  For i = 0 to uBound(aModTB())   'für alle Module
    oModuleCfgMgrSupplier =
createUnoService("com.sun.star.ui.ModuleUIConfigurationManagerSupplier")
    oModuleCfgMgr = oModuleCfgMgrSupplier.getUIConfigurationManager(aModTB(i))

    REM *** Create a settings container which will define the structure of our new
    REM *** custom toolbar.
    oToolbarSettings = oModuleCfgMgr.createSettings()

    REM *** Set a title for our new custom toolbar
    oToolbarSettings.UIName = "AL_aktuell"

    REM *** Create a button for our new custom toolbar
    oToolbarItem = AL_CreateToolbarItem( ALCHGURL, "AL: " & sALakt)
    oToolbarSettings.insertByIndex( 0, oToolbarItem )

    REM *** Set the settings for our new custom toolbar. (replace/insert)
```

```
    if ( oModuleCfgMgr.hasSettings( AL_TOOLBAR )) then
      oModuleCfgMgr.replaceSettings( AL_TOOLBAR, oToolbarSettings )
    else
      oModuleCfgMgr.insertSettings( AL_TOOLBAR, oToolbarSettings )
    endif
    oModuleCfgMgr.store
  next

  REM Flag setzen
  bToolbarFlag = true
  MAK017_def.MAK017_SaveNewPref()   'Werte sichern
End Sub

'/**   AL_CreateToolbarItem
'*************************************************************************.
'* @kurztext erzeugt ein Symbol/Schaltfläche im AL Toolbar
'* Die Funktion erzeugt ein Symbol/Schaltfläche im AL Toolbar
'*
'* @param1  Command as string   der verknüpfte Befehl (Makro)
'* @param2  Label as string   der angezeigte Name des Buttons
'*
'* @return  aToolbarItem   PropertyValue des Controlelementes
'*
'*************************************************************************
'*/
Function AL_CreateToolbarItem( Command as String, Label as String ) as Variant
  Dim aToolbarItem(3) as new com.sun.star.beans.PropertyValue

  aToolbarItem(0).Name = "CommandURL"
  aToolbarItem(0).Value = Command
  aToolbarItem(1).Name = "Label"
  aToolbarItem(1).Value = Label
  aToolbarItem(2).Name = "Type"
  aToolbarItem(2).Value = 0
  aToolbarItem(3).Name = "Visible"
  aToolbarItem(3).Value = true

  AL_CreateToolbarItem = aToolbarItem()
End Function

'/**  ToolbarEntfernen
'*************************************************************************.
'* @kurztext entfernt die AL Symbolleiste aus den Modulen
'* Die Funktion entfernt die AL Symbolleiste aus den Modulen
'*
'*************************************************************************
'*/
sub ToolbarEntfernen
  dim oModuleCfgMgrSupplier as variant
  dim oModuleCfgMgr as variant
  dim i%

  MAK017_InitAL
  For i = 0 to uBound(aModTB())
    oModuleCfgMgrSupplier =
createUnoService("com.sun.star.ui.ModuleUIConfigurationManagerSupplier")
    oModuleCfgMgr = oModuleCfgMgrSupplier.getUIConfigurationManager(aModTB(i))
    if ( oModuleCfgMgr.hasSettings( AL_TOOLBAR )) then
      oModuleCfgMgr.removeSettings( AL_TOOLBAR )   'entfernen
```

```
        oModuleCfgMgr.store
      end if
    next

    REM Flag entfernen
    bToolbarFlag = false
    MAK017_def.MAK017_SaveNewPref()    'Werte sichern
end sub
```

5.2.6 Fehlerquellen und Behandlung

Die am häufigsten auftretende Fehlerquelle ist die nicht korrekte Zuordnung eines Befehls/Symbols einer benutzerdefinierten Symbolleiste zu dem passenden Makro. In diesem Fall erzeugt OOo die folgende Fehlermeldung:

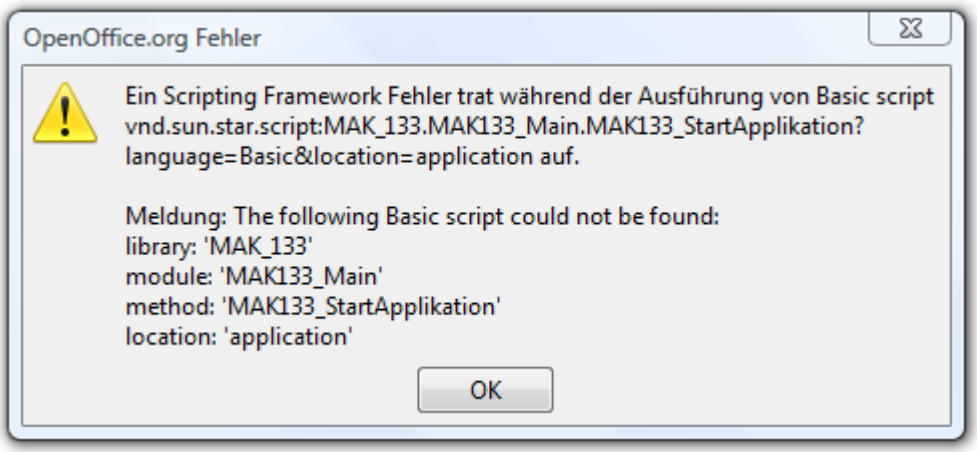


Abbildung 5.4: Fehlermeldung - Makro nicht gefunden

Da diese Fehlermeldung reproduzierbar ist, lässt sich auch leicht die Ursache dafür feststellen:

Das vom Icon/Befehl aufgerufene Makro wird ja im Klartext dargestellt:

```
vnd.sun.star.script:MAK_133.MAK133_Main.MAK133_StartApplikation?
language=Basic&location=application
```

Also: Es wird in der Bibliothek **MAK_133** im Modul **MAK133_Main** das Makro **MAK133_StartApplikation** aufgerufen**.**

Der Parameter **language** zeigt, um welche Script-Sprache es sich handelt (hier Basic) und der Parameter **location** definiert den Speicherort (hier „application“, also im User-Profil).

Man prüft nun also, ob die Funktion im angegebenen Modul in der Bibliothek vorhanden ist, und korrigiert entweder den Funktionsaufruf oder den Namen der Funktion.

Ein solcher Fehler tritt entweder beim Erstellen der addon.xcu auf (vertippt, nicht UTF-8, kopiert aus einer anderen Extension und vergessen, Pfade anzupassen) oder auch manchmal nach dem nicht korrekten Beenden einer OOo-Instanz. Dabei kann es passieren, dass die Makroverwaltungsdateien script.xlb und script.xlc nicht mehr korrekt zurückgeschrieben wurden und somit „zerstört“ wurden – in diesem Fall sind die Makros für OOo nicht mehr zu finden. Die

Dateien können manuell repariert (siehe hierzu auch Kapitel 4.4.3) oder durch Neuinstallation der Extension neu geschrieben werden.

6 Dialoge und Benutzerinterface

Die wenigsten Makros kommen ohne Benutzerinteraktion aus. Meist ist mindestens der Start eine „echte“ Benutzerinteraktion, in der Regel sind Applikationsprogrammierungen jedoch abhängig von Benutzereingaben und -reaktionen.

Jeder Dialog und jeder Hinweis, der vom Programm ausgegeben wird, ist ein Benutzer-Interface und erfordert normalerweise auch eine Aktion desselben. Oft ist es auch nötig, dem/der Benutzer/in zumindest eine Rückmeldung zu geben, dass ein Makro gerade läuft und diverse Arbeiten durchführt. Auch dies wird mit Hinweisfenstern (Dialogen) erledigt.

6.1 Definition

Bevor Details zu den Dialogen folgen, zunächst eine gemeinsame Begriffsdefinition für die Möglichkeiten einer Benutzerinteraktion:

6.1.1 Dialog

Der Begriff „Dialog“ wird am häufigsten auftreten. Unter einem Dialog versteht man ein eigenständiges Fenster auf der Benutzeroberfläche, das ähnlich wie andere Fenster die typischen Eigenschaften der Oberflächen besitzt, es kann verschoben werden, eventuell verkleinert oder skaliert und es kann geschlossen werden (entsprechend der Oberfläche – in der Regel durch ein Schließen-Kreuz in der rechten oberen Ecke). All diese Funktionen werden direkt von der UI bereitgestellt.

Der Dialog wiederum besitzt Steuerelemente, die unterschiedlicher Art sein können:

- reine Informationsfelder (Label-Felder)
- Eingabefelder (Text-, Datums.-; Zahlenfelder und ähnliche)
- Auswahllisten
- Schaltflächen

Der Dialog dient dazu, Benutzereingaben zu erfragen oder diverse Informationen auszugeben. Jeder Dialog besitzt mindestens eine Schaltfläche, die den Dialog beendet. Zwar ginge dies theoretisch auch über das Schließen-Kreuz, diese Möglichkeit ist jedoch außen vor zu lassen.

Ein Dialog ist immer ein Kind-Prozess – abgeleitet vom aktuellen Dokument (bzw. vom Modul, das beim Starten des Makros gerade aktiv war). Der Dialog „hängt“ also mit dem Eltern-Prozess direkt zusammen und wird mit diesem auf der Oberfläche bisweilen verkleinert oder in den Hintergrund verschoben.

Ein typischer Dialog wird **ausgeführt** – das heisst, der Makroablauf stoppt an der Stelle, an dem der Dialog aktiv geschaltet wird,

```
odlg.execute()
```

und wird fortgesetzt in der Zeile danach, sobald der Dialog geschlossen wird. Das „Ausführen“ übergibt die Kontrolle an interne Strukturen, die dafür Sorge tragen, dass Befehle der Oberfläche – also Mausklicks, Klick auf den Schließen-Button etc. – korrekt ausgeführt werden. Gleichzeitig werden alle anderen Aktivitäten in OpenOffice.org „gestoppt“, das bedeutet, dass der Dialog nun der dominante und aktive Prozess ist, der Eltern-Prozess ruht. Das bedeutet, Eingaben im Dokument sind nicht mehr möglich, solange der Dialog geöffnet ist, auch andere Aufrufe der Menü- oder Symbolleisten des Eltern-Prozesses funktionieren nicht.

Da der Dialog als Model und Objekt vorher aufgebaut werden muss, gibt es noch eine zweite Möglichkeit für die Darstellung des Dialoges – er wird nicht ausgeführt, sondern nur **sichtbar** geschaltet (man spricht jetzt von einem „schwebenden Dialog“).

```
oDlg.setVisible(true)
```

In diesem Fall stoppt das Makro nicht, die nächste Code-Zeile wird unmittelbar nach dem Befehl „setVisible()“ ausgeführt und es „läuft“ weiter. Da der Dialog aber auch ein Objekt des aktuellen Makros ist und mit dem Beenden dieses Makros das Objekt auch „zerstört“ wird, endet die Sichtbarkeit des Dialoges mit dem Ende des Makros. Dieser Prozess ist selten erwünscht, in diesem Fall muss der/die Programmierer/in dann selbst dafür sorgen, dass das Makro eben nicht beendet wird (zum Beispiel durch eine Endlosschleife – siehe hierzu Kapitel 6.3 - „Schwebende“ Dialoge).

Der Vorteil der „schwebenden“ Dialoge ist, dass sie auch die anderen Prozesse nicht blockieren, d.h. der Eltern-Prozess (meist ein Dokument) läuft normal weiter, der/die Benutzer/in kann im Dokument arbeiten und auch alle Menüs oder Symbolleisten nutzen. Das Dialogfenster bleibt aber immer im Vordergrund (vor dem Eltern-Prozess) – eben schwebend.

Für beide Fälle gibt es diverse Einsatzmöglichkeiten.

Ein Dialog wird vom Programmierer / von der Programmiererin in seiner Größe festgelegt – und kann typischerweise vom Benutzer / von der Benutzerin nicht geändert werden. Dadurch wird auch sichergestellt, dass der/die Benutzer/in alle Kontrollelemente sieht, die für ihn/sie vorgesehen sind, und die Bedienung gelingt.

6.1.2 Formular

Auch Formulare können ein Benutzer-Interface darstellen. Formulare sind meist eigene Dokumente oder Teile eines Dokumentes, die Steuerelemente wie etwa Eingabefelder,

Auswahllisten oder Buttons enthalten können. Auch über Formulare kann der/die Benutzer/in Informationen eingeben, Informationen empfangen oder den Makroablauf steuern.

Anders als Dialoge sind Formulare ja Teil eines Dokumentes – und folgen somit den Aktionen des Dokumentes. Ein Dokument wiederum ist Teil eines Fensters, das unabhängig vom Inhalt entsprechend den Möglichkeiten der UI-Oberfläche skaliert, verkleinert oder sonst wie manipuliert werden kann. Außerdem „erbt“ das OOo-Modul alle Eigenschaften vom zuletzt geöffneten Dokument gleicher Machart – also zum Beispiel des letzten Writer-Dokumentes. Also, selbst wenn der/die Programmierer/in entsprechend Vorsorge trifft und die Fenstergröße in seinem/ihrem Skript passend einstellt sowie die Symbolleisten ausblendet, so ist noch immer nicht sichergestellt, dass der/die Benutzer/in tatsächlich das passende Formular wie gewünscht sieht. Er/Sie muss nur eine eigene neue Symbolleiste in Writer geschaffen haben – schon bleibt etwas übrig, was nicht gewünscht war.

Und: Der/Die Programmierer/in sollte unbedingt dafür Sorge tragen, dass nach Abschluss der Formularaktivitäten wieder die ursprüngliche Umgebung (also das Aussehen der Komponente „Writer“ zum Beispiel) hergestellt wird (Größe, Symbol- und Menüleisten etc.). Sonst erlebt der/die Benutzer/in beim Öffnen oder Erstellen des nächsten Writer-Dokumentes eine herbe Überraschung.

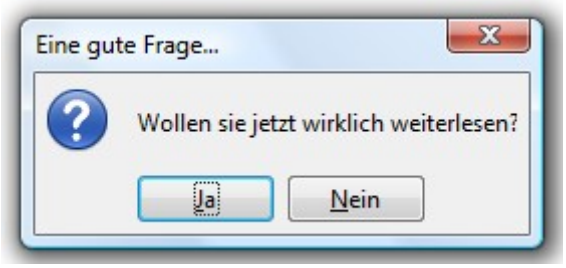
Formulare besitzen zusätzlich die Möglichkeit, direkt mit Datenbanken verbunden zu werden – in der Regel reichen die eingebauten Funktionen aber nicht aus und es muss sowieso zusätzlich programmiert werden.

6.1.3 Message-Boxen – Hilfsdialoge

Die dritte Gruppe der Interaktionen sind einfache Meldungsfenster. Hier bietet OOo die Möglichkeit, eine Mitteilung an den/die Benutzer/in zu senden und entsprechend seiner/ihrer Aktion (mindestens eine ist erforderlich) fortzufahren.

Eine Hinweisbox lässt sich recht einfach mit dem Basic-Befehl msgbox aufbauen:

```
sInfoText = "Wollen sie jetzt wirklich weiterlesen?"
sTitelText = "Eine gute Frage..."
msgbox (sInfoText, 4+32+128, sTitelText)
```



Bei der Verwendung der „Messagebox“ (msgbox) braucht der/die Programmierer/in sich keine Gedanken um den Aufbau des Fensters etc. zu machen – das alles ist intern vorbelegt. Die Hinweis-Box bietet diverse Möglichkeiten der individuellen Anpassung (Anzahl Schaltflächen, Schaltflächentexte, Symbole etc.) und kann eine Rückmeldung geben, welcher Button gedrückt wurde. In diesem Fall muss der Aufruf der Msgbox rechts einer Zuordnung erfolgen. Die Box liefert dann immer einen Rückgabewert – auch beim Abbruch des Dialoges durch das Schließen-Kreuz:

```
n = msgbox (sInfoText, 4+32+128, sTitelText)
```

so nimmt n den Wert 6 an bei Klick auf „Ja“, 7 bei Klick auf „Nein“ und 2 bei Abbruch über das Schließen-Kreuz.

Zu den Hilfsdialogen zählen auch Status-Dialoge, die zum Beispiel den/die Benutzer/in lediglich über einen Fortschritt informieren – also ihm/ihr einen Hinweis geben, dass sich noch etwas tut und er/sie einfach noch warten soll. In diesem Fall muss aber der Dialog zunächst aufgebaut werden und dann per „visible“-Eigenschaft sichtbar gestellt werden. Jetzt läuft das Makro weiter, der Statusbalken kann regelmäßig aktualisiert werden und am Ende der Aktivität wird der Dialog wieder unsichtbar geschaltet.

Der folgende Code zeigt den Aufbau und den Einsatz eines 10-stufigen Wartedialoges – als reine Prinzipskizze:

```
Sub WartedialogTest
  dim oDlg as variant
  dim i%

  DialogLibraries.loadLibrary("CookBook")
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_warten)

  oDlg.setVisible(true)  'Dialog sichtbar schalten

  REM Schleife für die Wartezeit
  for i = 2 to 10
     with oDlg
      .getControl("pgb_1").model.ProgressValue = i
      .getControl("lbl_wtext").text = "Vorgang " & i & " von 10"
     end with
     wait(1000) 'eine Sekunde warten (oder Programm ausführen)
  next

  oDlg.setVisible(false)  'Dialog unsichtbar schalten

End Sub
```

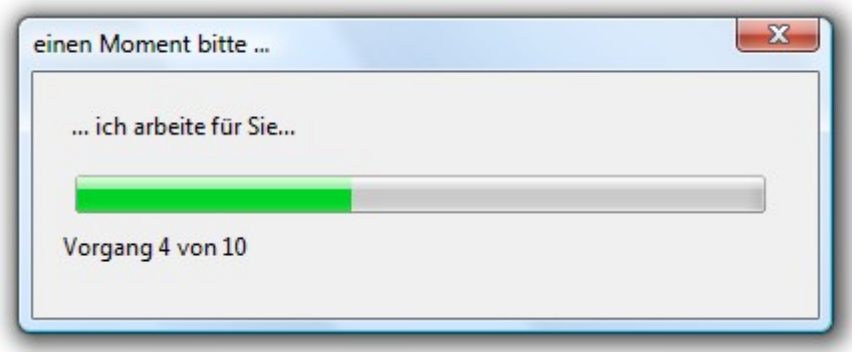


Abbildung 6.1: Beispiel Fortschrittsbalken

In solchen Dialogen fehlen oft Schaltflächen – der Dialog verschwindet ja von selbst ohne Eingriff des Benutzers / der Benutzerin.

Auf gleiche Art und Weise lassen sich auch Hilfsdialoge aufbauen, die dem/der Benutzer/in lediglich Informationen liefern – ohne selbst eine Aktion von ihm/ihr zu erwarten. In diesem Fall dürfen Dialoge nie mit „execute()“ aufgerufen werden, denn diese Methode muss und kann nur durch Benutzerinteraktion beendet werden.

6.2 Dialoge erzeugen und benutzen

Dialoge können mit dem in der IDE integrierten grafischen Dialogeditor sehr einfach erzeugt werden. Der Dialogeditor allerdings erzeugt lediglich eine xml-Datei – die Beschreibung und das Model des Dialoges, integriert im Basic-Ordner, erkennbar an dem Namen des Dialoges ergänzt um die Dateierweiterung *.xdl. Also wird ein erzeugter Dialog mit dem Namen „dlg_main“ zu finden sein als Beschreibung in der Datei dlg_main.xdl.

Ein so beschriebener Dialog muss von OOo zunächst intern in ein Objekt verwandelt werden, bevor er auf der Benutzeroberfläche angezeigt werden kann. Dazu wird zunächst ein „model“ erzeugt, das dann vom aktuellen Controller in einen „View“ gewandelt und angezeigt wird.

Damit ein Dialog überhaupt genutzt werden kann, muss immer zunächst die Dialog-Bibliothek geladen werden. Anders als beim Makro-Code wird diese nämlich nicht automatisch mitgeladen, wenn ein Makro der aktiven Bibliothek ausgeführt wird. Wird die (Dialog-)Bibliothek nicht geladen, so existieren für OOo überhaupt keine Dialoge für diese Bibliothek – es wird immer zu einem Laufzeitfehler kommen.

Achtung Entwicklungsfalle:

Wird der Dialog während der Entwicklung erst designed, wird in der Regel die Bibliothek auch geladen. Eine einmal geladene Grafikbibliothek bleibt aber aktiv bis zum vollständigen Beenden

von OOo – so fällt es in der Regel gar nicht auf, dass Code-Zeilen fehlen (Laden der Bibliothek). Wird eine solche Version ausgeliefert, kommt es aber dann zum Fehler!

Dialoge müssen nicht in der aktiven Bibliothek untergebracht werden, sie können auch in einer eigenen Bibliothek stehen oder man kann auf den Dialog einer anderen Bibliothek zugreifen. Ob oder inwieweit das sinnvoll ist, entscheiden die Umstände. In der Regel sind die benötigten Dialoge jedoch in der selben Bibliothek zu finden.

Und so aktivieren Sie einen Dialog, der mit dem Dialog-Designer erzeugt wurde. Der Name des Dialoges ist „dlg_test“ (ist auch der Name des Dialog-Moduls in der IDE), die Bibliothek heisst „CookBook“:

```
dim oDlg as variant
dim oBib as variant
dim oDlgModul as variant

DialogLibraries.loadLibrary("CookBook")    'laden der Dialog-Bibliothek
oBib = DialogLibraries.getByName("CookBook") 'das Objekt der Bibliothek
oDlgModul = oBib.getByName("dlg_warten")     'das Modul des Dialoges
oDlg = createUnoDialog(oDlgModul)           'der Dialog - als Objekt "UnoDialogControl"
```

In der Variablen oDlg ist nun das Objekt des Dialoges gespeichert – und kann entsprechend bearbeitet und verändert werden. Das Objekt wird erzeugt aus den Informationen und Daten der *.xdl Datei. Werden die Objektdaten nun verändert, so erfolgt keine Rückschreibung in die *.xdl Datei – jede Änderung geht also verloren. Wird der Dialog neu aufgebaut, so werden wieder die ursprünglichen Daten verwendet.

Die oben dargestellten vier Zeilen lassen sich leicht auf zwei reduzieren – das einzelne Zwischenspeichern der Objekte kann entfallen und mit Hilfe der Punktnotation direkt aufgerufen werden:

```
DialogLibraries.loadLibrary("CookBook")    'laden der Dialog-Bibliothek
oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_warten) 'der Dialog "UnoDialogControl"
```

Dieser zweite Weg ist vorzuziehen, es sei denn, einzelne Begriffe (wie z.B. der Name der Bibliothek) werden über eine Variable übergeben. In diesem Fall wäre nur der erste Weg machbar.

In der Variablen oDlg ist das Objekt des Dialoges nun gespeichert – der/die Benutzer/in sieht aber noch nichts. Nachdem also der Dialog erzeugt wurde, sollte er nun im Detail zunächst angepasst werden – das bedeutet, alle noch nicht feststehenden Text- und Label-Felder werden gefüllt, ebenso alle Control-Elemente vordefiniert, soweit dies nicht bereits beim Design erfolgt ist. Erst wenn alles fertig ist, sollte der Dialog für den/die Benutzer/in freigegeben und sichtbar geschaltet werden.

Beispiel einer Dialoganpassung:

```
   REM Abteilungsliste auslesen
   aAbtCtl = array("lst_2fabt","lst_3abt1", "lst_3abt2", "lst_4abt", "lst_5abt", "lst_6abt")
'Liste der Abteilungscontrols
```

```
aAbtListe = db_sql.getAbteilungsListeAlle()
REM Dialog erzeugen
oDlg = createUnodialog(dialogLibraries.MAK_140.dlg_mak140)
with oDlg   'vorladen diverser Parameter
  .model.title = .model.title & " - " & sVersNr
  .model.step = 8   'Startbildschirm
  .getControl("lbl_version").text = sVersNr
  for i = 0 to uBound(aAbtCtl)
    .getControl(aAbtCtl(i)).model.stringItemList = aAbtListe
  next
end with
```

Der Dialog wird nun ausgeführt. In der Regel dient ein Dialog dazu, dem/der Benutzer/in Gelegenheit zu geben, Informationen zu erhalten bzw. Informationen einzugeben, die dann weiterbearbeitet werden. Das Makro „wartet“ also, bis der/die Benutzer/in mit dem Dialog „fertig“ ist – und arbeitet erst dann weiter. Im einfachsten Fall wird der Dialog dann aufgerufen:

```
oDlg.execute()
```

Der Dialog wird nun als „View“ erzeugt und angezeigt und die Bearbeitung des Makros bleibt genau an dieser Zeile stehen, bis der Dialog geschlossen wird („Schließen-Kreuz“, Buttons „OK“ oder „Abbrechen“ etc.).

Anschließend verarbeitet das Makro alle weiteren Zeilen bis zum „end sub“ oder „end function“. Der Dialog ist zwar nicht mehr sichtbar, als Objekt aber natürlich nach wie vor vorhanden und auf dieses kann auch problemlos zugegriffen werden. Es lassen sich also alle Eingaben auslesen, ebenso wie die letzten aktiven Control-Elemente oder andere Eigenschaften.

```
n = oDlg.execute()
```

Die Methode „execute()“ liefert allerdings auch einen Rückgabewert – der kann zusätzlich dazu dienen, festzustellen, wie der Dialog beendet wurde. Dabei bedeuten:

1 – beendet mit „OK“

0 – beendet mit „Abbrechen“ bzw. mit dem „Schließen-Kreuz“

„OK“ und „Abbrechen“ sind dabei spezielle vordefinierte Eigenschaften, die einem Button zugeordnet werden können (im Design-Modus)

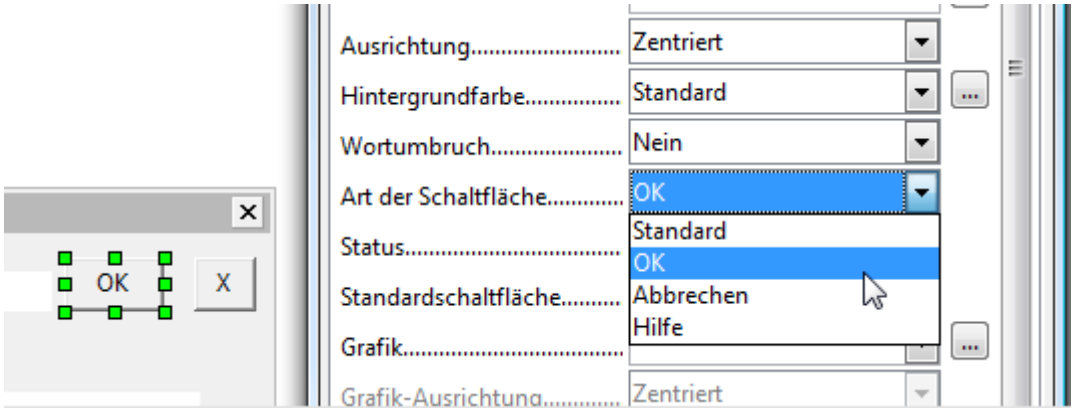


Abbildung 6.2: Schaltfläche - Typzuweisung

Wird die Vorgabe „Standard“ gewählt, dann muss man selbst ein Makro dem Ereignis „Auslösen“ zuordnen – sonst passiert gar nichts beim Klick auf den Button. Sowohl „OK“ als auch „Abbrechen“ beenden den Dialog – mit entsprechenden Rückgabewerten – „Hilfe“ ruft die URL auf, die in den Eigenschaften „Hilfe URL“ definiert wurde.

Wird ein reiner Informationsdialog verwendet mit einer einzigen Schaltfläche, so ist sowohl die Beschriftung als auch die zugeordnete „Art der Schaltfläche“ egal, solange bei dieser entweder „OK“ oder „Abbrechen“ gewählt wurde. Für die Fortsetzung des Makros ist es unerheblich, welche Wahl getroffen wurde – es muss nur weitergehen:

```
Sub WartedialogTest
  dim oDlg as variant

  DialogLibraries.loadLibrary("CookBook")    'laden der Dialog-Bibliothek
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_hinweis)
  odlg.getControl("lbl_1").text = "Achtung:" & chr(13) & "Dies ist ein sehr wichtiger Hinweis!"
  oDlg.execute()      'Dialog ausführen
  'REM Codeverarbeitung wird jetzt hier fortgeführt
  '...
end sub
```

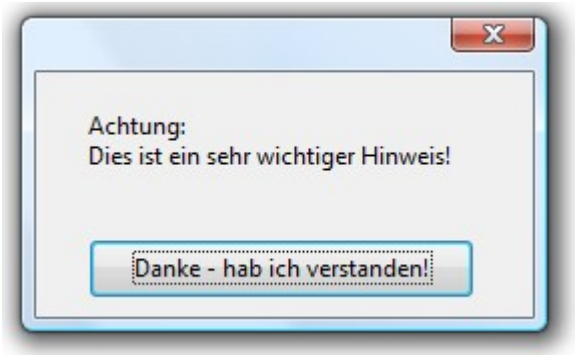


Abbildung 6.3: Eine Hinweis-Box

Auch das Abbrechen über das Schließen-Kreuz ändert am Fortgang des Makros nichts – auch hier „verschwindet“ lediglich der Dialog.

Eine „echte“ Wahl hingegen erzielt man über die Auswertung der „Art des Schließens“ – wie das folgende Beispiel zeigt:

```
Sub HinweisdialogTest
  dim oDlg as variant

  DialogLibraries.loadLibrary("CookBook")   'laden der Dialog-Bibliothek
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_hinweis)
  odlg.getControl("lbl_1").text = "Achtung:" & chr(13) & _
       "Dies ist ein sehr wichtiger Hinweis!" & chr(13) & "Makro wirklich fortsetzen?"
  if NOT (oDlg.execute() = 1) then exit sub    'Dialog ausführen und Ende bei Abbruch
  'Codeverarbeitung wird jetzt hier fortgeführt
  'wenn Dialog mit "OK" beendet wurde...
  '...
end sub
```

Abbildung 6.4: Dialog – Rückgabewerte

Nochmals der Hinweis:

Ein mit „execute()“ aufgerufener Dialog „blockiert“ den aktiven Controller – das heisst, weder das darunterliegende Makro „läuft“ weiter, noch ist es möglich, in der aktiven „Parent-Applikation“ (also dem Fenster, aus dem das Makro aufgerufen wurde) irgendeine Änderung vorzunehmen.

Wohl aber lassen sich andere, bereits gestartete OOo-Prozesse aktivieren, also zum Beispiel ein schon geöffnetes anderes Dokument – und dort können beliebige Eingaben gemacht oder Prozesse gestartet werden.

Dies trifft dann leider aber auch für das Makro zu. Als Beispiel: In OpenOffice ist eine Extension installiert, die eine Startfunktion für einen Dialog in einer eigenen Symbolleiste in Writer bereitstellt. Es sind zwei Writer-Dokumente aktuell geöffnet – Test1 und Test2. Beide laufen jetzt ja in eigenen Fenstern und haben einen eigenen aktuellen Controller. In Test1 wird nun das Makro gestartet und der Dialog wird geöffnet. Der/Die Benutzer/in kann nun nicht auf das Dokument „Test1“ zugreifen – der Controller ist blockiert – wohl aber kann er/sie auf dem Desktop das Dokument „Test2“ aktivieren (in den Vordergrund holen) und nun darin arbeiten

und auch das Makro erneut starten – der Controller des Dokumentes ist ja noch aktiv. Da Basic aber nicht multitask-fähig ist, die Variablen somit doppelt belegt sind, ist das Chaos vorgezeichnet – im Ergebnis wird es zu einem „Absturz“ führen.

Gerade bei Extensions ist dieses Szenerium gar nicht so abwegig: Man „vergisst“, dass das Makro noch läuft und startet es ein zweites Mal.

6.2.1 Doppelstart verhindern

Sinnvoll verhindern lässt sich das nur mit eigenem Code. Ist die Situation möglich, dass ein Makro doppelt gestartet wird – zum Beispiel aus verschiedenen Dokumenten heraus – und besteht die Gefahr, dass zum Beispiel ein Makro steht (dialog.execute()), das andere aber weiterläuft, **muss** der/die Programmierer/in den Doppelstart verhindern.

In der Regel gibt es eindeutige Startfunktionen für jedes Makro. Um einen Doppelstart zu verhindern, muss geprüft werden, ob das Makro bereits läuft. Hierfür gibt es aber leider keine eingebaute, eindeutige Prüfprozedur, man muss also selbst einen „Flag“ setzen, der besagt, dass das Makro bereits läuft. Dies passiert beispielsweise durch das Setzen einer Boolschen Variablen (global bibliotheksweit) und das Abfragen derselben. Ist der Wert gesetzt, darf das Makro nicht noch einmal starten. Andererseits werden zu jedem Start die Variablen neu initialisiert – und somit nach Ablauf des Makros auch zurückgesetzt. Das Vorgehen hat sich bewährt:

```
public bMakroRunFlag as boolean

sub Start_MeinMakro_Main
  if  bMakroRunFlag then
    msgbox ("Sorry, eine Instanz des Makros läuft bereits." & chr(13) & _
            "Ein Doppelstart ist nicht möglich!", 64, "Start nicht möglich...")
    exit sub
  else
     bMakroRunFlag = true
  end if
  ...
```

Gerade in verschachtelten Strukturen sind weitere Prüfprozeduren nötig, wenn zum Beispiel aus einem Dialog heraus ein anderer Dialog gestartet wird – auch hier ist es technisch möglich, den Child-Dialog mehrfach zu starten – dies muss aber ebenfalls verhindert werden.

Prüfstrukturen sind insbesondere wichtig, wenn der Start des Dialoges den aktiven Controller erst gar nicht stoppt (siehe hierzu auch Kapitel 6.3). Dann wäre theoretisch der Doppelstart auch im selben Prozess möglich – mit gleichen chaotischen Folgen.

6.2.2 Dialog beenden

Auch wenn der Controller „stoppt“ und die Ausführung des Makros stoppt, während der Dialog aktiv ist, ist es dennoch möglich, andere Makros zu starten und auszuführen. Jedem Kontrollelement kann ja zu einem Ereignis auch ein Makro zugeordnet werden – und dieses wird auch ausgeführt.

Insofern kann der Dialog auch programmatisch jederzeit beendet werden – hierzu genügt die Funktion „endExecute()“. Allerdings muss die Variable des Dialoges global definiert sein – sonst würde ein Fehler auftreten.

```
sub DialogEnde
  oDlg.endExecute()
end sub
```

Eine solche Funktion, verknüpft mit einem Button, bewirkt das gleiche, wie wenn der Button als Typ „OK“ bekommen hätte – mit dem Vorteil, dass man hier selbst zusätzliche Aufgaben unterbringen könnte, zum Beispiel die Prüfung von Eingaben oder ähnliches, und das Beenden vom Ergebnis der Prüfung abhängig machen kann.

Ein Beispiel wäre ein Passwort-Prüf-Dialog, siehe hierzu auch Kapitel 6.8.2.

Im Zusammenhang mit diesen „Beenden-Strukturen“ muss man sich aber immer im Klaren sein, wie die Prozesse intern ablaufen. Mit der Methode „endExecute()“ wird zunächst der Dialog „invisible“ gestellt – er verschwindet von der Benutzeroberfläche – und zwar unmittelbar beim Aufruf der Methode „endExecute()“. Dennoch „läuft“ das ursprüngliche Startmakro nicht unmittelbar weiter, sondern zunächst wird der „Tochterprozess“ – also das Makro mit der „endExecute()“ Anweisung – beendet, dann erfolgt der Rücksprung auf den „Hauptprozess“, also zu der Anweisung „execute()“, und das Makro wird dort fortgesetzt. Verschachtelt man mehrere solche Prozesse (mit mehreren unabhängigen Dialogen), sollte man sich immer sehr genau bewusst sein, wie die Prozesse intern ablaufen.

Verschachtelte Dialoge werden auf der Benutzeroberfläche auch in ihrer Reihenfolge abgebildet – ein Tochterprozess schwebt immer über dem Vaterprozess – solange der Tochterprozess aktiv ist, ist der Vaterprozess blockiert – Eingaben sind nicht möglich und Schaltflächen nicht aktiv.

6.3 „Schwebende“ Dialoge

Wurde ein Dialog bisher aufgerufen über die „execute()“ Methode, so gibt es eine weitere Methode, den Dialog dem/der Benutzer/in sicht- und nutzbar zu machen: Das Objekt des Dialoges wird nicht ausgeführt, sondern „nur“ sichtbar geschaltet. „Sichtbar“ heißt, der Dialog „schwebt“ als Tochterprozess im Vordergrund der aktiven Anwendung (des aktiven Dokumentes), ohne den Controller aber zu blockieren. Der/Die Benutzer/in kann sowohl im Dialog als auch im Dokument arbeiten – das Makro läuft im Hintergrund weiter.

Die Vorteile liegen auf der Hand – man kann dem/der Benutzer/in direkte, laufende Informationen liefern, der/die Benutzer/in kann im Dokument weiterarbeiten etc.

Man darf allerdings nicht vergessen:

- Der Prozess ist ein Tochterprozess des aktuellen Prozesses (Dokumentes). Wird das Dokument inaktiv (durch Aktivierung eines anderen Dokuments), so wird auch der Dialog inaktiv – und umgekehrt.
- Ein nur sichtbar geschalteter Dialog ist nicht wirklich lange lebensfähig – das Makro läuft ja im Hintergrund weiter – und spätestens bei dem Erreichen der „end sub“-Anweisung – also dem Ende des Makros – werden alle Makroprozesse beendet, also auch der Dialog.

Während der erste Punkt in der Regel auch so gewünscht wird, ist der zweite eher unerwünscht. Hier muss also der/die Programmierer/in Sorge tragen, dass der Prozess solange erhalten bleibt, wie der Dialog benötigt wird.

In der Regel löst man das mit einem Flag (Boolsche Variable), dessen Wert umgeschaltet wird, wenn der Dialog geschlossen wird (eigene Funktion). Über eine „Endlosschleife“ im Hauptprozess wird der Makroablauf künstlich am Leben gehalten – solange der Dialog noch offen ist. Das folgende Beispiel zeigt einen kleinen Dialog, der regelmäßig die aktuelle Anzahl der Wörter in einem Textdokument anzeigt:

```
public oDlg as variant
public bDlgFlag as boolean

Sub HinweisdialogTest
  dim oDoc as variant

  oDoc = thisComponent  'aktuelles Dokument

  DialogLibraries.loadLibrary("CookBook")   'laden der Dialog-Bibliothek
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_hinweis)

  bDlgFlag = true
  oDlg.setVisible(true)  'Dialog sichtbar schalten

  REM Schleife für die Wartezeit
  Do while bDlgFlag
     with oDlg
      .getControl("lbl_num").text = odoc.WordCount
     end with
     wait(100) 'eine 10tel Sekunde warten
  loop

  oDlg.setVisible(false)  'Dialog unsichtbar schalten

End Sub


sub Dlg_Ende
   bDlgFlag = false
```

```
end sub
```

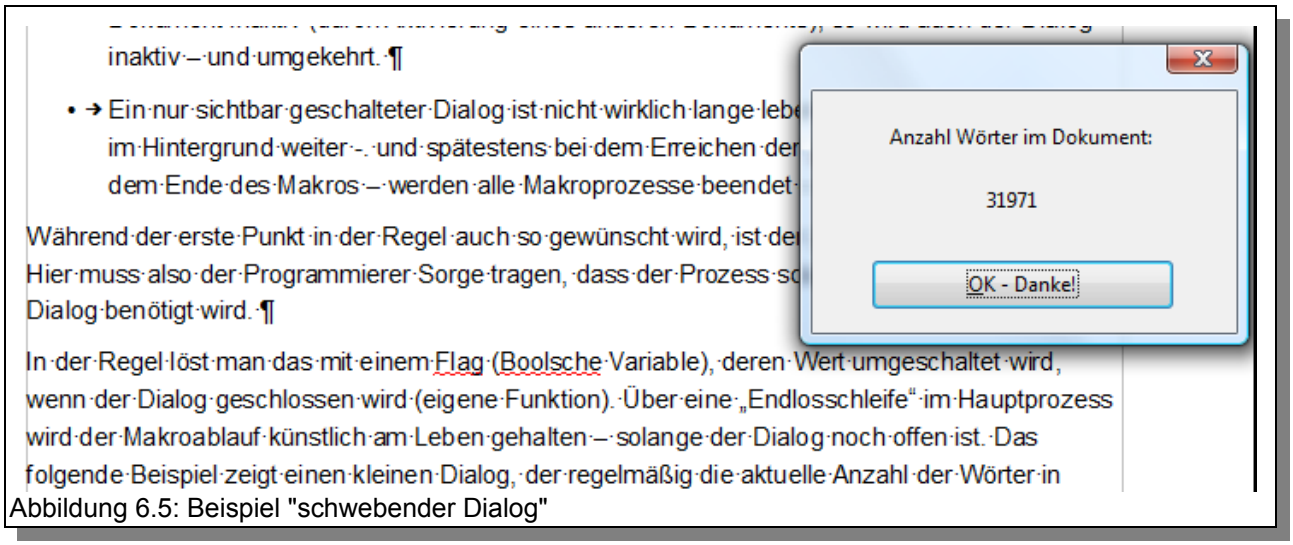


Abbildung 6.5: Beispiel "schwebender Dialog"

Während also im Dokument gearbeitet wird, informiert der Dialog über die Anzahl der Wörter – und aktualisiert sich alle 1/10 Sekunde.

Wichtig:

Hier, wie in fast allen verwendeten Dialogen: Die Variable, die den Dialog als Objekt aufnimmt, muss global definiert werden – nur so ist gewährleistet, dass Funktionen, die im Dialog aufgerufen werden und selbst wieder auf den Dialog zugreifen, überhaupt funktionieren.

Zwar ist Basic an sich nicht multitask-fähig, OOo kann aber sehr wohl mehrere Prozesse gleichzeitig starten – auch wenn sie sich auf die gleiche Bibliothek und auf das gleiche Modul beziehen.

Achtung!

Bei nur „Visible" geschalteten Dialogen ist der/die Programmierer/in selbst verantwortlich für alle Aktivitäten des Fensters/Dialogs. Auch das „Schließen-Kreuz" ist nicht mehr mit einer Funktion verknüpft – das heißt, es funktioniert nicht direkt. Zwar liefert das Fenster (eben der Dialog) ein entsprechendes Ereignis, nur muss man dies nun selbst abfangen und eine entsprechende Funktion schreiben – das bedeutet hier einen Listener registrieren und entsprechend auf das Ereignis „close" reagieren.

Ein lediglich mit „setVisible(true)" aktivierter Dialog kann natürlich genauso wieder programmatisch mit „setVisible(false)" ausgeblendet werden – er bleibt dann als Objekt erhalten und kann erneut eingeblendet werden.

„Visible" geschaltete Dialoge werden immer dann benötigt, wenn parallel zum neuen Dialog auch im Dokument oder im Hauptdialog gearbeitet werden soll.

6.4 Mehrstufige Dialoge

Zwar ist es möglich, mehrere Dialoge auf der Benutzeroberfläche aufzurufen (sie sozusagen zu verschachteln) – dies ist aber vom Arbeitsablauf her oft nicht nötig. Dialoge erfüllen einzelne Aufgaben – und der Grundsatz „Ein Dialog für eine Aufgabe“ ist sicher gut und zu beherzigen. Das heißt aber nicht, dass jeweils tatsächlich ein (physikalischer) Dialog für jede einzelne Aufgabe gebraucht wird. Man kann sich ohne weiteres auf eine Oberfläche einigen und dort dann – aufgabenbezogen – die entsprechenden Kontrollelemente einblenden. Das erleichtert dem/der Benutzer/in die Handhabung und stellt ein einheitliches Design (Rahmen) dar.

In diesem Fall spricht man von **mehrstufigen Dialogen**. Allen mehrstufigen Dialogen ist jedoch ein Merkmal gemeinsam: Es bedarf einer für den/die Benutzer/in erkennbaren Steuerungsstruktur für die einzelnen Stufen – und der Ablauf ist nicht linear und nur in eine Richtung möglich, sondern der/die Benutzer/in kann auch wieder zurückschalten.

Zwar wäre es technisch möglich, alle Kontrollelemente im Dialogmodell (in der IDE) auf einer Ebene zu platzieren und je nach Bedarf die einzelnen Elemente ein- bzw. auszublenden, dies wäre aber zu viel Aufwand und sehr fehleranfällig.

Das Modell des Dialoges bietet hingegen eine einfache Möglichkeit – das Step-Modell. So wird jedem Kontrollelement zusätzlich eine „Step“-Eigenschaft übergeben – und der Controller wird später automatisch nur die Elemente einblenden, deren Step-Eigenschaft mit der des aktuellen Steps übereinstimmt.

Bereits beim Design des Dialoges kann man die Ebenen-Eigenschaft recht gut nutzen – wählt man den Dialog an sich aus (Klick auf den Dialograndl), kann man in den Eigenschaften die aktuelle Ebene (unglücklich gewählter Name „Seite“ (Step)) bereits einstellen. Das Designmodell zeigt dann auch nur diese „Seite“ an und alle darauf befindlichen Kontrollelemente – und alle neu hinzugefügten erhalten automatisch diese „Seite“ als Vorgabe.

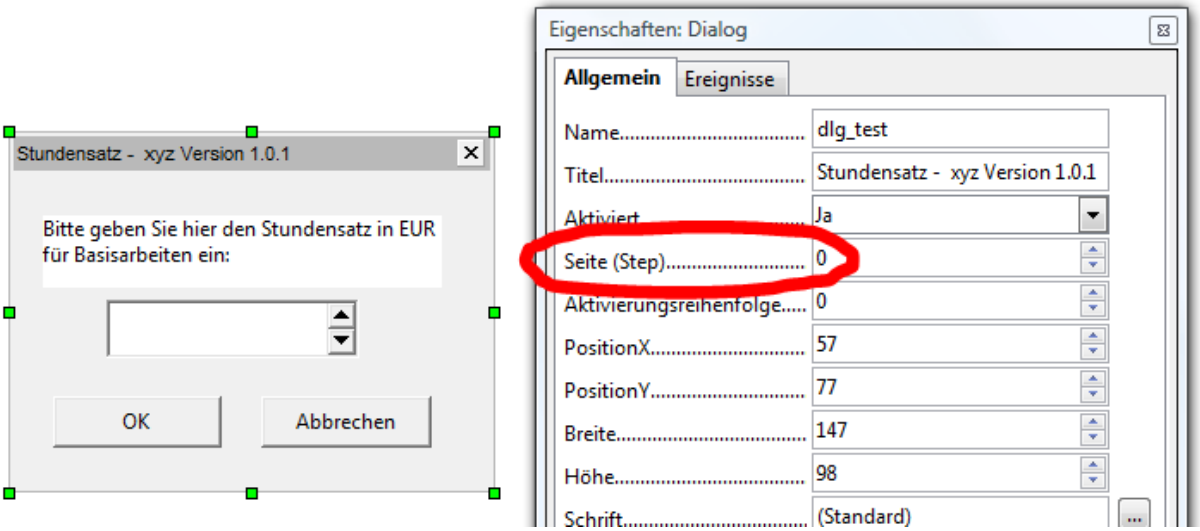


Abbildung 6.6: Step-Eigenschaft

Ich werde im folgenden eher von „Ebene“ oder „Step“ reden als von „Seite“, gemeint ist aber immer diese Eigenschaft.

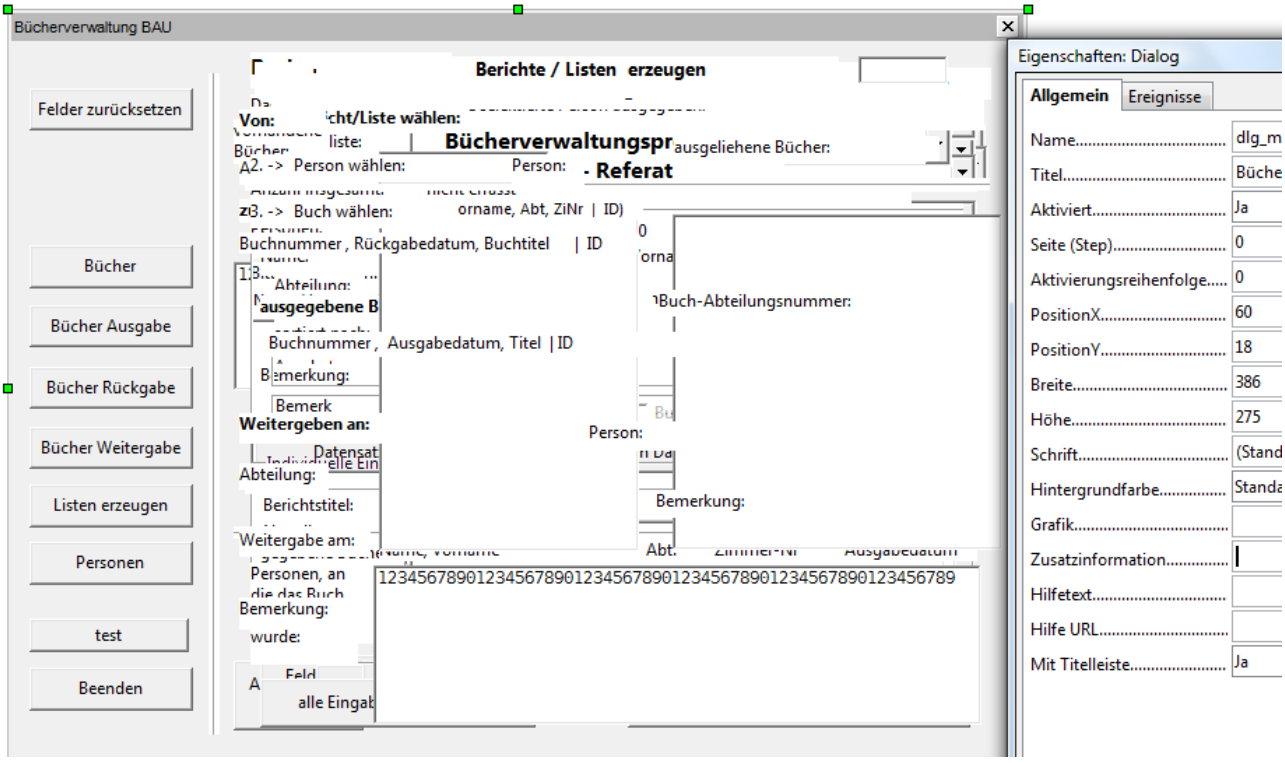
Wird ein neues Dialogmodell erzeugt (neues Modul), so ist die Vorgabe immer Step 0. Diese nullte Ebene (Step 0) hat eine Besonderheit: Alle Elemente auf dieser Ebene werden immer angezeigt – egal auf welcher Ebene sich der Dialog aktuell befindet. Es ist quasi die „Hintergrundebene“.

Dieser Step 0 eignet sich also ideal, um die Navigation sowie die Elemente unterzubringen, die immer sichtbar sein müssen.

Arbeiten Sie mit mehrstufigen Dialogen, so achten Sie darauf, wirklich nur die Elemente auf Step 0 unterzubringen, die immer sichtbar sein sollen – und beginnen Sie mit diesen!

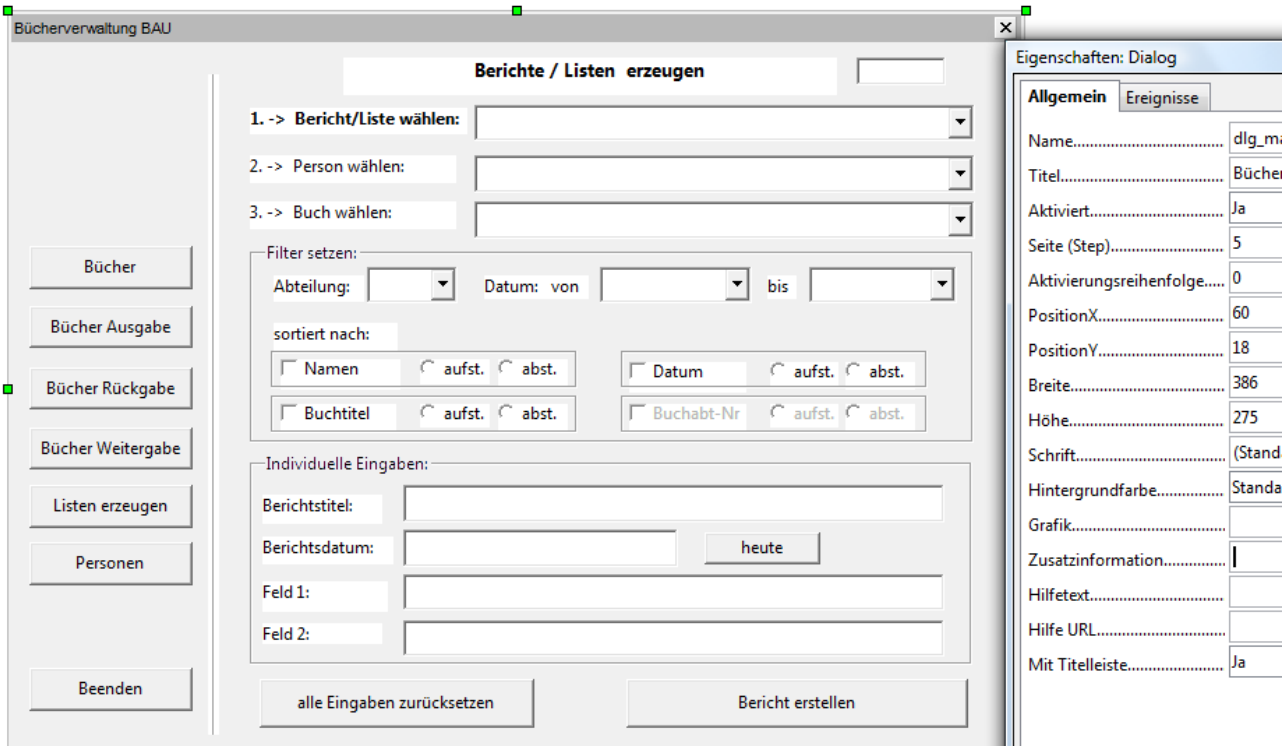
Schalten Sie später einmal auf Ebene 0, so sehen sie alle Kontrollelemente des Dialoges – egal welche Step-Eigenschaft sie besitzen. Auf Seite 0 sind alle Elemente sichtbar – entsprechend ist das „Chaos“ vorprogrammiert.

Beispiel eines Dialoges mit 10 Steps – aktiviert ist Stufe 0:



Einzig sauber sichtbar sind die Navigationselemente auf der linken Seite (alle Kontrollelemente Step 0) sowie der Trennstrich. Die Überlagerung aller anderen Elemente lässt keine Zuordnung mehr zu.

Ganz anders sieht die Sache auf einem festen „Step“ aus – hier Step 5:



Alle Elemente sind klar zu erkennen – so, wie sie der/die Benutzer/in später sieht.

Die Step-Eigenschaft wird im Betrieb durch die Buttons links „umgestellt“ und somit dem/der Benutzer/in die entsprechenden Oberflächen angeboten. Das „Umschalten“ erledigt eine entsprechende Funktion, die mit allen Buttons verbunden ist.

```
Sub MAK140_dlgButton
  dim i%

  REM aktiven Button suchen
  for i = 1 to 6
    if oDlg.getControl("cmd_0" & i).hasFocus() then exit for
  next i

  REM Prüfen, ob Änderungen vorhanden
  if bChangeflag then
    if MAK140_Helper1.MAK140_ChangeAbbruch then exit sub
  end if

  Select case i
    case 1  'Bücher
      code_step1.MAK140_Step1_Start
    case 2  'Bücher-Rückgabe
      code_step2.MAK140_Step2_Start
    case 3  'Bücher Weitergabe
      code_step3.MAK140_Step3_Start
    case 4  'Bücher Ausgabe
      code_step3.MAK140_Step4_Start
    case 5  'Listen erzeugen
      code_step5.MAK140_Step5_Start
    case 6  'Personen
      code_step6.MAK140_Step6_Start
```

```
    end select
   oDlg.model.step = i
 end sub
```

Die Buttons haben Namen, die auf ihre zu schaltende Step-Eigenschaft hinweisen – also hier „cmd_03“ für Step 3. Über die „For-Schleife“ wird zunächst geprüft, welcher Button eigentlich gedrückt wurde – dieser besitzt aktuell den Fokus – dann wird auf den entsprechenden Step umgeschaltet, wobei zunächst die Initialisierungen für den jeweiligen Step durchgeführt werden (eigenständige Funktionen, die entsprechend aufgerufen werden).

Erst nach der Initialisierung wird auf den entsprechenden Step umgestellt (`oDlg.model.step = i)`

Mehrstufige Dialoge werden in größeren Projekten fast immer benötigt!

Achtung!

Der „bloße“ Aufruf des Dialoges (execute()) lässt den Dialog in dem Step starten, der in der IDE zuletzt angezeigt wurde! Wenn man also mit mehrstufigen Dialogen arbeitet, muss vor dem Ausführen (execute oder visible) unbedingt der Step wunschgemäß eingestellt werden! Über die Eigenschaft: oDlg.model.step = 2 – die Step-Eigenschaft existiert nur im Modell.

6.4.1 Roadmap-Element

Bei mehrstufigen Dialogen bietet es sich an, ein sogenanntes „Roadmap“-Kontrollelement zu verwenden, das dem/der Benutzer/in bei einer linearen Aufgabenkette die einzelnen Stufen anzeigt und die entsprechende Navigation zulässt.

Das entsprechende Roadmap-Kontrollelement ist leider nicht direkt im Dialogeditor erzeugbar (per Drag&Drop) sondern muss nachträglich per Code integriert werden – dennoch ist es ein sinnvolles Element.

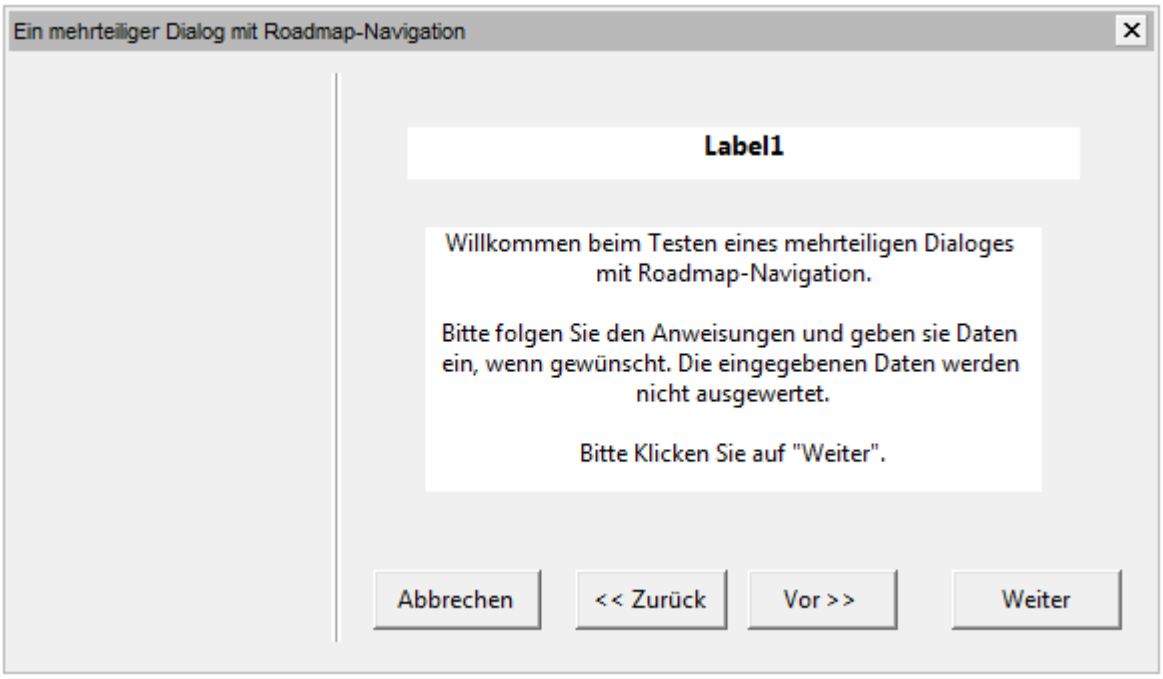
Die Vorgehensweise ist dann zweistufig:

Zunächst erzeugt man im Dialogeditor einen entsprechenden mehrstufigen Dialog, für jede Aufgabe wird ein Step genutzt. Im Dialog selbst wird ausreichend Platz gelassen, um das Roadmap-Element zu platzieren – und zwar auf jedem Step an der gleichen Stelle!

Im Code wird dann der Dialog zunächst als Objekt erzeugt und erhält schließlich dann erst das Roadmap-Kontrollelement mit allen seinen Eigenschaften.

Jetzt ist der Dialog einsatzbereit und wird angezeigt.

Das folgende Beispiel zeigt einen solchen Einsatz. Zunächst wird der Dialog designed – zu sehen ist hier Step 1 – und links ist Platz für das später hinzuzufügende Roadmap-Kontrollelement:



Der Code besteht nun aus diversen Teilen: Zunächst wird sowohl der Dialog als auch das Dialog-Model erzeugt und zwischengespeichert, dann wird die Roadmap-Navigation aufgebaut:

```
public oDlg as object
public oDlgModel as Object


sub mehrteiligenDialogAnzeigen

	DialogLibraries.LoadLibrary("Standard")
	oDlg = createUnoDialog(DialogLibraries.Standard.Dialog)
	oDlgModel = oDlg.getModel()
	InitializeRoadmap
	Schritt1
	oDlg.execute
End sub
```

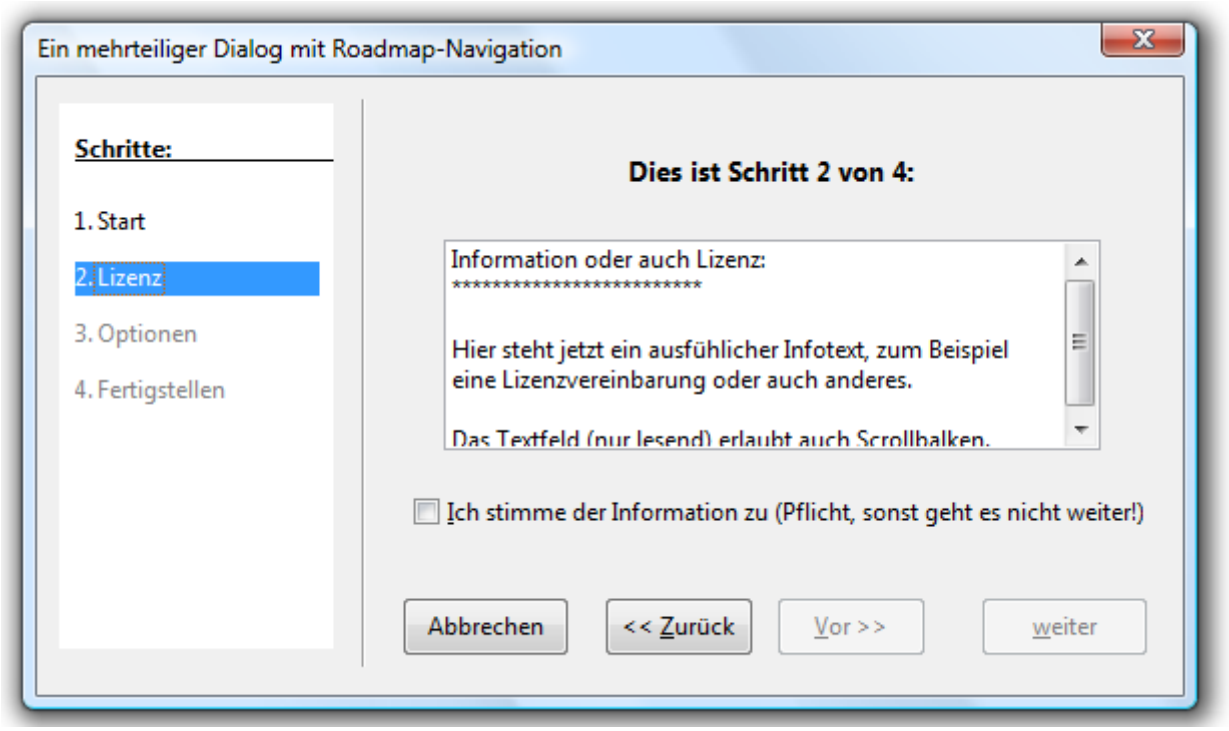
Nun zu der Erzeugung des Roadmap-Controls:

```
REM Roadmap Code
Sub InitializeRoadmap
	Dim oRoadmapControl As Object
	Dim oRoadmapModel As Object
	Dim oListener As Object

	REM Das Roadmap-Control erzeugen
	oRoadmapModel = oDlgModel.createInstance("com.sun.star.awt.UnoControlRoadmapModel")
	oDlgModel.insertByName( "Roadmap", oRoadmapModel )
	With oRoadmapModel
		.Step = 0
		.PositionX = 6
		.PositionY = 7
```

```
                .Width = 73
                .Height = 145
                .Text = "Schritte:"
                .Name = "Roadmap"
                .TabIndex = 1
        End With

        REM Roadmap-Einträge vornehmen
        InsertRoadmapItem(0, 1, "Start", True)
        InsertRoadmapItem(1, 2, "Lizenz", True)
        InsertRoadmapItem(2, 3, "Optionen", False)
        InsertRoadmapItem(3, 4, "Fertigstellen", False)
        REM Listener auf Ereignisse registrieren
        oRoadmapControl = oDlg.getControl("Roadmap")
        oListener = CreateUnoListener("Roadmap_", "com.sun.star.awt.XItemListener")
        oRoadmapControl.addItemListener(oListener)

End Sub

REM Einen Eintrag in die Roadmap integrieren
Sub InsertRoadmapItem(iPos As Integer, nID As Integer, sLabel As String, bEnable As Boolean)
        Dim oRoadmapModel As Object
        Dim oRoadmapItem As Object

        oRoadmapModel = oDlgModel.Roadmap
        oRoadmapItem = oRoadmapModel.createInstance("com.sun.star.awt.RoadmapItem")
        With oRoadmapItem
                .ID = nID
                .Label = sLabel
                .Enabled = bEnable
        End With
        oRoadmapModel.insertByIndex(iPos, oRoadmapItem)

End Sub

REM Auswahl über Roadmap
Sub Roadmap_itemStateChanged(oEvent)
        Select Case oEvent.ItemID
        Case 1
                Schritt1
        Case 2
                Schritt2
        Case 3
                Schritt3
        Case 4
                Schritt4
        end select
End Sub
```

Die Funktionen „Schritt1“ bis „Schritt4“ initialisieren dann die jeweiligen Steps des Dialoges und die entsprechenden Voreinstellungen. Die Codes werden hier nicht abgebildet.

Der Dialog im Einsatz sieht dann wie folgt aus (hier jetzt Step 2):



6.5 Dialoge zur Laufzeit verändern

Auch wenn der Dialog mit Hilfe der IDE grafisch designed und erzeugt wurde, so ist er doch nicht „festgemeißelt“, sondern kann problemlos zur Laufzeit geändert werden.

Zwar liefert die Hilfe in OOo die Aussage, dass alle Dialoge und deren Kontrollelemente intern über die Maßeinheit „ma“ (Map AppFont) definiert werden – dies ist auch die Maßeinheit aller Zahlenanzeigen bei den Größenwerten – doch leider hilft das dem Designer nur sehr wenig.

Die Einheit Map AppFont (ma) ist definiert als ein Achtel der durchschnittlichen Höhe eines Zeichens des durch das Betriebssystem definierten Systemschriftsatzes (system font) sowie einem Viertel seiner Breite. Durch die Nutzung der ma-Einheit soll sichergestellt sein, dass die Dialoge auf unterschiedlichen Systemen und bei unterschiedlichen Systemeinstellungen gleich aussehen. Soviel zur Theorie – die Praxis sieht leider anders aus.

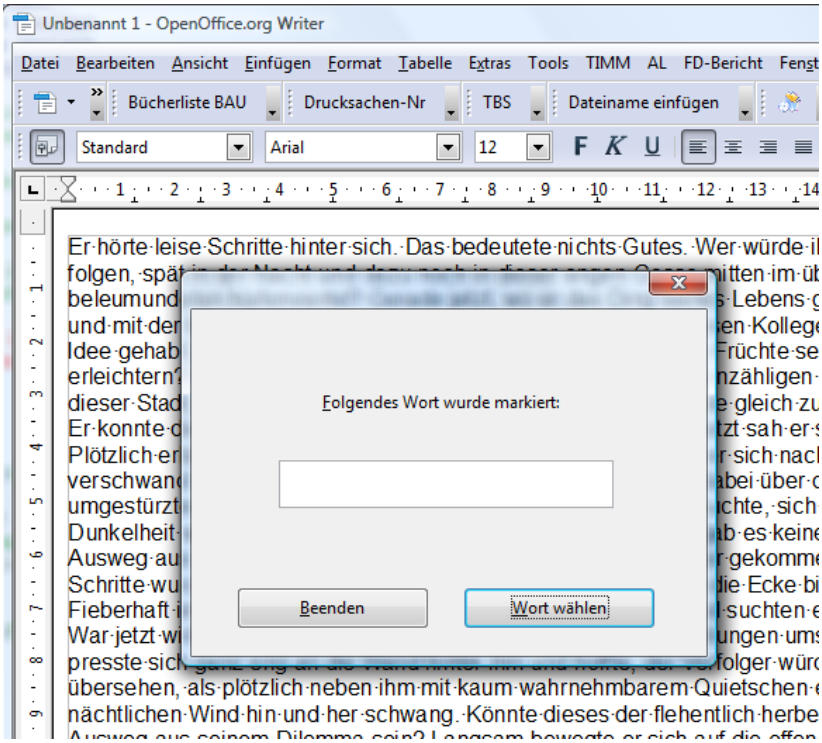
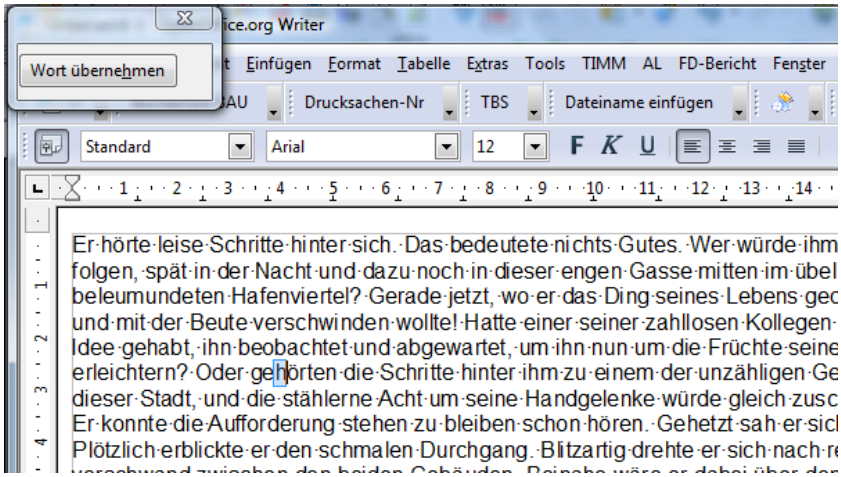
Jeder Dialog muss unbedingt auf allen verwendeten Systemen überprüft werden – und entsprechend optimiert werden. Dies geht meist nur durch Ausprobieren. Im Programm selbst müssen dann entsprechende Weichen eingebaut werden.

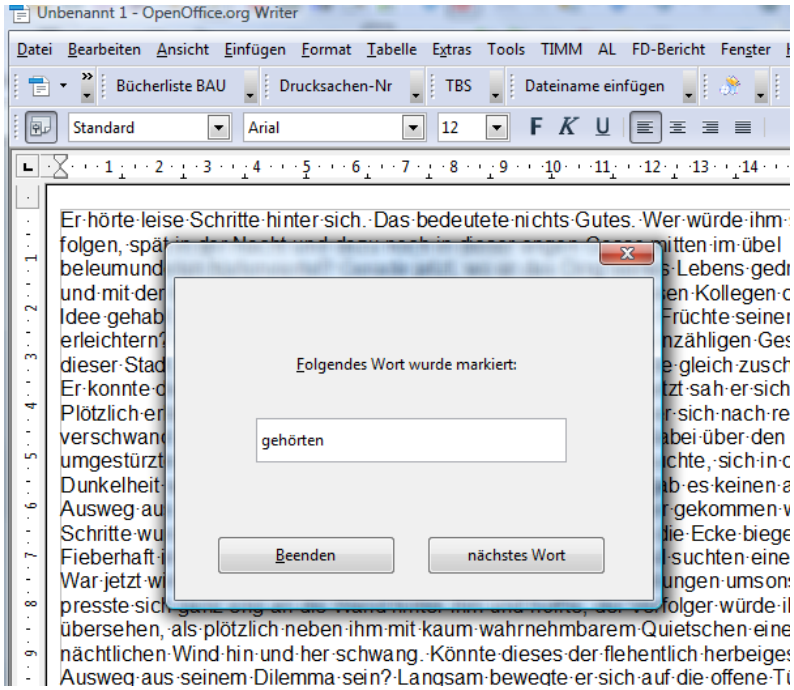
Doch nun zum praktischen Einsatz:

Ein Dialog soll einen Begriff aus dem aktuellen Dokument übernehmen und zwar den, den der/die Benutzer/in aktiv markiert hat, bzw. den, wo der Cursor gerade steht. Um das zu

gewährleisten, wird der Dialog auf Knopfdruck verkleinert – der/die Benutzer/in kann das Wort markieren bzw. den Cursor setzen – dann wird der Dialog wieder hergestellt.

Damit der Dialog das Bearbeiten des Dokumentes nicht verhindert, wird er nur „sichtbar“ gestellt.

Start des Makros:	
Verkleinern:	

Wort übernehmen:	

Der Code:

```
REM  *****  BASIC  *****
public oDlg as variant
public bDlgFlag as boolean
public oDlgSize as variant
public oDoc as variant

Public const iBreiteKlein = 140  'Breite kleiner Dialog
Public const iHoeheKlein = 40  'Höhe kleiner Dialog

Sub Start_DlgWortAufnahme
  oDoc = thisComponent  'aktuelles Dokument

  DialogLibraries.loadLibrary("CookBook")   'laden der Dialog-Bibliothek
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_hinweis)
  with oDlg
    .model.step = 1
    .getControl("cmd_1").label = "Wort wählen"
  end with

    bDlgFlag = true
  oDlg.setVisible(true)  'Dialog sichtbar schalten

  REM Schleife für die Wartezeit
  Do while bDlgFlag
     wait(500) 'eine 10tel Sekunde warten
  loop

  oDlg.setVisible(false)  'Dialog unsichtbar schalten

End Sub
```

```
' Ende Dialog
sub Dlg_Ende
   bDlgFlag = false
end sub

'Dialog verkleinern
sub Dialog_Klein
  oDlgSize = oDlg.getPosSize()  'Auslesen Größe und Position
  with oDlg
    .setPosSize(0, 0, iBreiteKlein, iHoeheKlein, 15)
    .model.step = 2
  end with

end sub

'Dialog wieder vergrößern
sub dialog_Gross
  dim oTCur as variant, oText as object

    oText = oDoc.getCurrentController().getView-Cursor().getText()
  oTCur = oText.createTextCursorByRange(oDoc.getCurrentController().getView-Cursor())
  oTCur.gotostartOfWord(false)
  oTCur.gotoendofWord(true)

  oDlg.getControl("txt_1").text = oTCur.string
  REM Dialog wieder vergrößern
  oDlg.model.step = 1
  oDlg.getControl("cmd_1").label = "nächstes Wort"
  oDlg.setPosSize(oDlgSize.X, oDlgSize.Y, oDlgSize.Width, oDlgSize.Height, 15)
end sub
```

Das Struct „PosSize“ besitzt die vier benötigten Werte: X-Position, Y-Position, Breite und Höhe des Dialoges – eben in der Einheit „MA“. Hier ist also immer etwas „Experimentieren„ angesagt – inklusive entsprechender Weichen für unterschiedliche Betriebssysteme.

Das könnte dann wie folgt aussehen:

```
…
Public const iBreiteKleinWin = 140  'Breite kleiner Dialog Windows
Public const iHoeheKleinWin = 40  'Höhe kleiner Dialog Windows
Public const iBreiteKleinUnix = 160  'Breite kleiner Dialog Linux/Unix
Public const iHoeheKleinUnix = 50  'Höhe kleiner Dialog Linux/Unix

…
'Dialog verkleinern
sub Dialog_Klein
  oDlgSize = oDlg.getPosSize()  'Auslesen Größe und Position
  with oDlg
    if GetGuiType() = 1 then     'Windows
     .setPosSize(0, 0, iBreiteKleinWin, iHoeheKleinWin, 15)
    elseif GetGuiType() = 4 Then      'Linux/Unix
     .setPosSize(0, 0, iBreiteKleinUnix, iHoeheKleinUnix, 15)
    else    'Standardwerte unbekanntes System
     .setPosSize(0, 0, 200, 100, 15)
    end if
    .model.step = 2
  end with
```

```
end sub
```

Ist die Größe des dargestellten Dialoges kleiner als die Fläche, die er benötigt, um alle Kontrollelemente anzuzeigen, so sind diese zwar existent (ähnlich wie auf einem anderen „Step“), nicht aber erreichbar und auch nicht sichtbar. Insofern lassen sich die Dialoge auch mit Hilfe der Größeneigenschaft flexibel einsetzen.

Noch eine Besonderheit:

Da ein Makro und damit auch ein Dialog stets ein Tochterprozess eines Dokumentes ist, bezieht sich die Positionsangabe des Dialoges immer relativ auf die Position des Hauptprozesses, also auf die linke obere Ecke des Dokumentes bzw. dessen Fenster. Will man den Dialog innerhalb des Bildschirms exakt positionieren, so muss man auch die Position des Fensters des Hauptprozesses auslesen und die Position aus der Mischung beider Daten berechnen.

Ein Fenster links oben in die Bildschirm-Ecke zu platzieren funktioniert meist aber auch einfacher (zumindest sicher unter Windows): Da das Fenster keinesfalls weiter als eben auf die Position 0, 0 herausgeschoben werden kann, ist es möglich, den Positionsangaben (X-Position, Y-Position) des Dialogfensters einfach hohe negative Werte zu übergeben:

```
oDlg.setPosSize(-500, -500, 0, 0, 3)
```

Dadurch wird der Dialog nach links und nach oben verschoben – eigentlich um 500 Pt jeweils – ist aber das Ende vorher erreicht, geht es auch nicht weiter. Der Dialog „sitzt“ jetzt oben links.

Hinweis:

In OOo 3.2.1 gibt es noch diverse Bugs bei Dialogen. So funktioniert beispielsweise die Sichtbarkeit von Buttons nicht zuverlässig. Eine Anweisung wie `„odlg.getControl(„cmd_1“).setVisible(true)“` bewirkt leider nichts – obwohl sie eigentlich einen Button sichtbar schalten sollte. In Folgeversionen ist der Bug behoben. Bis zur 3.2.1 muss man sich mit der Step-Eigenschaft behelfen und Schalter auf unterschiedlichen Steps unterbringen.

6.6 Vordefinierte Dialoge

Nicht alle Dialoge muss man selbst neu erstellen – es ist möglich, auf einige vordefinierte Dialoge zuzugreifen. Während die msgBox in Basic eingebaut ist, liefert OOo zwei interne vordefinierte Dialoge zur Auswahl: einen Dateiauswahl-Dialog (zum Öffnen oder Speichern einer Datei) sowie einen Verzeichnisauswahl-Dialog.

Genauer gesagt sind es keine „echten“ eigenen Dialoge, sondern Code-Sequenzen, die die betriebssystemeigenen verwenden. Dies entspricht dann den „Look & Feel“-Anforderungen des Betriebssystems und der/die Nutzer/in muss sich nicht umgewöhnen. Zur Sicherheit allerdings

hat OOo dennoch eigene Dialoge eingebaut, die alternativ genutzt werden können – in der UI muss hierzu die Option „Extras/Optionen/OpenOffice.org/Allgemein → OpenOffice.org Dialoge verwenden“ aktiviert werden, um diese Dialoge zu nutzen, im Makro-Code können Sie diese auch unabhängig nutzen.

Der Zugriff auf die Dialoge ist definiert im Modul com.sun.star.ui.dialogs, wobei es zwei interessante Services dort gibt: **FolderPicker** und **FilePicker**.

6.6.1 Verzeichnisauswahl-Dialog (FolderPicker)

Der Service FolderPicker spezifiziert dabei den Dialog, um ein Verzeichnis auszuwählen. Wichtig zu wissen ist, dass in der Standardeinstellung von OpenOffice.org diese Dialoge vom jeweiligen Betriebssystem (genauer gesagt von der grafischen Oberfläche, der GUI) genutzt werden, soweit diese eigene Dialoge bereitstellt. In diesem Fall werden also nur die entsprechenden Parameter weitergereicht und der betriebssystemeigene Dialog gestartet. Unter Windows beispielsweise nutzt man dann den Win32FolderPicker-Dialog, der natürlich auch von anderen Programmen verwendet wird.

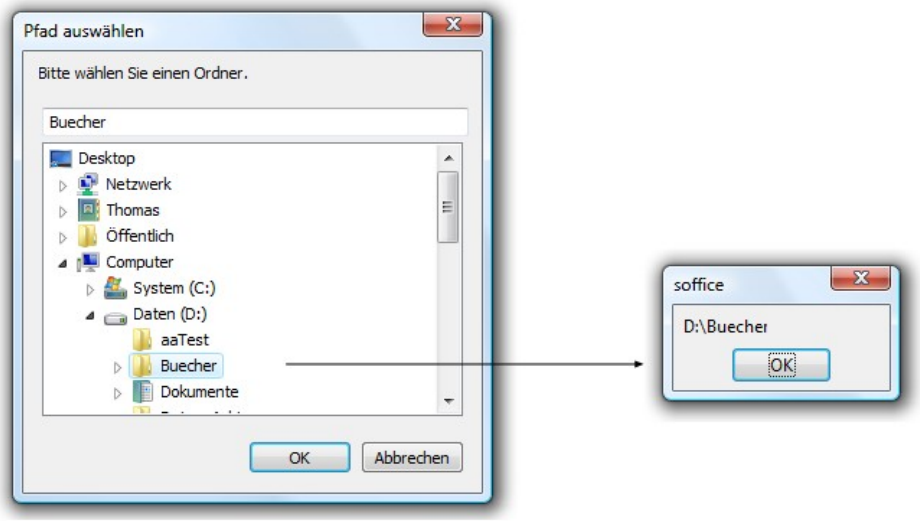
Während der Service com.sun.star.ui.dialogs.FolderPicker den Dialog (das Objekt) an sich darstellt, liefert das Interface com.sun.star.ui.dialogs.XFolderPicker einige Methoden, die den Dialog direkt nutzbar machen:

Methode	Beschreibung
setDisplayDirectory(sDir)	Setzt das Startverzeichnis, das beim Start des Dialoges angezeigt wird. Der String sDir muss im URL-Format übergeben werden.
getDisplayDirectory()	Liefert das eingestellte Startverzeichnis des Dialoges (im URL-Format).
getDirectory()	Liefert das ausgewählte Verzeichnis als String im URL-Format.
setDescription(sBeschreibung)	Hier kann eine optionale Beschreibung übergeben werden, die im Dialog angezeigt wird – als Text. Wird keine Beschreibung übergeben, zeigt der Dialog in der Regel eine Vorgabe an.

Ein Beispiel zur Auswahl eines Verzeichnisses, ausgehend vom im System eingestellten Arbeitsverzeichnis:

```
Sub GetFolder1
  Dim oDlg as Object, oPS as Object, sPfad as string
  oDlg = CreateUnoService("com.sun.star.ui.dialogs.FolderPicker")
  oPS = CreateUnoService("com.sun.star.util.PathSettings")
  oDlg.setDisplayDirectory(oPS.work)
  if oDlg.execute() then
     sPfad = oDlg.getDirectory
```

```
    msgbox ConvertFromURL(sPfad)
  end if
End Sub
```



Beispiel im Windows-eigenen Dialog.

Wer allerdings unter Windows arbeitet wird feststellen, dass der Vorgabepfad (an sich das Arbeitsverzeichnis oPS.work) trotz korrekter Übergabe nicht als Startverzeichnis genutzt wurde. Das liegt an einem Bug des Win32FolderPicker-Systems und ist auch bis jetzt nicht gefixt. Ein Vorgabeverzeichnis lässt sich hier leider nicht einstellen.

Achtung!

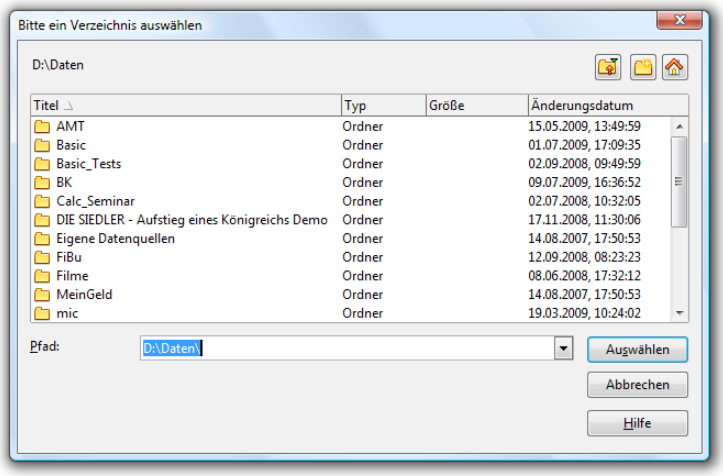
Bug in Windows-FolderPicker. Auch wenn man ein Vorgabeverzeichnis einstellt, wird es nicht angezeigt!

Abhilfe schafft hier nur die Wahl der OpenOffice.org-eigenen Dialoge, entweder generell in den Optionen eingestellt oder temporär durch Verwenden eines undokumentierten Services: Der com.sun.star.ui.dialoges.OfficeFolderPicker repräsentiert den OOo-eigenen Verzeichniswahl-Dialog und kann somit auch direkt aufgerufen werden. Dazu ändert man eine Zeile:

```
oDlg = CreateUnoService("com.sun.star.ui.dialogs.OfficeFolderPicker")
oDlg.setTitle("Bitte ein Verzeichnis auswählen")
```

Zusätzlich wurde auch noch der Dialogtitel geändert (die Vorgabe lautet einfach „Pfad wählen“). Anders als der normale FolderPicker-Dialog besitzt der OOo-eigene allerdings kein Description-Control, so dass es hier keine Rolle spielt, ob eine gesetzt wurde oder nicht.

Das Ergebnis:



Unter Linux wird in der Regel intern sowieso der hier dargestellte Dialog verwendet und insofern ist der „OfficeFolderPicker“ in der Regel eine gute Wahl.

6.6.2 Dateiauswahl-Dialog (FilePicker)

Neben dem reinen Verzeichnis spielt die Auswahl einer Datei oder eines Speicherortes für eine Datei eine sehr große Rolle. Dieses löst man mit dem Dateiwahl-Dialog, dem FilePicker. Dieser Service (com.sum.star.ui.dialogs.FilePicker) implementiert diverse Interfaces, die wiederum einige Methoden zur Verfügung stellen, die Dialoge entsprechend anpassen. Auch hier gilt wieder, dass es neben dem FilePicker-Dialog auch noch den (undokumentierten) Service com.sum.star.ui.dialogs.OfficeFilePicker gibt, der den OpenOffice.org-eigenen Dialog anzeigt statt des GUI-typischen. Allerdings sind alle Interfaces und Methoden identisch.

Ein erstes wichtiges Interface ist das com.sun.star.lang.XInitialization, das genau eine Methode zur Verfügung stellt: initialize(aListe), wobei aListe ein Array darstellt, dessen Inhalt nicht so wesentlich ist, wohl aber die Anzahl der Einträge. Nur die wird ausgewertet! Diese Methode sollte direkt nach dem Erzeugen des Objektes ausführt werden – und bestimmt das Aussehen und die Vorgaben des Datei-Dialoges. In Kurzform sieht das dann in etwa so aus:

```
oDlg = createUnoService("com.sun.star.ui.dialogs.FilePicker")
oDlg.initialize(array(3))
oDlg.execute
```

Diese Vorgaben werden – wie gesagt – als Listenanzahl übergeben, wobei die Werte festgelegt sind in der Konstantengruppe com.sun.star.ui.dialogs.TemplateDescription:

Achtung!

Es gab Änderungen in der API ab Version 3.3 von OOo. Die hier dargestellten Varianten funktionieren so nur bis einschließlich Version 3.2.1.

Int	Konstante	Beschreibung
0	FILEOPEN_SIMPLE	Einfacher Datei-Öffenen-Dialog
1	FILESAVE_SIMPLE	Einfacher Datei-Speichern-Dialog
2	FILESAVE_AUTOEXT-ENSION_PASSWORT	Ein Datei-Speichern-Dialog mit zusätzlichen Checkboxen für Passwort und automatischer Dateierweiterung
3	FILESAVE_AUTOEXT-ENSION_PASSWORT_ FILTEROPTION	Ein Datei-Speichern-Dialog mit zusätzlichen Checkboxen für Passwort, automatischer Dateierweiterung und Filterbearbeitung
4	FILESAVE_AUTOEXT-ENSION_SELECTION	Ein Datei-Speichern-Dialog mit zusätzlichen Checkboxen, automatischer Dateierweiterung und Selektion
5	FILESAVE_AUTOEXT-ENSION_TEMPLATE	Ein Datei-Speichern-Dialog mit zusätzlicher Checkbox, automatischer Dateierweiterung sowie zusätzlichen Listbox-Vorlagen
6	FILEOPEN_LINK_PRE-VIEW_IMAGE_TEMPLATE	Ein Datei-Öffnen-Dialog mit zusätzlichen Checkboxen für „Verknüpfen“ und „Vorschau“, eine Listbox für „Stile“ und ein Platz, um Bilder darzustellen
7	FILEOPEN_PLAY	Ein Datei-Öffnen-Dialog mit zusätzlichem Schalter „Abspielen“
8	FILEOPEN_READONLY_ VERSION	Ein Datei-Öffnen-Dialog mit zusätzlicher Checkbox für „nur Lesen“ sowie einer Listbox „Versionen“
9	FILEOPEN_LINK_PREVIEW	Ein Datei-Öffnen Dialog mit zusätzlichen Checkboxen für „Verknüpfen“ und „Vorschau“ sowie einem Platz, um Bilder darzustellen
10	FILESAVE_AUTOEXTENSION	Ein Datei-Speichern-Dialog mit zusätzlicher Checkbox und automatischer Dateierweiterung

Aber: Der Initialisierungsstil bestimmt nur das Aussehen des Dialoges – es werden keinerlei Funktionalitäten hinterlegt! Darum muss sich der/die Programmierer/in selbst kümmern.

Insbesondere sollte unbedingt geprüft werden, ob die Datei schon existiert, ob sie evtl. schon geöffnet ist oder ob der Pfad überhaupt erreichbar ist!

Hinweis

Wird keine Initialisierung aufgerufen, so wird immer der Default-Wert (0 – also FILEOPEN_SIMPLE) verwendet.

Das Interface com.sun.star.ui.dialogs.XFilePicker liefert die Methoden zur Auswahl bzw. zum Setzen der Dateien.

Die wahrscheinlich am häufigsten benutzte Methode liefert die gewählte/n Datei/en: getFiles()

Geliefert wird eine Liste (array) aller selektierten Dateien, und zwar wie folgt:

- Ist der Dialog nicht im MultiSelection-Mode, so wird die einzige gewählte Datei im ersten Eintrag der Liste zurückgegeben – inklusive des Pfades in URL-Schreibweise.
- Ist der Dialog im MultiSelection-Mode gestartet worden, so trägt der erste Eintrag der zurückgegebenen Liste nur den Pfad (das Verzeichnis) in URL-Schreibweise, alle anderen Einträge enthalten dann die gewählten Dateinamen – jetzt aber ohne Pfadangabe.

Beispiel:

```
Sub Datei_Wahl
        dim aListe(), i%, s$
        oDlg = createUnoService("com.sun.star.ui.dialogs.FilePicker")
        oDlg.setMultiSelectionMode(true)   'alternativ: false
        oDlg.execute
        aListe = oDlg.getFiles()
        s = "Die Liste der gewählten Dateien:" & chr(10)
        for i = lbound(aListe) to uBound(aliste)
                s = s & "Eintrag " & i & ": " & aListe(i) & chr(10)
        next
        msgbox s
End Sub
```

Die Vorgabe des MultiSelection-Modes ist false – also keine Mehrfachwahl. Lässt man beide Möglichkeiten einmal durchlaufen, so gibt es folgende Ergebnisse:

Spätestens beim Test dieses kurzen Code-Stückchens wird Ihnen aufgefallen sein, dass zwar eine Dateityp-Liste vorhanden ist, um Filter zu definieren, diese aber leer ist. Das ist natürlich unschön – und muss vom Programmierer / von der Programmiererin geändert und manuell definiert werden.

Hierfür zuständig ist das Interface com.sun.star.ui.dialogs.XFilterManager, das drei Methoden zur Verfügung stellt:

Methode	Beschreibung
appendFilter(sName, sFilter)	Fügt einen Filter zur bestehenden Liste hinzu. Details siehe unten.
setCurrentFilter(sName)	Aktiviert den mit sName spezifizierten Filter.
getCurrentFilter()	Liefert den Namen des aktuell aktivierten Filters als String.

Die appendFilter()-Methode fügt der Listbox „Dateityp“ des Dateidialoges genau einen Filter hinzu, und zwar müssen zwei Parameter übergeben werden:

- Der Name des Filters als String (sName): Dies kann ein von Ihnen beliebig gewählter Name sein, er wird in der Listbox angezeigt und sollte den Filter beschreiben. Der Name soll jedoch kein Semikolon enthalten.
- Der Filter selbst: Das ist nun der String, der als Filter auch angewendet wird, typischerweise also so etwas wie "*.txt", es können aber auch mehrere Filterkriterien übergeben werden, die müssen dann aber mit einem Semikolon getrennt werden. Als „wildcard“ – also Platzhalter – wird der * verwendet.

Sie können beliebig viele Filter definieren und nacheinander „anhängen“.

Hinweis

Der definierte Filter wird nicht in der Listbox angezeigt, nur der Name! Wenn Sie das dennoch möchten, so bauen Sie den Namen entsprechend um.

Ergänzend zu den eben besprochenen Möglichkeiten bietet das Interface com.sun.star.ui.dialogs.XFilterGroupManager noch die Methode

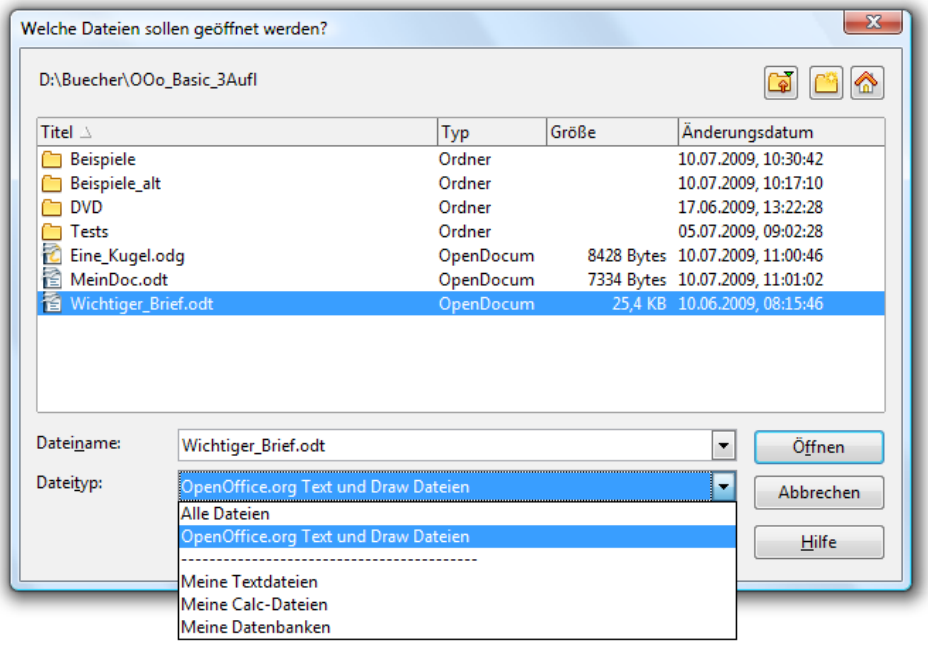
appendFilterGroup(sGuppenName, aFilterListe).

Hier wird zunächst ein Gruppenname (sGruppenname) gesetzt und anschließend die Liste der im Array aFilterListe (vom Typ com.sun.star.beans.StringPair) aufgelisteten Paare aus Filtername und Filter – wie oben. Nur können hier ganze Listen übergeben werden.

Beispiel:

```
sub DateiWahl_komplett
        dim oDlg as Object, s$, sPath as string
        dim MyFilterListe(2) as new com.sun.star.beans.StringPair
        REM Filterliste füllen
        MyFilterListe(0).first = "Meine Textdateien"
        MyFilterListe(0).second = "*.tk.odt"
        MyFilterListe(1).first = "Meine Calc-Dateien"
        MyFilterListe(1).second ="*.tk.ods"
        MyFilterListe(2).first = "Meine Datenbanken"
        MyFilterListe(2).second = "*.tk.odb"
```

```
        sPath = ConvertToUrl("E:\Daten\BasicBuch2A_Dateien")
        REM nun den Dialog erzeugen
        oDlg = createUnoService("com.sun.star.ui.dialogs.FilePicker")
        With oDlg
                .setMultiSelectionMode(true)
                .appendFilter("Alle Dateien", "*.*")
                .appendFilter("OpenOffice.org Text und Draw Dateien", "*.odt; *.ott; *.odg;
*.otg")
                .appendFilterGroup("Meine Dateien", MyFilterListe())
                .setDisplayDirectory(sPath)
                .setCurrentFilter("OpenOffice.org Text und Draw Dateien")
                .setTitle("Welche Dateien sollen geöffnet werden?")
        end with
        oDlg.execute
end sub
```



Genau wie das Öffnen funktioniert natürlich auch der Speichern-Dialog – und hier kann man gleich auch noch einen Dateinamen vorgeben.

oDlg.setDefaultName("EineWichtigeDatei.odt")

Nochmal zur Erinnerung: Nur das Aufrufen des Dialoges reicht im Programmierfall nicht – man muss selbst Sorge dafür tragen, wie und dass die Listboxen, Checkboxen, Text-Labels etc. ausgefüllt sind und auch entsprechend funktionieren. Die Dialoge stellen nur den grafischen Rahmen zur Verfügung.

6.7 Dialoge dynamisch erzeugen

Die bisherigen Betrachtungen drehten sich um Dialoge, deren Design zunächst innerhalb der IDE erstellt wurde und deren Daten dort zu finden sind. Dies ist sicherlich der normale und

sinnvollste Weg. Allerdings können Dialoge auch komplett im Programm-Code erzeugt und angezeigt werden, ohne dass vorher ein Design-Schritt stattgefunden hat, sie können also vollständig während der Laufzeit eines Programms erzeugt und geändert werden.

Praktisch ist dies immer dann, wenn der Dialog stark abhängig vom Kontext ist und erst zur Laufzeit die entsprechenden Parameter (z.B. Anzahl der Kontrollelemente) feststehen. Da jedoch der Programmieraufwand erheblich größer ist, schwindet die praktische Bedeutung dieses Weges im Fall von Makros.

In den meisten Fällen ist das Design in der IDE zu erstellen der bessere und genauere Weg. Dennoch soll das Prinzip hier kurz erläutert werden, wie man einen Dialog zur Laufzeit erstellt und anzeigt.

Um einen Dialog zu erstellen, gehen Sie wie folgt vor:

1. Zuerst erzeugen Sie das Model eines Dialoges mit der Methode createUnoService(„com.sun.star.awt.UnoControlDialogModel“). Dieses Model speichern Sie in einer Objektvariablen.
2. Jetzt können Sie dem erzeugten Model die gewünschten Eigenschaften zuweisen, wie zum Beispiel die Größe des Dialoges (Height, Width), die Position (PositionX, PositionY), den Titel und was Ihnen sonst so einfällt. Sie nutzen hierfür die Methode setPropertyValue(). Diese erwartet zwei Parameter, zunächst den Namen der Eigenschaft als String, dann den Wert.
3. Nun erzeugen Sie die einzelnen Kontrollfelder. Der Weg ist prinzipiell derselbe, allerdings müssen diese nun als Instanz des erzeugten Dialog-Models aufgerufen werden. Sie nutzen die Methode createInstance(), wobei dieser der Name des gewünschten Models übergeben wird, also zum Beispiel „com.sun.star.awt.UnoControlButtonModel“ für einen Button. Das erzeugte Objekt ist wieder ein Model eben der einzelnen Kontrollelemente, dem Sie jetzt die gewünschten Eigenschaften zuordnen.
4. Ist das Kontrollelement (beziehungsweise sein Model) komplett, wird es in den bestehenden Dialog integriert. Hierzu nutzen Sie die Methode insertByName("NameDesKontrollelementes") des Models der Dialogbox.
   Schritt 3 und 4 werden jetzt für jedes gewünschte Kontrollfeld wiederholt.
5. Ist das Model des Dialoges mit all seinen Kontrollelementen fertig, wird zunächst ein Dialog an sich erzeugt. Hierzu nutzen Sie die Methode CreateUnoService("com.sun.star.awt.UnoControlDialog"). Dies ist quasi nun die View-Ansicht, die aber noch nicht mit einem Model verbunden ist, also noch nichts anzeigen könnte.

6. Dem erzeugten Dialog wird das Model zugewiesen. Hierzu verwenden Sie die Methode setModel(). Dieser Methode wird das Model als Parameter übergeben. Dadurch werden dem Dialog alle Eigenschaften des Models zugewiesen.
7. Da der Dialog nun erzeugt ist (intern), müssen die Ereignismodelle noch zugeordnet werden. Dies ist in der IDE-Variante sehr einfach, bei zur Laufzeit erstellten Dialogen nicht ganz so trivial. Ereignismodelle sind nichts anderes als Listener, die auf die entsprechenden Ereignisse reagieren. Also werden jetzt genau diese erzeugt (mit der Methode CreateUnoListener(name)) und anschließend den entsprechenden Kontrollelementen des Dialoges zugeordnet.
8. Der Dialog ist eigentlich fertig, aber zur Anzeige fehlt noch ein entsprechendes Fenster. Dieses wird erzeugt mit der Methode createUnoService("com.sun.star.awt.Toolkit").
9. Über die Methode createPeer(oFenster, null) wird das erzeugte Fenster (oFenster) dem Dialog zugeordnet. Der Dialog eignet sich diese Eigenschaften an.
10. Jetzt kann der Dialog gestartet und angezeigt werden, hierzu nutzen Sie die bekannte Methode execute().

Das folgende Beispiel erzeugt eine einfache Dialogbox mit nur zwei Kontrollelementen, einem Beschriftungsfeld und einem Button. Für den Button wird der Typ OK-Button gewählt, so dass dieser den Dialog schließt und beendet. Hierfür ist kein Eventhandler nötig. Das Ergebnis der Box wäre sicher mit der vordefinierten Funktion msgBox() schneller und kürzer erreichbar gewesen, es geht aber hier um das Prinzip der Dialogerstellung.

Umfangreiche Dialoge sollten durch spezielle Routinen zum Erzeugen von Eigenschaften automatisiert und gekürzt werden, der Weg und das Prinzip zur Erstellung von Dialogboxen zur Laufzeit sollte aber erkennbar sein:

```
Sub LaufzeitDialogErstellen
  Dim oDlgM as Variant  ' das Model des Dialoges
  Dim oDlg as Variant   ' der Dialog an sich
  Dim oMod as Variant   ' nimmt jeweils das Model der Elemente auf

  REM Das Dialog-Model erzeugen:
  oDlgM=CreateUnoService("com.sun.star.awt.UnoControlDialogModel")
  REM Eigenschaften zuweisen
  oDlgM.setPropertyValue("PositionX", 30)
  oDlgM.setPropertyValue("PositionY", 30)
  DlgM.setPropertyValue("Width",  200)
  oDlgM.setPropertyValue("Height", 100)
  oDlgM.setPropertyValue("Title", "Ein Laufzeitdialog")

  REM ein Beschriftungsfeld erzeugen
  oMod = oDlgM.createInstance("com.sun.star.awt.UnoControlFixedTextModel")
  REM die Eigenschaften setzen
  oMod.setPropertyValue("Name", "Txt_Feld1")
  oMod.setPropertyValue("Align", 1)
  oMod.setPropertyValue("TabIndex", 1)
  oMod.setPropertyValue("PositionX", 15)
```

```
    oMod.setPropertyValue("PositionY", 15)
    oMod.setPropertyValue("Width",  170)
    oMod.setPropertyValue("Height", 30)
    oMod.setPropertyValue("Label", "Dieser Dialog wurde zur Laufzeit erstellt.")
    REM und dem Dialog-Model zuweisen
    oDlgM.insertByName("Txt_Field1", oMod)

    REM einen Button erzeugen
    oMod = oDlgM.createInstance("com.sun.star.awt.UnoControlButtonModel")
    REM die Eigenschaften setzen
    oMod.setPropertyValue("Name", "Button1")
    oMod.setPropertyValue("TabIndex", 2)
    oMod.setPropertyValue("PositionX", 50)
    oMod.setPropertyValue("PositionY", 65)
    oMod.setPropertyValue("Width",  100)
    oMod.setPropertyValue("Height", 20)
    oMod.setPropertyValue("Label", "OK")
    oMod.setPropertyValue("PushButtonType", com.sun.star.awt.PushButtonType.OK)
    REM und dem Dialog-Model zuweisen
    oDlgM.insertByName("Button1", oMod)

    REM den Dialog erzeugen
    oDlg = CreateUnoService("com.sun.star.awt.UnoControlDialog")
    oDlg.setModel(oDlgM)

    REM ein Fenster erzeugen und den Dialog zuweisen
    oWin = CreateUnoService("com.sun.star.awt.Toolkit")
    oDlg.createPeer(oWin, null)

    Rem Dialog aufrufen
    oDlg.execute()
End Sub
```

Wie gesagt, der Weg über den Dialog-Designer in der IDE ist wesentlich komfortabler und für die allermeisten Makros vollständig ausreichend.

6.8 Best practice Dialoge

Im folgenden werden nun einige bewährte Techniken „aus der Welt“ der Dialoge beschrieben, die häufig angewendet werden. Sie sollen helfen, bei zukünftigen Entwicklungen Zeit zu sparen und Ideen zu generieren.

6.8.1 Kontrollelemente Auswertung

Jeder Dialog besitzt typischerweise einige Kontrollelemente, also zum Beispiel Textfelder, Comboboxen oder andere Eingabefelder. Und in der Regel werden diese Eingaben weiterverarbeitet und oft erfolgen Aktivitäten (Programmverzweigungen) in Abhängigkeit des eingegebenen oder gewählten Eintrages.

Um hier Fehler zu vermeiden, sollte man folgende Überlegung anstellen: Für den/die Benutzer/in sind Leerzeichen oft nicht erkennbar und nicht wichtig. Der Name „Maier“ ist für ihn/sie identisch mit dem Namen „Maier " (also noch mit einem folgenden Leerzeichen). Er/Sie

würde auf eine Fehlermeldung beim zweiten Namen mit Unverständnis reagieren – und dann dem Programm an sich „Fehler“ unterstellen. Meist gilt dies im übrigen auch bei der Verwendung von Groß- und Kleinschreibung – auch hier muss das Verständnis des Benutzers / der Benutzerindas Maß aller Dinge sein – nicht die einfachere Programmiermöglichkeit. Bleiben wir beim Beispiel „Maier“: Für den/die Benutzer/in sind die folgenden Eingaben alle identisch – und sollten nicht zu einem Fehler führen:

„Maier“, „Maier ", „maier“, „MAIER“, „ Maier“ „MaieR“ und so weiter.

Es ist also unbedingt notwendig, die Eingaben zu „normalisieren“, bevor damit Vergleiche stattfinden. Beispiel:

```
Public Const sName = "Maier"   'Der Name als Prüfbedingung

...
sTxt = odlg.getControl("txt_eingabename").text
REM normalisieren
if lCase(Trim(sTxt)) = lCase(sName) then
  REM Bedingung erfüllt
else
  REM  Bedingung nicht erfüllt
end if
```

Ähnliches kann auch notwendig sein, wenn Sie mit Zahlen arbeiten. Denken Sie daran, dass Computer intern mit einer hohen Genauigkeit rechnen – es dann aber schnell zu minimalen Rundungsdifferenzen kommen kann – und das führt im Extremfall zu falschen Vergleichswerten. Typische Fälle sind die Rechnungen mit Währungszahlen!

Auf der sicheren Seite sind Sie in der Regel mit ganzen Zahlen (Int/Long) – und das kann man dann auch entsprechend nutzen:

Beispiel: Zu einem gegebenen oder berechneten Bruttowert soll der Nettowert eingegeben werden und intern verglichen werden – bei Gleichheit wird eine Aktion ausgelöst, bei Ungleichheit eine andere. Der/Die Nutzer/in wird die Zahlen mit zwei Nachkommastellen eingeben:

```
public const MwSt = 19   'Mehrwertsteuersatz

Sub Bsp_ButtoNetto
  Dim brutto as double, netto as double

  brutto = 1200.30
  netto = brutto/(1+ MwSt/100)

  msgbox netto    'Ergebnis: 1008,65546218487

  nDlgNetto = oDlg.getControl("num_netto").value   ' 1008.66

  if clng(netto * 100) = clng(nDlgNetto * 100) then
    msgbox "passt"
  else
    msgbox "Netto ist nicht korrekt!"
  end if
```

```
End Sub
```

Würde man als Vergleichsbedingung hier nur `netto = nDlgNetto` verwenden, so wäre das Ergebnis immer „False“ – im dargestellten Fall jedoch wird es als „True“ gewertet – durch die Umwandlung wird die interne Genauigkeit reduziert und der korrekte Wert ermittelt.

Es kann sich durchaus als sinnvoll erweisen, bei Währungsberechnungen intern nur mit Long-Variablen zu arbeiten, also wären Beträge zunächst mit 100 zu multiplizieren, dann alle Rechnungen durchzuführen und erst bei Ausgabe oder Darstellung wieder auf die übliche Darstellung (zwei Nachkommastellen) zu bringen (Division durch 100).

Direkte Prüfung von Eingaben

Eine andere oft genutzte Struktur ist das direkte Reagieren auf eine Eingabe. Die meisten Eingabefelder bieten ein Ereignis „Text modifiziert“, das immer dann aufgerufen wird, wenn es Änderungen bei der Eingabe gibt. So wird dieses Ereignis bei jedem Buchstaben/Zeichen einer Eingabe ausgelöst – und man kann quasi in „Echtzeit“ die Eingabe prüfen und entsprechend darauf reagieren. Zwei kleine Beispiele:

Der weiter oben aufgeführte Passwort-Dialog soll soweit modifiziert werden, dass das Passwort nicht mit „*“ angezeigt wird, sondern nur die ersten Zeichen – das zuletzt eingegebene hingegen verbleibt im Klartext. Ist die Länge des Passwortes (hier 10 Zeichen) hingegen erreicht, dann wird alles als „*“ angezeigt – für jede weitere Eingabe erfolgt eine Fehlermeldung.

```
dim oPwDlg as variant
dim bFlag as boolean
dim sPW as string

Function Eingabecheck_PW
  dim s as string, n as integer

  if bFlag then exit function
  bFlag = true

  s = oPWDlg.getControl("txt_pw").text
  n = len(trim(s))
  if n > 10 then
    msgbox "Sie haben schon 10 Zeichen eingegeben - mehr geht nicht!"
    oPWDlg.getControl("txt_pw").text = string(10, "*")
    oPWDlg.getControl("txt_pw").accessibleContext.setCaretPosition(10)
  end if

  if n = 1 then
    sPW = s
  elseif n > 1 AND n < 11 then
    sPW = left(sPW, n-1) & right(s,1)
    oPWDlg.getControl("txt_pw").text = string(n-1, "*") & right(s,1)
    oPWDlg.getControl("txt_pw").accessibleContext.setCaretPosition(n)
  end if

  bFlag = false
end function
```

Zur Erläuterung: Es wird ein globales Flag benötigt, das anzeigt, dass die Funktion bereits läuft. Da diese verknüpft ist mit dem Ereignis „Text modifiziert“ (Ereignis ruft diese Funktion auf), die Funktion aber wiederum den Text des Controlls modifiziert und sich somit erneut aufruft, würde eine endlose „Verschachtelung“ der Aufrufe erfolgen mit dem Ergebnis eines Speicherüberlaufes und des Zusammenbruchs des Programms. Insofern beendet sich ein weiterer Aufruf sofort, solange das globale Flag gesetzt ist.

Als nächstes wird die aktuelle Eingabe ausgelesen und die Länge (Anzahl Zeichen) berechnet.

Achtung!

Die Zeichen entsprechen nicht mehr den tatsächlichen Eingaben – lediglich das letzte Zeichen ist korrekt! In einer globalen Variablen wird das Passwort so Zeichen für Zeichen aufgebaut.

Schließlich prüft man die Länge der Eingabe – mit entsprechenden Meldungen.

Ist die maximale Länge nicht überschritten, dann wandelt man alle Zeichen des Strings in „*“ (oder andere Zeichen) um, bis auf das letzte Zeichen, und schreibt den String zurück. In diesem Fall wäre der Cursor des Eingabefeldes nun aber ganz am Anfang zu finden – also muss dieser hinter die letzte Position versetzt werden, damit der/die Benutzer/in problemlos weiter schreiben kann.

Mit der gleichen Technik lassen sich auch Eingaben korrigieren (Groß-/Kleinschreibung) oder automatische Weiterverarbeitungen starten.

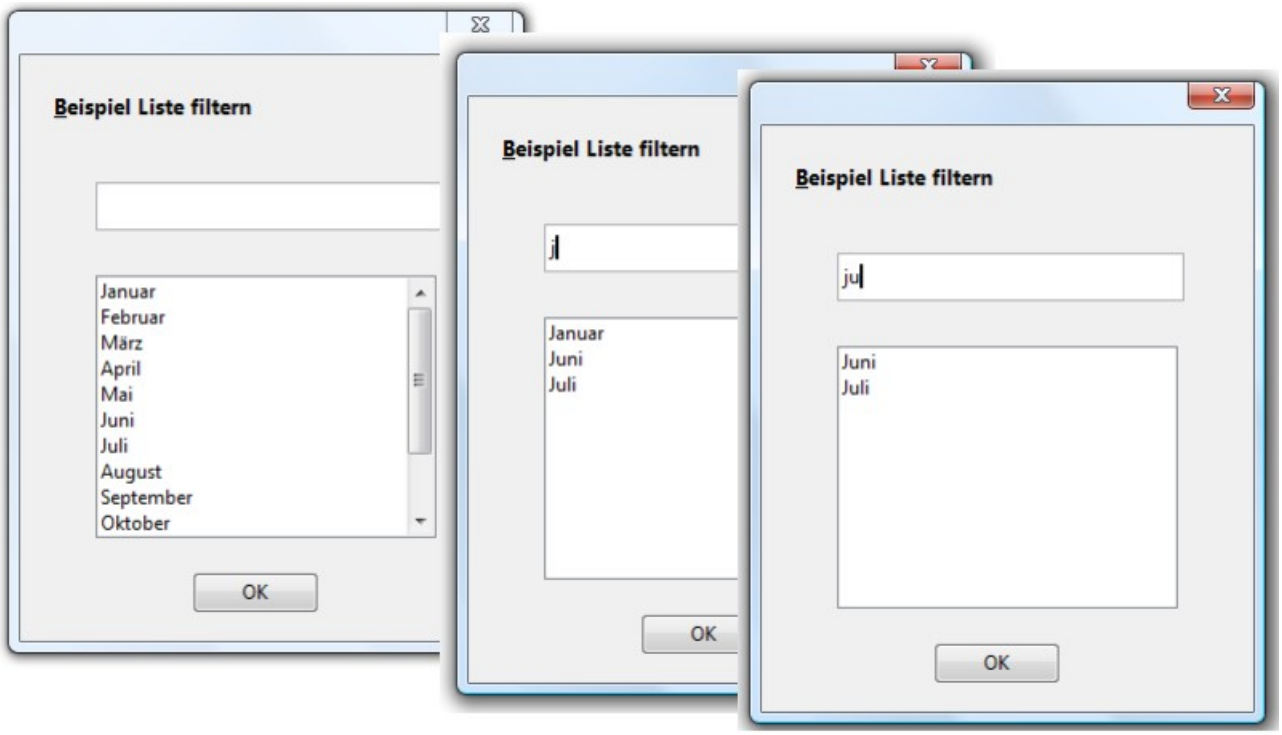
Live-Filtertechniken

Ein weiteres Beispiel soll genau dies zeigen. In einer Listbox sind jede Menge Einträge gespeichert – der/die Benutzer/in wählt einen bestimmten aus . Um ihm/ihr die „lästige“ Suche und das „Scrollen“ zu ersparen, soll die Liste immer nur die Einträge anzeigen, die mit den Anfangsbuchstaben der Filter-Eingabe übereinstimmt (also quasi ein Filter). Die Combobox bietet bereits eine solche Funktion automatisch – diese ist jedoch begrenzt auf den ersten Buchstaben. Das Beispiel hier arbeitet mit beliebig vielen.

```
sub DlgListeFiltern
  dim aListe()

  aListe = array("Januar", "Februar", "März", "April", "Mai", "Juni", "Juli", _
                 "August", "September", "Oktober", "November", "Dezember")

  DialogLibraries.loadLibrary("CookBook")   'laden der Dialog-Bibliothek
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_test3)
  with oDlg
    .getControl("lbl_1").text = "Beispiel Liste filtern"
    .getControl("lst_1").model.stringItemList = aListe
  end with

  oDlg.execute()
end sub
```

```
REM Liste filtern - verknüpft mit dem Ereignis "Text modifiziert"
sub listeFiltern
  dim s as String
  dim aListe1()
  dim aListe2()

  s = oDlg.getControl("txt_1").text
  aListe1 = oDlg.getControl("lst_1").model.stringItemList
  redim aListe2(uBound(aliste1()))
  n = 0
  for i = 0 to uBound(aListe1())
    if lcase(trim(s)) = lcase(trim(left(aListe1(i), len(s)))) then  'passt
      aListe2(n) = aListe1(i)
      n = n+1
    end if
  next

  redim preserve aListe2(n-1)
  oDlg.getControl("lst_1").model.stringItemList = aListe2

end sub
```

Das Ergebnis zeigt die Bilderfolge:



Man darf allerdings die Performance nicht außer Acht lassen – eine Liste mit 20.000 Einträgen benötigt eine gewisse Zeit, bis sie durchgelaufen ist. Hier heißt es einfach ein bisschen „versuchen“.

Der hier dargestellte Algorithmus hat noch eine zweite „Schwäche“ – die Rücktaste führt zum Absturz, da die Liste regelmäßig angepasst wurde und nur die aktuell vorhandene untersucht

wird. Wird also das letzte Zeichen gelöscht, muss wieder die komplette Liste durchsucht werden! Das lässt sich beispielsweise durch Zwischenspeicherung der Textlänge im Textfeld lösen – mit entsprechend angepasstem Code. Aber dies hier soll ja auch nur als Beispiel dienen und das Prinzip vorstellen.

Eingabekontrolle

Insbesondere bei Dialogen, die Benutzereingaben verlangen, ist es unablässig, diese auch zu überprüfen und falsche oder fehlende unbedingt zu bemängeln. Während das Prinzip klar ist – Eingaben auslesen, prüfen, und dann Fehlermeldung ausgeben – zunächst ein paar Überlegungen zum Vorgehen:

Wird ein Dialog per execute() ausgeführt und per OK oder Abbrechen beendet, ist eine Prüfung nicht so einfach möglich. Durch die interne Struktur wird der Dialog ja zunächst geschlossen – er muss also jetzt wieder neu aufgebaut und erneut angezeigt werden. Das lässt sich zwar mit einem rekursiven Aufruf ermöglichen (siehe folgendes Beispiel), bietet dann aber wenig Flexibilität:

```
sub DialogAufrufen
  REM vorbereiten des Dialoges , Initialisierungen etc.

  if NOT (oDlg.execute() = 1) then exit sub   'Ende bei Abbruch

  REM jetzt Eingaben prüfen
  if bFehler then   'Fehler sind aufgetreten
    DialogAufrufen    'Prozedur neu starten
    exit sub          'unbedingt Exit Anweisung anfügen!!
  end if

  REM weitere Verarbeitung
end sub
```

Der Dialog wird dabei jedesmal gleich aufgebaut, dies hilft dem/der Benutzer/in wenig. Oder es müsste durch gesetzte globale Flags der Aufbau unterschiedlich erfolgen – das ist nicht wirklich eine gute Programmierung.

Des weiteren ist unbedingt zu beachten, dass nach dem erneuten Aufruf der eigenen Funktion (rekursiver Aufruf) unbedingt eine „Exit-Anweisung“ erfolgen muss – sonst wird der Rest-Code entsprechend der Anzahl der Aufrufe später hintereinander abgearbeitet – und führt dann mit Sicherheit zu Fehlern!

Möchte man die Eingaben vor dem Schließen des Dialoges überprüfen, so geschieht dies in einer eigenen Funktion – verbunden mit dem normalen „Ende“-Button des Dialoges – der darf dann nicht auf „OK“ stehen, sondern muss auf „Standard“ bleiben. Nur dann wird die dem Ereignis „Aktion ausführen“ zugeordnete Funktion sicher ausgeführt.

Jetzt ist der Dialog ja noch offen – man kann nun direkt die Eingaben prüfen, Fehler in einer Liste sammeln, die betroffenen Eingabefelder entsprechend markieren (zum Beispiel durch eine rote Hintergrundfarbe) – und dann eine gesammelte Fehlermeldung ausgeben. Eine solche

Sammelfehlermeldung ist immer einer Einzelfehlermeldung vorzuziehen, um dem/der Benutzer/in einen Gesamtüberblick zu geben – und ihn/sie nicht viermal hintereinander mit den gleichen Fehlermeldungen für unterschiedliche Felder zu nerven.

Im folgenden Beispiel wird ein Dialog mit vier Eingabefeldern geprüft – erstens daraufhin, ob überhaupt eine Eingabe stattfand und zweitens, ob die Eingabe in einem bestimmten Feld (dem 3. Feld) genau vier Zeichen lang ist – anderenfalls erfolgt eine Fehlermeldung, der Dialog verbleibt aber sichtbar und an der gleichen Stelle – der/die Benutzer/in kann seine/ihre Eingaben entsprechend korrigieren.

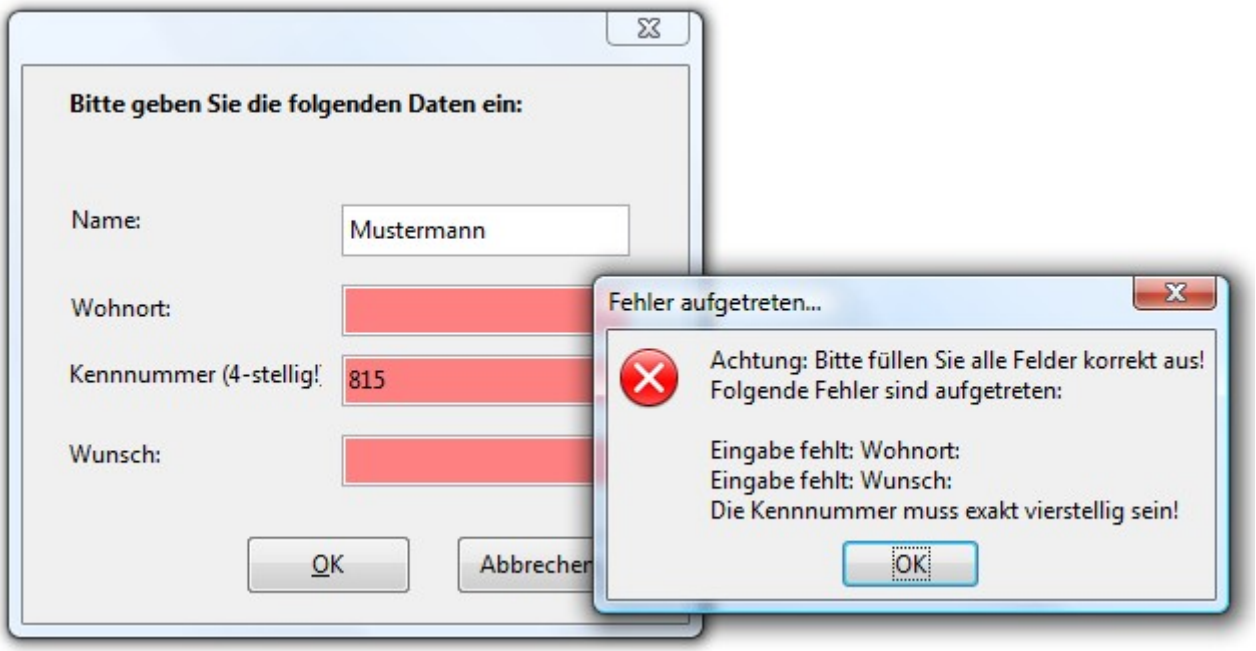
Zunächst der Dialog und die gemachten Eingaben:

Der folgende Code zeigt nun die Fehlerprüfung. In diesem Fall sind alle Beschriftungsfelder mit dem Namen „lbl_x“ benannt und alle Eingabefelder mit „txt_x“, wobei „x“ für die Zahlen 1-4 steht (von oben nach unten). Natürlich sind auch Einzelprüfungen möglich!

```
sub DlgEingabeCheck
  DialogLibraries.loadLibrary("CookBook")
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_test4)  'Dialog erzeugen

  if NOT (oDlg.execute() = 1) then exit sub
end sub

' Eingaben-Prüfung
sub CheckEingaben
  dim bFehlerflag as boolean
  dim aFehler()
  dim i%, n%, s as string

  n = 0
  For i = 1 to 4
    sTxt = oDlg.getControl("txt_" & i).text
    oDlg.getControl("txt_" & i).model.backgroundColor = void
    if trim(sTxt) = "" then 'keine Eingabe!
      bFehlerflag = true
```

```
      redim preserve aFehler(n)
      aFehler(n) = "Eingabe fehlt: " & oDlg.getControl("lbl_" & i).text
      n = n+1
      oDlg.getControl("txt_" & i).model.backgroundColor = RGB(255, 128, 128)  'lachs
    end if
  next i
  REM Länge der Eingabe Feld 3 prüfen
  sTxt =  oDlg.getControl("txt_3").text
  if len(sTXT) <> 4 AND trim(sTxt) <> "" then
    bFehlerflag = true
    redim preserve aFehler(n)
    aFehler(n) = "Die Kennnummer muss exakt vierstellig sein!"
    oDlg.getControl("txt_3").model.backgroundColor = RGB(255, 128, 128)  'lachs
  end if
  REM Fehlermeldung
  if bFehlerFlag then
    sTxt = "Achtung: Bitte füllen Sie alle Felder korrekt aus!" & chr(13) & _
           "Folgende Fehler sind aufgetreten:" & chr(13) & chr(13)
    for i = 0 to uBound(aFehler())
      sTxt = sTxt & aFehler(i) & chr(13)
    next
    msgbox (stxt, 16, "Fehler aufgetreten...")
    exit sub    'Ende der Funktion
  end if

  oDlg.endExecute()  'Dialog beenden

end sub
```

Das Ergebnis sähe dann wie folgt aus:

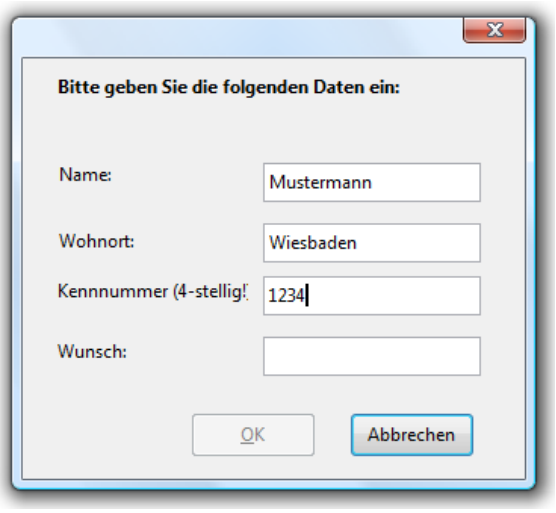


Die Fehlerprüfung läuft solange durch, bis alle Felder korrekt ausgefüllt werden. Bei jeder Prüfung werden die Farben zunächst zurückgesetzt und neu gesetzt, falls das entsprechende

Feld immer noch nicht in Ordnung ist. Dieses Verhalten führt zu einem für den/die Nutzer/in konsistenten Vorgehen.

Eine andere Möglichkeit der Benutzerführung besteht darin, über eine Prüffunktion den Inhalt der vier Textfelder abzufragen und den OK-Button erst dann freizuschalten, wenn alle vier Felder den gewünschten Inhalt besitzen. Der folgende Code ist mit dem Ereignis „Bei Focus Verlust“ der vier Textfelder verbunden:

```
sub DlgEingabeCheck
  DialogLibraries.loadLibrary("CookBook")
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_test4)  'Dialog erzeugen
  oDlg.getControl("cmd_ok").SetEnable(false)
  if NOT (oDlg.execute() = 1) then exit sub
end sub

' Prüfung der Eingaben in allen Textfeldern
sub Textfeldpruefung
  dim bFlag as boolean
  for i = 1 to 4
    if trim(oDlg.getControl("txt_" & i).text) = "" then bFlag = true
  Next
  if bFlag then
    oDlg.getControl("cmd_ok").SetEnable(false)
  else
    oDlg.getControl("cmd_ok").SetEnable(true)
  end if

end sub
```



Wichtig im Code ist hier, dass der Button auch wieder auf „enable = false“ gesetzt wird, falls ein Feld später wieder gelöscht wird!

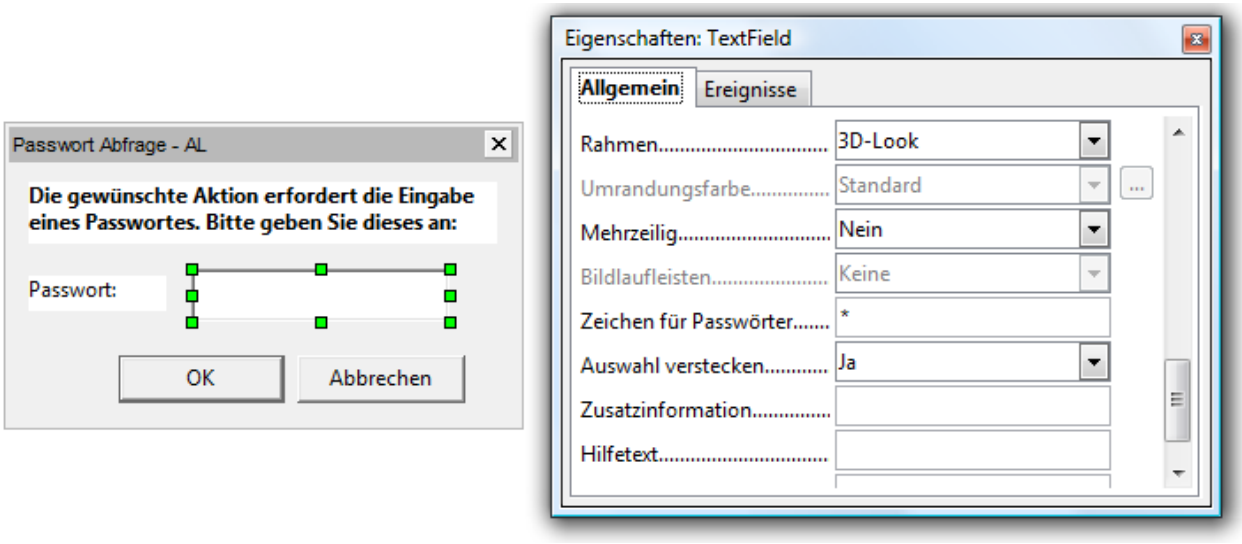
6.8.2 Passwort abfragen

Ein typisches Beispiel für einen eigenen Dialog ist eine Abfrage nach einem Passwort.

Sicherheitsüberlegung:

Das Passwort zum Vergleich muss natürlich irgendwo hinterlegt sein – und das wäre dann auch die Schwachstelle. Jedes Office-Dokument ist ja im Grunde „nur“ ein XML-File – und der lässt sich mit einem normalen Texteditor problemlos öffnen und durchsuchen. Damit wäre auch ein Passwort leicht zu finden. Die hier vorgestellten Programme eignen sich also nur sehr bedingt dazu, sicherheitsrelevante Daten zu „verbergen“. Ein gewisser Schutz besteht aber, wenn beispielsweise eine eigene Bibliothek in einem Dokument angelegt wurde, dort das Passwort als Konstante gespeichert wird und anschließend die Bibliothek „verschlüsselt“ wird. Jetzt ist zumindest diese Bibliothek verschlüssselt und mit dem Texteditor nicht mehr lesbar – die Makros aber können problemlos auf die Bibliothek zugreifen und auch das eingegebene Passwort mit der Konstanten vergleichen.

Doch zurück zum Passwort-Dialog. Der kann z.B.wie folgt aussehen:



Das Eingabetextfeld wird als „Passwort-Feld“ gekennzeichnet und bekommt einen Eintrag für die Eigenschaft „Zeichen für Passwörter“. Dadurch werden eingegebene Zeichen nicht angezeigt, sondern nur der entsprechende Platzhalter, die verhindert, dass ein Unbefugter „quasi über die Schulter“ das Passwort mitlesen kann.

Der Code vergleicht nun das Passwort mit der Vorgabe – und bietet insgesamt drei Möglichkeiten, das Passwort zu wiederholen – nach drei falschen Versuchen beendet sich der Dialog automatisch und könnte eine andere Aktivität auslösen:

```
'/**  Passwortcheck
'*************************************************************************
'* @kurztext prüft ein einzugebendes Passwort, mit Dialog
'* Die Funktion erzeugt einen Passwort-Dialog und prüft, ob das eingegeben
'* Passwort stimmt.
'* Wenn nicht, wird der Dialog maximal 3* aufgerufen - dann Ende
'*
'* @param1  sPasswort as string   das Passwort
'*
```

```
'* @return  bFlag as boolean   true, wenn PW ok, sonst false
'*
'*************************************************************************
'*/
function Passwortcheck(sPasswort as string)
  Passwortcheck = false    'Vorgabe
  DialogLibraries.loadLibrary("MAK017_AL")   'evtl. wieder löschen+++++
  oPWDlg = createUnoDialog(DialogLibraries.MAK017_AL.dlg_pw)  'Dialog erzeugen
  oPWDlg.model.title = oPWDlg.model.title & " - Version " &
MAK017_Helper2.GetMakroVersionsNummer(sBibName)
  if (oPWDlg.execute = 0) then exit function
  if NOT (oPWDlg.getControl("txt_pw").text = sPasswort) then
    with oPWDlg
      .getControl("lbl_pw").text = "Das Passwort ist falsch! Bitte erneut eingeben. Sie haben
noch zwei Versuche!"
      .getControl("txt_pw").text = ""
      .getControl("txt_pw").model.backgroundcolor = RGB(255,180,130)
    end with
    if (oPWDlg.execute = 0) then exit function
    if NOT (oPWDlg.getControl("txt_pw").text = sPasswort) then
      oPWDlg.getControl("lbl_pw").text = "Das Passwort ist falsch! Bitte erneut eingeben. Sie
haben noch einen Versuch!"
      oPWDlg.getControl("txt_pw").text = ""
      if (oPWDlg.execute = 0) then exit function
      if NOT (oPWDlg.getControl("txt_pw").text = sPasswort) then
         msgbox ("Das Passwort ist dreimal nicht korrekt gewesen." & chr(13) & "Die Funktion
wird abgebrochen", 16, "Falsches Passwort")
         exit function
      end if
    end if
  end if
  Passwortcheck = true    'alle ok
end function
```

Die Funktion liefert „True“ zurück, wenn das Passwort ok ist, und „False“, wenn der Dialog abgebrochen oder dreimal hintereinander ein falsches Passwort eingegeben wurde.

Aktuell erfolgt ein reiner Textvergleich des Passwortes – das bedeutet, Groß- und Kleinschreibung werden unterschieden und sind zu beachten. Es ist jedoch problemlos möglich, dieses zu erweitern.

6.8.3 Selbstlernende Listen

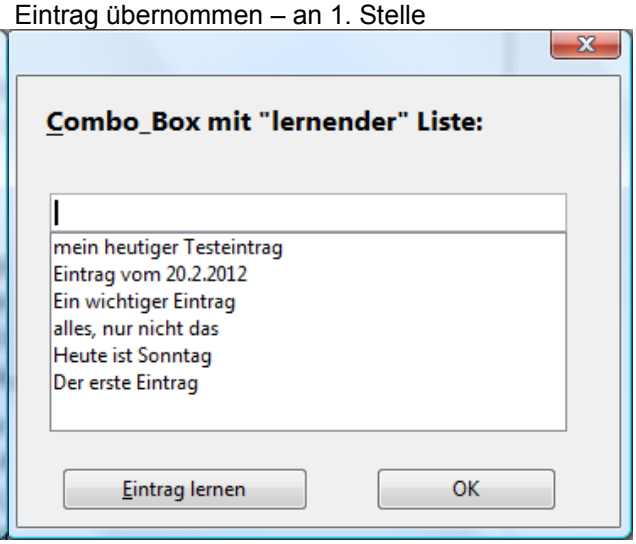
Eine weitere häufige Anwendung von Arrays sind „selbstlernende Listen“ - also Combo-Felder in Dialogen oder Formularen, in die der/die Benutzer/in Werte eingeben oder aus der Liste auswählen kann. Die Liste wiederum „lernt“ aus den Eingaben und behält sich die zum Beispiel letzten 10 Eingaben – die dann auch entsprechend ihrer Reihenfolge dargestellt werden.

Die Listeneinträge von Comboboxen sind Arrays, diese wiederum müssen nun regelmäßig aktualisiert werden – und zwar mit dem letzten Eintrag. Allerdings wird dieser an Position 0 (also an erster Position) eingefügt, falls er nicht schon in der Liste vorhanden ist. Ansonsten wird nur

umsortiert. Wird er neu eingefügt und hatte die Liste vorher schon die maximale Größe erreicht, so entfällt der letzte Eintrag. Das folgende Beispiel zeigt das gewünschte Ergebnis (der besseren Darstellung wegen wurde die Combobox nicht aufklappbar, sonder fix dargestellt):

ein neuer Eintrag wird ergänzt

Eintrag übernommen – an 1. Stelle



Der passende Code dazu:

```
'/** CBO_ListeLernen
'*************************************************************************.
'* @kurztext erweitert eine Liste einer Combobox um einen Begriff
'* Diese Funktion erweitert eine Liste einer Combobox (Dialog) um einen Begriff an der 1.
Stelle
'* Dafür wird zunächst geprüft, ob der Begriff bereits vorhanden ist (dann wird
'* er verschoben, ansonsten an Position 1 (Index 0) hinzugefügt.
'*
'* @param1  oCbo as object   das Objekt der Combo-Box
'* @param1  sEintrag as string   der neue bzw zu verschiebende Eintrag
'* @param1  iAnz as integer   optional  maximale Anzahl der Listeneinträge
'*
'*************************************************************************
'*/
sub CBO_ListeLernen( oCbo as object, sEintrag as String, optional iAnz as integer)
  Dim aListe(), i%

  with oCbo
    REM Liste der Combobox ist noch leer - dann Eintrag erster Eintrag
    if .getItemCount = 0 then
      .addItem(sEintrag, 0)
      exit sub
    end if
    REM Liste ist nicht leer, prüfen, ob Eintrag schon vorhanden, dann verschieben
    aListe = .getItems
    for i = lBound(aListe) to ubound(aListe)
      if sEintrag = aListe(i) then  'Eintrag vorhanden
        .removeItem(i, 1)  'bisherigen Eintrag entfernen
        .addItem(sEintrag, 0)  'Eintrag an Pos 1 neu schreiben
        exit sub
      end if
    next
```

```
    REM Liste nicht leer und Eintrag nicht vorhanden
    REM Einfügen an Pos 0 und evtl. kürzen der Liste
    .addItem(sEintrag, 0)
    if NOT isMissing(iAnz) then
      if .getItemCount = iAnz + 1 then .removeItems(iAnz, 1)
    end if
  end with
end sub
```

Aber Achtung: Die Einträge sind „kurzlebig“. Wird der Dialog geschlossen und das Makro beendet, wären die Einträge ebenfalls „weg“. Es liegt am/an der Programmierer/in, die Einträge zunächst auszulesen, zu sichern und eventuell für eine spätere Verwendung in eine Datei oder an einen anderen Ort zu schreiben. Möglichkeiten dazu wurden besprochen in Kapitel 4.5 und Kapitel 4.6.

6.8.4 Tabellarische Darstellung von Listen

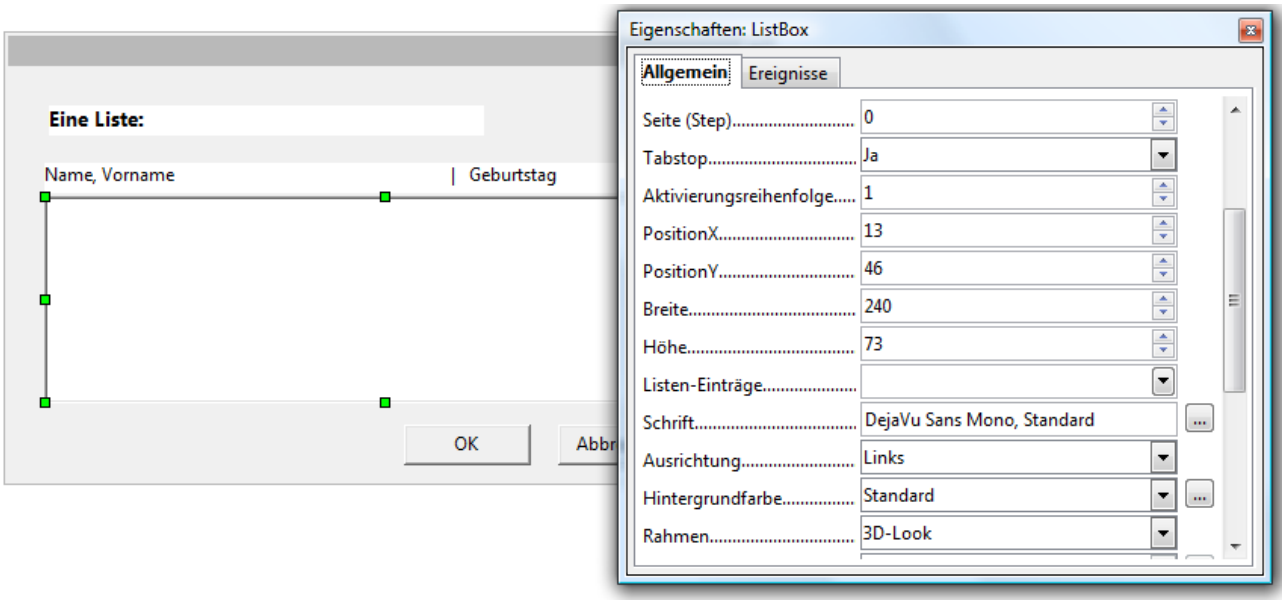
Werden in Dialogen Listen verwendet, so sind es oft nicht nur einfache „Textlisten“, sondern oft Informationen in Datensatzform. Beispiel: Die Liste von Namen, Vornamen sowie Geburtsdatum. Während man Name und Vorname problemlos mit einem Komma getrennt hintereinander hängen kann, soll das Geburtsdatum schön untereinander stehen.

Normalerweise würde man für die Darstellung ein Tabellen-Grit-Kontrollelement verwenden, leider gibt es dieses für Dialoge bis zur Version OOo 3.2.1 nicht. Für Formulare ist dieses bereits vorhanden, zur Verwendung in Dialogen wird es erst in Version 3.3 eingeführt, und selbst dort funktioniert es noch nicht vollständig und muss manuell sehr aufwendig programmiert werden.

So bleibt nur ein Umweg:

Man kann die eingebauten Listboxen (oder auch Comboboxen) auch für symmetrische Listen verwenden – benötigt aber ein paar „Tricks“.

Zunächst muss für die Schrift der Box eine Schriftart gewählt werden, die eine feste Zeichenbreite bereitstellt – nur so ist später gewährleistet, dass Zeichen Nummer 10 immer unter Zeichen Nummer 10 steht.



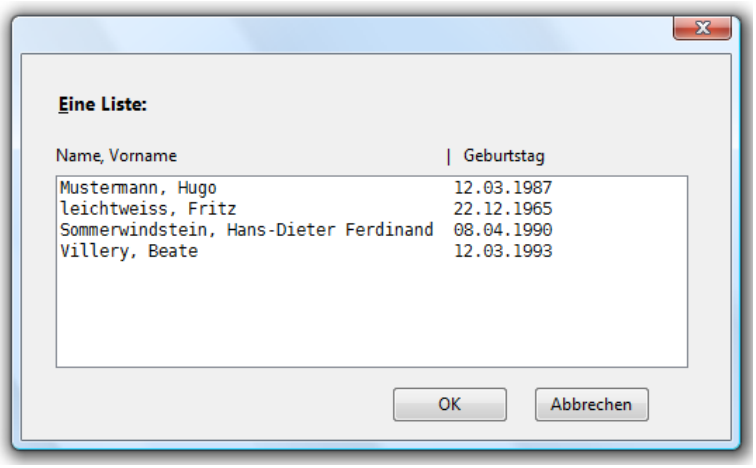
Die Schriftart „DejaVu Sans Mono“ wäre eine solche und wird mit OOo ausgeliefert. Sie sollte also auf jedem Rechner zur Verfügung stehen.

Als nächstes legt man die maximale Breite für jede Spalte fest – denn nur dann kann es eine symmetrische Aufarbeitung geben. Einträge, die über die maximale Spaltenbreite hinausragen würden, müssen später entsprechend gekürzt werden.

In der Regel liegen die späteren Zeilen als Liste der Einträge vor (Arrays), aus diesen Einträgen wird dann die anzuzeigende Liste generiert. Wichtig auch hierbei: Die angezeigte Liste kann evtl. gekürzte Einträge enthalten – es ist also notwendig, die Originalliste zu behalten und später nur damit weiterzuarbeiten. Das folgende Beispiel zeigt eine solche „Aufbearbeitung“. Die Liste selbst wird hier definiert – typischerweise wird sie jedoch durch andere Aktionen erzeugt – also zum Beispiel durch Lesen aus einer Calc-Datei, als Datenbankauszug oder anderes.

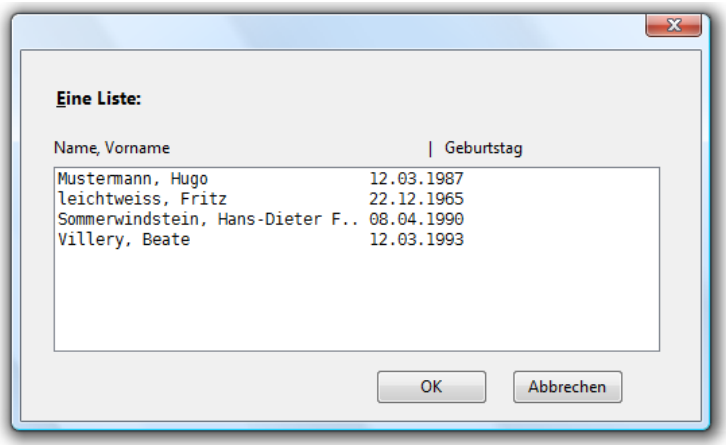
```
sub Dlgliste
  dim aListe()
  dim aFeldlen()

  aListe = Array(array("Mustermann, Hugo", "12.03.1987"), _
                 array("leichtweiss, Fritz", "22.12.1965"), _
                 array("Sommerwindstein, Hans-Dieter Ferdinand", "08.04.1990"), _
                 array("Villery, Beate", "12.03.1993"))

  aFeldLen = array(40, 20)

  DialogLibraries.loadLibrary("CookBook")
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_liste)  'Dialog erzeugen
  oDlg.getControl("lst_1").model.stringItemList = ListeSymAufbereiten(aListe, aFeldlen(), "", 0
)
  if NOT (oDlg.execute() = 1) then exit sub
end sub

'/** ListeSymAufbereiten()
'*************************************************************************.
'* @kurztext bereitet einen Array als Liste mit festen Zeichenspalten auf
'* Diese Funktion bereitet einen Array als Liste mit festen Zeichenspalten auf
'* Sind die Spalteneinträge länger als die Spaltenbreite, so werden die überfälligen
'* Zeichen abgeschnitten und zwei Punkte ".." ergänzt. Sind die Spalteneinträge kürzer,
'* so erfolgt ein Auffüllen mit Leerzeichen.
'* Wichtig: Anzahl der Spalten (Arrayelemente) müssen identisch sein des übergebenen
Längenarray!
'*
'* @param1  aListeAlt() as array   ein Array von (Zeilen-) Arrays (Werte der Spalten)
'* @param2  aFeldLen() as array  eine Liste der gewünschten Zeichenanzahlen pro Spalte (für
jede Spalte ein Eintrag!)
'* @param3  sSpTr as string   Spaltentrenner (kann leer sein)
'* @param4  iSpTrTyp as integer   Spaltentrenner Typ 0- keine, 1- nur erste Spalte, 2 - nur
letzte Spalte, 5 - alle
'*
'* @return  aListe() as array   die aufbereitete Liste
'*************************************************************************
'*/
function ListeSymAufbereiten(aListalt, aFeldlen(), sSpTr as string, iSpTrTyp as integer )
  dim aListe(), aDSalt()
```

```
  dim sZeile as string, n%, j%, i%, sTxt as string

  if uBound(aListalt()) > -1 then  redim aListe(uBound(aListalt()))

  for i = 0 to uBound(aListe())
    aDSalt = aListalt(i)
    sZeile = ""
    for j = 0 to uBound(aFeldLen())
      n = aFeldLen(j)
      sTxt = aDSalt(j)
      if len(sTxt) > n-1 then
        if NOT (len(sTxt) <= 3) then sTxt = left(sTxt, n-3) & ".. "
      else
        sTxt = sTxt & Space(n-len(stxt))
      end if
      sZeile = sZeile & sTxt
      select case iSptrTyp
        case 1   'erste Spalte trennen
          if (j=0) then sZeile = sZeile & sSpTr
        case 2   'letze Spalte trenen
          if (j=uBound(aFeldLen())-1) then sZeile = sZeile & sSpTr
        case 5   'alle Spalten Trennen
          if NOT (j = uBound(aFeldLen())) then sZeile = sZeile & sSpTr
      end select
    next j
    aListe(i) = sZeile
  next i

   ListeSymAufbereiten = aListe()
end function
```



Das Ergebnis ist eine entsprechend aufbereitete Liste und eine „hübsche“ Darstellung. In diesem Fall wurden die Spaltenbreiten auf 40 und 20 Zeichen festgelegt – eine Kürzung ist somit nicht erforderlich gewesen. Korrigieren wir jetzt einmal die Spaltenbreite auf 33 für den Namen, Vornamen, dann sieht das Ergebnis wie folgt aus:

```
  aFeldLen = array(33, 20)
```



Der Eintrag wird gekürzt, die Punkte signalisieren den längeren Eintrag.

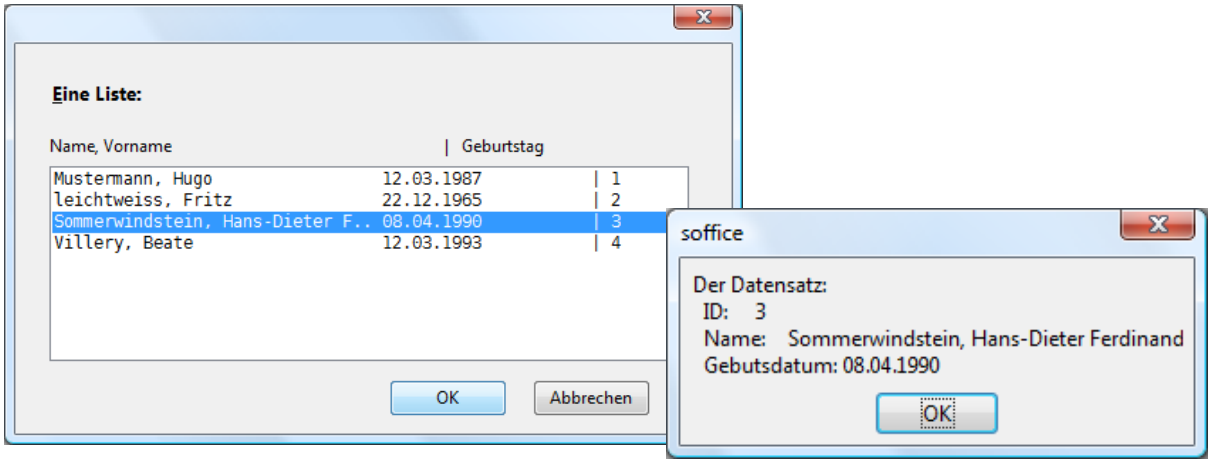
Typischer Einsatz:

Diese Listendarstellungen kommen überwiegend im Datenbankbereich vor. In diesem Fall werden Datensätze entsprechend aufbereitet, wobei jetzt ein eindeutiges Merkmal des Datensatzes mit übergeben werden muss – typischerweise die ID-Nummer. Anhand deren lässt sich ein Datensatz jederzeit „rekonstruieren“ – also neu aus der Datenbank einlesen. Es ist zwar möglich, die ID-Nummer zum Beispiel in einer zweiten Liste mitzuführen und dann über die Position des gewählten Eintrags die ID-Nummer zu identifizieren – schneller und einfacher geht es aber, die ID-Nummer als Teil des Eintrages anzusehen – entweder ganz vorne oder (meist besser) ganz hinten. Mit Hilfe der folgenden Funktion wird die ID-Nummer dann ganz einfach aus dem gewählten Datensatz wieder ausgelesen:

```
'/** MAK140_ExtractID()
'*************************************************************************.
'* @kurztext Extrahiert die ID aus einem Texteintrag
'* Diese Funktion Extrahiert die ID aus einem Texteintrag (typischerweise aus einem Listenfeld)
'*
'* @param1  sTxt as string   der Texteintrag mit der ID
'* @param2  iTyp as integer  Typ des Eintrags: 1: ID vorne, 2: ID letzter Eintrag
'* @param3  sTrenner as string  optional: das Trennzeichen  - wenn nicht übergeben, wird die
"Pipe" (|) verwendet
'*
'* @return  sID as string  die ID als Text
'*
'*************************************************************************
'*/
function MAK140_ExtractID(sTxt as string, iTyp as integer, optional sTrenner2 as string)
  dim sID as string, sTrenner as string
  dim a()

  if isMissing(sTrenner2) then
    sTrenner = "|"
  else
    sTrenner = sTrenner2
  end if
  a = split(sTxt, sTrenner)
```

```
  if uBound(a) > 0 then    'mehr als ein Eintrag
    if iTyp = 1 then  'ID erster Eintrag
      sID = trim(a(0))
    elseif iTyp = 2 then  'ID letzer Eintrag
      sID = trim( a(uBound(a)))
    end if
  end if
  MAK140_ExtractID = sID
end function
```

Ich füge mal den Datensätzen eine ID am Ende hinzu, und lese mit dem OK-Button den gewählten Datensatz aus – und zwar aus der ursprünglichen Liste:



In dem Beispiel zu erkennen: Obwohl der Name verkürzt dargestellt wird (im Dialog) und die Zeile nicht rekonstruierbar ist, wird über die ID der ursprüngliche Datensatz zur Auswertung herangezogen. Der folgende Code zeigt die entsprechenden Stellen:

```
..
  aListe = Array(array("Mustermann, Hugo", "12.03.1987", 1), _
                 array("leichtweiss, Fritz", "22.12.1965", 2), _
                 array("Sommerwindstein, Hans-Dieter Ferdinand", "08.04.1990", 3), _
                 array("Villery, Beate", "12.03.1993",4 ))

  aFeldLen = array(33, 20, 5)

  DialogLibraries.loadLibrary("CookBook")
  oDlg = createUnoDialog(DialogLibraries.CookBook.dlg_liste)  'Dialog erzeugen
  oDlg.getControl("lst_1").model.stringItemList = ListeSymAufbereiten(aListe, aFeldlen(), " |
", 2 )
  if NOT (oDlg.execute() = 1) then exit sub

  iID =  cInt(MAK140_ExtractID(oDlg.getControl("lst_1").getSelectedItem(), 2))

  for i = 0 to uBound(aliste)
    aZeile = aListe(i)
    if aZeile(2) = iID then  'gefunden
      sTxt = "Der Datensatz: " & chr(13) & _
             "  ID:      " & aZeile(2) & chr(13) & _
             "  Name:     " & aZeile(0) & chr(13) & _
             "  Gebutsdatum: " & aZeile(1)
```

```
      msgbox sTxt
      exit sub
    end if
  next
..
```

Gerade im Umgang mit Datenbanken und den direkten Select-Abfragen ist diese Technik sehr effektiv.

7 Best Practice Writer (Textverarbeitung)

Viele Makros der Textverarbeitung beziehen sich auf eine Teilverarbeitung der Inhalte – sei es, dass sie gelesen, geschrieben oder verändert werden.

Andererseits wird die Textverarbeitung aber auch gerne komplett als „Ausgabemedium“ verwendet. Die folgenden Ausführungen beschreiben typische Beispiele.

Zum Verständnis: Ein Writer-Dokument besteht aus einem „Textbereich“, dem Bereich, der typischerweise den Textinhalt aufnimmt sowie jeder Menge anderer Objekte (Grafiken, Textrahmen, Kopf- und Fusszeilen, weiteren Textbereichen und vielem mehr). Viele Objekte liegen dabei auf der „Drawpage“.

Ein Textbereich beinhaltet eine Abfolge von Absätzen – jeder Absatz wiederum besteht aus einer Abfolge von Absatzteilen (ein Absatzteil ist ein Textabschnitt mit gleichen Texteigenschaften). (Text-)Tabellen sind wie Absätze integriert und diesen gleichgestellt.

Ein Textdokument wird mit Hilfe von Formatvorlagen formatiert. Formatvorlagen sind ein Bestandteil des Dokumentes und werden auch in diesem gespeichert. Legt man also Wert auf bestimmte Formatierungen, so sollten diese zunächst passend erzeugt werden, wenn sie nicht schon vorhanden sind.

Dies geht ziemlich einfach mit zum Beispiel dem folgenden Code:

```
REM überprüft, ob die benötigten Vorlagen vorhanden sind, wenn nicht, werden sie erstellt
sub checkStyles
	REM Zunächst Zeichenvorlagen:
	vCharStyles = thisComponent.StyleFamilies.getByName("CharacterStyles")
	REM Master-Style
	if not vCharStyles.hasByName("BasicCode_Master") then
		vStyle = thisComponent.createInstance("com.sun.star.style.CharacterStyle")
		with vStyle
			.CharFontName = sBasicFont
			.CharHeight = iBasicFontHeight
			.CharScaleWidth = iBasicFontScaleWidth
		end with
		vCharStyles.insertByName("BasicCode_Master", vStyle)
	end if
	REM Kommentar-Style
	if not vCharStyles.hasByName("BasicCode_Comment") then
		vStyle = thisComponent.createInstance("com.sun.star.style.CharacterStyle")
		with vStyle
			.ParentStyle = "BasicCode_Master"
```

```
                .CharColor = RGB(102,102,102)
            end with
            vCharStyles.insertByName("BasicCode_Comment", vStyle)
        end if
        REM Strings oder Texte-Style
        if not vCharStyles.hasByName("BasicCode_Literal") then
            vStyle = thisComponent.createInstance("com.sun.star.style.CharacterStyle")
            with vStyle
                .ParentStyle = "BasicCode_Master"
                .CharColor = RGB(255,0,0)
            end with
            vCharStyles.insertByName("BasicCode_Literal", vStyle)
        end if
        REM Schlüsselwörter-Style
        if not vCharStyles.hasByName("BasicCode_Keyword") then
            vStyle = thisComponent.createInstance("com.sun.star.style.CharacterStyle")
            with vStyle
                .ParentStyle = "BasicCode_Master"
                .CharColor = RGB(0,0,128)
            end with
            vCharStyles.insertByName("BasicCode_Keyword", vStyle)
        end if
        REM Bezeichner-Style
        if not vCharStyles.hasByName("BasicCode_Ident") then
            vStyle = thisComponent.createInstance("com.sun.star.style.CharacterStyle")
            with vStyle
                .ParentStyle = "BasicCode_Master"
                .CharColor = RGB(0,128,0)
            end with
            vCharStyles.insertByName("BasicCode_Ident", vStyle)
        end if
        REM Jetzt Absatzvorlagen
        vParaStyles = thisComponent.StyleFamilies.getByName("ParagraphStyles")
        REM Master-Style
        if not vParaStyles.hasByName("BasicCode") then
            vStyle = thisComponent.createInstance("com.sun.star.style.ParagraphStyle")
            with vStyle
                .CharFontName = sBasicFont
                .CharHeight = iBasicFontHeight
                .CharScaleWidth = iBasicFontScaleWidth
                .ParaLeftMargin = 200
            end with
            vParaStyles.insertByName("BasicCode", vStyle)
        end if
        REM Zeilennummer
        if not vParaStyles.hasByName("BasicCode_Nummer") then
            vStyle = thisComponent.createInstance("com.sun.star.style.ParagraphStyle")
            with vStyle
                .ParentStyle = "BasicCode"
                .CharColor = RGB(255,255,255)
                .ParaAdjust = com.sun.star.style.ParagraphAdjust.RIGHT
                .ParaLeftMargin = 0
                .ParaRightMargin = 100
            end with
            vParaStyles.insertByName("BasicCode_Nummer", vStyle)
        end if
end sub
```

Selbstverständlich können in gleicher Weise auch Seiten- und Nummerierungsformate geprüft und erzeugt werden. Auch lassen sich vorhandene Vorlagen verändern (zum Beispiel Überschriften), dies birgt jedoch die Gefahr, vom Benutzer / von der Benutzerin bereits angepasste Vorlagen erneut zu verändern mit Wirkung auf das komplette Dokument. Dies sollte unbedingt vermieden werden.

Mit Hilfe der Formatvorlagen lassen sich dann einfach eigene (Text-)Teile formatieren.

7.1 View-Cursor und Textcursor

Um in Textdokumenten überhaupt etwas zu platzieren, wird entweder der View-Cursor oder ein Textcursor verwendet. Kurz zu den Unterschieden:

View-Cursor: Es kann immer nur einen View-Cursor im Dokument geben. Dieser wird vom CurrentController gesteuert und bereitgestellt. Nur der View-Cursor „kennt“ seine exakte Position – also zum Beispiel, auf welcher Seite er sich befindet, in welcher Zeile und so weiter. Der View-Cursor ist der sichtbare, blinkende Cursor im Dokument.

Textcursor: Ein Dokument kann gleichzeitig mehrere Textcursor verarbeiten – der Textcursor arbeitet in dem Textbereich, in dem er erzeugt wurde – und kann nur dort Aktionen auslösen. Ein Textcursor ist auf der Oberfläche nicht sichtbar. Ein Textcursor kann nicht auf die Möglichkeiten des sichtbaren Bereiches zugreifen – also zum Beispiel das „Ende der Zeile“. Ein Textcursor kann nur die Möglichkeiten des Textbereiches abdecken (also zum Beispiel „Ende des Absatzes“).

Typischerweise arbeitet man mit dem Textcursor, der wiederum mit Hilfe des View-Cursors im gewünschten Bereich erzeugt wird. Es gibt aber auch Arbeiten, die ohne View-Cursor nicht durchführbar sind.

Wichtig:

Verwendet man im Code den View-Cursor, so muss unbedingt vorher geprüft werden, ob es diesen überhaupt gibt! Der/Die Benutzer/in könnte aktuell eine Ansicht eingestellt haben, in der es keinen View-Cursor gibt (zum Beispiel: Seitenvorschau-Ansicht). Startet jetzt ein Makro, das den View-Cursor nutzt, erfolgt ein Basic-Laufzeitfehler!

Das folgende Beispiel druckt die aktuelle Seite (die man gerade auf dem Bildschirm sieht) direkt am Standarddrucker aus. Die Seiteneigenschaft wird dabei mit Hilfe des View-Cursors ermittelt.

```
'/** WT_SeiteDrucken
'*************************************************************************.
'* @kurztext  druckt die aktuelle Seite (dort, wo sich der Viewcursor befindet)
'* Das Makro druckt die aktuelle Seite auf dem aktuellen Drucker aus. Die aktuelle
'* definiert sich dabei durch die Position des Viewcursors (dem blinkenden Cursor),
'* diese kann abweichen von der tatsächlichen Sichtseite.
'*************************************************************************
'*/
Sub WT_SeiteDrucken
  dim oViewC as variant
```

```
  REM ausschließen, dass die "Seitenansicht" eingeschaltet ist
  if thisComponent.currentController.frame.layoutManager.isElementVisible( _
          "private:resource/toolbar/previewobjectbar") then
     msgbox "Die Zusatzfunktion ""aktuelle Seite drucken"" ist in der" & chr(13) & _
            "Seitenvorschauansicht nicht verfügbar. Bitte verlassen Sie diese" & chr(13) & _
            "Darstellung (über den Button ""Seitenansicht schließen"") oder nutzen" & chr(13) &
_
            "Sie die Funktion ""Seitenansicht drucken"" - zweites Icon links" & chr(13) & _
            "neben ""Seitenansicht schließen""."
     exit sub
  end if

  REM Fehler sonstiger Art ausschließen (kein Viewcursor)
  on error goto fehlerVC
  oViewC = thisComponent.getCurrentController.getViewCursor()
  dim arg(0) as new com.sun.star.beans.PropertyValue
  arg(0).name = "Pages"
  arg(0).value = """" & oViewC.page & """"
  thisComponent.print(arg())

exit sub
fehlerVC:
  msgbox "Die Funktion ""aktuelle Seite drucken"" lässt sich derzeit nicht" & chr(13) & _
         "ausführen. Es liegt ein unbekannter Fehler vor."

End Sub
```

Oft ist es notwendig zu wissen, wo sich der/die Benutzer/in gerade befindet – mit seinem View-Cursor. Leider gibt es dazu viele Möglichkeiten – und nicht alle sind tatsächlich für die Weiterverarbeitung nutzbar. Möchte man beispielsweise ein Textfeld an der Position des Cursors (View) einfügen, so geht dies nur in einem Textbereich – eine aktuell markierte Grafik besitzt aber keinen Textbereich.

Der folgende Code untersucht die aktuelle Position und liefert das aktuelle Objekt:

```
'/** MIC_GetObjectPosition
'*************************************************************************.
'* @kurztext Analysiert die Position eines Objektes im Dokument
'* Analysiert die Position eines Objektes im Dokument
'* Übergeben wird als Parameter ein Objekt (hier Viewcursor oder Textmarke) sowie
'* der Name des Objektes (wird für evt. Fehlermeldungen genutzt)
'* liefert das Textobjekt, in dem sich das Objekt befindet, zurück
'* Kann ein leeres Objekt zurückliefern, soweit hier ein Abbruch erfolgen soll.
'* @param1  Obj as object  übergeben wird das Objekt, dessen Position im Dokument gesucht
'*  werden soll. Hier entweder der ViewCursor oder die Textmarke
'* @param2  sName as string   übergeben wird auch ein String, der das übergebene Objekt
beschreibt.
'*                    Dieser String wird für evtl. Fehlermeldungen gebraucht - und zur
Identifikation
'* @return  oTxtObj as object  zurückgeliefert wird das Textobjekt der Position, also zum
Beispiel
'*                    eine Tabellenzelle, ein Rahmen, Kopf-oder Fusszeile, Fliesstext ...
'*************************************************************************
'*/
function MIC_GetObjectPosition(Obj as object, sName as string)
  dim oTxtObj as object  'leeres Objekt, wird bei der Rückgabe benötigt
```

```
                        'nimmt evtl. das Textobjekt auf, in dem sich das Obj befindet
  if sName = "Der Cursor " then   'nur beim Cursorobjekt
    If ThisComponent.CurrentSelection.SupportsService("com.sun.star.drawing.ShapeCollection")
then
         REM in einem Zeichnungsobjekt oder ein solches ist markiert
         MIC_FehlerPosition(sName & "befindet sich in einem Zeichnungsobjekt oder ein solches
ist markiert")
         MIC_GetObjectPosition = oTxtObj
         exit Function
    ElseIf
ThisComponent.CurrentSelection.SupportsService("com.sun.star.text.TextGraphicObject") then
         REM in einem Grafikobjekt oder ein solches ist selektiert.
         MIC_FehlerPosition(sName & "befindet sich in einem Grafikobjekt oder ein solches ist
selektiert.")
         MIC_GetObjectPosition = oTxtObj
         exit Function
    end if
  end if
  REM jetzt für alle
  if not isEmpty(obj.textTable) then  ' in einer Texttabelle
       oTxtObj = Obj.cell.text
  elseIf not isEmpty(Obj.textFrame) then  'in einem Textrahmen
       oTxtObj = Obj.textFrame.text
  elseIf not isEmpty(Obj.textSection) then  'in einem Textbereich
       oTxtObj = ThisComponent.text
  elseIf not isEmpty(Obj.textField) then 'in oder am Anfang eines Textfeldes
       MIC_FehlerPosition(sName & "befindet sich in einem (Text-) Feld oder ein solches ist
selektiert.")
  ElseIf obj.getText.ImplementationName = "SwXFootnote" then 'in einer Fußnote
    oTxtObj = Obj.getText()
  ElseIf obj.getText.ImplementationName = "SwXEndnote" then 'in einer Endnote
    oTxtObj = Obj.getText()
  ElseIf obj.getText.ImplementationName = "SwXHeadFootText" then 'in der Kopf- oder Fusszeile
    oTxtObj = Obj.getText()
  else
       oTxtObj = ThisComponent.text
  end if

  MIC_GetObjectPosition = oTxtObj   'Rückgabe: das Textobjekt, evtl. leer
end function
```

Allerdings reicht es oft auch aus, ohne explizites Wissen der Position des View-Cursors an genau dieser Stelle einen Textcursor zu erzeugen. Schließlich trägt das Objekt, der Viewcursor, ja die Information in sich:

```
  ...
  oViewC = oDoc.getCurrentController().getViewCursor()
  oTextC = oViewC.text.createTextCursorByRange(oViewC)
  …
```

Nun kann mit beiden Objekten gearbeitet werden.

Textcursor spielen im weiteren Verlauf eine wichtige Rolle – werden doch viele eingebaute Möglichkeiten als „Textcursor“ ausgeführt und können so direkt bearbeitet werden (zum Beispiel Suchen und Ersetzen).

7.1.1 Besonderheiten Tabellen

Texttabellen (siehe auch nächsten Abschnitt) werden intern als Absatz behandelt und angesprochen – aber eben leider ohne die typischen Absatzmerkmale. Da ein Absatz im Textbereich nur über eine Enumeration erreichbar ist und nicht über eine Zahlvariable, muss im Hinterkopf immer behalten werden, dass es auch Texttabellen im Text geben kann. Das folgende Beispiel entfent alle harten Formatierungen aus einem markierten Textbereich, wobei immer der ganze Absatz der Markierung betroffen ist. Die Herausforderung ist nun also, zunächst die betroffenen Absätze zu identifizieren. Über die Markierung (View-Cursor) kann der Anfang des ersten Absatzes und das Ende des letzen Absatzes mit Textcursorn „markiert“ werden. Anschließend erfolgt die „Enumeration“ der Absätze und es wird geprüft, ob der aktuelle Absatz (Start bzw. Ende) übereinstimmt mit dem Start/Ende-Textcursor. Ist dies der Fall,wird der Absatz zurückgesetzt.

Wäre der Absatz eine Texttabelle, so gibt es kein „Text-Objekt“ – die Methode würde zu einem Basic-Laufzeitfehler führen. Daher muss zunächst ausgeschlossen werden, dass der aktuelle Absatz eine Texttabelle ist (eine Texttabelle unterstützt den Service "com.sun.star.text.TextTable").

```
'/** WT_DELHarteFormatierungen
'*************************************************************************.
'* @kurztext entfernt harte Formatierungen aus den markierten Absätzen
'* Das Makro entfernt alle harten Formatierungen aus den markierten Absätzen bzw.
'* aus dem kompletten Textbereich. Dabei werden "weiche Formatierungen" - also
'* Formatvorlagen - erhalten.
'*
'*************************************************************************
'*/
Sub WT_DELHarteFormatierungen
  dim oViewC as variant   'Viewcursor
  dim oAbsaetze as variant
  dim oAbs as variant
  dim oTC1 as variant, oTC2 as variant, oTC3 as variant, oTC4 as variant

  oDoc = thisComponent
  REM Selektion auslesen
on Error goto Fehler  'falls kein ViewCursor existiert!
  oViewC = oDoc.currentController.getViewCursor
  if oViewC.isCollapsed() then   'nichts markiert
    n = msgbox("Sie haben nichts markiert - soll das komplette" & chr(13) & _
               "Dokument (Basistext) korrigiert werden?",4+32+256, "Keine Textmarkierung")
    if NOT (n=6) then exit sub

    oAbsaetze = oDoc.text.createEnumeration
    Do while oAbsaetze.hasMoreElements
      WT_AbsatzZuruecksetzen(oAbsaetze.nextElement, oDoc)
    loop
  else
    otc1 = oViewC.getText.createTextCursorbyRange(oViewC.getStart())  'Anfang
    otc2 = oViewC.getText.createTextCursorbyRange(oViewC.getEnd())  'Ende
    oTC1.gotostartofParagraph(false)  'Start
    oTC2.gotoendofParagraph(false)   'Ende
```

```
    oAbsaetze = oViewC.gettext().createEnumeration

    Do while oAbsaetze.hasMoreElements
      oAbs = oAbsaetze.nextElement
      if oAbs.supportsService("com.sun.star.text.TextTable") then
        'Absatz ist eine Tabelle
      else
        oTC3 = oAbs.text.createTextCursorByRange(oAbs.getStart())
        oTC3.gotoRange(oTC1, true)
        if oTC3.isCollapsed then flag = true  'Anfang gefunden
        if flag then     'jetzt abarbeiten
          WT_AbsatzZuruecksetzen(oAbs, oDoc)
          'xray oAbs
          oTC4 = oAbs.text.createTextCursorByRange(oAbs.getEnd())
          oTC4.gotoRange(oTC2, true)
          if oTC4.isCollapsed then exit do
        end if
      end if
    loop
  end if
  exit sub
fehler:
  msgbox ("Bitte markieren Sie einen Text oder Textteil!", 16, "keine Markierung")
End sub

'/** WT_AbsatzZuruecksetzen
'*************************************************************************.
'* @kurztext entfernt harte Formatierungen aus dem übergebenen Absatz
'* Das Makro entfernt alle harten Formatierungen aus dem übergebenen Absatz
'* Dabei werden "weiche Formatierungen" - also Formatvorlagen - erhalten.
'*
'* @param1  oAbs as object   der Absatz als Objekt
'* @param2  oDoc as object   das Dokument
'*************************************************************************
'*/
sub WT_AbsatzZuruecksetzen(oAbs as object, oDoc as object)
  dim oFrame as variant
  dim sAbsVorlage as string, sCharVorlage as String
  dim oZeichen as variant
  dim dispatcher as variant
  dim oAbsTeile as variant


  oFrame    = oDoc.CurrentController.Frame
  dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

    if oAbs.supportsService("com.sun.star.text.Paragraph") then  'Absatz
      sAbsVorlage = oAbs.paraStyleName
      oAbsTeile = oAbs.createEnumeration
      Do while oAbsTeile.hasMoreElements
        oZeichen = oAbsTeile.nextElement
        sCharVorlage = oZeichen.CharStyleName
        oDoc.currentController.select(oZeichen)
        dispatcher.executeDispatch(oFrame, ".uno:ResetAttributes", "", 0, Array())

        if NOT (sCharVorlage = "") then oZeichen.CharStyleName = sCharVorlage   'Rückschreiben
Formatvorlage
      loop   'nächster Absatzteil
      oAbs.ParaStyleName = sAbsVorlage  'Rückschreiben Absatzvorlage
```

```
    end if
End Sub
```

7.2 Tabellen

Während einfacher Text recht einfach in ein Dokument integriert werden kann, ist das mit den Tabellen schon etwas komplizierter.

Tabellen werden in Textdokumenten wie Absätze behandelt – folgen also dem Textfluss. Eine Tabelle wird zunächst als Objekt erzeugt, dann in das Dokument integriert und anschließend angepasst. Diese Reihenfolge ist wichtig, da einige Eigenschaften und Möglichkeiten der Tabelle erst nach Integration in den Textbereich möglich bzw. verfügbar sind.

Eine Tabelle wird beim Einfügen automatisch in ihrer Größe definiert – und zwar über die gesamte Breite des zur Verfügung stehenden Textbereiches. Die Spalten werden dann zunächst gleichgroß (Breite) entsprechend verteilt.

Anders als Tabellen in der Tabellenkalkulation Calc besitzen Tabellen in Writer keine echten Spalteneigenschaften, sondern jede Zeile ist eigentlich nur eine Spalte (Breite entspricht der Textobjekt-Breite abzüglich der definierten Ränder der Tabelle), die einzelnen Zellen entstehen durch Teilungen innerhalb der Zeile. Diese „Teilungen“ können dann per Skript „verschoben“ werden – wobei sie typischerweise keine festen Maßeinheiten erhalten, sondern Prozentwerte der Gesamtbreite. Das Arbeiten mit Tabellen ist also etwas anders als in Calc-Tabellen – andererseits gibt es viele Ähnlichkeiten bei den einzelnen Zellaktivitäten.

Im folgenden erläutere ich das Vorgehen zum Erstellen einer Tabelle in einem Writer-Dokument – im Haupttextbereich (Breite 17 cm), mit einem Abstand von 1 cm links und rechts, 3 Spalten, wobei die erste Spalte ca. 1 cm breit sein soll und der Rest mittig aufgeteilt wird.

Die Tabelle erhält 5 Datensätze (übergeben manuell per Array) sowie eine Kopfzeile.

```
'Texttabelle in Writer erzeugen
sub TextInTabelle
  dim aTabInhalt(), aInhalt(), aKopfZeile()
  dim sTxt as String
  dim i as integer, oTab as variant, oTxtRange as variant

  aKopfzeile() = Array("Nr", "Bundesland", "Hauptstadt")
  aInhalt = array(array("1", "Hessen", "Wiesbaden"), _
                  array("2", "Bayern", "München"), _
                  array("3", "Sachsen", "Dresden"), _
                  array("4", "Berlin", "Berlin"), _
                  array("5", "Saarland", "Saarbrücken"))

  REM Tabelleninhalt als Array aufbauen
  redim aTabInhalt(uBound(aInhalt) + 1)
  aTabInhalt(0) = aKopfzeile
  for i = 0 to Ubound(aInhalt)
    aTabInhalt(i+1) = aInhalt(i)
  next
```

```
  REM Tabelle am Ende des Textbereiches einfügen
  oTxtRange = ThisComponent.text.getEnd()
  oTab = thisComponent.createInstance("com.sun.star.text.TextTable")
  oTab.HoriOrient = 0
  oTab.leftMargin = 1000
  oTab.rightMargin = 1000
  oTab.initialize(UBound(aInhalt)+1+1, 3)   'Anzahl Datensätze + 1 Kopfzeile
  oTxtRange.getText().insertTextContent(oTxtRange, oTab, false)

  With oTab
    .setDataArray(aTabInhalt())
    .setName("MeineTabelle")
    REM jetzt Tabelle formatieren
    oTabTrenner = .TableColumnSeparators
    oTabTrenner(0).position = 667
    oTabTrenner(1).position = 5334
    .TableColumnSeparators = oTabTrenner
  end with

end sub
```

Das Ergebnis:

¶
¶

Nr¶	Bundesland¶	Hauptstadt¶
1¶	Hessen¶	Wiesbaden¶
2¶	Bayern¶	München¶
3¶	Sachsen¶	Dresden¶
4¶	Berlin¶	Berlin¶
5¶	Saarland¶	Saarbrücken¶

¶

Selbstverständlich lassen sich so auch umfangreichere Tabellen erzeugen – und nachträglich mit Formatvorlagen gestalten.

Allerdings bieten Tabellen auch „Tücken“:

Die Initialisierung einer Tabelle erfolgt von der Basis 1 – nicht wie sonst üblich der Basis 0. Eine Tabelle mit 2 Zeilen und 3 Spalten wird also wie folgt initialisiert:

```
oTab. initialize(2,3)
```

Der zu schreibende Daten-Array muss aber die folgende Dimension besitzen: Max-Index Zeilen = 1, Max-Index Spalten = 2

Wird mit der Funktion „setDataArray()“ gearbeitet (performant und unbedingt zu empfehlen), muss aber darauf geachtet werden, dass das letzte Element des Daten-Arrays nicht leer ist – sonst kann es einen Laufzeitfehler geben! Zur Not ergänzt man ein Leerzeichen.

Wird – wie hier im Beispiel – ein Tabellenname übergeben, so sollte vorher geprüft werden, ob es nicht bereits eine Tabelle mit diesem Namen gibt. Wenn ja, muss der Name geändert werden

(z.B. mit einer laufenden Nummer ergänzt), sonst gibt es einen Fehler. Verzichtet man auf einen eigenen Namen, so wird die Tabelle intern mit dem Namen „TabelleX“ bezeichnet, wobei „X“ eben eine eindeutige Nummer ist (die nächste folgende). Ein eigener Name ist deswegen sinnvoll, weil so die Tabelle leicht im Navigator identifiziert werden kann.

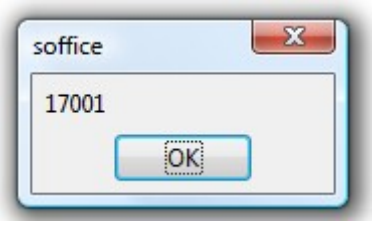
Die Berechnung der Tabellentrenner ist ziemlich „trickreich“, insbesondere dann, wenn man die Daten nicht als Fixwerte zur Verfügung stehen hat.

Die Breite einer Zeile in der Tabelle beträgt immer 10000! (also quasi 100%). Ein Tabellentrenner wird nun in relativem Maß von vorne (also dem linken Rand der Tabellenzeile) gemessen und gesetzt. Da die Gesamtbreite der Tabelle (und damit jeder Zeile) aber wiederum von der Seitenvorlage abhängt – jede Tabelle ist zunächst einmal genauso breit wie der Textbereich – sind mehrere Berechnungen notwendig, um die passende Spaltenbreite festzulegen.

Zurück zum dargestellten Beispiel:

Die Breite des Textbereiches kann nur über die Seitenvorlage extrahiert werden:

```
oPage = thisComponent.StyleFamilies.getByName("PageStyles").
        getByName(thisComponent.text.end.pageStyleName)
iBreite = oPage.width - oPage.leftMargin - oPage.rightMargin
msgbox iBreite
```



Ergebnis: 17 cm.

Unsere eingefügte Tabelle ist also immer 17 cm breit – abzüglich der übergebenen Abstände (left und right margin), in unserem Fall also: 17 cm - 1 cm - 1 cm = 15 cm.

Diese 15 cm bilden nun die Basis und sind 100% (oder 10000). Um eine gewünschte Spaltenbreite von 1 cm (für Spalte A) zu bekommen, ist nun Dreisatz-Rechnung gefragt: Spaltenbreite = (10000 * 1000) / 15000 – im Langtext: die 10000 sind das Maß der Breite der kompletten Zeile – diese entspricht dem 15000 (in 100stel Millimeter). Nun ist die Frage, wieviel Teile von den 10000 entsprechen nun 1 cm (1000 100stel Millimeter). Ergebnis: 667 – das ist nun die Position des ersten Tabellentrenners (Index Null).

Der zweite Tabellentrenner (trennt Spalte 2 und 3) soll nun in der Mitte der verbleibenden Breite positioniert werden: 15 cm - 1 cm = 14 cm – Spaltenbreite der Spalte 2 also 7 cm, der Trenner muss also 7cm + 1 cm (erste Spalte) vom linken Rand positioniert werden – also 8000 100stel Millimeter. Rechnung wie oben – Ergebnis: 5334

Die Arbeit mit Writer-Tabellen ist aufwendig und teilweise kompliziert.

7.2.1 Tabelle sortieren

Nicht so kompliziert ist das Sortieren einer Texttabelle – solange die Matrixstruktur der Tabelle nicht verändert wurde (zum Beispiel durch Zusammenfassen oder Teilen von Zellen).

Dann geht das ähnlich einfach wie in Calc:

```
Sub TabelleSortieren(oTab as variant)
  Dim oRange as variant, oSortKrit as variant

'  oTab = ThisComponent.getTextTables().getByName("MeineTabelle")
  if oTab.rows.count < 2 then
    msgbox "nur Titel-Zeile vorhanden - kann Tabelle nicht sortieren!"
    exit sub
  end if

  oRange = oTab.getCellRangeByName("B2:C" & oTab.rows.count)

  oSortKrit = oRange.createSortDescriptor()

  dim oSortFeld(0) as new com.sun.star.table.TableSortField

  oSortFeld(0).Field = 1
  oSortFeld(0).isAscending = true
  oSortFeld(0).FieldType = com.sun.star.util.SortFieldType.AUTOMATIC

  oSortKrit(2).Value = false
  oSortKrit(3).Value = 1    'max. Anzahl Sortfelder
  oSortKrit(4).Value = oSortFeld()

  oRange.sort(oSortKrit())

End Sub
```

Das Ergebnis ist in diesem Fall die sortierte Liste, wobei die erste Spalte nicht mit sortiert wurde.

7.3 Textmarken, Feldbefehle und Platzhalter

Geht es um Writer, so sind Textmarken ein typisches „Positionierungsmerkmal“. Textmarken sind leicht anzuspringen und einfach mit Inhalt zu füllen.

7.3.1 Textmarken

Textmarken können zwei verschiedene Zustände annehmen – expandiert und nicht expandiert. Dies ist für das Verständnis sehr wichtig:

Eine **nicht expandierte Textmarke** (Anfang- und Ende-Position liegen übereinander) ist „unsichtbar“ im Text. Man kann nun per Code oder auch in der UI dorthin springen und ab da weiterarbeiten. Die Textmarke wird nun je nach Befehl entweder „hinten“ angehangen oder verbleibt vor dem ersten neuen Zeichen. Wird das Zeichen gelöscht, verschwindet auch die

Textmarke. Ein einmal an einer Textmarke eingegebener Text (per Code) wird Bestandteil des Haupttextes und ist nicht wieder auslesbar – weiterer Text wird immer zusätzlich eingegeben.

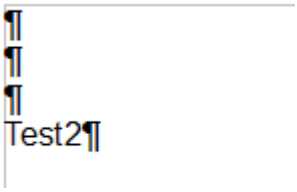
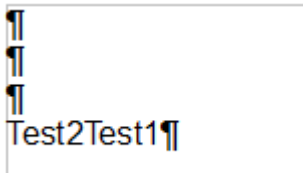
Anders ist das Verhalten einer **expandierten Textmarke**. In diesem Fall gibt es mindestens ein Zeichen zwischen dem Start der Textmarke und dem Ende der Textmarke (dies kann auch ein Leerzeichen sein!). Wird nun Text an der Textmarke eingegeben, so verbleibt dieser immer zwischen den beiden Enden – und kann jederzeit ausgelesen oder überschrieben werden. Jetzt wirkt diese Textmarke zu so etwas wie einem Textfeld.

In der UI wird eine expandierte Textmarke erzeugt, indem man den (View-)Cursor erweitert (also etwas markiert) und dann den Menübefehl **Einfügen/Textmarke** aufruft.

Die folgende Funktion schreibt Text an die Position, die durch die Textmarke bestimmt wird. Dabei wird vorher überprüft, ob der Name der Textmarke überhaupt existiert – ansonsten passiert nichts.

```
'/** setTextMarkeInhalt()
'************************************************************************.
'* @kurztext schreibt Text an eine vorhandene Textmarke in Writer
'* Funktion schreibt einen Text (String) an eine vorhandene Textmarke (Bookmark).
'*
'* @param1 oDokument as object  das Objekt es Writer-Dokumentes mit den Textfeldern
'* @param2 sName as string   Name der Textmarke - eindeutiges Kennzeichen!
'* @param3 sInhalt as string   der einzutragende Inhalt
'************************************************************************
'*/
sub setTextMarkeInhalt(oDokument as object, sName as string, sInhalt as string)
  dim oBookmark as variant

  if oDokument.getBookmarks().hasByName(sName) then  'Textmarkennamen ist eindeutig und gleich
    oBookmark = oDokument.getBookmarks().getByName(sName)
    oBookmark.anchor.string = sInhalt
  end if
end sub
```

Die folgenden beiden Bilder zeigen den zweifachen Aufruf der Funktion, einmal auf eine expandierte Textmarke, einmal auf eine nicht expandierte Textmarke: Die Textmarke steht jeweils in der vierten Zeile am Anfang, übergeben wird zunächst der String "Test1" und anschließend (2. Aufruf) der String "Test2".

Expandierte Textmarke	nicht expandierte Textmarke
	
oBookmark.anchor.string liefert hier nun "Test2"	oBookmark.anchor.string liefert hier nichts ("") – also einen leeren String.

Leider gibt es kaum eine Möglichkeit, eine Textmarke nachträglich zu expandieren, es ist also wichtig, dies schon bei der Erzeugung zu erledigen, wenn es gewünscht ist.

Eine Textmarke selbst wird wie folgt im Dokument erzeugt:

```
REM Fügt am Ende des Textes eine Textmarke ein
Sub TextmarkeEinfuegen(odoc as variant, sTextmarke as string)
  dim oBookmark as variant
  oBookmark = oDoc.createInstance("com.sun.star.text.Bookmark")
  oBookmark.setName(sTextmarke)
  oDoc2.text.insertTextContent(oDoc2.text.end, oBookmark, false)
End Sub
```

Wichtig hierbei sind zwei Punkte:

1. Die Position der Einfügung muss bekannt sein (hier Dokument, Haupttextbereich)
2. die Textmarke wird in diesem Fall nicht expandiert eingefügt! Die Einfügeposition ist abstrakt ein Punkt (.end!)

Praktischer Nutzen solcher Textmarken:

Neben der klassischen Verwendung kann man aber Textmarken auch im Makro nutzen. Folgendes Beispiel: Ein Dokument, das noch per Makro „zusammengebaut“ wird, enthält einen Teil, der später erneut kopiert werden soll. Dieser Teil ist aber noch nicht bekannt. Das Vorgehen ist nun wie folgt:

Der Beginn des Kopierteils ist definiert. Es wird also zunächst eine Textmarke in das Dokument eingefügt, dann folgt der Textteil, der kopiert werden soll. Dieser wird nun Stück für Stück aufgebaut. Am Ende erfolgt erneut das Setzen einer Textmarke.

Zum Kopieren muss nun der Teil „markiert“ werden: Dazu springt man die erste (gesetzte) Textmarke an, setzt dort den View-Cursor hin, sucht jetzt die zweite Textmarke, und expandiert den View-Cursor bis dort hin. Nun ist der – vorher unbekannte – Teil markiert und kann entsprechend kopiert werden.

Das folgende Code-Beispiel erledigt genau dies:

```
'/** copyDoc
'*************************************************************************.
'* @kurztext  kopiert Teile des Dokumentes an eine Textmarke
'* Diese Funktion kopiert Teile eines Dokumentes, die eingeschlossen sind, in die
```

```
'* Textmarken sBMCopyStart und sBMCopyEnd (jeweils Konstanten) an die Position
'* der Textmarke aBMCopyZiel.
'* benutzt wird dazu der ViewCursor und die TransferabeDatas.
'*
'* @param  oCCDoc as object  Das Objekt des Dokumentes, in dem kopiert wird
'*
'* @return  Flag as Boolean   True, wenn alles ok, False bei Fehler
'************************************************************************
'*/
function copyDoc(oCCDoc as variant)
  dim oCur as variant  'ViewCursor
  dim oTCur as variant  'Textcursor
  dim oCopyDaten as  variant

  REM Textmarke "Start" und "Ende" finden
  oTextmarken = oCCDoc.getBookmarks()
  if NOT oTextmarken.hasByName(sBMCopyStart) AND _
     NOT oTextmarken.hasByName(sBMCopyEnd)  then
    copyDoc = false
    exit function
  end if
  REM Textmarke Ziel abfragen
  if NOT oTextmarken.hasByName(sBMCopyZiel) then
    REM Keine Zielmarke
    copyDoc = false
    exit function
  end if
  oCur = oCCDoc.getCurrentController().getViewCursor()
  oCur.gotoRange(oTextmarken.getByName(sBMCopyStart).getAnchor(), false)
  oCur.gotoRange(oTextmarken.getByName(sBMCopyEnd).getAnchor(), true)

  oCopyDaten = oCCDoc.getCurrentController().getTransferable()
  oCur.gotoRange(oTextmarken.getByName(sBMCopyZiel).getAnchor(), false)
  oTCur = oCCDoc.text.createTextcursorByRange(oCur)

  if NOT oTCur.isStartOfParagraph() then 'kein leerer Ansatz
    oCCDoc.text.insertControlCharacter(oCCDoc.text.end, _
            com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, false)

    oCur.gotoRange(oTextmarken.getByName(sBMCopyZiel).getAnchor(), false)
  end if
  oCCDoc.getCurrentController().insertTransferable(oCopyDaten)
  copyDoc = true
End function
```

Um eine expandierte Textmarke einzufügen, bedarf es eines „Tricks“: Die Textmarke wird jetzt über einer Textpassage (markiert, expandiert) eingefügt, dadurch ist die Textmarke selbst nun auch expandiert. Im Beispiel wird ein Leerzeichen als „Anker“ genutzt und mit Hilfe eines Textcursors eingefügt:

```
REM Fügt am Ende des Textes eine expandierte Textmarke ein
Sub TextmarkeEinfuegen(odoc as variant, sTextmarke as string)
  dim oBookmark as variant, oTextCursor as variant

  oBookmark = oDoc.createInstance("com.sun.star.text.Bookmark")
```

```
  oBookmark.setName(sTextmarke)
  REM Einfügeposition definieren - als Bereich
  oTextCursor = oDoc.text.createTextCursorByRange(oDoc.text.end)
  oTextCursor.setString(" ")
  REM Textmarke einfügen
  oDoc.text.insertTextContent(oTextCursor, oBookmark, true)
  oBookMark.anchor.string = "Hallo"  'Beispiel einer Texteingabe
End Sub
```

Aber auch hier gilt: Die Position der Textmarke muss bekannt sein! Im Beispiel wurde der Einfachheit halber immer das Textobjekt des Dokumentes verwendet, dies geht aber selbstverständlich auch in allen anderen Objekten – wie Textrahmen, Bereichen, Tabellenzellen und so weiter.

Zur praktischen Anwendung einer expandierten Textmarke:

Sie dient als Sprungstelle für austauschbaren Text und ist dabei sehr flexibel und sehr performant. Im Grunde sind Feldbefehle oder Platzhalter auch nichts anderes als expandierte Textmarken, die allerdings intern mit Funktionen vorbelegt sind.

7.3.2 Feldbefehle oder auch Textfelder

Zu den eingesetzten Feldbefehlen zählen typischerweise nur die wichtigsten vordefinierten – wie Dateinamen, Pfade, Datum und Uhrzeit sowie Seitenzahlen.

Feldbefehle werden wie andere Textinhalte auch einfach in den Textbereich integriert. Allerdings muss man auch hier vorher wissen, in welchen Textbereich man diese einfügen möchte.

Das folgende Beispiel fügt einige Felder am Beginn des Textbereiches des Dokumentes ein – dadurch wird das Prinzip klar:

```
Sub Textfelder
  DIM oDoc as Variant, oFeldDatum as Variant, oFeldAutor as Variant
  dim oAnker as Variant, oFeldWorte as Variant

  oDoc = thisComponent

  oFeldDatum = oDoc.createInstance("com.sun.star.text.TextField.DateTime")
  oFeldDatum.IsFixed = True
  oFeldDatum.IsDate = True

  oFeldAutor = oDoc.createInstance("com.sun.star.text.TextField.DocInfo.ChangeAuthor")

  oFeldWorte = oDoc.createInstance("com.sun.star.text.TextField.WordCount")
  oFeldWorte.NumberingType = com.sun.star.style.NumberingType.ARABIC

  oAnker = oDoc.text.getStart()

  With oDoc.text
    .insertString(oAnker,"Datum: ", false)
    .insertTextContent(oAnker, oFeldDatum, false)
    .insertString(oAnker," - , Bearbeiter: ", false)
```

```
    .insertTextContent(oAnker, oFeldAutor, false)
    .insertString(oAnker," - , Anzahl Wörter: ", false)
    .insertTextContent(oAnker, oFeldWorte, false)
    .insertControlCharacter(oAnker, 0, false)
  end with

End Sub
```

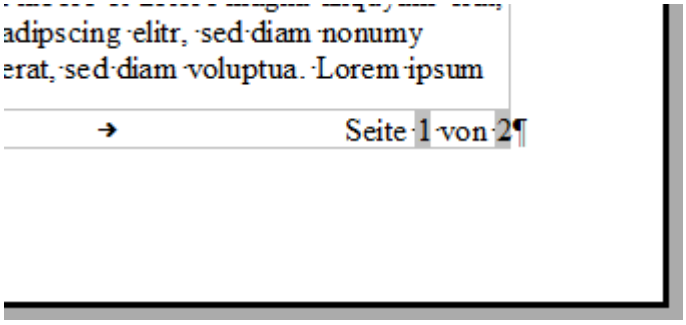
Eine typische Anwendung ist beispielsweise das Einfügen der Seitennummern in die Fußzeile einer Seitenvorlage, hier der Einfachheit halber die Standard-Seitenvorlage.

```
Sub FusszeileUndSeitenzahlEinfuegen
  DIM oDoc as Variant,  oCursor as Variant
  DIM oSVorlage as Variant, oSVorlagen as Variant
  DIM oFeldSNr as Variant, oFEldSAnz as Variant
  oDoc = ThisComponent

  oSVorlagen = oDoc.getStyleFamilies.getByName("PageStyles")

  oSVorlage = oSVorlagen.getByName("Standard")
  oSVorlage.FooterIsOn = True

  oCursor = oSVorlage.FooterText.Text.CreateTextCursor()

  oFeldSNr = oDoc.createInstance("com.sun.star.text.TextField.PageNumber")
  oFeldSNr.NumberingType = com.sun.star.style.NumberingType.ARABIC
  oFeldSNr.SubType = com.sun.star.text.PageNumberType.CURRENT

  oFeldSAnz = oDoc.createInstance("com.sun.star.text.TextField.PageCount")
  oFeldSAnz.NumberingType = com.sun.star.style.NumberingType.ARABIC

  s = chr(09) & "Seite "

  With oSVorlage.FooterText.Text
    .insertString(oCursor, s, false)
    .insertTextContent(oCursor, oFeldSNr, false)
    .insertString(oCursor, " von ", false)
    .insertTextContent(oCursor, oFeldSAnz, false)
  end with

End Sub
```

In diesem Fall wird nicht nur die Fußzeile der Seitenvorlage aktiviert (eingeschaltet), sondern es werden auch zwei Felder aktiviert und eingefügt: Die aktuelle Seite sowie die Anzahl aller Seiten.

Damit der Text rechtsbündig erscheint, werden der Fußzeile zunächst zwei Tabulatoren (chr(09)) übergeben. Da die Fußzeile standardmäßig drei Bereiche besitzt (vorne – linksbündig, mittig – zentriert, hinten – rechtsbündig) und dieses durch entsprechende Tabs realisiert wird, gelingt dies mit den zwei Tabulatoren. Selbstverständlich kann dies ebenfalls per Skript geregelt werden.

Das Ergebnis sieht dann wie folgt aus:



Sucht man ein spezielles Textfeld im Dokument, so muss man zunächst eine Enumeration erzeugen – und dann entsprechende Vergleichswerte definieren. Der folgende Code liest alle vorhandenen Textfelder aus und listet sie in einer Message-Box:

```
sub TextFelderanzeigen
  Dim oDoc as Variant, oTextFelder as Variant, s2 as String
  Dim oFeld as Variant, s as string, iAnz as integer

  oDoc = ThisComponent

  oTextFelder = oDoc.getTextFields.createEnumeration()
  iAnz = 0

  Do while oTextFelder.hasMoreElements()
    iAnz = iAnz + 1
    oFeld = oTextFelder.nextElement()
    s = s & oFeld.getPresentation(true) & " : "
    if oFeld.supportsService("com.sun.star.text.TextField.Annotation") then
      s = s & "Autor: " & oFeld.Author &  CHR(13)
      S = s & oFeld.getPresentation(true) & " : Bemerkung: " & oFeld.Content & CHR(13)
    else
      s = s & oFeld.getPresentation(false) & CHR(13)
    end if
  loop
  s = "Diese Dokument enthält " & CStr(iAnz) & " Textfelder:" & CHR(13) & s

  msgbox (s, 0, "Textfelder in diesem Dokument")

end sub
```

7.3.3 Platzhalter

Auch Platzhalter sind nur „spezielle“ Textfelder, sie werden aber gerne dazu genutzt, um den/die Benutzer/in zu „führen“ und spezielle Eingaben zu tätigen. Platzhalter sind normalerweise mit einem Hinweistext im Text sichtbar – gibt der/die Benutzer/in aber etwas an dieser Stelle ein, so geht die Feldeigenschaft verloren und die Eingabe überschreibt das Feld.

7.4 Grafiken

Grafiken sind Teile der Drawpage – eines Sammelobjektes des Textdokumentes. Sie liegen „quasi“ auf dem Dokument und werden an einer speziellen Stelle „verankert“. Ansonsten sind es eigenständige Objekte mit eigenen Eigenschaften und Methoden.

Als Grafikobjekte zählen quasi alle Zeichnungen, Zeichenelemente wie Linien und Formen, aber auch (Formular-)Kontrollelemente, Icons und vieles mehr. Auch Bilder sind typischerweise Elemente der Drawpage, wobei es leider zwei verschiedene Arten von „Bildern“ gibt:

- Bilder, die über den Menü-Befehl **Einfügen/Bild** dem Dokument hinzugefügt wurden. Diese Bilder unterstützen den Service "com.sun.star.text.TextGraphicObject"
- Bilder, die direkt der Drawpage hinzugefügt wurden (zum Beispiel über den Umweg Draw, per „Copy&Paste“ oder auch aus der Gallery.). Alle diese Grafiken und Bilder unterstützen den Service "com.sun.star.drawing.ShapeCollection"

Während der erste Fall (TextGraficObject) nur relativ wenige Möglichkeiten der Manipulation erlaubt (so lässt sich dieses Bild weder drehen noch wirklich „beschneiden“), bieten die Objekte der ShapeCollection jede Menge Manipulationsmöglichkeiten.

Doch zur Praxis:

Einfügen eines Bildes in ein Dokument an der Ende-Position des Textbereiches – zentriert. Das Bild wird dabei skaliert auf 5 cm Breite – falls es größer ist.

```
sub BildEinfuegen
  dim oDoc as variant
  dim oViewC as variant, oTC as variant  'Viewcursor, Textcursor
  dim vStyle as variant, vParaStyles as variant
  dim sBildURL as string
  dim oBild as variant, oBildO as variant  'Bildobjekt, Originalbildobjekt
  dim oBH as variant, oBB as variant  'Bildhöhe, Bildbreite
  dim iBreite as long
  dim iKorrFaktor as long   'Korrekturfaktor

  iKorrFaktor = 26.45

  odoc = thisComponent
  sBildURL = convertToURL("N:\allg_Daten\Bilder\Apfelbaum.jpg")
  iBreite = 5000

  REM Absatzvorlage für Bild erzeugen
  vParaStyles = oDoc.StyleFamilies.getByName("ParagraphStyles")
  if not vParaStyles.hasByName("Bildanker") then
       vStyle = oDoc.createInstance("com.sun.star.style.ParagraphStyle")
       with vStyle
      .CharHeight = 2
      .ParaAdjust = com.sun.star.style.ParagraphAdjust.CENTER
       end with
       vParaStyles.insertByName("Bildanker", vStyle)
  end if
  REM Bild einfügen
  oTC = oDoc.text.createTextCursor
```

```
  oTC.gotoEnd(false)
  oTC.ParaStyleName = "Bildanker"
  oBildO = getGraphFromUrl(sBildURL)
  REM Bildgröße Lesen
  if oBildO.size100thMM.Height = 0 then
    oBH = oBildO.size.Height * iKorrFaktor
    oBB = oBildO.size.Width * iKorrFaktor
  else
    oBH = oBildO.size100thMM.Height
    oBB = oBildO.size100thMM.Width
  end if
  oBild = oDoc.createInstance("com.sun.star.text.GraphicObject")
  oBild.GraphicURL = sBildURL
  REM skalieren
  if oBH >= oBB then   'Hochformat
    if oBH > iBreite then 'nur skalieren, wenn Bild zu groß
      oBild.height = iBreite
      oBild.width = iBreite * oBB / oBH
    else
      oBild.height = oBH
      oBild.width = oBB
    end if
  else  'Querformat
    if oBB > iBreite then 'nur skalieren, wenn Bild zu groß
      oBild.height = iBreite * oBH / oBB
      oBild.width = iBreite
    else
      oBild.height = oBH
      oBild.width = oBB
    end if
  end if
  oDoc.Text.insertTextContent(oTC, oBild, false )
end sub


REM liefert den Input-Stream eines Bildes
function getGraphFromUrl(sFileUrl as String) as Object
  dim oProvider as variant
  oProvider = createUnoService("com.sun.star.graphic.GraphicProvider")
  Dim oPropsIN(0)as new com.sun.star.beans.PropertyValue
  oPropsIN(0).Name  = "URL"
  oPropsIN(0).Value = sFileUrl
  getGraphFromUrl = oProvider.queryGraphic(oPropsIN())
end function
```

Das Bild wurde nun als „TextGraficObjekt“ eingefügt – simuliert also den Weg der UI mit dem Menübefehl **Einfügen/Bild...**.

Als nächstes soll dieses Bild gedreht werden. Bei einem Grafikobjekt ginge dies über die Eingabe eines „Rotationangle“ – also eines Drehwinkels. Bei einem TextGrafikObjekt geht dieses aber nicht. Insofern muss das Bild zunächst in eine Grafik umgewandelt werden. Dies erledigt man am einfachsten über den Umweg mit einer Draw-Datei (versteckt öffnen).

Der folgende Code „dreht“ das markierte Bild um 45° (gegen den Uhrzeigersinn). Dazu wird dem Bild zunächst ein eindeutiger Name verpasst – anschließend wird es in eine Draw-Datei kopiert – dort wird es automatisch in ein GraphicShape-Objekt umgewandelt (Draw kennt keine

anderen Objekte!) und wieder zurückkopiert. Jetzt ist das Bild auch in Writer ein GraphicShape-Objekt und kann gedreht werden. Über den Namen kann es nun auch eindeutig identifiziert werden.

Der Code berücksichtigt bereits den Fall, dass das markierte Bild schon ein GrapicShape-Objekt ist – dann wird es sofort gedreht

```
'/** WT_BildDrehen()
'*************************************************************************.
'* @kurztext dreht ein markiertes Bild in Writer
'* Diese Funktion dreht ein markiertes Bild in Writer. Dazu wird ein Grafik-Objekt
'* wie dies durch Einfügen - Bild entstanden ist, zunächst in ein Shape
'* Objekt verwandelt (via Draw) - dann gedreht.
'*************************************************************************
'*/
Sub WT_BildDrehen
  dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")
  oDoc = thisComponent
  oSel = oDoc.getCurrentSelection

  if oSel.supportsService("com.sun.star.drawing.ShapeCollection") then
    REM Grafic-Element - kann direkt gedreht werden - nur erstes Element!
    oBild = oSel.getByIndex(0)
    if NOT oBild.supportsService("com.sun.star.drawing.Shape") then
      msgbox ("Es wurde kein Bild oder Grafik Objekt markiert!", 16, "Fehler")
      exit sub
    end if

  elseif NOT oSel.supportsService("com.sun.star.text.TextGraphicObject") then
    msgbox ("Es wurde kein Bild oder Grafik Objekt markiert!", 16, "Fehler")
    exit sub
  else    'Das Bild muss zunächst in eine Grafik-Shape umgewandelt werden - hier über den Umweg
über Draw!
    sBN = "Bild_" & format(Now(),"ddmmhhmmss")  'eindeutiger Bildname
    dispatcher.executeDispatch(oDoc.CurrentController.Frame, ".uno:Copy", "", 0, Array())
    REM Draw Dokument hidden öffnen
    dim Arg(0) as new com.sun.star.beans.PropertyValue
    arg(0).name = "Hidden"
    arg(0).value = true
    oDoc2 = StarDesktop.loadComponentFromURL("private:factory/sdraw","_blank", 0, Arg())
    dispatcher.executeDispatch(oDoc2.CurrentController.Frame, ".uno:Paste", "", 0, Array())

    oDoc2.getCurrentSelection.getByIndex(0).name = sBN  'eindeutigen Namen vergeben
    wait(200) 'kurz warten - damit OOo die Änderung mitbekommt!
    dispatcher.executeDispatch(oDoc2.CurrentController.Frame, ".uno:Copy", "", 0, Array())
    oDoc2.close(true)  'Draw Dokument schließen
    dispatcher.executeDispatch(oDoc.CurrentController.Frame, ".uno:Paste", "", 0, Array())

    REM Bild suchen
    for i = 0 to odoc.drawPage.count -1
      oBild = odoc.drawPage.getByIndex(i)
      if oBild.name = sBN then exit for
    next
    if NOT (oBild.name = sBN) then exit sub  'Ende, falls Bild nicht gefunden wurde
    oDoc.getCurrentController.select(oBild)  'Bild auswählen
  end if
  REM Drehwinkel 45°
```

```
    nDrehW = 45 * 100

    REM Jetzt drehen
    oBild.RotateAngle  = Int(oBild.RotateAngle + nDrehW)

End Sub
```

und das Ergebnis (verkleinert):

Neben dem „klassischen“ Bild gehört auch die Integration von Logos zu den praktischen Aufgaben der Grafik-Verarbeitung in Writer.

Bilder (egal in welcher Form) haben immer auch ein Ankerobjekt. Beachten Sie unbedingt die Informationen in Abschnitt 7.8.

7.5 Sonstige Objekte

Neben Bildern spielen insbesondere Textrahmen eine große praktische Rolle in Writer-Dokumenten. (Text-)Rahmen sind eigenständige Textbereiche, die ähnlich wie Bilder beliebig im Dokument verankert werden können und dann Text oder andere Inhalte aufnehmen können. Textrahmen werden in einem eigenen Sammelcontainer pro Dokument organisiert (TextFrames()) und können direkt über den Namen identifiziert werden.

Ähnlich wie Grafiken können Textrahmen gedreht und beliebig positioniert werden. Da sie ebenfalls fix im Dokument verankert werden können (Seitenverankerung) eignen sie sich ideal, um feste Layout-Teile zu erzeugen. Diese „wandern“ nicht mit dem Textfluss und sind einfach zu befüllen.

Das folgende Beispiel erzeugt eine „linke Randleisten-Information“, die – um 90° gedreht – am linken Seitenrand platziert wird und auf Seite 1 erscheinen soll. Die Informationen der Abmessungen etc. sind im Code-Beispiel fest eingetragen – in der Praxis sollen diese natürlich durch Variablen oder Konstanten ersetzt werden!

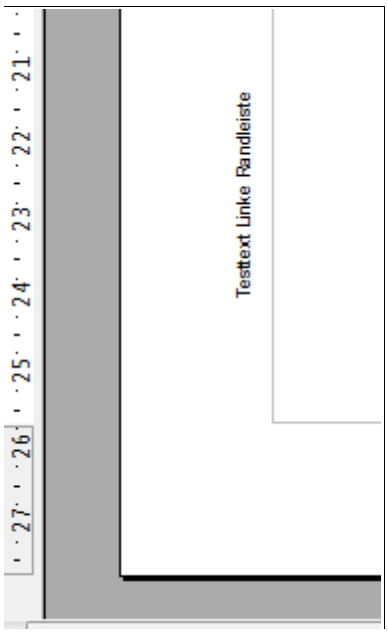
```
'/** MAK131_LRLErzeugen()
'*************************************************************************.
'* @kurztext erzeugt die Randleiste im Dokument auf der benannten Seite
```

```
'* Diese Funktion erzeugt die Randleiste im Dokument auf der benannten Seite
'*
'* @param1  oDokument as variant   Das Dokument
'* @param2  iSeite as integer      Seite, auf der die RL eingefügt werden soll
'* @param3  sText as string        Text der Randleiste
'*
'*************************************************************************
'*/
sub MAK131_LRLErzeugen(oDokument as variant, iSeite as integer, sText as string)
  dim oTextShape as variant, oDrawPage as variant

  oDrawpage = oDokument.getDrawPage()

  oTextShape = oDokument.createInstance("com.sun.star.drawing.TextShape")

  with oTextShape
    .AnchorPageNo = iSeite   'Seite
    .AnchorType = 2     'An der Seite
    .Position = MAK131_Helper.erzeugePunkt(1500, 26050)  'Position Punkt oben links (x =15 mm,
y = 260,5 mm) (aus Gestaltungsrichtlinien)
    .size = MAK131_Helper.erzeugeSize(19000, 450)        'Größe (100 mm lang, 4,5 mm hoch)
    .RotateAngle = 9000   '90° nach links drehen
    .name = "LHMLRL_S" & iSeite
    .TextWrap = com.sun.star.text.WrapTextMode.THROUGHT    'Textdurchlauf: Durchlauf
    .MoveProtect = true
    .SizeProtect = true
  end with
  REM Objekt der Seite zufügen
  oDrawPage.add(oTextShape)    'Objekt wird im Hintergrund (LayerID = 0) eingefügt!

  REM Objekt nun entsprechend anpassen (aus Gestaltungsrichtlinien)
  with oTextShape
    .string = sText
    .CharFontName = "Arial"
    .CharHeight = 7   '7pt
    .LayerID = 1    'Vordergrund
  end with
end sub
```

Der Aufruf lautet dann wie folgt:

```
MAK131_LRLErzeugen(thisComponent, 1, "Testtext Linke Randleiste")
```

Und das Ergebnis:



Zur Vervollständigung: Positionierungen und Größenobjekte werden bei der Arbeit mit Grafiken und sonstigen verankerten Objekten immer wieder benötigt. Die folgenden beiden Funktionen liefern die erforderlichen internen Objekte:

```
'/** ErzeugePunkt()
'*************************************************************************.
'* @kurztext  erzeugt einen Punkt als Objekt
'* Diese Funktion erzeugt einen Punkt als Objekt
'*
'* @param1 x as long    X-Koordinate
'* @param1 y as long    Y-Koordinate
'*
'* @return  oPunkt as com.sun.star.awt.Point
'************************************************************************
'*/
Function ErzeugePunkt(byVal x as long, byVal y as long) as com.sun.star.awt.Point
  dim oPunkt as new com.sun.star.awt.Point
  oPunkt.X = x
  oPunkt.Y = y
  ErzeugePunkt = oPunkt
End Function

'/** ErzeugeSize()
'*************************************************************************.
'* @kurztext  erzeugt ein Größenobjekt
'* Diese Funktion erzeugt ein Größenobjekt
'*
'* @param1 breite as long   Breite des Objektes in 1000st Milimetern
'* @param1 hoehe as long    Höhe des Objektes in 1000st Milimetern
'*
'* @return  oSize as com.sun.star.awt.Size
'************************************************************************
'*/
Function ErzeugeSize(byVal breite as long, byVal hoehe as long) as com.sun.star.awt.Size
```

```
  dim oSize as new com.sun.star.awt.Size
  oSize.Width = breite
  oSize.Height = hoehe
  ErzeugeSize = oSize
End Function
```

7.6 Suchen und Ersetzen

Suchen und Ersetzen im Writer-Dokument ist eine sehr performante Methode, (bekannte) Textteile schnell zu identifizieren und an dieser Position Aktionen durchzuführen.

Neben dem „klassischen“ Suchen und Ersetzen – auf das ich hier nicht näher eingehe – bietet sich zum Beispiel die Möglichkeit an, mit speziellen Platzhaltern (Codes) in Dokumenten und Vorlagen zu arbeiten. Mit solchen „Codes“ lässt sich flexibler arbeiten als mit Feldern, so können zum Beispiel auch komplexere Ausdrücke dort untergebracht werden.

Das Prinzip ist jedoch immer das gleiche: Das Suchergebnis ist immer ein Textcursor-Objekt, expandiert und als Inhalt das Suchergebnis. Dieses kann direkt bearbeitet oder überschrieben werden.

So lassen sich beispielsweise in einer Vorlage folgende Textinhalte definieren:

„Die folgende Ordungswidrigkeit wurde *BCT(in der Wohnung|auf der Straße|im Wald)*ECT begangen.“

Es wurden in diesem Fall zwei Platzhalter definiert, die eine Bedingung einschließen. Der Bedingungstext besteht aus drei Alternativen, getrennt durch die Pipe (|). Es ist nun einfach möglich, den Text zunächst nach dem Startpaltzhalter (*BCT) zu durchsuchen, dann von dieser Position ausgehend zum nächsten Endplatzhalter (*ECT) und dann den Text zwischen den Markierungen auszulesen und zu ersetzen (durch die Wahl – hier soll fest gelten: 2. Eintrag, Index 1 – in der Praxis wird dies entweder über einen Dialog erfragt oder von anderen Kriterien abgeleitet).

Der Code:

```
public const sBDS as string = "*BCT"  'Wenn-Bedingungen Identifier Start
public const sBDE as string = "*ECT"  'Wenn-Bedingungen Identifier Ende

'/** WennBedingungenAufloesen
'*************************************************************************.
'* @kurztext  löst die bedingten Texte im Dokument auf
'* Diese Funktion löst die bedingten Texte im Dokument auf. Dabei wird nach den
'* Schlüsselwörtern für bedingte Texte gesucht, dieser Bereich ausgelesen,
'* die Bedingung ausgeführt und das Ergebnis an Stelle dieses Textes geschrieben.
'*************************************************************************
'*/
sub WennBedingungenAufloesen
  dim txt as string
  dim oWSucheStart as variant
  dim oWSucheEnd as variant
```

```
  dim oWSErg as variant
  dim oWSErgEnde as variant
  dim oDoc as variant

  oDoc = thisComponent

  oWSucheStart = oDoc.createSearchDescriptor()
  oWSucheEnd = oDoc.createSearchDescriptor()
  oWSucheStart.setSearchString(sBDS)
  oWSucheEnd.setSearchString(sBDE)
  oWSErg = odoc.findfirst(oWSucheStart)        'erster Treffer

  if isNull(oWSErg) then exit sub    'Ende, wenn keine Wenn-Bedingung vorhanden ist

  oWSErgEnde = odoc.findfirst(oWSucheEnd)    'erste Endemarkierung

  if isNull(oWSErgEnde) then exit sub    'Ende auch, wenn keine passende Wenn-Ende-Bedingung
vorhanden ist

  oWSErg.gotoRange(odoc.findnext(oWSErg.End, oWSucheEnd).end, true)
  REM für ersten Treffer - Bed. auflösen
  WennBedingungAusfuehren(oWSErg)

  REM für jeden weiteren Treffer ausführen
  Do until isNull(oDoc.findNext(oWSErg.End, oWSucheStart))
    oWSErg = oDoc.findNext(oWSErg.End, oWSucheStart)
    oWSErg.gotoRange(odoc.findnext(oWSErg.End, oWSucheEnd).end, true)
    WennBedingungAusfuehren(oWSErg)
  loop

end sub

REM lediglich Rumpf-Basis Code! Detaillierte Auflösung möglich.
sub WennBedingungAusfuehren(oSuchErg as variant)
  dim aBedArray()

  aBedArray = split(mid(oSuchErg.string,len(sBDS)+2, len(oSuchErg.string)-len(sBDS)-len(sBDE)-
2),"|")

  REM Suchergebnis jetzt durch den passenden Eintrag (hier 2. Eintrag) ersetzen
  oSuchErg.string = aBedArray(1)

end sub
```

Da die Formatierung des ursprünglichen Textbereiches erhalten bleibt, können auf diese Weise sehr komplexe Lösungen gebaut werden.

So wäre es durchaus möglich, den zwischen den Identifiern Start und Ende untergebrachten Daten auch noch Bedingungen oder Handlungsanweisungen mitzugeben – ebenfalls getrennt durch spezielle Zeichen. Somit lassen sich ziemlich komplexe Vorlagen erstellen, die beim /bei der Benutzer/in entweder vollautomatisiert (anhand von Datenbank-Parametern oder Arbeitsplatzwerten) oder mittels manueller Definition mit Hilfe von Dialogen aufgebaut werden.

7.7 Textbereiche

Der Vorteil von Textbereichen in Writer liegt auf der Hand: Es können sehr komplexe Textteile in einem Dokument gespeichert sein und nur bei Bedarf ein- oder ausgeblendet werden. Dies ist insbesondere immer dann hilfreich, wenn die Textbereiche umfangreiche Formatierungen und andere Besonderheiten aufweisen (Tabellen, Grafiken etc.). Der Textbereich ist ja quasi ein eigenständiges Dokument – aber ohne Kopf- und Fußzeilen.

Per Makro muss nur das Textobjekt des Textbereichs angesprochen werden – dann stehen einem im Textrahmen oder im Dokument selbst alle Möglichkeiten dieses Textobjektes zur Verfügung.

Ein Bereich wird erzeugt als Instanz des Services com.sun.star.text.TextSection und wird im Dokument verankert (Haupttextbereich).

```
Sub BereichErzeugen
  oDoc = thisComponent

  if oDoc.getTextSections().hasByName("MeinNeuerBereich") then exit sub

  oBereich = oDoc.createInstance("com.sun.star.text.TextSection")
  with oBereich
    .backColor = rgb(250,150,150)
    .isProtected = true
    .name = "MeinNeuerBereich"
  end with

  oDoc.Text.insertTextContent(oDoc.text.getEnd(), oBereich, false)

  oBereich.getAnchor().setString("mein neuer Bereich..." & _
              chr(13) & chr(13) & "für den Benutzer schreibgeschützt!")

End Sub
```

Bereiche müssen einen eindeutigen Namen besitzen – ist der Name schon vergeben, so führt das erneute Erzeugen allerdings nicht zu einem Fehler, sondern der Name wird einfach ignoriert und der Bereich erhält einen automatischen Namen (also so etwas wie „Bereich1“ etc.).

Wird der Name also noch benötigt, so muss vorher unbedingt geprüft werden, ob es diesen schon gibt, und dann müssen entsprechende Maßnahmen ergriffen werden (löschen nach Rückfrage, Namensanpassung oder Abbruch – um nur einige zu nennen).

Der Bereich selbst wird jetzt über den Namen angesprochen und aktiviert:

```
oTextBereich = oDoc.getTextSections().getByName("MeinNeuerBereich")
```

Um einen Basic-Laufzeitfehler zu vermeiden, muss allerdings auch hier zunächst geprüft werden, ob es den Bereich tatsächlich gibt:

```
if Doc.getTextSections().hasByName("MeinNeuerBereich") then
  oTextBereich = oDoc.getTextSections().getByName("MeinNeuerBereich")
```

```
else
  '(Aktion - Bereich existiert nicht...)
end if
```

So wie Bereiche erzeugt wurden, können sie auch gelöscht werden – man muss dabei aber bedenken, dass ein Bereich nicht wie ein Textrahmen-Objekt komplett gelöscht wird, sondern dass nur das Bereichsobjekt (als eigenständiges Objekt) aufgelöst, der Inhalt aber in das Parent-Objekt integriert wird.

Die Code-Zeile

```
oDoc.Text.removeTextContent(oDoc.getTextSections().getByName("MeinNeuerBereich"))
```

führt also nicht dazu, dass der Inhalt des Textbereiches inklusive des Textbereiches „verschwindet“, sondern lediglich dazu, dass die eigenständige Position des Inhaltes im Rahmen des Gesamtdokumentes aufgelöst und dieser Fließtext integriert wird. Dabei gehen dann auch eigenständige Eigenschaften wie beispielsweise eine Hintergrundfarbe verloren.

Will man einen Textbereich vollständig entfernen, so muss man zunächst das Textobjekt des Textbereiches leeren (alles markieren – löschen), dann alle im Textrahmen verankerten Objekte entfernen und schließlich das Textrahmen-Objekt selbst entfernen. Eventuell muss man anschließend noch einen übriggebliebenen leeren Absatz im Hauptdokument löschen.

Textbereiche eignen sich also weniger, um Informationen per Makro in ein Dokument zu bringen und diese später wieder zu löschen – dazu nutzt man besser Textrahmen.

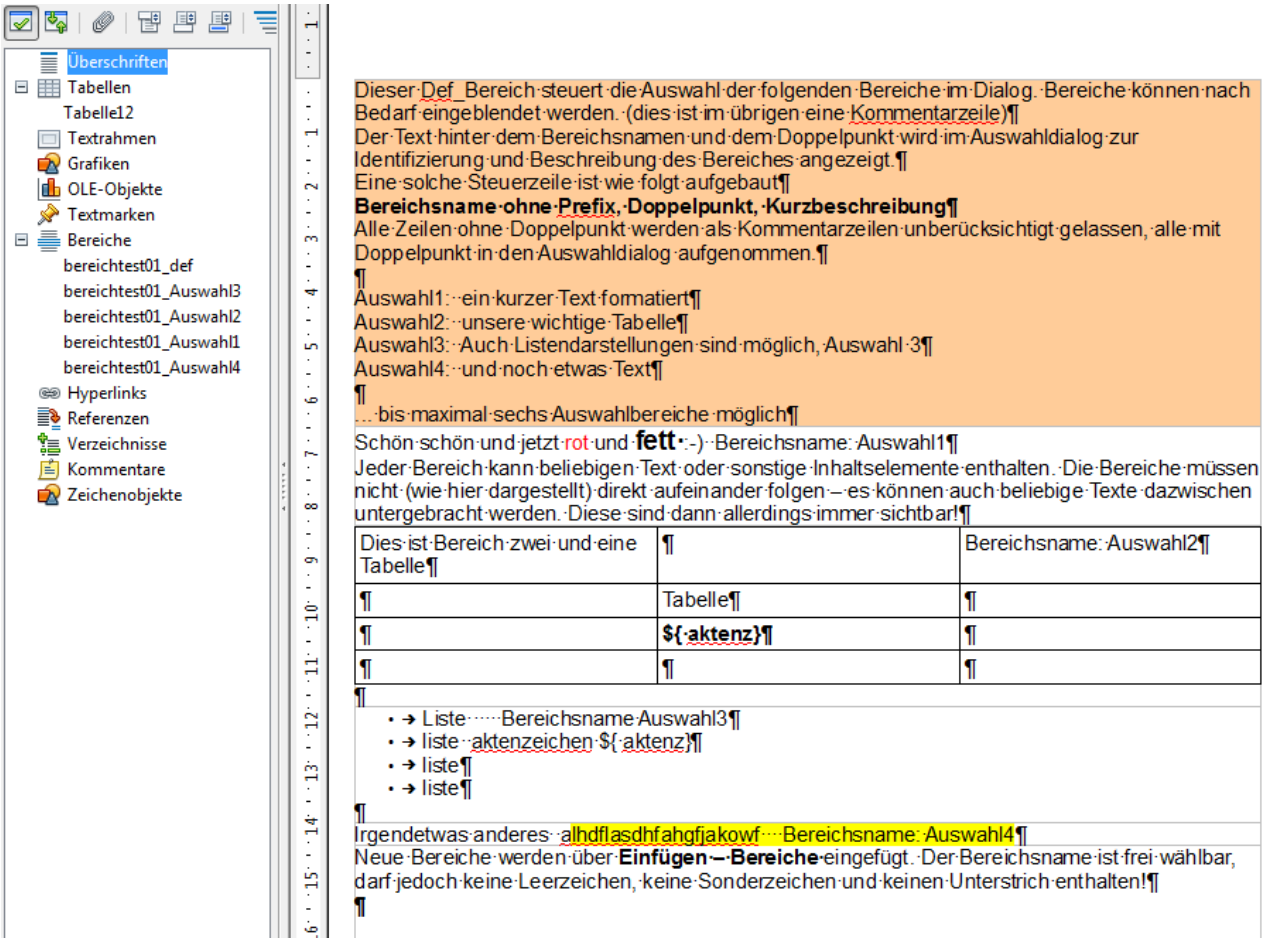
Textbereiche sind jedoch aufgrund ihrer Möglichkeit, ausgeblendet zu werden, ideal dazu geeignet, große Vorlagendokumente zu erzeugen und Informationen dann individuell per Makro ein- oder auszublenden, auch abhängig von anderen Informationen. Das folgende Beispiel zeigt eine Vorlage, die diverse Textbereiche beinhaltet mit unterschiedlichen Textinhalten. Zusätzlich gibt es einen sogenannten „Def“-Bereich, der für die interne Verwaltung der Bereiche zuständig ist und Hilfetexte beinhaltet:

Ein solches Dokument kann nun als Vorlage dienen – und von geschultem Personal leicht selbst erzeugt und erweitert werden.

Wird das Dokument später geöffnet (Vorlage aufrufen – jetzt wird ja ein neues Dokument auf der Basis der Vorlage erzeugt), startet ein Makro, das zunächst alle Bereiche ausblendet, dann den Def-Bereich sucht, einliest und auswertet. Die dort definierten Hilfetexte zu den Bereichen werden nun in einem Dialog angezeigt, über den der/die Benutzer/in entscheidet, welche Teile er/sie im Dokument benötigt.

Das Dokument ist fertig – nur die gewünschten Textbereiche (inklusive der ihnen eigenen Formatierung) sind sichtbar, das Dokument kann gedruckt werden.

Bereiche können somit quasi als Textbausteine genutzt werden, sind aber aufgrund ihrer vielfältigen Möglichkeiten (Formatierung, Objekte) deutlich flexibler.



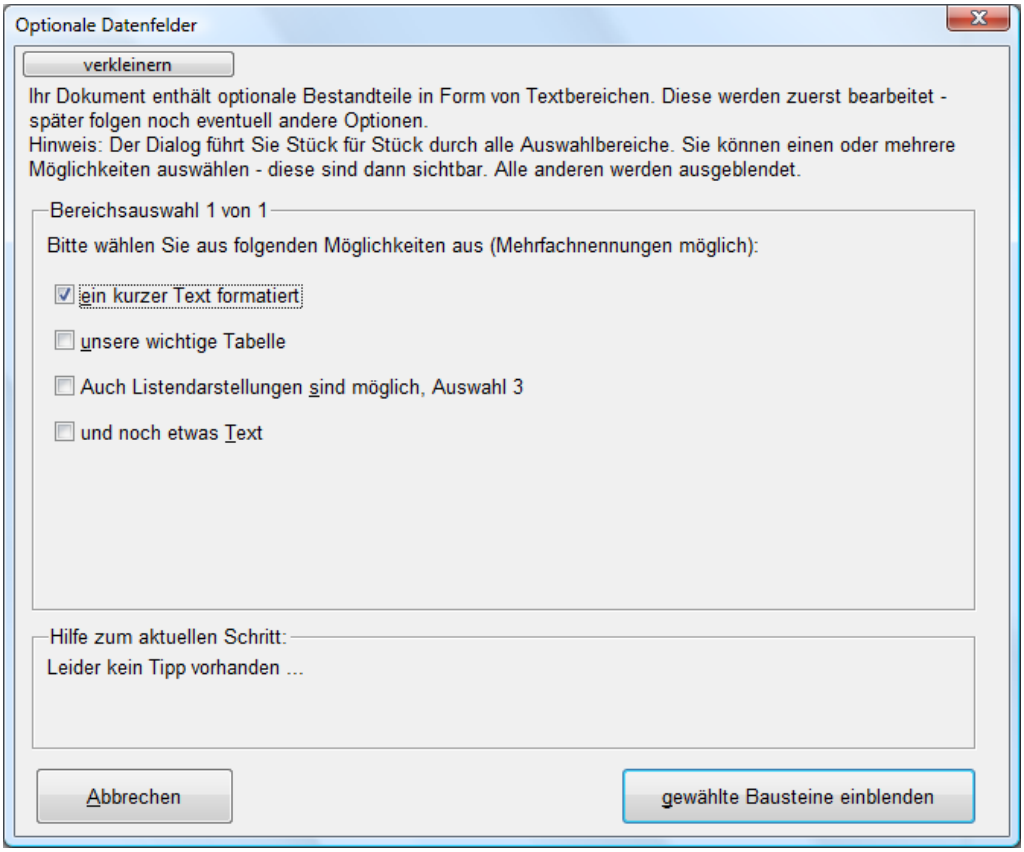
Der passende Code zur Bereichsauswahl (ohne Details des Dialogaufbaus):

```
'/** CheckDocBereiche
'*************************************************************************.
'* @kurztext  Prüft, ob auswählbare Bereiche im Dokument vorhanden sind
'* Diese Funktion prüft, ob auswählbare Bereiche im Dokument vorhanden sind,
'* zeigt einen entsprechenden Dialog an, und löscht nicht benötigte.
'*
'*************************************************************************
'*/
function CheckDocBereiche
  dim oBereich as object   'ein einzelner Bereich
  dim a()                  'Arrayplatzhalter

  dim i%, az%, iB%, t%, sTxt as string

  CheckDocBereiche = true  'Vorgabe

  REM prüfen, ob das Dokument Bereiche enthält
  if oDoc.getTextSections().getCount() = 0 then exit function    'keine Bereiche -> Ende

  REM Vorgaben
  az = 0 : iB = 0

  REM prüfen, ob bei den Bereiche "def" Bereiche vorhanden sind
```

```
  REM gleichzeitig einlesen der Bereichsnamen in einen Array mit zwei Elementen
  oBereiche = oDoc.getTextSections()
  for i = 0 to oBereiche.getCount() - 1
    oBereich = oBereiche.getByIndex(i)
    if inStr(oBereich.name, "_") > 0 then  'nur Bereiche mit Unterstrich werden ausgewertet
      a = split(oBereich.name, "_", 2)     'maximal 2 Elemente!
      redim preserve aBerListe(iB)
      aBerListe(iB) = a
      REM Prüfen, ob Def-Bereich, wenn ja, aufnehmen in zusätzliche Liste
      if lcase(trim(a(1)))= "def" then   'Def-Bereich
        redim preserve aPosDef(az)
        aPosDef(az) = iB
        az = az + 1
      end if
      iB = iB + 1
      REM Bereich ausblenden - Vorgabe
      oBereich.isVisible = false
    end if
  next i

  if az = 0 then exit function  ' keine Def-Bereiche vorhanden -> Ende

  REM jetzt Def-Bereiche auswerten und Auswahldialog starten
  REM  Dialog initialisieren
  oDlg = createUnodialog(DialogLibraries.SFr_LK.dlg_BedText)

  oDlg.model.step = 9     'neunte Stufe
  sTxt = "Ihr Dokument enthält optionale Bestandteile in Form von Textbereichen. " & _
         "Diese werden zuerst bearbeitet - später folgen noch eventuell andere Optionen." &
chr(13) &_
         "Hinweis: Der Dialog führt Sie Stück für Stück durch alle Auswahlbereiche. Sie können
" & _
         "einen oder mehrere Möglichkeiten auswählen - diese sind dann sichtbar. Alle anderen
werden " & _
         "ausgeblendet."
  oDlg.getControl("lbl_main").text = sTxt
  for i = 1 to az               ' Für jede Fundstelle
      bUndoFlag = true     'kein Undo
      BedingterText.initDlg()
      DialogBereichAktualisieren(i,az, join(aBerListe(aPosDef(i-1)), "_"))
      a = aBerListe(aPosDef(i-1))
      oDlg.getControl("txt_def").text = a(0)
      oDlg.getControl("cmd_weiter").label = "gewählte Bausteine einblenden"

      bDlgEnd = false   'Vorgabe Flag
      oDlg.setVisible(true)
      REM Schleife für Dialog Sichtbarkeit
      do while NOT bDlgEnd
        wait(iDlgPause)
      loop
      oDlg.setVisible(false)  'Dialog ausblenden

      REM Bei Dialog-Abbruch - Ende
         if bDlgAbbruch then
        CheckDocBereiche = false
        exit function
      end if

         BereicheEintragen
```

```
  next
end function

'/** BereicheEintragen
'*************************************************************************.
'* @kurztext  blendet die gewählten Bereiche ein
'* Diese Funktion blendet die ausgewählten Bereiche im Dokument ein
'*
'*************************************************************************
'*/
sub BereicheEintragen
  dim i%

  For i = 1 to 6    'maximal 6 Positionen
    if (oDlg.getControl("chk_B" & i).state = 1) AND _
       (oDlg.getControl("chk_B" & i).isVisible()) then    'eintragen

       if oBereiche.hasByName(oDlg.getControl("txt_def").text & _
            "_" & oDlg.getControl("txt_B" & i).text) then    'prüfen ob es einen solchen
Bereich gibt

          oBereiche.getByName(oDlg.getControl("txt_def").text & _
               "_" & oDlg.getControl("txt_B" & i).text).IsVisible = true
       else
          Msgbox ("Der Bereich """ & oDlg.getControl("txt_def").text & _
            "_" & oDlg.getControl("txt_B" & i).text & """ konnte" & chr(13) & _
            "nicht identifiziert werden! Bitte prüfen Sie Groß- und Kleinschreibung!" & chr(13)
& _
            "Der Bereich wird nicht eingeblendet!", 16, "Fehler bei Bereichsnamen!")
       end if
    end if
  next

end sub
```

Das Ergebnis für den/die Benutzer/in:



Die Auswahl erfolgt nun bequem über die Checkboxen – die angekreuzten Bereiche werden anschließend eingeblendet.

7.8 Dokumente einfügen

Statt mit Textbereichen können auch mit eigenen Dokumenten größere Textbausteine mit umfangreicher Formatierung und Objekten als Basis zur Gestaltung neuer Dokumente dienen. Über die UI nutzt man hier den Menübefehl **Einfügen/Datei...**, per Makro geht dies auch. Allerdings muss man dabei ein paar „Tücken“ beachten.

Grundlagen des Einfügens eines Dokumentes:

Ein Dokument mit bekannter URL wird an der Cursor-Position eingefügt mit der Methode

```
oCursor.insertDocumentFromURL(sUrlVorl, arg())
```

Dabei wird der Inhalt des Dokumentes beginnend an dieser Position in das Dokument integriert – Text, Formate und Objekte.

1. Falle: Mit der Integration des Dokumentes werden die Standardvorlagen auf die gemeinsame Basis zurückgesetzt – und das bedeutet, die Zeichenvorlage (Standardabsatz) erhält die vordefinierte Schriftart (im Fall von Windows wäre dies TimesNewRoman).

Dadurch wird also sowohl das Ursprungsdokument als auch das zu integrierende Dokument auf diese Schriftart zurückgesetzt – ein selten gewünschter Effekt.

Abhilfe schafft hierbei der folgende Weg:

Vor der Integration des Dokumentes liest man die Schriftart des aktiven Dokumentes aus (Standardvorlage – Absatz), speichert den Namen zwischen, integriert das neue Dokument und weist nun die vorherigen Standardeinstellungen wieder dem Dokument zu.

```
REM Standard-Absatzvorlage Schriftart auslesen
 sStyle =
oDoc.getStyleFamilies().getByName("ParagraphStyles").getByName("Standard").CharFontName
 ..
 oCursor.insertDocumentFromURL(sUrlVorl, arg())
 ..
REM Standardschrift zurückschreiben
 oDoc.getStyleFamilies().getByName("ParagraphStyles").getByName("Standard").CharFontName =
sStyle
```

Normalerweise sind alle anderen Absatzvorlagen abhängig von der Standardvorlage – insofern stimmen nun alle Schriften wieder.

2. Falle: verankerte Objekte:

Neben der Standardschriftart muss an alle verankerten Objekte gedacht werden. Beinhaltet das zu integrierende Dokument zum Beispiel ein Bild (oder Logo), das seine Verankerung an Seite 1 enthält, und wird ein solches Dokument in ein anderes auf Seite 3 integriert, so behält das Objekt leider seine Eigenschaft „Verankerung auf Seite 1“. Das Objekt wird also nicht – wie wahrscheinlich gewünscht – auf Seite 3 erscheinen, sondern auf Seite 1.

Um das zu verhindern, sind mehrere Schritte nötig. Zunächst muss das einzufügende Dokument geöffnet und auf seitenverankerte Drawpage-Objekte hin untersucht werden. Gibt es welche (und hier zählen jetzt nicht nur Grafiken, sondern eben auch Textrahmen etc. dazu), so müssen diese eindeutig identifizierbar markiert werden. Dies geschieht beispielsweise über den Namen, den man eindeutig vergibt. Wird der bisherige Name weiterhin benötigt, so muss dieser zwischengespeichert werden – eindeutig zugeordnet zum neuen Namen. Das so geänderte Dokument speichert man nun als „Zwischenversion“ temporär ab.

Nun ermittelt man die Einfügeposition (und insbesondere die Seitennummer – geht nur über den View-Cursor!) und fügt das gespeicherte Zwischenversionsdokument ein. Alle ursprünglich dort an Seite 1 verankerten Objekte behalten diese Eigenschaft – sind jetzt also auf Seite 1 zu sehen. Über den vorher eindeutig zugewiesenen Namen sind sie nun aber einfach zu identifizieren, die Seiteneigenschaft kann entsprechend der neuen Position leicht geändert werden und der ursprüngliche Name wird wiederhergestellt.

Der folgende Code zeigt ein solches Beispiel, in dem ein Briefkopf mit an der Seite verankerten Elementen erneut in ein Dokument eingefügt wird:

```
'/** KopfbogenErneutEinfuegen
'*************************************************************************.
'* @kurztext  Fügt erneut einen Kopfbogen in das Dokument ein
'* Diese Funktion fügt einen erneuten Kopfbogen in das Dokument ein. Dazu wird
'* die Vorlage zunächt in einem Temp-Verzeichnis geöffnet und unter einem bestimmten
'* Platzhalternamen gespeichert. Dann werden alle Objekte, die an der Seite
'* verankert sind, ausgelesen und umbenannt. Anschließend wird diese Datei
'* in das Dokument eingefügt und die aktuelle Seitenzahl ermittelt.
'* jetzt werden die speziell benannte Objekte in dem Hauptdokument wieder gesucht
'* und auf die aktuelle Seite verankert.
'*
'* @param1 sVorlURL as string   VorlageURL, die eingefügt werden soll
'* @param2 oCur as object  Cursorobjekt an dem die Vorlage eingefügt werden soll
'*
'* @return  Flag as Boolean   True, wenn alles ok, False bei Fehler
'*************************************************************************
'*/

function KopfbogenErneutEinfuegen(sVorlURL as string, oCur as object, oDoc as object)
REM Vorlage ist gecheckt, existiert. Öffnen möglich
  dim sTempVorlURL as string      'Pfad und Name der Temp-Vorlage
  dim oTempVorl as object         'Objekt der Vorlage
  dim sSeitenvorlage as string    'Name der Seitenvorlage
  dim oElem as object             'ein Element der Drawpage
  dim args()                      'leere Argumentenliste
  dim oVCur as variant            'Viewcursor
  dim iSeite as integer           'aktuelle Seite
  dim i%, n%

  sTempVorlURL = convertToURL(environ("temp") & "\" & sBK2TempName)
  dim arg2(0) as new com.sun.star.beans.PropertyValue
  arg2(0).name = "Hidden"
  arg2(0).value = true
  '#### Datei laden
  wait(iWarten)
  oTempVorl = starDesktop.LoadComponentFromURL(sVorlURL, "_blank", 0, arg2())
  sSeitenvorlage = oTempVorl.getText().getStart().PageDescName
  REM Jetzt prüfen, ob Drawpage-Objekte vorhanden
  n = 0
  if oTempVorl.drawpage.hasElements() then    'mindestens ein Element vorhanden
    For i = 0 to oTempVorl.drawpage.count -1
      oElem = oTempVorl.drawpage.getByIndex(i)
      if oElem.AnchorType = com.sun.star.text.TextContentAnchorType.AT_PAGE then      'Objekt
an der Seite verankert
        REM Objektname auslesen, und umbenennen
        redim preserve aBKObjName(n)
        redim preserve aBKObjekte(n)
        aBKObjName(n) = oElem.name
        aBKObjekte(n) = sObjName & "_" & n
        oElem.name = sObjName & "_" & n
        n = n + 1
      end if
    next
  end if
  REM Datei jetzt temporär speichern
  oTempVorl.storeAsURL(sTempVorlURL, args())
  oTempVorl.close(true)

  REM jetzt Vorlage in das Dokument importieren
```

```
  REM zunächst aktuelle Seite herausfinden
  oVCur = oDoc.getCurrentController().getViewCursor()
  oVCur.gotoEnd(false)
  iSeite = oVCur.page + 1    'die Seite, auf der die neue Vorlage eingefügt wird
(Seitenumbruch!)

  oCur.insertDocumentFromURL(sTempVorlURL, args())

  if  uBound(aBKObjName()) > -1  then   'Dokument enthielt Objekte an der Seite verankert
    oCur.BreakType = com.sun.star.style.BreakType.PAGE_BEFORE  'Seitenwechsel davor
    oCur.pageDescName = sSeitenvorlage                         'Seitenvorlage zuweisen

    REM jetzt Objekte wieder auf passende Seite platzieren und Rückbenennen
    if odoc.drawpage.hasElements() then    'mindestens ein Element vorhanden
     FOR i = 0 to oDoc.drawpage.count -1
       oElem = oDoc.drawpage.getByIndex(i)
       if indexInArray(oElem.name, aBKObjekte()) >= 0  then      'Objekt der neuen Vorlage
         REM Objekt umbenennen und auf aktueller Seite verankern
         n = indexInArray(oElem.name, aBKObjekte())
         oElem.name = "S" & iSeite & "_" & aBKObjName(n)
         oElem.AnchorPageNo = iSeite
       end if
     next
    end if
  end if
  REM Aufräumen
  oSFA.kill(sTempVorlURL)
End function
```

Es empfiehlt sich unbedingt, ausreichende Tests durchzuführen, wenn man andere Dokumente per Code in ein Dokument einfügen will – und auch die unterschiedlichsten Szenarien zu berücksichtigen (zum Beispiel einen Seitenumbruch mitten im Dokument!).

8 Best Practice Calc (Tabellenkalulation)

Makros entstehen häufig in Tabellenkalkulationen – zunächst, um Routineaufgaben zu vereinfachen, später auch um Abläufe komfortabler zu gestalten. Im Ergebnis werden Tabellenkalkulationen dann auch als „Datenbank“ missbraucht – das heißt, es werden Daten gespeichert (ein Datensatz = eine Zeile, auch als sehr komplexe Tabelle) und Auswertungen über diese Daten errechnet.

Im Grunde sind die Möglichkeiten der Tabellenkalkulation dazu heute auch problemlos in der Lage, soweit es sich um eine „überschaubare“ Anzahl von Datensätzen handelt, die Datei lediglich alleine genutzt wird (Single-Zugriff), Rollenkonzepte nicht oder nur in schwächerer Form nötig sind und die Datei transferierbar bleiben muss.

Bei allen Calc-Anwendungen ist aber unbedingt auf Performance zu achten. Eine Datenbank mit 20.000 Datensätzen und regelmäßigen Suchroutinen kann nicht mehr performant sein bzw. werden!

8.1 Zellinhalte, Datumsdarstellungen, Formate

Zuerst die Grundlagen: Eine Zelle kann drei verschiedene Inhalte aufnehmen (nur alternativ – nicht gleichzeitig!):

- Texte
- Werte
- Formeln

Genauer gesagt gibt es auch noch einen vierten Inhalt – nämlich gar nichts. Die Darstellung des Inhalts ist davon völlig losgelöst und auch die entsprechenden Eigenschaften versprechen keine eindeutige Identifizierung.

Befindet sich beispielsweise in der Zelle A1 der Wert „123“ – also eine Zahl – so liefern die entsprechenden Eigenschaften die folgenden Ergebnisse:

```
oZelle = oDoc.sheets(0).getCellbyPosition(0,0)

s = oZelle.string     '123 als Text-
n = oZelle.value      '123 als Zahl
f = oZelle.formula    '123 als Text
```

Alle Eigenschaften sind somit belegt – der Inhalt der Zelle ist nicht wirklich definiert.

Neben der „formula“, die eine Formel in internationaler (englischer) Schreibweise darstellt, gibt es noch die Eigenschaft FormulaLocal, die die internationale Formelbezeichnung auf eine nationale „übersetzt“ und umgekehrt.

Die einzig sichere Inhaltsbestimmungsmethode ist die Typ-Eigenschaft der Zelle. Diese bestimmt den Inhaltstyp und die Möglichkeiten (als Enum der Gruppe com.sun.star.table.CellContentType):

Enum Int	Type	Bedeutung
0	EMPTY	Der Zellinhalt ist leer
1	VALUE	Die Zelle beinhaltet einen Wert (Zahl)
2	TEXT	Die Zelle beinhaltet einen Text
3	FORMULA	Die Zelle beinhaltet eine Formel

Noch ein Beispiel: Die Eingabe lautet diesmal in Zelle A1 10.5. – dies wird intern als Datum interpretiert.

```
oZelle = oDoc.sheets(0).getCellbyPosition(0,0)

s = oZelle.string     '10.05.12 als Text-
n = oZelle.value      '41039 als Zahl
f = oZelle.formula    '41039 als Text
fl = oZelle.formulaLocal   '10.05.2012 als Text
```

Wenn aber die Inhalte in verschiedenen Eigenschaften abgespeichert werden, so ist es eben wichtig zu wissen, was genau in der Zelle steht.

Die Analyse des letzten Beispiels würde ergeben:

oZelle.getType() → 1 (ein Wert)

oZelle.Value → 41039 (der Inhalt der Zelle als Double)

oZelle.NumberFormat → 37 (Anzeige TT.MM.YY)

Nur mit diesen Informationen kann man schlüssig auf den Inhalt der Zelle sowie auf die Darstellung schließen (mal abgesehen von den (Text-)Formaten).

8.1.1 Nummernformate

Insofern spielen die Nummernformate eine wichtige Rolle – insbesondere auch dann, wenn man Werte schreiben und später korrekt darstellen möchte.

Leider sind diese Codes nicht einheitlich und ändern sich öfter mal. Es empfiehlt sich also immer, diese ab und zu zu prüfen.

Der folgende Beispiel-Code liest alle Format-Codes aus und listet diese in einer Calc-Datei:

```
Sub FormateAuflisten
        on error goto Fehler
        DIM oDoc as variant, oNumberFormats as variant, oBereich as  variant
        Dim l%, n%, i%, Flag%
        Dim oNumF as variant

        l = 150
        DIM aFormate(l)
        oDoc = ThisComponent
        oNumberFormats = oDoc.NumberFormats

        n = 0
        For i = 1 to l
            Flag = 1
               oNumF = oNumberFormats.getByKey(i)
               if  flag = 1 then
                      aFormate(n) = array(cstr(i), oNumF.FormatString)
                      n = n+1
               end if
        next
        if n < l then
          for i = l-n to l
            aFormate(i) = array("","")
          next
        end if

        oBereich=oDoc.sheets(0).getCellRangeByPosition(0,1,1,l+1)
        oBereich.setDataArray(aFormate())
    exit sub
Fehler:
        flag = 0
```

```
        resume next
End Sub
```

Das Ergebnis sieht dann wie folgt aus:

Der Format-String entspricht der Eingabe in der UI – unter Zellformat.

	A	B	C
1	**Indexnummer**	**Formatstring**	
2	1	0	
3	2	0,00	
4	3	#.##0	
5	4	#.##0,00	
6	5	#.###,00	
7	10	0%	
8	11	0,00%	
9	20	#.##0 DM;-#.##0 DM	
10	21	#.##0,00 DM;-#.##0,00 DM	
11	22	#.##0 DM;[ROT]-#.##0 DM	
12	23	#.##0,00 DM;[ROT]-#.##0,00 DM	
13	24	#.##0,00 CCC	
14	25	#.##0,-- DM;[ROT]-#.##0,-- DM	
15	30	TT.MM.JJ	
16	31	NN TT.MMM JJ	
17	32	MM.JJ	
18	33	TT.MMM	
19	34	MMMM	
20	35	QQ.JJ	

8.1.2 Textformatierung der Anzeige

Neben der Zahlendarstellung lässt sich auch der Text einer Zelle individuell formatieren – jede Zelle besitzt nämlich auch ein Textobjekt. Üblicherweise erledigt man die Formatierung mit Hilfe einer Zellvorlage (Hintergrund, Schrift, Farben etc.), es ist aber auch möglich, individuell die Anzeige unterschiedlich zu gestalten.

Im folgenden Beispiel-Code wird der Zelle A1 der String "Hallo, ein Test – hallo" zugewiesen, der Begriff „ein Test“ soll aber fett und rot erscheinen. In diesem Fall hilft keine Formatvorlage für die Zelle, denn diese Eigenschaften gelten für den kompletten Inhalt. Jetzt muss man also einen Textcursor erzeugen und die gewünschten Textteile „markieren“ – und dann entsprechende Zeichenvorlagen oder harte Eigenschaften zuweisen.

```
Sub ZelleFormatieren
  dim oZelle as variant, oTC as variant    'Zelle, Textcursor

  oZelle = thiscomponent.Sheets(0).getCellbyPosition(0,0)
  oZelle.string = "Hallo, ein Test - hallo"
  oTC = oZelle.createTextCursor()
  with oTC
    .gotoStart(false)   'Textcursor vor das erste Zeichen setzen
```

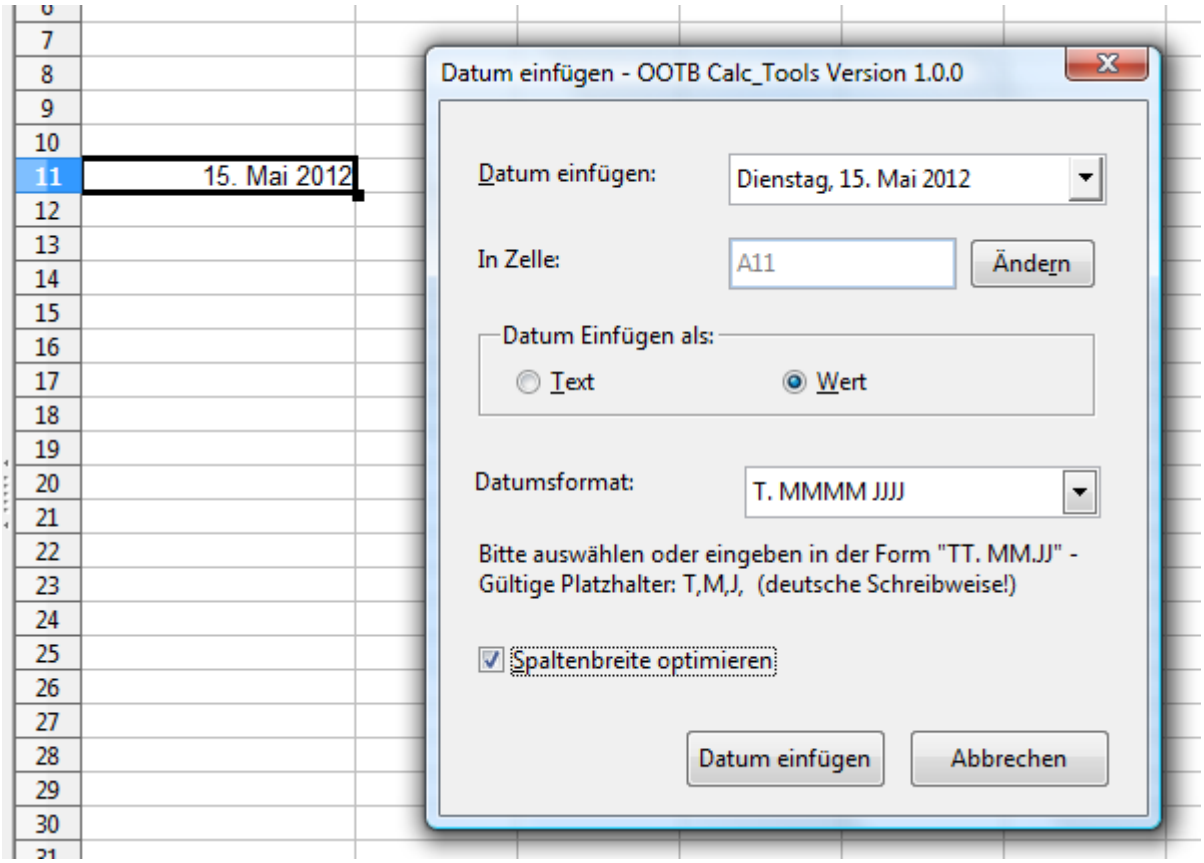
```
    .goright(7, false)   'Textcursor 7 Zeichen nach rechts
    .goright(8, true)    'markieren - die nächsten 8 Zeichen
    .CharWeight = 150    'Markierung Fett
    .CharColor = RGB(255,0,0)   'Markierung rot
  end with
End Sub
```

Das Ergebnis:

	A	B
1	Hallo, **ein Test** - hallo	
2		

8.1.3 Datumsformate

Datumswerte in Zellen einzugeben ist so eine Sache. Intern werden Datumswerte als (Double-)Zahlen gespeichert – und entsprechend dann formatiert ausgegeben. Die automatische Zahlenerkennung ist dabei nicht wirklich hilfreich – liefert „nur“ das Standardformat und irrt oft. Für den/die Benutzer/in ist es oft leichter, Datumswerte konkret und über einen Dialog passend einzugeben. Das folgende Beispiel zeigt so einen Dialog und den passenden Code, der die unterschiedlichen Varianten in die Calc-Tabelle einträgt.



In der Combo-Box zum Datumsformat sind bereits die gängigsten Codes vorgegeben – eigene können aber entsprechend den Calc-Regeln auch eingetippt werden.

```
'/** DatumEinfuegenDialog()
'*************************************************************************.
'* @kurztext fügt das aktuelle Datum in die aktuelle Zelle ein, mit Auswahl
'* Diese Funktion fügt das aktuelle Datum in die aktuelle Zelle oder die erste
'* Zelle eines markierten Zellbereiches ein. Der/Die Benutzer/in kann dabei über einen
'* Dialog diverse Einstellungen vornehmen
'*************************************************************************
'*/
Sub DatumEinfuegenDialog
  dim oZelle as variant   'Objekt der aktuellen Zelle
  dim sZelle as string    'Zellname
  dim oTab as object      'aktuelles Tabellenblatt
  dim sFormat as String   'Formatcode
  dim oNum as variant     'Formatcode-Objekt
  dim n%

  if NOT Helper.CheckCalcDokument(thisComponent) then exit sub
  oZelle = Helper.GetFirstSelectedCell(thisComponent.getCurrentSelection())
  oTab = thisComponent.sheets.getByIndex(oZelle.CellAddress.sheet)
  sZelle = deleteStr(Mid(oZelle.AbsoluteName, instr(oZelle.AbsoluteName, ".")+1), "$")
  REM Dialog erzeugen
  dialogLibraries.loadLibrary("OOTB_CT")
  oDlg = createUnoDialog(dialogLibraries.OOTB_CT.dlg_datum1)
  with oDlg     'Voreinstellungen
    .model.title = oDlg.model.title & sVersion
    .getControl("cbo_datf").text = "TT.MM.JJ"
    .getControl("txt_zelle").text = sZelle
    .getControl("dat_d1").date = CDateToIso(now())
  end with

  if not (oDlg.execute()= 1) then exit sub
on local error goto fehler:
  sFormat = oDlg.getControl("cbo_datf").text
  REM Datum einfügen
  oZelle = oTab.getCellRangeByName(oDlg.getControl("txt_zelle").text)
  if oDlg.getControl("opt_1").state then  'als Text einfügen
    sFormat = helper.getIntFormatCodes(sformat)  ' für Text englische Formate verwenden!
    oZelle.string = format(CDateFromIso(oDlg.getControl("dat_d1").date), sFormat)
  else   'als Wert einfügen
    oZelle.Value = CDateFromIso(oDlg.getControl("dat_d1").date
    rem Formate eintragen
    dim aLocale as new com.sun.star.lang.Locale
    oNum = thiscomponent.NumberFormats
    n = oNum.queryKey(sFormat, aLocale , true)    ' Deutsche Formate verwenden!
    if n = -1 then n = oNum.addNew(sFormat, aLocale)  'Format existiert noch nicht - erstellen
    oZelle.NumberFormat = n
  end if
  REM Spaltenbreite optimieren
  if oDlg.getControl("chk_spo").state = 1 then
    oTab.columns.getByIndex(oZelle.CellAddress.Column).OptimalWidth = true
  end if
  exit sub
fehler:
  msgbox ("Ein Fehler ist aufgetreten. Möglicherweise haben Sie keinen" & chr(13) & _
          "gültigen Zellnamen eingegeben?", 16, "Fehler aufgetreten")
```

```
end sub
```

Wichtig ist hierbei, dass unerwartete Fehler abgefangen werden – zum Beispiel, falls keine Zelle selektiert ist.

Ansonsten lassen sich aus dem Code viele häufig benutzte Techniken auslesen.

8.2 Kopieren und Einfügen

Kopieren und Einfügen ist in Calc nicht ganz so trivial. Wie schon dargestellt gibt es eine strenge Trennung zwischen Inhalt der Zelle und Format/Darstellung des Inhaltes. Hinzu kommt die Möglichkeit, Formeln in Zellen einzugeben, die dann wieder auf andere Zellen zugreifen.

Natürlich besteht die einfachste Möglichkeit des Kopierens und Einfügens in den Dispatcher-Kommandos „uno:copy“ und „uno:paste“, diese sollen hier jedoch nicht weiter betrachtet werden.

8.2.1 Nur Daten kopieren und einfügen

Im ersten Schritt sollen nur die Daten übertragen werden, also weder Formeln noch Formate. Um das zu realisieren, müssen Quellbereich und Zielbereich gleich groß sein. Dann kann der „Daten-Array“ ausgelesen und wieder geschrieben werden – die Daten werden übertragen.

```
sub KopieUndInsertDaten
  dim aData()
  dim oTab as variant, oBereich as Variant, oZielbereich as Variant

  oTab = thisComponent.sheets(0)

  oBereich = oTab.getCellRangeByName("A1:A15")
  aData = oBereich.getDataArray()
  oZielbereich = oTab.getCellRangeByName("D1:D15")
  oZielbereich.setDataArray(aData)

end sub
```

In diesem Fall sind alle Formeln verloren gegangen, in den Zellen stehen nun die originalen oder berechneten Inhalte des Quellbereichs. Allerdings wird die Zelleigenschaft – also die Information, ob die Zelle einen Text oder eine Zahl beinhaltet – übernommen, auch die Datums- und Zahlenformate werden übertragen, nicht aber die Textformate des Textobjektes (also zum Beispiel Fettdruck).

Die Methode eignet sich also zum Übertragen der Zellinhalte – nicht der darunterliegenden Formeln. Das Einfügen der Daten führt zum kommentarlosen Überschreiben des bisherigen Inhaltes des Zielbereiches.

8.2.2 Auch Formeln mit kopieren

Sollen auch die Formeln mit kopiert werden, so nutzt man statt der Methode „DataArray()“ die Alternative „FormulaArray()“. In diesem Fall werden die Daten der Eigenschaft „formula“ der Zelle übertragen – und die beinhaltet entweder die Formel oder den Text (bei Textinhalt) oder den Wert (bei Zahleninhalt).

```
sub KopieUndInsertDaten
  dim aData()
  dim oTab as variant, oBereich as Variant, oZielbereich as Variant

  oTab = thisComponent.sheets(0)
  oBereich = oTab.getCellRangeByName("A1:A15")
  aData = oBereich.getFormulaArray()
  oZielbereich = oTab.getCellRangeByName("D1:D15")
  oZielbereich.setFormulaArray(aData)

end sub
```

Auch hier wird die Zelleigenschaft mit übertragen, ebenso wie die Datums- und Zahlenformate, nicht aber die Textformate des Textobjektes.

Auch wenn das Ergebnis zunächst so aussieht wie bei Kapitel 8.2.1, so zeigt sich bei der Analyse eben doch, dass auch die Formeln mit kopiert wurden.

Aber Achtung!

Die Formeln wurden nicht relativ angepasst und verweisen nach wie vor auf die im Quellbereich angegebenen Zellen! Dies kann zu schweren Fehlern führen.

Auch hier führt das Einfügen der Daten zum kommentarlosen Überschreiben des bisherigen Inhaltes des Zielbereiches. Man hat also selbst dafür zu sorgen, dass dieser Bereich entweder leer ist oder überschrieben werden kann!

8.2.3 Alles kopieren

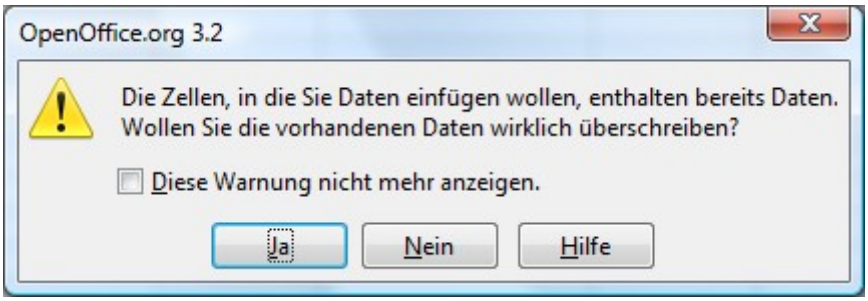
Als letzte Methode schließlich soll alles kopiert werden – also sowohl die Formeln, die Texte und die Zahlen, aber eben auch eventuell eingebundene Objekte, die Formate und alles andere. Das jetzt beschriebene Verhalten entspricht dem Vorgehen eines Benutzers / einer Benutzerin bei Copy&Paste.

Der zu kopierende Bereich muss diesmal „markiert“ werden – dies geschieht durch Selektion im aktuellen Controller. Soll in einer praktischen Anwendung der Cursor später wieder an die aktuelle Stelle gesetzt werden, so ist die Cursorposition (markierte Zelle oder Zellbereich) zunächst zwischenzuspeichern, dann die Kopieraktion auszuführen und schließlich die ursprüngliche Selektion wieder herzustellen.

```
sub KopieUndInsert
  dim oTab as variant, oBereich as Variant, oZielbereich as Variant
  dim oController as variant
```

```
    oTab = thisComponent.sheets(0)

    oBereich = oTab.getCellRangeByName("A1:A15")
    oController = ThisComponent.getCurrentController()
    oController.select(oBereich)

    oObjekt = oController.getTransferable()


    oZielbereich = oTab.getCellRangeByName("D1")
    oController.select(oZielbereich)

    oController.insertTransferable(oObjekt)

end sub
```

Auch muss bei diesem Vorgehen der Quell- und Zielbereich nicht automatisch gleich groß übergeben werden, es reicht, wenn die linke obere Ecke des Zielbereiches übergeben wird. In diesem Fall wird der Zielbereich automatisch bestimmt – und wenn dieser nicht leer ist, so erfolgt die folgende Warnung:



Zusammenfassend:

Die größte praktische Bedeutung des „Kopierens“ kommt dem DataArray() zu. Diese aus Zeilen-Arrays bestehende Liste lässt sich sehr performant in einen Zellbereich einfügen und ebenso auslesen. Lediglich die Dimensionen müssen passen! Auch muss darauf geachtet werden, dass die Zeilen-Arrays alle die gleiche Länge besitzen.

8.2.4 Markierungen

Ähnlich wie in der Textverarbeitung sind auch in Calc oft die markierte Zelle oder der markierte Bereich per Skript zu bearbeiten. Es ist also wichtig, Kenntnis von der aktuellen Selektion (Markierung) zu haben: Das sind Teile des CurrentControllers, also der View-Ansicht. Mit der aktuellen Selektion ist auch bei Calc-Dokumenten das gemeint, was mit dem View-Cursor (also dem sichtbaren Cursor) markiert wurde. Um dieses Objekt zu erhalten, unterstützen alle Calc-Dokumente die Methode getCurrentSelection(), wobei das zurückgelieferte Objekt unterschiedlich ausfallen kann.

Da es in einem Tabellendokument grundsätzlich zwei Möglichkeiten gibt, Markierungen zu setzen, müssen die Fälle unterschieden werden: Entweder ist die Zelle an sich markiert oder Teile des Inhaltes, also beispielsweise des Textes oder der Formel. Leider können auch diese beiden grundsätzlichen Fälle nochmals diverse Unterfälle beinhalten:

1. Teile des Inhaltes sind markiert: Da die Inhalte immer in einer Zelle stehen, ist automatisch auch diese selektiert. Vorteil: Es ist genau eine Zelle selektiert, die dann auch den View-Cursor beinhaltet. Auch wenn die Zelle offensichtlich leer erscheint und der Cursor nur blinkt (so dass jetzt Zeichen eingegeben werden können), ist die Zelle selektiert. In diesen Fällen liefert die Methode getCurrentSelection() immer das Zellobjekt der markierten Zelle vom Typ com.sun.star.sheet.SheetCell, mit dem dann alle weiteren Manipulationen vorgenommen werden können.
2. Mehr als eine Zelle ist markiert. Diese können entweder zusammenhängend sein (Mausklick in eine Zelle, linke Maustaste gedrückt halten, Maus ziehen→, es entsteht ein zusammenhängender Bereich markierter Zellen) oder auch nicht zusammenhängend (Zellen markieren, Strg-Taste drücken und gedrückt halten, weitere Zellen oder Zellbereiche markieren). In diesen Fällen liefert die Methode getCurrentSelection() entweder einen Zellbereich (Objekt) vom Typ com.sun.star.sheet.SheetCellRange (zusammenhängender Bereich) oder vom Typ com.sun.star.sheet.SheetCellRanges (nicht zusammenhängende Bereiche). In diesem Fall können Sie mit der getCount()-Methode die Anzahl der selektierten Bereiche feststellen und anschließend diese einzeln bearbeiten.

Da die Weiterverarbeitung der verschiedenen Objekte unterschiedlich ist, sollte man also zunächst eine Abfrageschleife einbauen. Das folgende Code-Beispiel liefert eine Liste aller markierten Zellen – und stellt das Prinzip der Abfrage dar:

```
'/** getCellListeSelektion()
'*************************************************************************
'* @kurztext  liefert eine Liste aller markierten Zellen zurück
'* Diese Funktion liefert eine Liste aller markierten Zellen zurück.
'* Übergeben wird eine beliebige Zellauswahl, typischerweise die aktuelle
'* Markierung, diese wird aufgeschlüsselt und für jede Zelle ein Array mit
'* folgenden Werten aufgebaut: Tabelle, Spalte, Zeile (jeweils Indexe)
'*
'* @param  oSel as object   Die zu prüfende Zellauswahl
'*
'* @return aListe as array  Liste der Zellen
'*************************************************************************
'*/
function getCellListeSelektion(oSel)
  dim aListe()
  dim n%

  if oSel.supportsService("com.sun.star.sheet.SheetCell") then  'einzelne Zelle
    redim preserve aListe(n)
    aListe(n) = array(oSel.cellAddress.sheet,oSel.cellAddress.Column,oSel.cellAddress.row)
    n = n+1
  elseif oSel.supportsService("com.sun.star.sheet.SheetCellRange") then  'zusammenhängender
```

```
Zellbereich
    for i = oSel.rangeAddress.startRow to oSel.rangeAddress.EndRow
      for j = oSel.rangeAddress.startColumn to oSel.rangeAddress.EndColumn
        redim preserve aListe(n)
        aListe(n) = array(oSel.rangeAddress.sheet,j ,i)
        n = n+1
      next j
    next i
  elseif oSel.supportsService("com.sun.star.sheet.SheetCellRanges") then  'nicht
zusammenhängender Zellbereich
    for k = 0 to oSel.count -1
       oBereich = oSel.getByIndex(k)  'erster Bereich
       if oBereich.supportsService("com.sun.star.sheet.SheetCell") then  'einzelne Zelle
         redim preserve aListe(n)
         aListe(n) =
array(oBereich.cellAddress.sheet,oBereich.cellAddress.Column,oBereich.cellAddress.row)
         n = n+1
       elseif oBereich.supportsService("com.sun.star.sheet.SheetCellRange") then
'zusammenhängender Zellbereich
         for i = oBereich.rangeAddress.startRow to oBereich.rangeAddress.EndRow
           for j = oBereich.rangeAddress.startColumn to oBereich.rangeAddress.EndColumn
             redim preserve aListe(n)
             aListe(n) = array(oBereich.rangeAddress.sheet,j ,i)
             n = n+1
           next j
         next i
       end if
     next k
  end if

  getCellListeSelektion = aListe()
end function
```

Nicht zusammenhängende Zellbereiche können nicht in einem Stück bearbeitet werden – sie müssen nacheinander manipuliert werden.

8.3 Suchen und Ersetzen

Auch Suchen nach speziellen Begriffen sowie das Ersetzen diverser Werte ist eine typische Aufgabe in Calc. Gesucht werden kann nach Texten, Zahlen, aber auch nach Formaten und anderem.

Die Suche wird genauso aufgebaut wie auch in anderen Modulen. Es wird ein Suchobjekt erzeugt, dieses erhält entsprechende Eigenschaften und dann wird die Suche gestartet. Das Ergebnis ist typischerweise ein Zellobjekt – eben die Zelle, auf die die Suche zutrifft – oder ein leeres Objekt (wenn die Suche nicht erfolgreich war).

Beispiel-Code:

```
 oSearch = oTab.createSearchDescriptor()
  With oSearch
    .SearchType = 0
    .SearchString = "abc"
    .SearchByRow = true
    .SearchCaseSensitive = false
```

```
end with

oObj = oTab.findFirst(oSearch)
```

In Calc ist der SearchType entscheidend:

SearchType	Bedeutung
0	gesucht wird in Formeln (Zelleigenschaft „Formula“ bzw. „FormulaLocal“) – insofern sind auch lokale Bezeichnungen einer Formel möglich!
1	gesucht werden nur Werte (Zelleigenschaft „value“)
2	gesucht wird in Notizen

Der voreingestellte SearchType ist Null (0) – es wird also die Formel-Eigenschaft durchsucht. Zur Erinnerung: Die ist immer belegt – auch wenn nur Text oder eine Zahl in der Zelle steht. Die Zelle selbst bietet keine Unterscheidung, ob es sich um einen Wert, einen Text oder eine Formel handelt! Diese Prüfung muss nachgelagert erfolgen!

Wird der SearchType allerdings auf „1“ gestellt, wird nur nach Werten gesucht – jetzt spielt der Inhalt der Formel keine Rolle mehr. Die Zelle wird auch gefunden, wenn das Ergebnis der Formel den gesuchten Zellwert ergibt.

Aber Achtung!

Auch bei der Suche nach Werten erfolgt eine „Textübereinstimmungssuche“ – es wird keine Wert-Übereinstimmung geprüft. Lediglich der Zelltyp muss einen Wert unterstützen.

So findet die Suche nach dem Wert 123 eben auch den Zellwert 1234 oder 5123! Um das zu verhindern, muss die Suche auf „ganze Wörter“ hin verändert werden:

```
.searchWords = true
```

Jetzt wird die Zelle nur gefunden, wenn der SearchString exakt übereinstimmt. Das hat nun aber den Nachteil, dass gerundete Werte quasi nie gefunden werden können.

Beispiel: In einer Zelle steht das Ergebnis der Division 1/3 – also 0.333333333....

Möchten Sie dieses Ergebnis suchen, können Sie SearchWords = true nicht mehr angeben – Sie kennen die exakte Anzahl der intern gespeicherten Stellen nicht, auch nicht die letzte gerundete Stelle. Die Suche nach „0,333“ würde aber zu keinem Ergebnis führen – die „Strings“ stimmen nicht überein.

In diesem Fall wäre die Suche nach dem Wert „0,333“ sinnvoll ohne SearchWords = true und die anschließende Überprüfung des Wertes der Zelle mit dem gewünschten Suchwert – nur so kann exakt die korrekte Zelle bestimmt werden!

Ersetzen

Ähnlich wie die Suche funktioniert auch das Ersetzen. In diesem Fall wird eben statt eines SearchDescriptor() ein ReplaceDescriptor() (createReplaceDescriptor()) erzeugt. Dieser besitzt noch als zusätzliche Eigenschaft ReplaceString() – dieser wird dann genutzt zur Ersetzung.

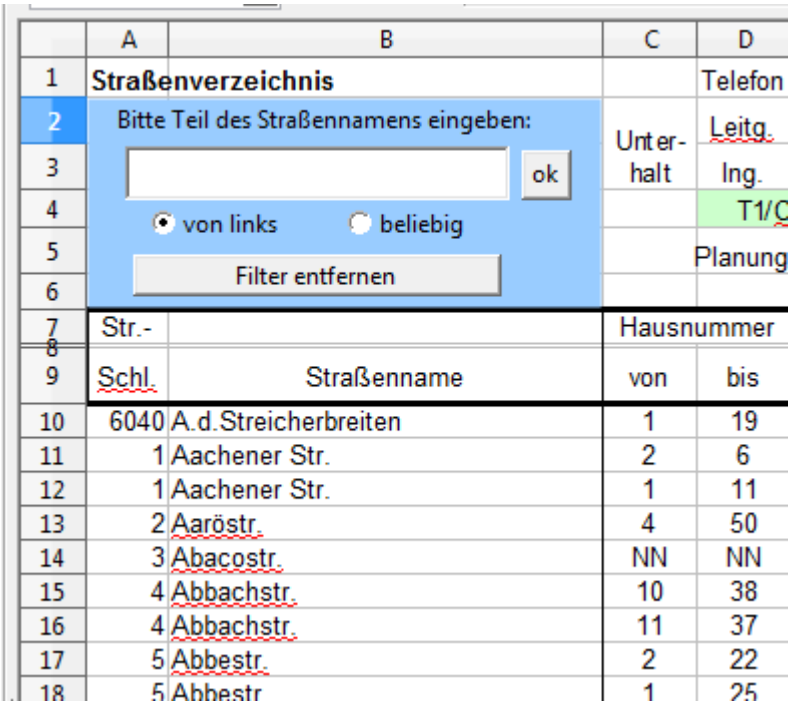
8.4 Filtern

Statt „Suchen und Ersetzen“ ist das Filtern von Informationen eine ebenfalls gern genutzte Methode, Informationen in Calc-Tabellen effizient zu suchen. Während beim „Suchen“ eine einzelne Zelle zurückgeliefert wird und der Inhalt dann verarbeitet werden kann, blendet die Filterfunktion Daten entsprechend aus, die den Suchkriterien nicht entsprechen. Dadurch ändert sich die Darstellung für den/die Benutzer/in – er/sie kann gesuchte Daten schnell finden.

Als Beispiel dient die Straßensuche:

In einer Calc-Tabelle sind alle Straßen eines Ortes (oder Bezirks) aufgelistet – teilweise mit zusätzlichen Informationen wie Hausnummern etc.

Über ein Suchfeld soll nun eine gewünschte Straße schnell gefunden werden:



Insgesamt enthält die Liste ungefähr 11.100 Einträge.

Beim Suchfeld handelt es sich um ein Formularfeld, ebenso bei den Optionsboxen. Um Filter überhaupt nutzen zu können, bedarf es eines Datenbereiches – dieser muss OOo bekannt sein.

Jeder zusammenhängende Zellbereich kann als Datenbereich genutzt werden. Dieser wird dann mit einem eindeutigen Namen belegt und steht anschließend zur Verfügung.

Im hier dargestellten Fall beginnt der Datenbereich in Zeile neun (Index 8), also mit der Überschriftzeile. Das Ende des Datenbereiches ist nicht wirklich bekannt – schließlich können jederzeit zusätzliche Daten hinzugefügt werden. Insofern muss dieser zunächst ermittelt werden.

Dazu nutzt man die Möglichkeit, die letzte benutzte Zelle (unten rechts) zu identifizieren. Dies funktioniert allerdings nur, wenn die Tabelle nicht auch noch anderen Inhalt als die Datensätze trägt und diese wahllos verteilt sind. Mit der Identifikation der letzten benutzten Zeile (geht über den View-Cursor), der bekannten Startzeile sowie der bekannten Start- und Endspalte ist nun der Datenbereich hinreichend beschrieben und kann festgelegt werden. Dies passiert beim Start des Programms:

```
const sStrTabName = "Gesamt_Verz"
const sStDaten = "Strassendaten"

dim oStrTab as variant   'Tabelle mit den Straßennamen
dim oDatBereich as variant  'Datenbereich


'/** MAK091_Init
'*************************************************************************.
'* @kurztext initialisiert den Datenbereich neu
'* Funktion wird beim Starten der Datei ausgeführt und aktualisiert den Datenbereich
'* indem es die letzte Zeile ausrechnet und den Datenbereich entsprechend anpasst.
'*************************************************************************
'*/
Sub MAK091_init
  dim iLZ as long   'letzte Zeile als Index
  dim oCur as variant   'Cursor
  dim oTab as variant
  dim oDatArea as variant

  REM check Tab
  if NOT thisComponent.sheets.hasByName(sStrTabName) then
    msgbox ("Die Tabelle """ & sStrTabName & """ konnte nicht identifiziert" & chr(13) & _
          "werden, diese beinhaltet aber die Straßennamen. Das Makro wird abgebrochen!", 16,
"Fehler!")
    exit sub
  end if
  REM letzte Zeile extrahieren
  oTab = thisComponent.sheets.getByName(sStrTabName)
  oCur = oTab.createCursor()
  oCur.gotoEndofUsedArea(false)
  iLZ = oCur.rangeAddress.EndRow
  Rem Datenbereich überprüfen
  if not thisComponent.databaseRanges.hasByName(sStDaten) then
     REM Datenbereich anlegen
     dim oBereich as new com.sun.star.table.CellRangeAddress
     with oBereich
       .sheet = oTab.RangeAddress.sheet
       .StartColumn = 0
       .startRow = 8
       .EndColumn = 17
       .EndRow = iLZ
     end with
```

```
    thisComponent.databaseRanges.addNewByName(sStDaten, oBereich )
  else      'Bereich schon vorhanden
    oDatBereich = thisComponent.databaseRanges.getByName(sStDaten)
    oDatArea = oDatBereich.dataArea
    oDatArea.EndRow = iLZ
    oDatBereich.dataArea = oDatArea
  end if
  REM Dokument speichern und somit sichern der Einstellungen
  REM Falls es zu einem Fehler kommt, speichern ignorierern
  on error resume next
  thisComponent.store()

end sub
```

Der Datenbereich ist nun bekannt, Filter können nun darauf angesetzt werden. Dabei werden die Zeilen, die den Filterkriterien nicht entsprechen, ausgeblendet. Es muss also nur der Filter sowie die entsprechenden Kriterien gesetzt werden:

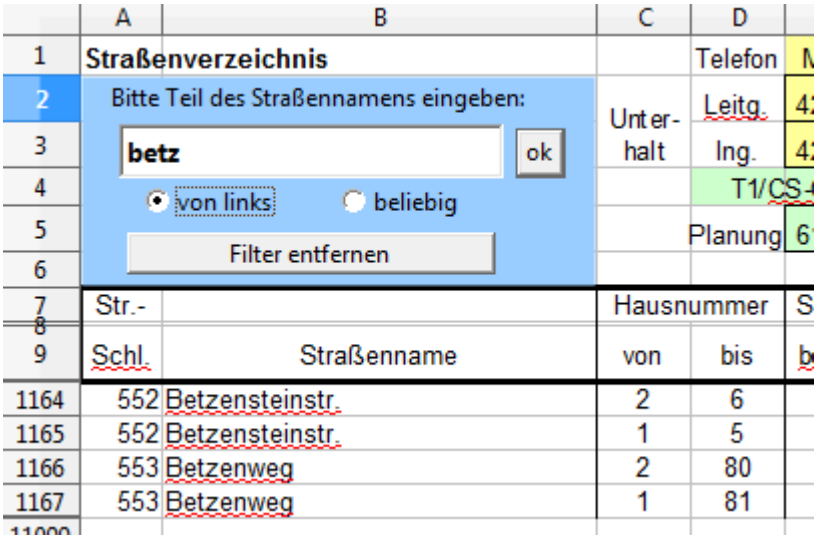
```
'/** MAK091_FilterSetzen()
'*************************************************************************.
'* @kurztext Hauptfunktion - setzt den Straßenfilter
'* Funktion setzt den Straßenfilter auf den Datenbereich. Dabei wird unterschieden, ob
'* die Begriffe von vorne oder generell im Straßennamen vorkommen sollen.
'*
'* @param stxt as string  Suchtext des Straßennamens
'*************************************************************************
'*/
Sub MAK091_FilterSetzen(sTxt as string)
  dim sPre as string

  if NOT thisComponent.sheets.hasByName(sStrTabName) then
    msgbox ("Die Tabelle """ & sStrTabName & """ konnte nicht identifiziert" & chr(13) & _
          "werden, diese beinhaltet aber die Straßennamen. Das Makro wird abgebrochen!", 16,
"Fehler!")
    exit sub
  end if

  REM Optionsboxen auslesen und entscheiden, ob von links oder im Wort gesucht wird
  sPre = ""
  if MAK091_CheckOption then sPre = ".*"

  REM Suchfeld initialisieren
  With oFFeld(0)
    .Field = 1
    .IsNumeric = false
    .Operator = com.sun.star.sheet.FilterOperator.EQUAL
    .StringValue = sPre & sTxt & ".*"
  end with

  REM Datenbankbereich prüfen und entsprechend filtern
  if not thisComponent.databaseRanges.hasByName(sStDaten) then
     msgbox ("Der Datenbereich """ & sStDaten & """ konnte nicht identifiziert" & chr(13) & _
          "werden, diese beinhaltet aber die Straßennamen. Das Makro wird abgebrochen!", 16,
"Fehler!")
    exit sub
  end if

  oDatBereich = thisComponent.databaseRanges.getByName(sStDaten)
```

```
    With oDatBereich.FilterDescriptor
      .containsHeader = true
      .IsCaseSensitive = false
      .UseRegularExpressions = true
      .FilterFields = oFFeld()
    end with

    oDatBereich.refresh()

End Sub
```

Das Makro ist mit dem Ereignis „Text modifiziert“ des Textfeldes verknüpft. Dadurch wird bei Eingabe eines Buchstaben sofort die Liste aktualisiert – und bei jedem weiteren Buchstaben verfeinert. Gesucht wird mit Hilfe von Regulären Ausdrücken – wobei noch unterschieden wird, ob der Text von vorne oder beliebig erscheinen kann (entsprechend der Auswahl der Optionsbox).

Mit Anwendung des Filters würde zunächst gar nichts passieren – erst ein „refresh()“ aktualisiert die Bildschirmanzeige. Die Anwendung eines Filters ist hochgradig performant – das Ergebnis ist sofort (innerhalb weniger 1000stel Sekunden) sichtbar – eine Schleife über 10.000 Datensätze würde mehrere Minuten laufen!

Und so könnte das Ergebnis aussehen.



8.5 Typumwandlungen

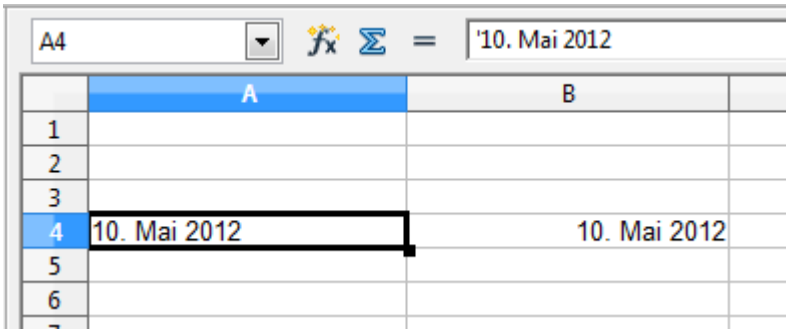
Eine in Calc oft benötigte Aufgabe ist die Umwandlung von Zellinhalten in einen anderen „Typ“. Beispiel:

Durch Einfügen einer CSV-Datei wurde in einer Spalte ein Datumswert als Text eingetragen – Sie wollen aber mit diesem Wert Berechnungen durchführen – hierzu bedarf es einer Umwandlung des Strings in einen internen Datumswert (ein Doublewert).

Das folgende Beispiel erledigt diesen Vorgang – wobei es lediglich auf eine Zelle zugreift und der Format-Code fest integriert wurde. Das Beispiel dient als Prinzip-Skizze:

```
sub Typumwandlung
  Dim oZelle as variant, oZelle1 as variant
  dim oNum as variant, n as integer
  dim sInhalt as string, dDat as double

  oZelle1 = thiscomponent.Sheets(0).getCellRangebyName("A4")
  sInhalt = oZelle1.string
  dDat = cDate(sInhalt)
  'msgbox sInhalt & chr(13) & dDat
  oZelle = thiscomponent.Sheets(0).getCellRangebyName("B4")
  with oZelle
    .value = dDat
    REM Format der Zelle
    dim aLocale as new com.sun.star.lang.Locale
    oNum = thiscomponent.NumberFormats
    n = oNum.queryKey("T. MMMM JJJJ", aLocale , true)    ' Deutsche Formate verwenden!
    if n = -1 then n = oNum.addNew(sFormat, aLocale)  'Format existiert noch nicht - erstellen
    .NumberFormat = n
  end with
end sub
```

Zur Demonstration wurde der umgewandelte Inhalt nicht in die Ausgangszelle zurückgeschrieben sondern in die rechts danebenliegende. In der Praxis würde man über eine Schleife alle Zellinhalte direkt überschreiben.



Oder: Die Spaltenwerte sind das Ergebnis einer Formel – und beziehen sich auf sehr unterschiedliche Zellen. Das Tabellenblatt muss nun so aufbereitet werden, dass in den Zellen zwar die Werte (also die Ergebnisse der Formeln) stehen, nicht mehr aber die Formeln selbst.

Nun ist es zwar so, dass in diesem Fall das korrekte Ergebnis bereits in der „value“-Eigenschaft steht und dort auch extrahiert werden kann, solange der Typ der Zelle jedoch „Formel“ lautet, ist es immer noch eine Formelzelle. Nun ist es leider nicht möglich, den Typ der Zelle direkt zu setzen – das erledigt OOo intern selber entsprechend dem der Zelle zugewiesenen Inhalt. Wird der Zelle also ein „Wert“ (Zahl) zugewiesen, so ändert sich der Typ auf 1 (Wert) – aber eben nicht in jedem Fall.

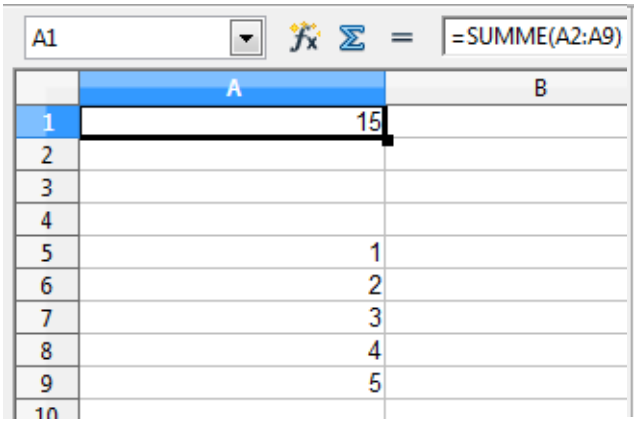
Die Zuweisung oZelle.value = oZelle.value führt leider nicht zum Erfolg, obwohl der Eigenschaft ein Wert zugewiesen wird. Eine solche Zuweisung wird einfach „ignoriert“.

Wählt man jedoch den Umweg über eine Variable, so funktioniert die Umwandlung:

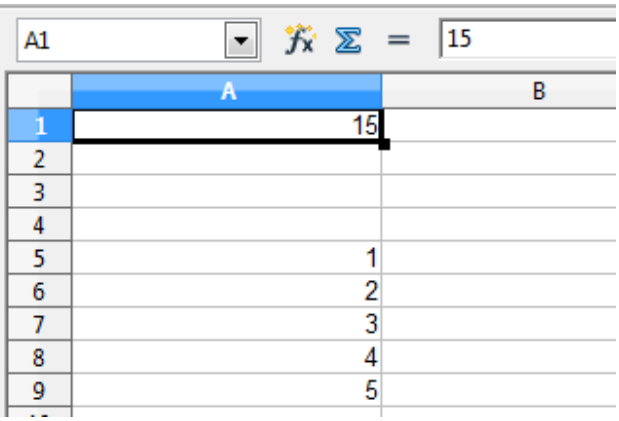
```
sub Typumwandlung2
  dim n as double, oZelle as variant
  oZelle = thiscomponent.Sheets(0).getCellRangebyName("A1")
  n = oZelle.value
  oZelle.value = n
end sub
```

Jetzt wird aus der ursprünglichen Formelzelle eine reine „Werte“-Zelle:

Vor der Umwandlung:



Nach der Umwandlung



Auf gleiche Art und Weise lassen sich auch Texte in Zahlen umwandeln oder auch umgekehrt.

Dabei kann der Typ der Zelle auch über die Eigenschaft „Formula“ bzw. „FormulaLocal“ definiert werden. In diesem Fall prüft OOo intern, welcher Inhalt übergeben wird und passt so den Typ entsprechend an. Formeln beginnen immer mit einem Gleichheitszeichen oder dem Pluszeichen (+) bzw. dem Minuszeichen(-), Werte sind in Zahlen umwandelbar und bestehen lediglich aus bestimmten ASCII-Zeichen, Datumswerte können umgewandelt werden, der Rest ist Text.

Der Vorteil der Eigenschaft „FormulaLocal“ besteht darüber hinaus darin, dass landesspezifische Besonderheiten berücksichtigt werden. So liefern beispielsweise CSV-Dateien den String „12,40“ – in Deutschland wäre das zum Beispiel die Zahl 12,40, in Amerika aber 12.400. FormulaLocal berücksichtigt die nationalen Schreibweisen und würde den String „12,40“ korrekt in 12,4 umwandeln.

Beispiel einer Funktion, die alle Texte eines markierten Bereiches in Werte umwandelt – soweit möglich:

```
'/** Calc_TexteZuWerten()
```

```
'************************************************************************
'* @kurztext konvertiert Zelltexte zu Zellwerten
'* Diese Funktion  konvertiert Zelltexte zu Zellwerten
'************************************************************************
'*/
sub Calc_TexteZuWerten()
  dim aZellen()    'Liste der Zellen
  dim Zelle()      'ein Zellarray
  dim oZelle as variant  'Zelle als Objekt
  dim s as string, i as long

  if NOT Helper.CheckCalcDokument(ThisComponent) then exit sub
  aZellen = Helper.getCellListeSelektion(thisComponent.getCurrentSelection())
  for i = 0 to UBound(aZellen)
    Zelle = aZellen(i)
    oZelle = thisComponent.getSheets().getByIndex(Zelle(0)).getCellByPosition(Zelle(1),
Zelle(2))
    if oZelle.getType() = 2 then 'nur bei Textzellen
      s = oZelle.string
      oZelle.FormulaLocal = s
    end if
  next
end sub
```

Auch hier wieder wichtig: Bevor ein Makro ausgeführt wird, sollte immer geprüft werden, ob das zugrundeliegende Dokument das korrekte ist. Die Funktion CheckCalcDokument() übernimmt diesen Test:

```
'/** CheckCalcDokument()
'************************************************************************
'* @kurztext  Prüft, ob das übergebene Dokument ein Calc-Dokument ist
'* Diese Funktion prüft, ob das übergebene Dokument ein Calc-Dokument ist.
'*
'* @param  oDoc as object   Das zu prüfende Dokument als Objekt
'*
'* @return  bFlag as boolean  true, wenn das Doc Calc ist, sonst false
'************************************************************************
'*/
function CheckCalcDokument(oDoc as object)

  if NOT oDoc.supportsService("com.sun.star.sheet.SpreadsheetDocument") then
    msgbox ("Diese Funktion ist nur in einem Tabellendokument verfügbar!", _
            64, "Fehler")
    CheckCalcDokument = false
  else
    CheckCalcDokument = true
  end if
end function
```

8.5.1 Aufteilen von Zellinhalten

Zur Typumwandlung gehört auch das Aufteilen von Zellinhalten. Oft werden Zahlenkolonnen als Strings angeliefert und in Calc eingefügt – ohne beim Import bereits eine entsprechende Aufteilung vornehmen zu können. Das passiert beispielsweise, wenn noch Hinweistexte ohne

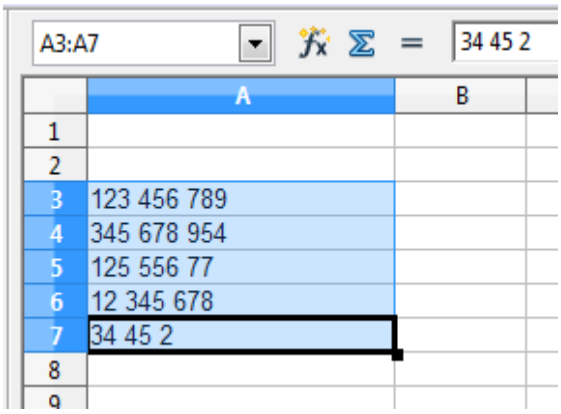
Spaltenbegrenzung vor oder nach dem eigentlichen Inhalt die automatische Erkennung blockieren – oder aber auch, weil Inhalte (zum Beispiel ein Feld mit Vor- und Zunamen) geliefert werden, die nachträglich in mehrere Felder aufgeteilt werden müssen (beispielsweise um eine Sortierung nach Zunamen durchführen zu können).

So nimmt der folgende Code eine Trennung der Zellinhalte am Leerzeichen vor und schreibt die Daten in die Zellen rechts neben der ursprünglichen Spalte – unter Einbeziehung der Quellspalte:

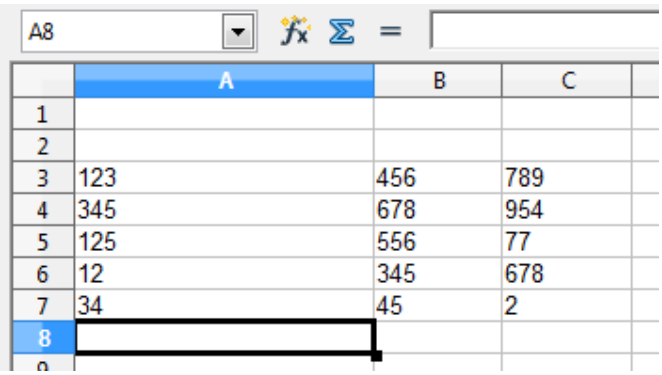
```
sub Typumwandlung3
  dim oSel as object  'aktuelle Selektion
  dim oTab as object  'aktuelle Tabelle
  dim sTr as string   'Trennzeichen
  dim aDaten()
  dim aZeile()
  dim n%, i%, i2%, i3%, j%, a()

'  if NOT Helper.CheckCalcDokument(thisComponent) then exit sub
  oSel = thisComponent.getCurrentSelection

  if oSel.supportsService("com.sun.star.sheet.SheetCell") then  'einzelne Zelle
    oTab = thisComponent.getSheets.getByIndex(oSel.CellAddress.sheet)
    oSel = oTab.getCellRangeByPosition(oSel.CellAddress.column, _
              oSel.CellAddress.row, oSel.CellAddress.column, oSel.CellAddress.row)
  elseif oSel.supportsService("com.sun.star.sheet.SheetCellRange") then  'zusammenhängender
Zellbereich
    oTab = thisComponent.getSheets.getByIndex(oSel.rangeAddress.sheet)
    if (oSel.RangeAddress.startColumn <> oSel.RangeAddress.EndColumn) then
      Msgbox ("Bitte markieren Sie nur einen Bereich mit einer Spalte!" & chr(13) &_
              "Es kann nur eine Spalte aufgeteilt werden", 16, "Fehler, zu viel Spalten
markiert")
      exit sub
    end if
  elseif oSel.supportsService("com.sun.star.sheet.SheetCellRanges") then  'nicht
zusammenhängender Zellbereich
    msgbox ("Dieses Makro unterstützt nur einen zusammenhängenden, einspaltigen Zellbereich!",
16, "Zu viel Zellen markiert")
    exit sub
  end if

  sTr = " "  'Trenner ist das Leerzeichen
  n = 2  'zwei Spalten
  REM Daten aufteilen
  j = 0
  For i = oSel.RangeAddress.startRow to oSel.RangeAddress.endRow
    if oTab.getCellByPosition(oSel.rangeAddress.StartColumn, i).type <> 0 then
       aZeile = split(oTab.getCellByPosition(oSel.rangeAddress.StartColumn, i).string, sTr,
n+1)
       if (uBound(aZeile) < n) then
         i3 = uBound(aZeile)
         redim preserve aZeile(n)
         for i2 = i3 to uBound(aZeile()
           aZeile(n) = ""
         next
       end if
    else
```

```
      redim aZeile(n)
      For i2 = 0 to uBound(aZeile)
        aZeile(i2) = ""
      next
    end if
    redim preserve aDaten(j)
    aDaten(j) = aZeile
    j = j+1
  next i

  REM Daten eintragen
  oTab.getCellRangeByPosition(oSel.rangeAddress.startColumn, oSel.rangeAddress.startRow, _
        oSel.rangeAddress.startColumn+n, oSel.rangeAddress.endRow).setDataArray(aDaten())
end sub
```

Vor der Umwandlung:



Nach der Umwandlung



Wichtig: In diesem Schritt wurde nur die Aufteilung vorgenommen – noch keine Umwandlung in Werte. Das wäre nun noch als nächster Schritt durchzuführen, wurde aber bereits erläutert.

8.6 Listen, aktive Bereiche

Calc-Tabellen können „riesig“ sein – eine Makroverarbeitung ist somit immer geprägt von Schleifen und Listen.

Allerdings können solche „Schleifen“ ziemlich aufwendig programmiert werden – oder einfacher und sinnvoller. Bedenkt man, dass die Anzahl der möglichen Zeilen in Calc von 32.000 in der Version 2 auf 64.000 in der Version 3 erhöht wurde und ab der Version 3.3 sogar mehr als 1 Million Zeilen möglich sind, so wird schnell klar, dass eine feste Grenze nicht sinnvoll ist.

Schleifen, die beispielsweise wie folgt aufgebaut sind:

```
for i = 0 to 65.000
  oZelle = oTab.getCellByPosition(iSpalte, i)
  if oZelle.string = "" then exit for
next
```

führen zwar auch zum Erfolg (solange nicht mehr als 65.000 Zeilen belegt sind) – zeugen jedoch nicht von einem guten Programmierstil.

Wenn schon mit einer For-Schleife gearbeitet wird, sollte man „echte“ und realistische Grenzen einsetzen. So wird auch intern weniger Speicher vorreserviert.

So lässt sich beispielsweise das Ende des benutzen Bereiches recht einfach über einen Blatt-Cursor erreichen:

```
oCursor = oTab.createCursor()
```

Der kann nun problemlos an das Ende des benutzen Bereiches gesetzt werden:

```
oCursor.gotoEndofUsedArea(false)
```

Dabei muss der Bereich nicht markiert werden (es schadet aber auch nichts). Die letzte benutzte Zeile ist nun im RangeAdress-Objekt gespeichert und kann problemlos zur Schleifenbildung genutzt werden:

```
iLztZeile = oCursor.RangeAdress.Endrow
for i = 0 to iLztZeile
...
```

Achtung!

Achten Sie bei Calc immer darauf, Zeilennummer-Variable ausschließlich als „long“ zu definieren. Eine Integer-Variable ist schnell zu klein!

Diese Variante ist immer dann sinnvoll, wenn zu erwarten ist, dass die gesuchten Daten den Bereich ausfüllen, nicht aber einheitlich strukturiert sind, wenn also beispielsweise durchaus Leerzeilen oder -zellen vorkommen, die aber die Schleife nicht abbrechen sollen. Bei solchen „Lückendaten“ ist die Ermittlung des Schleifenendes über diesen Weg die beste Methode.

Gibt es keine „Lücken“, so ist eine Do...While-Schleife eine echte Alternative. In diesem Fall wird das Ende überhaupt nicht ermittelt, sondern ergibt sich zum Beispiel durch die erste gefundene leere Zelle:

```
dim i as long
i = 0   'Startwert
do until oTab.getCellByPosition(iSpalte, i).getType() = 0
  oZelle = oTab.getCellByPosition(iSpalte, i)
  '...weitere Verarbeitung
  i = i +1   'nächstes Element
loop
```

Auch hier wichtig: die Zählvariable (= Zeilenindex) muss als Long definiert werden!

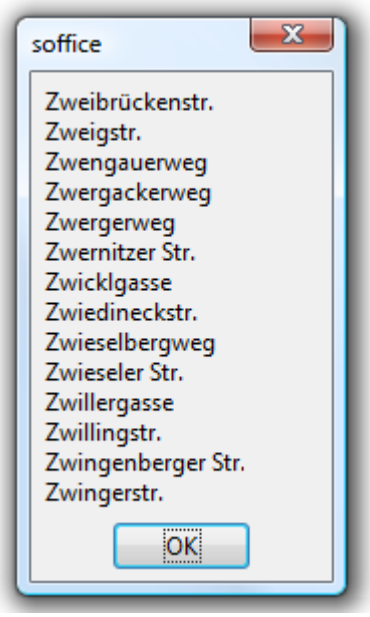
Wie im Detail eine solche „Liste“ aussehen könnte, soll am Beispiel der schon weiter oben genutzten Straßendatei dargestellt werden.

Die Datei umfasst rund 10.000 Datensätze, der Straßenname ist dabei in Spalte B untergebracht. Aus diesen Daten soll nun eine Liste (Array) aller Straßen ausgelesen werden,

die mit „zw“ beginnen. Sie erinnern sich: Die Straßenliste beinhaltete die meisten Straßennamen doppelt, da es zusätzlich eine Aufteilung nach Hausnummern gab. Die neue Liste soll natürlich die Straßennamen nur einfach enthalten.

Der Code könnte so aussehen – und liefert die gewünschte Liste:

```
sub Strassennamen
  dim aListe()  'die Liste der passenden Straßennamen
dim oTab as variant, oZelle as variant, sFilter as string
dim n as long, iZe as long, sStrasse as string

  oTab = thisComponent.sheets(0)
  sFilter = "zw"
  n = 0    'Listenanzahl
  iZe = 9   'Liste startet ab Zeile 10
  do until oTab.getCellByPosition(1, iZe).getType = 0
    oZelle =  oTab.getCellByPosition(1, iZe)
    if lcase(left(trim(oZelle.string),2))= lcase(sFilter) then
      sStrasse = trim(oZelle.string)
      if indexInArray(sStrasse, aListe) = -1 then
        redim preserve aListe(n)
        aListe(n) = sStrasse
        n = n+1
      end if
    end if
    iZe = iZe + 1
  loop

  msgbox join(aliste, chr(13))

end sub
```



Die Liste wird zunächst leer initialisiert. Die Do...Loop-Schleife durchläuft die Spalte B (Index 1) ab dem ersten Datensatz (in Zeile 10, Index 9) und wird beendet, sobald die erste leere Zelle erreicht wird.

Nun wird der Zellinhalt (String) geprüft, ob er mit den Suchkriterien übereinstimmt. Dabei müssen zwei Dinge beachtet werden: Der/Die Benutzer/in (und Ersteller/in der Liste) könnte zusätzliche Leerzeichen in der Zelle vor oder nach dem Straßennamen untergebracht haben, unbewusst oder zu Formatierungszwecken. Diese dürfen nicht berücksichtigt werden! Auch ist es nicht sicher, dass die Straßennamen immer korrekte Groß- und Kleinschreibung besitzen – auch hier müssen Eventualitäten abgefangen werden.

Die Aufgabe lautet also: Vom Inhalt der Zelle, verkürzt um die White-Spaces, normiert auf Kleinschreibung, müssen die ersten beiden Buchstaben mit dem Filter übereinstimmen.

Korrekterweise ist sogar die Länge des Filters nicht bekannt und müsste mit berücksichtigt werden:

```
    if lcase(left(trim(oZelle.string),len(sFilter)))= lcase(sFilter) then
```

Stimmt der Eintrag mit dem Filter überein, dann muss geprüft werden, ob die Straße nicht schon in der Liste vorhanden ist (Funktion IndexInArray() ist eine Funktion der globalen Bibliothek Tools und dort zu finden im Modul Strings). Ist dies nicht der Fall – die Funktion liefert die Indexnummer des Eintrages zurück, wenn dieser schon in der Liste vorhanden ist und -1 wenn nicht – dann wird der Eintrag aufgenommen. Dazu wird zunächst die Liste erweitert und dann der Eintrag hinzugefügt.

In diesem Fall werden also zwei „Listenindices“ geführt, die alte Liste – hier der Zeilenindex – sowie die neue Liste.

Zur Performance:

Die hier dargestellte Liste mit 14 Ergebnissen aus 10.000 Einträgen benötigt auf meinem Rechner (Win Vista, 2,8 Ghz, 1024 MB RAM) ca. 23 Sekunden beim ersten Start – also schon eine „fühlbare“ Zeit. Bei jedem weiteren Start sinkt die Zeit aber auf etwa 8 Sekunden (gerade die Tool-Funktionen sind jetzt im Speicher schon geladen und noch präsent!)

Würde eine Liste mit 100 Einträgen als Ergebnis erzeugt, dann werden aus den 8 Sekunden knapp 10 Sekunden – also immer noch eine gute Zeit. Eine Verbesserung auf knapp 8 Sekunden wäre nun erreichbar, würde man den Ergebnis-Array nicht jedes Mal neu dimensionieren, sondern eben nur in „größeren“ Abständen – in diesem Fall von vornherein auf 150 Elemente auslegen. Siehe hierzu auch Kapitel 4.2.3.

8.7 Calc-Funktionen

Calc dient ja überwiegend zur Berechnung von Daten – dazu gibt es eine große Anzahl vordefinierter Funktionen. Diese lassen sich alle in Calc direkt nutzen (als Formel in einer Zelle) und natürlich auch per Makro eintragen.

Alle Formeln in Zellen sind reine „Strings“ – also Texte. Sie können entsprechend aufgearbeitet und der Zelle zugewiesen werden. Damit OOo den Inhalt korrekt interpretiert, muss die Eigenschaft „formula“ oder „formulaLocal“ bzw. die Methode setFormula() bzw. setFormulaLocal() genutzt werden.

Wird eine Formel der String-Eigenschaft zugewiesen, so wird sie nicht als Formel erkannt!

Beispiele:

```
sub Formel
  dim sFormel as string, oZelle as variant

  sFormel = "=summe(A2:A10)"

  oZelle = thisComponent.sheets(0).getCellRangeByName("A1")
  oZelle.formulaLocal = sFormel
end sub
```

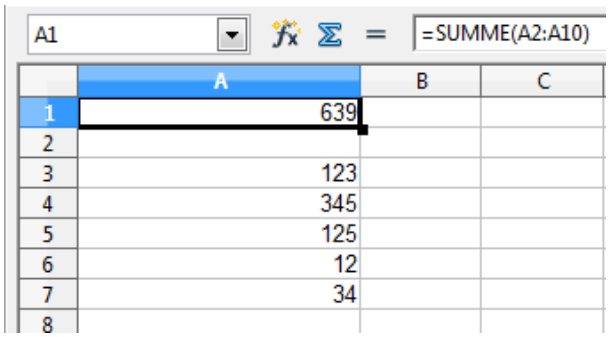
Die folgende Abbildung zeigt die entsprechenden Zuweisungen und die Ergebnisse: Wird die Formula übergeben, müssen die Bezeichner der Funktionen in Englisch definiert sein (hier also =sum()) – dann würde es funktionieren. FormulaLocal erkennt hingegen die lokalisierte Version und „Übersetzt“. Das funktioniert natürlich nur, wenn alle Einstellungen (sowohl von OOo als auch des Betriebssystems) korrekt auf die verwendete Sprache eingestellt sind (hier also DE, bzw. de-DE).

Die Übergabe der Formel als String erfolgt natürlich entsprechend den Regeln in Calc, das bedeutet, alle Parameter werden durch das Semikolon getrennt – nicht wie in Basic mit dem Komma. Auch doppelte Hochzeichen müssen maskiert mit übergeben werden (Beispiel):

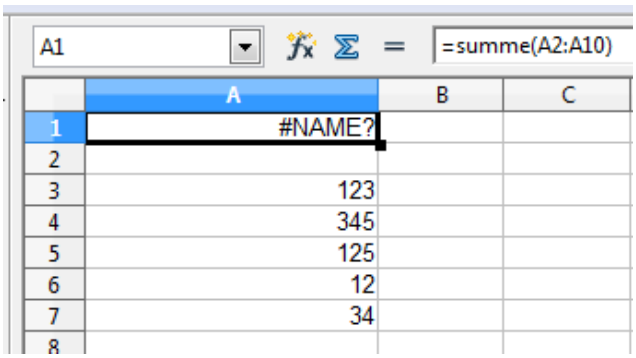
```
sFormel = "=wenn(D3 = ""Hallo""; ""Guten Tag""; """")"
```

Eine solche Formel wird korrekt eingetragen (FormelLocal).

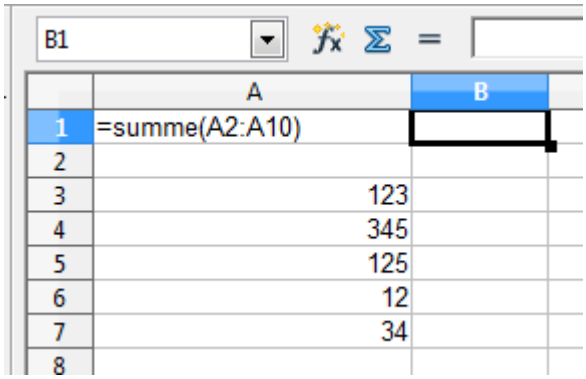
FormulaLocal:



Formula:



String



Viele Berechnungen lassen sich aber auch problemlos direkt in Basic-Code vornehmen – dadurch ist es oft gar nicht nötig, Formeln in das Dokument zu schreiben.

Und wer zur Berechnung auf die „bequemen“ Calc-Formeln nicht verzichten kann, der/die kann sie auch direkt in Basic aufrufen und nutzen. Zuständig hierfür ist der Service com.sun.star.sheet.FunctionAccess. Dessen Interface XFunctionAccess bietet genau eine Methode:

Methode	Beschreibung
callFunction(sName, aArgumente)	ruft eine Calc-Funktion auf. Erwartet werden zwei Parameter, zunächst der Funktionsname (als String – sName), dann eine Liste von Argumenten – entsprechend der Funktion.

Zurückgeliefert wird das Ergebnis der Funktion – so wie es auch in einer Zelle stehen würde.

Als Beispiel möchte ich die Funktion „Fakultät“ nutzen – die gibt es nämlich so in Basic nicht. Wesentlich ist jetzt allerdings, dass man den Originalnamen der eingebauten Funktion kennt – und nicht den lokalisierten! In deutschsprachigen OpenOffice.org-Varianten heißt die Funktion „Fakultät()“, in englischen einfach „Fact()“. Benötigt wird hier der englische Begriff.

Tipp

Kennt man den Original-Funktionsnamen nicht, so kann man diesen einfach auslesen: Geben Sie dazu zum Beispiel in Zelle A2 die lokalisierte Formel/Funktion ein. Dann schreiben Sie ein kurzes Makro und lesen die „Formula“ aus – das ist dann der englische Funktionsname. Hier die Code-Zeile:

```
msgbox thisComponent.sheets(0).getCellRangeByName("A2").formula
```

Doch zurück zum Makro. Ich nutze hier eine Funktion, die flexibler in der Handhabung ist, und möchte die Fakultät von 8 berechnen:

```
sub Start
  zahl = 8
  msgbox ("Die Fakultät von " & zahl & " lautet: " & Format(fakultaetBerechnen(zahl),"#,##0"),
0, "Fakultät")
end sub

function FakultaetBerechnen(wert as integer)
  oFunctionAccess = createUnoService("com.sun.star.sheet.FunctionAccess")
  dim arg(0)     'die Argumente, nur eins wird benötigt
  arg(0) = wert
  FakultaetBerechnen = oFunctionAccess.callFunction("FACT", arg())
end function
```

Die Funktion „Fakultät” benötigt nur einen Parameter – und das war es dann auch schon. Hier das Ergebnis:

Werden mehr Parameter für die Funktion benötigt, so verlängert man entsprechend die Argumentenliste. Im folgenden Beispiel nutze ich die Mittelwertsfunktion (AVERAGE), um den Durchschnittswert einer Zahlenreihe zu ermitteln:

```
sub Mittelwert
  dim aListe(), iMw as double
  aListe = array(4,6,8,9,30,2,5,7,9,13,22,16)
  oFunctionAccess = createUnoService("com.sun.star.sheet.FunctionAccess")
  iMw = oFunctionAccess.callFunction("AVERAGE", aListe())
  msgbox "Der Mittelwert lautet: " & Format(iMw,"#,##0"), 0, "Mittelwert")
end sub
```

Ergebnis:

Tipp

Die Calc-Funktionen funktionieren natürlich auch in allen anderen Modulen – OpenOffice.org hat ja nur einen Programmkern. Sie können also auch problemlos den FunctionAccess-Service in den Modulen Writer, Base oder Impress aufrufen – die Ergebnisse sind die gleichen.

Noch eine Besonderheit: Nicht immer sind die Funktionen so einfach aufzurufen, das funktioniert nur mit den eingebauten, quasi fest verdrahteten Calc-Funktionen. Einige interessante jedoch werden durch AddIns zur Verfügung gestellt – da geht das dann nicht mehr so einfach. Jetzt wird nämlich der interne Aufrufname benötigt – und der hat wenig mit der „Übersetzung“, dem lokalisierten Namen, zu tun. Im Zweifel lesen Sie die „Formula“ aus und schreiben den Aufruf ab.

8.8 Drucken von Teilinformationen

Eine Sache für sich ist der Druck unter Calc. Wird nur ein „Print“-Befehl aufgerufen, so wird der komplette benutzte Bereich, aufgeteilt in entsprechende Seitengrößen, ausgedruckt.

Dies ist in der Regel nicht wünschenswert.

Einen selektiven Druck in Calc kann man lediglich durch Definition von Druckbereichen erreichen.

Besitzt ein Dokument Druckbereiche, so werden nur diese gedruckt.

Jedes Tabellenblatt kann einen oder mehrere Druckbereiche beinhalten, die beim Druck ausgedruckt werden. Nochmals zur Erinnerung: Sind keine Druckbereiche definiert, wird

automatisch immer alles ausgedruckt, also alle mit Inhalt gefüllten Zellen (Ausnahme: `AutomaticPrintArea` ist auf `False` gesetzt und es gibt Druckbereiche in anderen Tabellen – dann wird nichts ausgedruckt).

Besitzt ein Tabellendokument keine Tabellen mit definierten Druckbereichen, so werden immer alle benutzten Blätter ausgedruckt, ist aber in mindestens einem Tabellenblatt mindestens ein Druckbereich definiert, werden nur noch die definierten Druckbereiche ausgedruckt, Tabellenblätter ohne solche, aber mit Inhalt, werden dann (in Abhängigkeit der Eigenschaft `AutomaticPrintArea`) ebenfalls ignoriert.

Alle Methoden für Druckbereiche werden im Interface com.sun.star.sheet.XPrintAreas definiert. Unabhängig davon funktionieren natürlich zusätzlich die Möglichkeiten, Druckoptionen im Dokumentobjekt festzulegen.

Methoden	Beschreibung
getPrintAreas()	Liefert eine Sequenz (Array) aller definierten Druckbereiche, wobei es sich hierbei um eine Liste von com.sun.star.table.CellRangeAddress-Objekten handelt.
setPrintAreas(aListe)	Definiert die Druckbereiche des Tabellenblattes. Übergeben wird eine Liste der Druckbereiche als Array von com.sun.star.table.CellRangeAddress-Objekten.
getPrintTitleColumns() getPrintTitleRows()	Als Boolean – liefert **True**, wenn Spalten/Zeilen auf jeder Seite wiederholt werden (Titelspalten/Titelzeilen)
setPrintTitleColumns(bFlag) setPrintTitleRows(bFlag)	(bFlag als Boolean) – wenn **True**, werden Spalten/Zeilen auf jeder Seite wiederholt (Titelspalten/Titelzeilen), anderenfalls **False**.
getTitleColumns() getTitleRows()	Liefert jeweils einen Bereich (Objekt als **com.sun.star.table.CellAddress**),der die Titelspalten/Titelzeilen definiert
setTitleColumns(oRange) setTitleRows(oRange)	Setzt die Titelspalten/Titelzeilen, die auf jeder Seite wiederholt werden, wenn die jeweiligen Flags auf **True** gesetzt sind. Übergeben wird ein Zellbereich (als Objekt des Typs **com.sun.star.table.CellAddress**), wobei die Methode setTitleColumns() nur die Spalten auswertet (und die Zeilen ignoriert) und setTitleRows() umgekehrt agiert.

Der Druck eines Teils einer Tabelle (und das ist der Normalfall) ist also wie folgt vorzunehmen:

- Identifikation und Beschreibung der zu druckenden Inhalte – als Ergebnis erhält man meist einen Zellbereich.
- Auslesen aller existierenden Druckbereiche und Zwischenspeichern der Daten. Dieser Schritt ist immer dann notwendig, wenn das Tabellendokument vom Benutzer / von der

Benutzerin geändert oder bearbeitet werden kann. In diesem Fall muss es ja nach dem Drucken wieder in den Ausgangszustand zurückversetzt werden.

- Einfügen des neuen Druckbereichs – dadurch gehen alle anderen verloren!
- Konfiguration der Druckeigenschaften (Kopf- und Fußzeile, evtl. Seitenvorlage)
- Druck des Dokumentes
- Rückscheiben der zwischengespeicherten Druckbereiche. Die Datei befindet sich jetzt wieder im Ausgangszustand.

Im folgenden Beispiel soll zunächst ein Druckbereich festgelegt werden (A1:N50), dieser wird dann als Hinweis ausgegeben, anschließend werden die erste Spalte und die ersten zwei Reihen als Wiederholungszeilen definiert.

```
Sub DruckBereicheFestlegen
  Dim oDoc as Variant, oSheet as Variant, oProps as Variant
  Dim oDrBereich as Variant, oDrB as Variant, s as String, sDrucker as string
  Dim aDruckbereiche(0) 'Druckbereiche-Array mit einem Element
  Dim aDruckbereicheAlt() 'Platzhalter für die „alten" Druckbereiche
  dim args2(1) as new com.sun.star.beans.PropertyValue

  oDoc = ThisComponent
  oSheet = oDoc.sheets(0)
  aDruckbereicheAlt = oSheet.getPrintAreas()   'Druckbereiche auslesen

  oDrBereich = oSheet.getCellRangeByName("A1:N50").rangeAddress   'Druckbereich
  aDruckbereiche(0) = oDrBereich
  oSheet.setPrintAreas(aDruckbereiche())
  aDruck = oSheet.getPrintAreas()
  s = "Insgesamt enthält dieses Tabellenblatt " &_
       lbound(aDruck) - ubound(aDruck) +1 &_
       " Druckbereich(e)" & CHR$(13)
  For i=lBound(aDruck) to ubound(aDruck)
    oDrB = aDruck(i)
    s = s & "Druckbereich " & i+1 & ": " &_
        ZellBereichZuName(aDruck(i))
  Next
  MsgBox s
  Dim oTitel as new com.sun.star.table.CellRangeAddress
  With oTitel
    .sheet = 0
    .startColumn = 0
    .endColumn = 0
    .startRow = 0
    .endRow = 1
  End with
  With oSheet
    .setTitleColumns(oTitel)
    .setTitleRows(oTitel)
    .setPrintTitleColumns(True)
    .setPrintTitleRows(True)
  End with
  REM jetzt Dokument drucken, zuerst Drucker auslesen
  oProps = oDoc.getPrinter()
```

```
    sDrucker = oProps(0).value

    Args2(0).name = "Name"
    Args2(0).value = "<" & sDrucker & ">"
    Args2(1).name = "Wait"
    Args2(1).value = true

    oDoc.print(args2())  'Drucken

    REM alte Druckbereiche wieder zurückschreiben
    oSheet.setPrintAreas(aDruckbereicheAlt())

    REM Dokument evt. Speichern oder Änderungsflag zurücksetzen

End Sub
```

In diesem Beispiel werden die zwei verschiedenen Möglichkeiten verwendet, ein Zellbereich-Adressobjekt zu erhalten: Einmal über die Zellen direkt und einmal als neues Objekt.

Die For-Schleife durchläuft alle definierten Druckbereiche des Tabellenblattes und listet diese auf. Da aber vorher nur ein Bereich gesetzt wurde, wäre dies eigentlich nicht nötig. Der Code kann aber auch für andere Zwecke eingesetzt werden und dann ist es schön, ein Beispiel zu haben. In der Schleife selbst wird die selbst geschriebene Funktion ZellBereichZuName() aufgerufen, die aus dem Adressobjekt eine lesbare Zellbereichsadresse der Form „A1:B4“ als String erstellt. Diese Funktion wird hier nicht beschrieben.

Es werden dann die Titelzeilen und -spalten fixiert und aktiviert. Für beide Methoden wird die gleiche Range-Adresse genutzt, jede Methode extrahiert daraus die für sie gültigen Werte.

Schließlich wird das Dokument gedruckt. Dazu wird zunächst der aktive Drucker ausgelesen und anschließend mit dem „Wait“-Befehl als Eigenschaft dem Druckbefehl übergeben.

Zum Schluss werden die „alten“ Druckbereiche wieder reaktiviert.

Hinweis

Die Funktion setPrintAreas() überschreibt alle bereits definierten Druckbereiche. Soll ein Druckbereich nur hinzugefügt werden, muss zunächst mit getPrintAreas() das aktuelle Array ausgelesen, dieses dann um das zusätzliche Element erweitert und anschließend komplett zurückgeschrieben werden.

8.8.1 Drucker auslesen

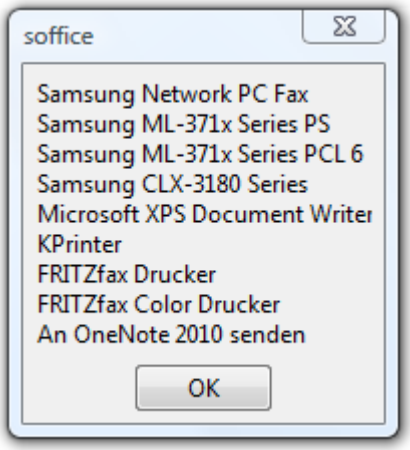
Im letzten Beispiel wurde es schon gezeigt – beim Druck kann ja der zu verwendende Drucker definiert werden. In der Regel sollte man den dem Dokument zugewiesenen nutzen. Das ist – wenn nicht anders definiert – der Standarddrucker.

Die Druckerverwaltung selbst und Ihre angezeigten Namen werden ebenfalls vom Betriebssystem bereitgestellt – in vielen früheren OOo-Versionen war es so per Basic nicht

möglich, die Liste der installierten Drucker auszulesen und daraus den passenden zu bestimmen. Ab der Version 3.x ist dies über einen „Umweg“ möglich:

```
sub Druckerliste
  Dim oServer as variant, oCore as variant
  Dim oClass as variant, oMethod as variant
  dim aNames()

  oServer = CreateUnoService("com.sun.star.awt.PrinterServer")
  oCore = CreateUnoService("com.sun.star.reflection.CoreReflection")
  oClass = oCore.forName("com.sun.star.awt.XPrinterServer")
  oMethod = oClass.getMethod("getPrinterNames")
  aNames = oMethod.invoke(oServer, Array())

  msgbox join(aNames(), chr(13))
end sub
```

Obiges liefert zum Beispiel:



Die Namen entsprechen den im Betriebssystem angemeldeten – und den zu übergebenden Namen im Print-Argument.

Mit Hilfe der Liste kann man nun recht einfach einen eigenen Druckdialog schreiben, in dem der/die Benutzer/in seinen/ihren Wunsch-Drucker auswählt – auf der Basis der installierten.

8.8.2 Listendruck

Es gibt noch weitere „Fallen“ beim Drucken von Calc-Dokumenten. Während es beispielsweise einfach ist, in Writer eine Tabelle auch über einen Seitenumbruch zu drucken und zwar so, dass die Kopfzeile auch auf der Folgeseite passend auftaucht, ist dies in Calc gar nicht so einfach zu realisieren. Zwar gibt es auch hier – wie oben gesehen – die Möglichkeit, Druckwiederholungszeilen zu definieren, doch gelten die dann immer für alle Seiten und sind wenig flexibel einsetzbar. Beispiel:

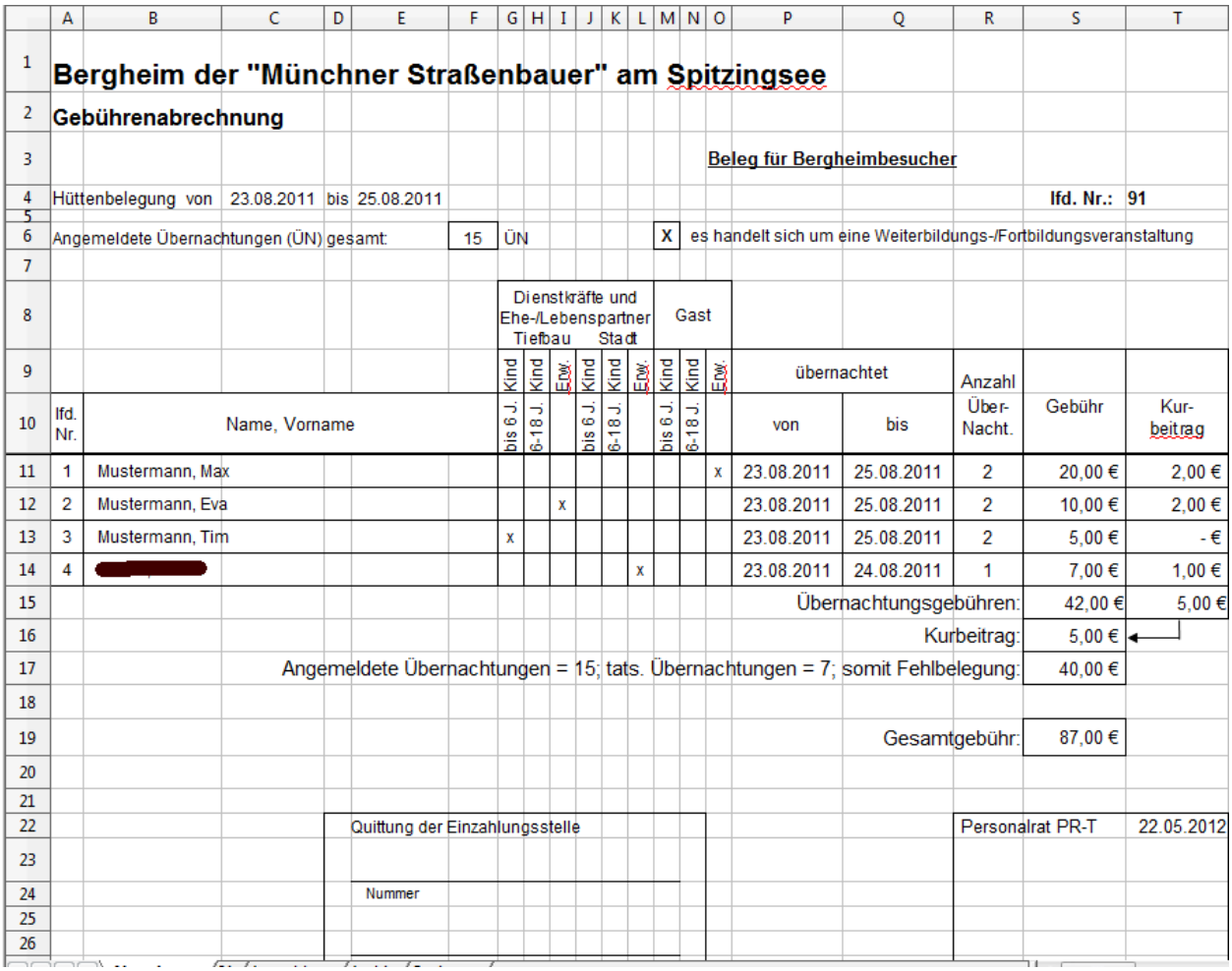
Sie haben eine Calc-Tabelle, in der es neben Informationen im Kopfbereich auch noch eine in der Länge nicht bestimmte Datentabelle sowie einen Fußbereich gibt. Dieses soll nun ausgedruckt werden und zwar so, dass – falls die Datentabelle umbricht – auch auf der Seite 2 der Kopf der Datentabelle vor den Daten steht.

Dieses Szenario ist mit den herkömmlichen Mitteln nicht realisierbar.

Das folgende Bild zeigt den Grundaufbau – innerhalb der Tabellenkalkulation:

Zeilen 8-10 sind die Kopfzeilen der Datentabelle, die aktuell vier Datensätze (Zeilen 11-14) beinhaltet, es können aber bis zu 150 Datensätze werden.

Die Kopfzeilen können als Wiederholungszeilen definiert werden (Zeilen 8-10), werden aber dann auf jede neue Seite oben gedruckt.



Bergheim der "Münchner Straßenbauer" am Spitzingsee

Gebührenabrechnung

Beleg für Bergheimbesucher

Hüttenbelegung von 23.08.2011 bis 25.08.2011 — **lfd. Nr.: 91**

Angemeldete Übernachtungen (ÜN) gesamt: 15 ÜN — X es handelt sich um eine Weiterbildungs-/Fortbildungsveranstaltung

lfd. Nr.	Name, Vorname	Dienstkräfte und Ehe-/Lebenspartner Tiefbau: Kind bis 6 J.	Tiefbau: Kind 6-18 J.	Tiefbau: Ew.	Stadt: Kind bis 6 J.	Stadt: Kind 6-18 J.	Stadt: Ew.	Gast: Kind bis 6 J.	Gast: Kind 6-18 J.	Gast: Ew.	übernachtet von	übernachtet bis	Anzahl Über-Nacht.	Gebühr	Kur-beitrag
1	Mustermann, Max									x	23.08.2011	25.08.2011	2	20,00 €	2,00 €
2	Mustermann, Eva			x							23.08.2011	25.08.2011	2	10,00 €	2,00 €
3	Mustermann, Tim	x									23.08.2011	25.08.2011	2	5,00 €	- €
4	[redacted]						x				23.08.2011	24.08.2011	1	7,00 €	1,00 €

Übernachtungsgebühren: 42,00 € 5,00 €

Kurbeitrag: 5,00 €

Angemeldete Übernachtungen = 15; tats. Übernachtungen = 7; somit Fehlbelegung: 40,00 €

Gesamtgebühr: 87,00 €

Quittung der Einzahlungsstelle

Nummer

Personalrat PR-T 22.05.2012

Das ist aber gar nicht immer gewünscht. Beispiel: Aufgrund der Datenzeilen erfolgt der automatisch berechnete Seitenumbruch nach Zeile 16 (im Beispiel). Jetzt würden auf der neuen Seite also zunächst die Wiederhohlungszeilen 8-10 gedruckt, dann würden die Zeile 17

und folgende Zeilen bis Zeile 28 (letzte benutzte Zeile) folgen – ein völlig unerwünschtes Ergebnis.

Jetzt also ist der/die Programmierer/in gefordert – der Seitenumbruch wird manuell gesetzt und zwar so, dass immer ein sinnvolles Ergebnis erzielt wird. Ok, das ist nur begrenzt möglich – hängt vom Drucker ab, von den verwendeten Schriftgrößen und so weiter. Vieles aber hat der/die Programmierer/in in der Hand – zum Beispiel die Formate für die Seite, Schriftgrößen etc. Lediglich der Drucker könnte einen Strich durch die Rechnung machen – aber hier nimmt man „übliche“ Standardwerte an.

In diesem Fall besteht die Kunst dann eben darin, passende Seitenumbrüche manuell zu setzen, also wenn zu erwarten ist, dass nicht mehr alle Informationen auf eine Seite passen (lässt sich nur durch Ausprobieren und Abschätzen realistisch ermitteln), dann trennt man „sinnvoll“, also so, dass die Datentabelle noch einige Einträge auf Seite zwei hat. In gleicher Art und Weise würde man jetzt auch eventuelle weitere Seiten abspalten.

Die folgenden Code-Zeilen sind ein Ausschnitt aus dem MAK_110 – Bergheim und zeigen das Prinzip. Die exakten Werte müssen natürlich für jedes Projekt neu ermittelt und passend eingebaut werden:

```
  ...
REM Druckbereich festlegen
  oCur = oTab.createCursor()
  oCur.gotoEndofUsedArea(true)
  iZe = oCur.RangeAddress.endRow
  REM Druckbereich erzeugen
  oTab.setPrintAreas(array(oCur.RangeAddress))
  REM Wiederholungszeilen $8:$10
  oKopfBereich.startRow = 7
  oKopfBereich.endRow = 9
  oTab.setTitleRows(oKopfBereich)
  REM evt. Seitenumbrüche einfügen
  REM Drucken bis 30 Personen (insgesamt Zeilen 56) - eine Seite, dann Seitenumbruch nach der
25. -30. Person,
  REM je nach Gesamtzahl. dann wieder Umbruch nach ... Zeilen
  if iZe >= 55 AND iZe < 68 then   '1. Zeilenumbruch, fliessend
    oTab.getRows.getByIndex(iZe - 17).isManuelPageBreak = true
  elseif iZe >= 68 then   '1. Zeilenumbruch fix
    oTab.getRows.getByIndex(53).isManuelPageBreak = true
  end if
  if iZe >= 105 AND iZe < 115 then   '2. Zeilenumbruch, fließend
    oTab.getRows.getByIndex(iZe - 17).isManuelPageBreak = true
  elseif iZe >= 115 then   '2. Zeilenumbruch fix
    oTab.getRows.getByIndex(100).isManuelPageBreak = true
  end if
  ...
```

Zwar werden in diesen Fällen nicht immer die Seiten bis zum Schluss bedruckt, das Gesamtergebnis ist aber schlüssig und passend.

8.9 Variable und Passwörter in Calc

Während bei Extensions alle einstellbaren Variablen typischerweise mit SimpleConfig ausgelagert werden sollen, ist dies nicht immer möglich bei Calc-Applikationen, wenn der Code direkt in der Datei gespeichert wird.

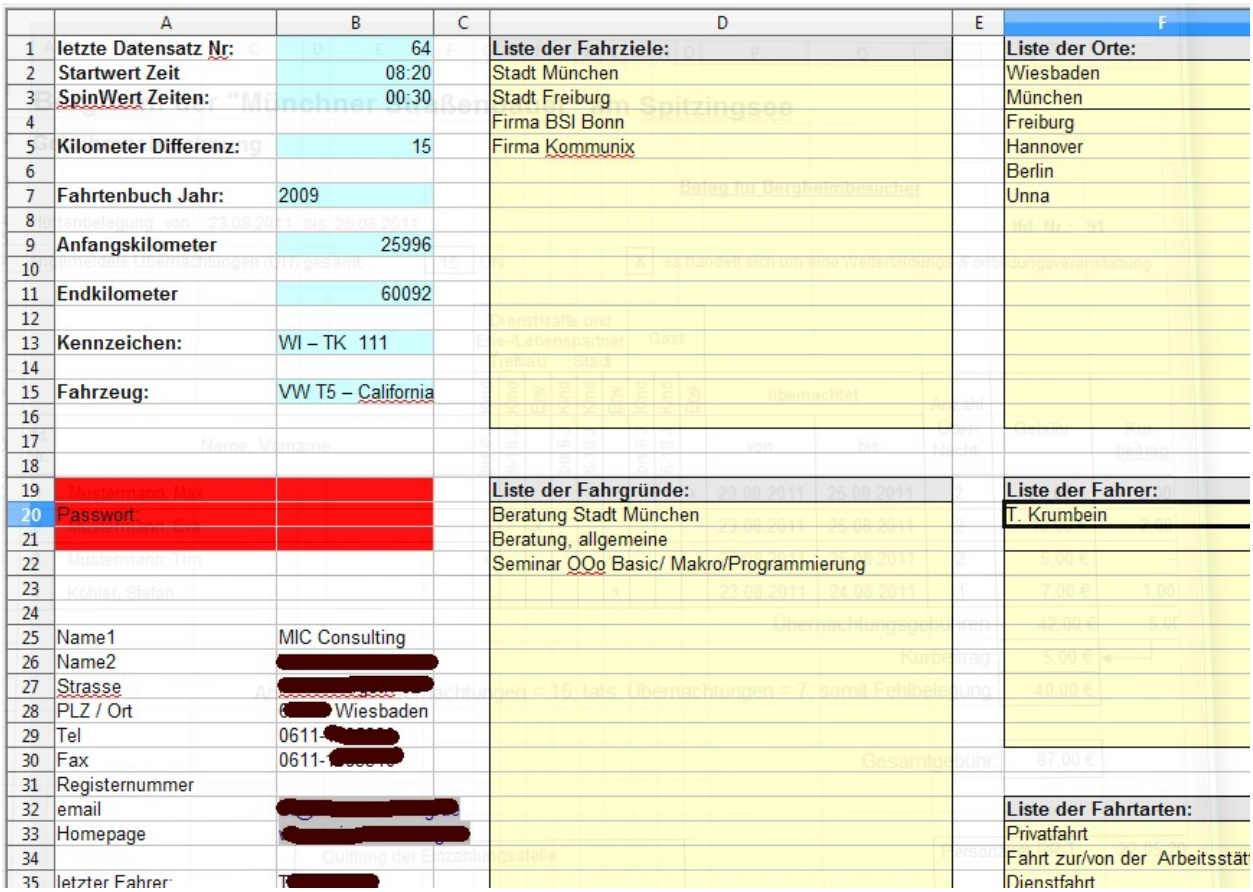
Eine solche Applikation kann nämlich weitergegeben werden – und hätte dann keinen Zugriff auf SimpleConfig selbst oder auch auf die entsprechenden Datendateien.

Aber in der Regel ist dies auch gar nicht nötig. Da Calc selbst ja einen nahezu unendlichen Datenspeicher darstellt, lassen sich all diese Informationen auch direkt in der Datei unterbringen.

Dazu nutzt man eine eigene Tabelle – nennen wir sie „Admin“. in dieser werden alle Variablen gespeichert und können dort auch geändert oder ergänzt werden. Auf diese Tabelle lässt sich per Code leicht zugreifen – und für den/die Benutzer/in lässt sie sich einfach sperren bzw. vollständig „verstecken“.

Das folgende Bild zeigt eine solche „Optionstabelle“ für eine in Calc realisierte Applikation „Fahrtenbuch“.

Der Eingabedialog für einzelne Fahrten greift nun auf die hier vorliegenden Informationen zurück: Die Listen werden als Listen in die vorhandenen Combo-Boxen geladen, sie sind „selbstlernend“ (vergleiche hierzu auch Abschnitt 6.8.3) und die Ergebnisse werden mit Beenden des Eingabedialoges wieder in die Tabelle geschrieben. Die Anzahl der Einträge ist dabei limitiert, die am wenigsten genutzten entfallen.



	A	B	C	D	E	F
1	letzte Datensatz Nr:	64		Liste der Fahrziele:		Liste der Orte:
2	Startwert Zeit	08:20		Stadt München		Wiesbaden
3	SpinWert Zeiten:	00:30		Stadt Freiburg		München
4				Firma BSI Bonn		Freiburg
5	Kilometer Differenz:	15		Firma Kommunix		Hannover
6						Berlin
7	Fahrtenbuch Jahr:	2009				Unna
8						
9	Anfangskilometer	25996				
10						
11	Endkilometer	60092				
12						
13	Kennzeichen:	WI – TK 111				
14						
15	Fahrzeug:	VW T5 – California				
16						
17						
18						
19	[redacted]			Liste der Fahrgründe:		Liste der Fahrer:
20	Passwort:			Beratung Stadt München		T. Krumbein
21	[redacted]			Beratung, allgemeine		
22				Seminar OOo Basic/ Makro/Programmierung		
23						
24						
25	Name1	MIC Consulting				
26	Name2	[redacted]				
27	Strasse	[redacted]				
28	PLZ / Ort	6[redacted] Wiesbaden				
29	Tel	0611-[redacted]				
30	Fax	0611-[redacted]				
31	Registernummer					
32	email	[redacted]				Liste der Fahrtarten:
33	Homepage	[redacted]				Privatfahrt
34						Fahrt zur/von der Arbeitsstät
35	letzter Fahrer:	T[redacted]				Dienstfahrt

Zusätzlich speichert die Tabelle diverse Statistikdaten und Stammdaten.

Eine Besonderheit stellt das Passwort dar. Normalerweise sind solche Datentabellen schreibgeschützt und versteckt , damit der/die normale Benutzer/in weder hier Daten ändern (und eventuell die Struktur verändern, was dann zum Programmfehler führen könnte) noch sonstige Manipulationen vornehmen kann.

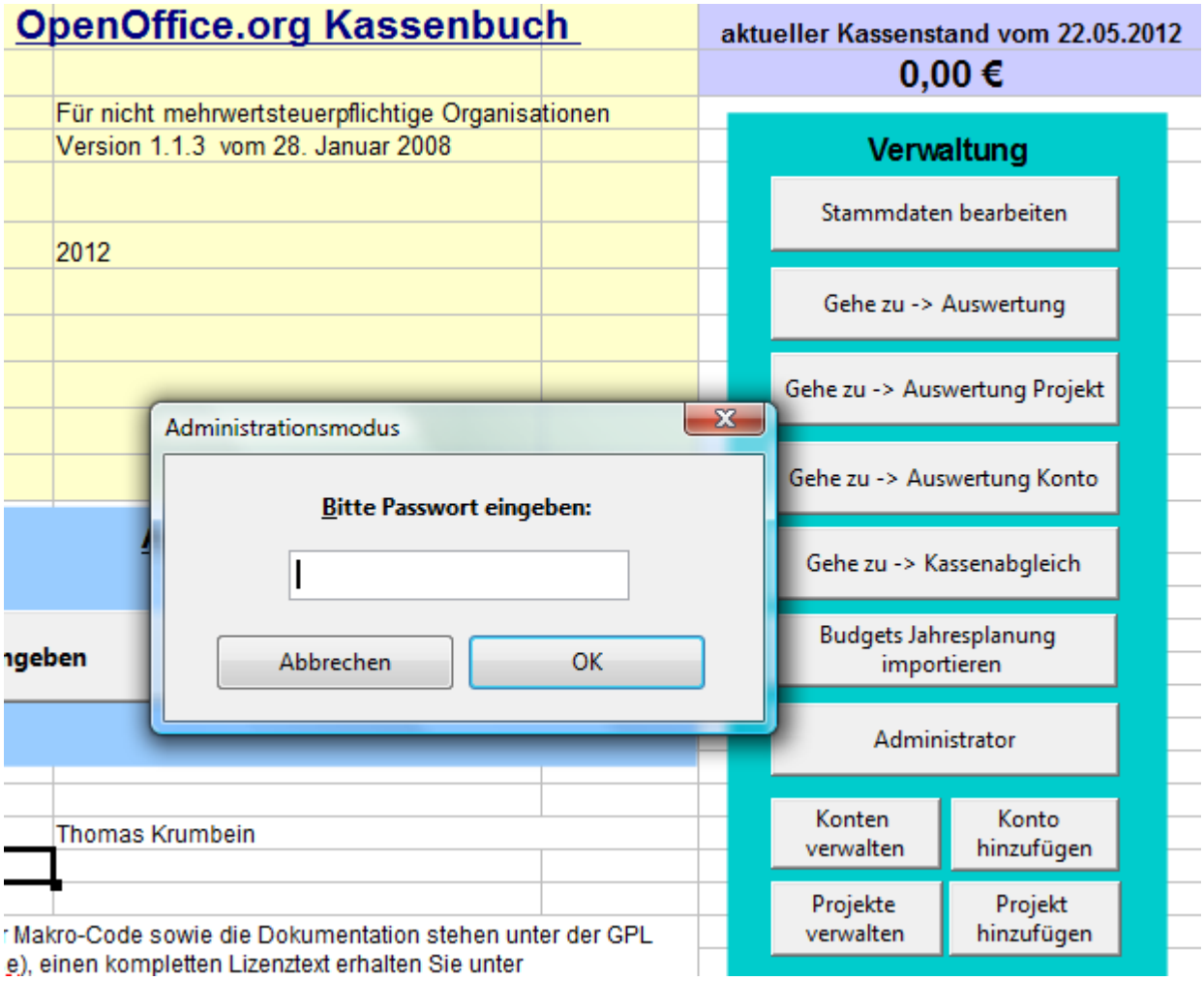
Zum Schutz der Tabelle kann ein Passwort angegeben werden. Dieses wird von OOo verschlüsselt abgelegt – und ist somit nicht „extrahierbar“. Das Passwort wird erstmalig vom Ersteller / von der Erstellerin vergeben – kann aber natürlich später vom Administrator des Programms geändert werden. Und genau hier liegt eine große Gefahr: Damit auch OOo per API auf die Seite schreibend zugreifen kann, darf die Tabelle nicht gesperrt sein. Der Ablauf ist also wie folgt:

```
oTab.unprotect( sPassWort )
REM jetzt Schreibzugriff durchführen
...
oTab.protect( sPassWort )
```

Soweit so gut. Das Passwort kann jetzt irgendwo als Konstante hinterlegt sein – und wenn der Administrator das Passwort in der UI ändert, dann funktioniert das Programm nicht mehr. Will man zusätzlich verhindern, dass der Administrator den Code ändern kann oder muss (zum

Ändern des Passwortes), braucht man eine andere Strategie. Hier wird eine Möglichkeit dargestellt:

Über einen eigenen Administrator-Button kann die Sperre der Tabellen aufgehoben sowie die versteckten Tabellen eingeblendet werden. Das geht jetzt „in einem Stück“ und nicht erst manuell für jede Tabelle einzeln. Mit Klick auf den Administrator-Button wird zunächst das Passwort erfragt (siehe hierzu auch Abschnitt 6.8.2), dieses wird verglichen mit dem in der Calc-Tabelle hinterlegten Passwort – stimmt es, werden die Optionen freigeschaltet.



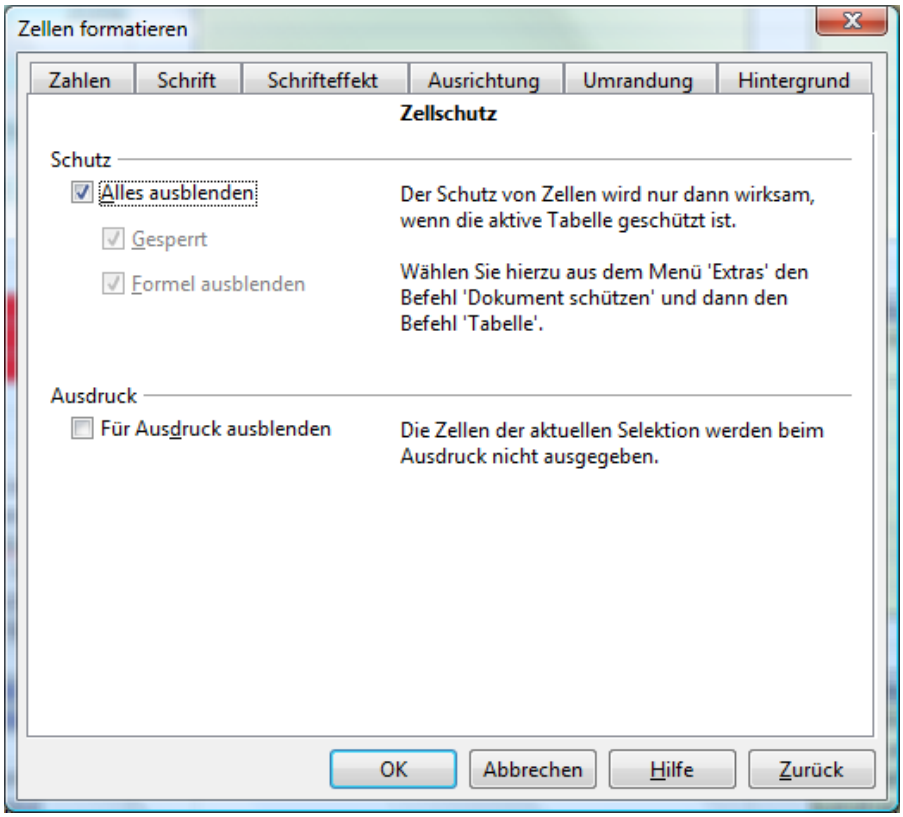
Das Passwort selbst wird in einer Zelle gespeichert in einer Schriftfarbe, die identisch zum Hintergrund (also beispielsweise rot auf rot, oder auch weiss auf weiss) ist. Dadurch ist es auch dann nicht lesbar, wenn der Administrator auf dieser Seite arbeitet und jemand über die Schulter schaut.

Wird der Administrator-Modus wieder verlassen, wird die Tabelle wieder gesperrt und die „geheimen“ Daten werden ausgeblendet.

Die Methode ist hinreichend ausgereift für die meisten Anwendungsfälle – sie ist jedoch nicht wirklich „sicher“. Mit entsprechender krimineller Energie kann ein/e Nutzer/in den XML-File auspacken, die content.xml in einem Editor laden und die Passwort-Zelle suchen (in der XML-Datei). Hat er/sie diese identifiziert, kann er/sie das Passwort im Klartext auslesen. Aber für hochsicherheitsrelevante Applikationen ist diese Methode ja auch gar nicht gedacht. Sie dient vielmehr dem unbewussten oder ungewollten Verändern der Daten/Struktur durch den/die Benutzer/in – und dafür ist sie sicher genug.

8.9.1 Tabellen schützen

Es ist sinnvoll, in Calc-Applikationen alle Tabellenblätter zu schützen, bis auf die Zellen, die der/die Nutzer/in aktiv ändern soll. Normalerweise lassen sich im Makro nicht alle Eventualitäten abprüfen und der Zugriff auf eine zunächst vom Programmierer / von der Programmiererin vorgesehene Struktur ist oft im Makro festgelegt. Ändert der/die Nutzer/in nun etwas dort (fügt zum Beispiel Zeilen oder Spalten ein), so bekommt dies die Applikation in der Regel nicht mit – Fehler sind die Folge.



So kann man im Vorfeld durchaus festlegen, welche Zellen geändert werden können – der Rest wird gesperrt – und man blendet auch gleich noch die Formeln aus.

Dann werden die Tabellen alle per Makro geschützt:

```
'/** TabellenSperren
'*************************************************************************.
```

```
'* @kurztext  sperrt/entsperrt alle Tabellen des aktuellen Dokumentes
'* Diese Funktion sperrt bzw. entsperrt alle Tabellen des aktuellen Dokumentes.
'* Dabei wird das in den Optionen gespeicherte PW genutzt
'*
'* @param  Flag as boolean  True = sperren, false = entsperren
'*************************************************************************
'*/
sub TabellenSperren(flag as Boolean)
  dim i%
  for i = 0 to thisComponent.sheets.count - 1
    if flag then
      thiscomponent.sheets.getByIndex(i).protect( sPW )
    else
      thiscomponent.sheets.getByIndex(i).unprotect( sPW )
    end if
  next
end sub
```

Wichtiger Hinweis:

Auch per Makro kann man nicht in eine gesperrte Zelle schreiben! Vor dem Schreibzugriff muss also zuerst immer der Tabellenschutz aufgehoben werden!

Schützt man aber die Tabellen und hebt diesen Schutz per Makro auf, um zu schreiben, muss man auch den Fall berücksichtigen, dass genau in diesem Moment ein Fehler OOo zum Absturz bringt.

Jetzt wäre das Dokument plötzlich ungeschützt – und Änderungen durch den/die Benutzer/in wären wieder möglich. Um das zu verhindern, sollte immer eine Initialisierungsroutine starten, wenn das Dokument geöffnet wird (Ereignis „Dokument öffnen“). Diese schützt zunächst wieder alle Tabellen – dann ist man auf der sicheren Seite.

8.9.2 Tabellen verstecken

Das gleiche sollten Sie auch machen, wenn in dem Dokument „versteckte“ Tabellen existieren. Auch hier sollte zum Start eine Initialisierungsroutine sicherstellen, dass alle gewünscht ausgeblendeten Tabellen auch wirklich ausgeblendet sind.

Hinweis und Tipp

Enthält das Calc-Dokument auch nur eine Tabelle, die einen Tabellenschutz besitzt, dann können die Formatvorlagen im kompletten Calc-Dokument nicht mehr geändert oder gelöscht werden. Die Befehle zum Aufruf (Ändern, Löschen) funktionieren alle nicht.

Möchten Sie also sicherstellen, dass alle Ihre definierten Formatvorlagen so bleiben, wie sie sind, dann fügen sie dem Dokument eine zusätzliche Tabelle hinzu, schützen diese und blenden sie anschließend jeweils mit Passwort aus. Die Formatvorlagen sind dann sicher :-).

8.10 Datenverarbeitung

Calc wird häufig als „Datenverarbeitungsmodul“ missbraucht – Daten werden wie in einer Datenbank zeilenweise eingegeben, teilweise verknüpft und über Formeln ausgewertet. Auch die Druckaufbereitung findet dann oft in einer eigenen Tabelle statt, die dann auch Ausgabemedium ist.

Bei all diesen Verfahren ist zunächst zu prüfen, ob nicht eine Datenbank-Applikation den besseren Lösungsweg darstellt und der aktuellen sowie der zukünftigen Aufgabe besser gewachsen ist.

Gerade für kleine Datenbestände kann sich aber auch die Calc-Datei als praktikable Lösung erweisen:

- Daten haben einen zeitlichen Horizont und können entsprechend gruppiert werden – ohne Verbindung zu anderen Perioden (Beispiel: Jedes Jahr eine neue Datei).
- Datensätze haben einen mengenmäßigen Horizont (Faustregel: 5.000 Datensätze mit maximal 10 Feldern (Spalten)).
- Datensätze sind normalisiert bzw. benötigen keine aufwendigen Verknüpfungen.
- Rechtemanagement und Historie ist nicht notwendig.

In all diesen Fällen könnte die Datenverarbeitung in Calc eine sinnvolle Alternative darstellen.

Die Entscheidung bei der Umsetzung ist dann nur noch wie folgt zu treffen:

1. Benutzer/in arbeitet komplett in den Tabellen, gibt dort manuell Daten ein und startet schließlich Makros für Auswertungen und/oder Druckaufbereitungen.
2. Benutzer/in arbeitet komplett in einer eigenen GUI (Dialoge oder Formulare), wird Punkt für Punkt geführt und die Calc-Tabellen werden lediglich als Datenbank und Speicher genutzt. Auf die Tabellen und die Daten greifen nur die Makros zu – der/die Benutzer/in hat keinen direkten Zugriff.

Neben diesen beiden „Extremen“ gibt es natürlich noch eine große Grauzone der Kombinationen aus beiden Lösungen.

Für die Entscheidung ist relevant:

- Wie viel „Ahnung“ hat der/die Benutzer/in von der Tabellenkalkulation Calc, der Arbeitsweise, den Formeln und den Möglichkeiten. Je höher sein/ihr Wissen und Können ist, um so eher ist Lösung 1 umsetzbar.
- Wie hoch ist die „Fehlertoleranz“ – also wird beispielsweise gefordert, dass keine Basic-Laufzeitfehler entstehen dürfen, so muss Lösung 2 programmiert werden. Kann der/die Benutzer/in in der Datenstruktur und in den Tabellen direkt Änderungen vornehmen und hat er/sie nicht selbst das Makro programmiert, so sind Basic-Fehler nicht zu vermeiden.

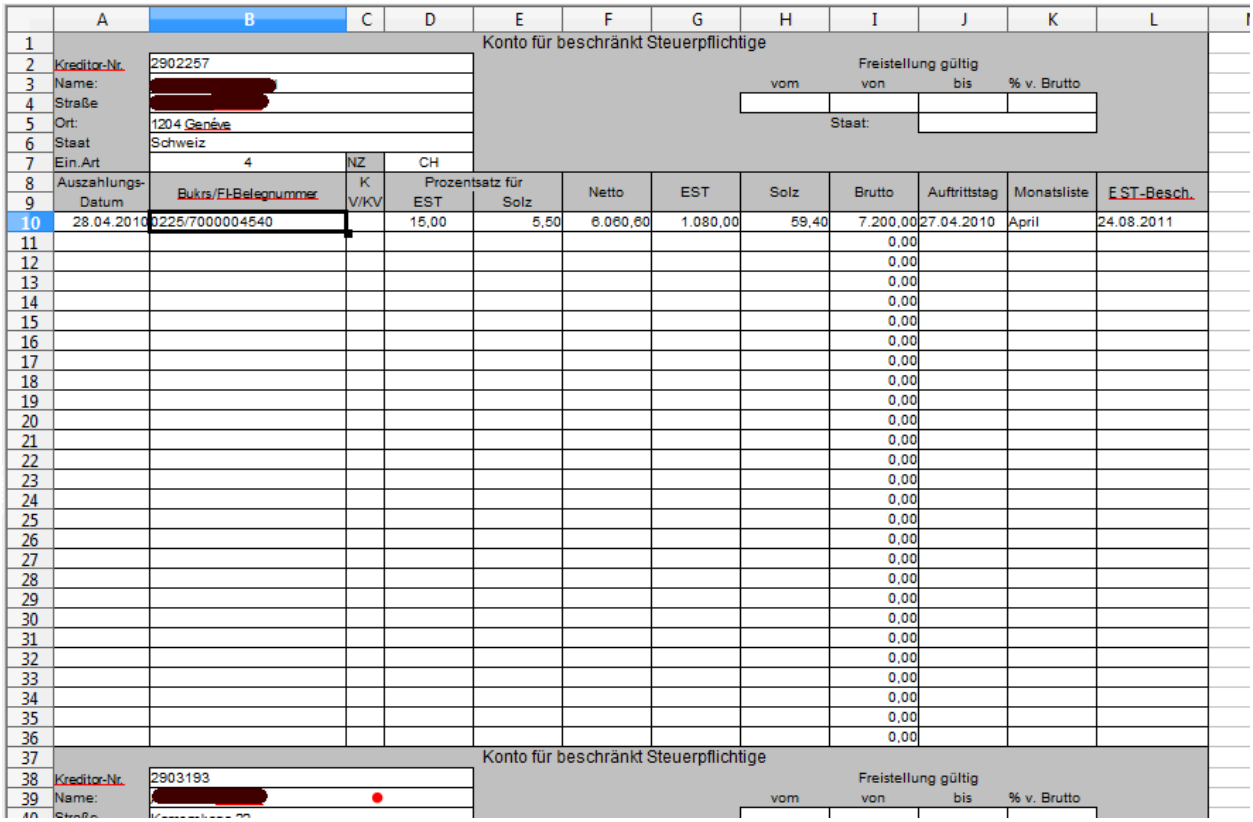
- Wie stark ist die Applikation „durchstrukturiert“, sind die Arbeitsabläufe vollständig beschrieben und entsprechend abbildbar. Wenn alle Arbeitsschritte feststehen, spricht dies für Lösung 2.
- Werden hingegen Aktionen auf vom Benutzer / von der Benutzerin markierte oder eingegrenzte Bereiche automatisiert, so ist dies in der Regel nur mit Lösung 1 erreichbar.

Für den/die Benutzer/in ist es immer besser, möglichst stark „geführt“ zu werden – arbeitet man mit einer Mischumgebung, dann müssen an sich beide Wege programmiert und realisiert werden.

Ich zeige jetzt einmal an einem praktischen Beispiel die „Mischlösung“. Die Technik ist übertragbar, die Details wahrscheinlich nicht. Es handelt sich um MAK_160.– Auslandssteuer.

- Die Datei verwaltet Ausgaben an Künstler/innen, die ihren Wohnsitz im Ausland haben. Erfasst werden alle steuerlich relevanten Daten.
- Für jede/n Künstler/in wird ein eigenes „Konto“ (Datenblatt) geführt, mit einem speziellen „Kopf“. Das Datenblatt muss druckbar sein (DIN A4 Querformat). Die Anzahl der Buchungen pro Künstler/in und Jahr ist eher gering (durchschnittlich maximal 10). Buchungen erfolgen als „Zeilenbuchungen“.
- Konten werden alle in einem Tabellenblatt verwaltet – alphabetisch geordnet (diese Form der Speicherung ist so „gewachsen“ und sollte nicht geändert werden).
- In regelmäßigen Abständen (monatlich) werden Steuermitteilungen erzeugt. Diese Form ist vorgeschrieben und mit Summen- und Einzeldaten zu befüllen. Das Formblatt wurde in einer eigenen Tabelle nachgebaut – die Felder entsprechen Tabellenzellen, die später die Inhalte aufnehmen.
- Neben der „Summenanmeldung“ gibt es auch eine „Anlage“, also eine Liste aller im Zeitraum geleisteten/empfangenen Zahlungen, ebenfalls als Formblatt mit speziellen Feldern.

Die folgende Abbildung zeigt den Start der „Kontenliste“. Es ist erkennbar, dass diese Darstellung nicht der üblicher Datenbanken entspricht:



	A	B	C	D	E	F	G	H	I	J	K	L
1	Konto für beschränkt Steuerpflichtige											
2	Kreditor-Nr.	2902257							Freistellung gültig			
3	Name:	[redacted]						vom	von	bis	% v. Brutto	
4	Straße	[redacted]										
5	Ort:	1204 Genève							Staat:			
6	Staat	Schweiz										
7	Ein.Art	4	NZ	CH								
8	Auszahlungs-		K	Prozentsatz für								
9	Datum	Bukrs/FI-Belegnummer	V/KV	EST	Solz	Netto	EST	Solz	Brutto	Auftrittstag	Monatsliste	EST-Besch.
10	28.04.2010	0225/7000004540		15,00	5,50	6.060,60	1.080,00	59,40	7.200,00	27.04.2010	April	24.08.2011
11									0,00			
12									0,00			
13									0,00			
14									0,00			
15									0,00			
16									0,00			
17									0,00			
18									0,00			
19									0,00			
20									0,00			
21									0,00			
22									0,00			
23									0,00			
24									0,00			
25									0,00			
26									0,00			
27									0,00			
28									0,00			
29									0,00			
30									0,00			
31									0,00			
32									0,00			
33									0,00			
34									0,00			
35									0,00			
36									0,00			
37	Konto für beschränkt Steuerpflichtige											
38	Kreditor-Nr.	2903193							Freistellung gültig			
39	Name:	[redacted]						vom	von	bis	% v. Brutto	

Ein Konto ist auf 36 Calc-Zeilen limitiert – spätestens dann beginnt das nächste Konto.

Der Arbeitsablauf war nun wie folgt:

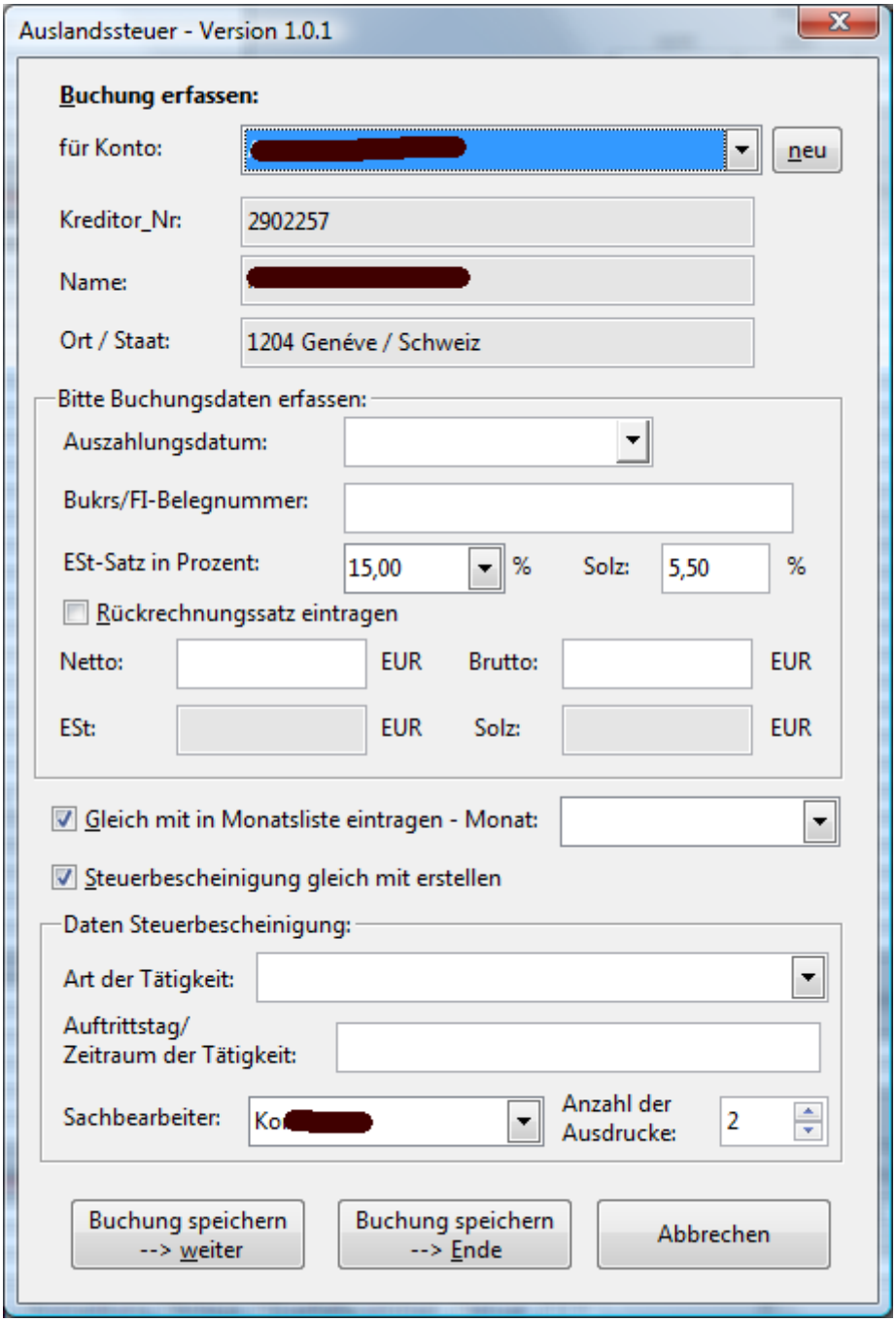
- Der/Die Sachbearbeiter/in sucht zunächst das passende Konto – entweder manuell oder über ein Makro, dem er/sie den **Namen** übergiebt (kleiner Eingabedialog). Das Makro springt dann zum gesuchten Konto und setzt dort den View-Cursor – dadurch erscheint das Konto auf dem Bildschirm.
- Der/Die Sachbearbeiter/in sucht nun manuell die nächste leere Eintragungszeile und füllt sie entsprechend aus (bis auf die letzten beiden Spalten – Monatsliste, EST-Bescheinigung). Eingabehilfen gibt es keine.
- Die Zeile wird schließlich „gebucht“. Der Vorgang kann, muss aber nicht, zeitlich nacheinander stattfinden. Dazu wird ein Makro aufgerufen – dieses wiederum nimmt die Zeile, in der der View-Cursor steht, und bucht diese. Am Ende des Buchungsvorganges werden auch die letzten beiden Spalten vom Makro ausgefüllt und eine Steuerbescheinigung erstellt (Eintrag der Werte in eine eigene Tabelle – Formblatt Ausdruck). „Buchen“ bedeutet, dass der Eintrag entsprechend des Auftrittsmonats in die Liste der entsprechenden Monatstabelle eingetragen wird.
- In regelmäßigen Abständen werden dann – ebenfalls durch ein Makro – die Daten der Monatslisten summiert und entsprechend in eine „Steueranmeldung“ ausgegeben.

- Wird ein neues Konto benötigt, sucht der/die Sachbearbeiter/in die alphabetisch passende Stelle in der Kontentabelle und fügt dort zunächst die entsprechenden Leerzeilen ein. Dann kopiert er/sie ein leeres Musterkonto dort hinein.

Diese Vorgehensweise beinhaltet diverse „Schwachstellen“. Die Makros funktionierten perfekt, wenn der/die Benutzer/in sich an der korrekten Stelle befindet und seine/ihre manuellen Eingaben immer richtig waren. Die Lern- und Erfahrungskurve des Benutzers / der Benutzerin ist also als hoch einzuschätzen.

Bei der Umsetzung soll die Schwelle nun „reduziert“ werden und auch eine/n „nicht erfahrene/n“ Nutzer/in in die Lage versetzen, ebenfalls Buchungszeilen einzugeben und Monatsabschlüsse durchzuführen.

Die einzugebenden Daten sind überschaubar: 10 Datenfelder (Spalten). Hier bietet sich ein Dialog an, der natürlich zunächst das passende Konto auswählen muss – und Optionen bieten sollte, die nachgelagerten Arbeiten gleich mit durchzuführen:

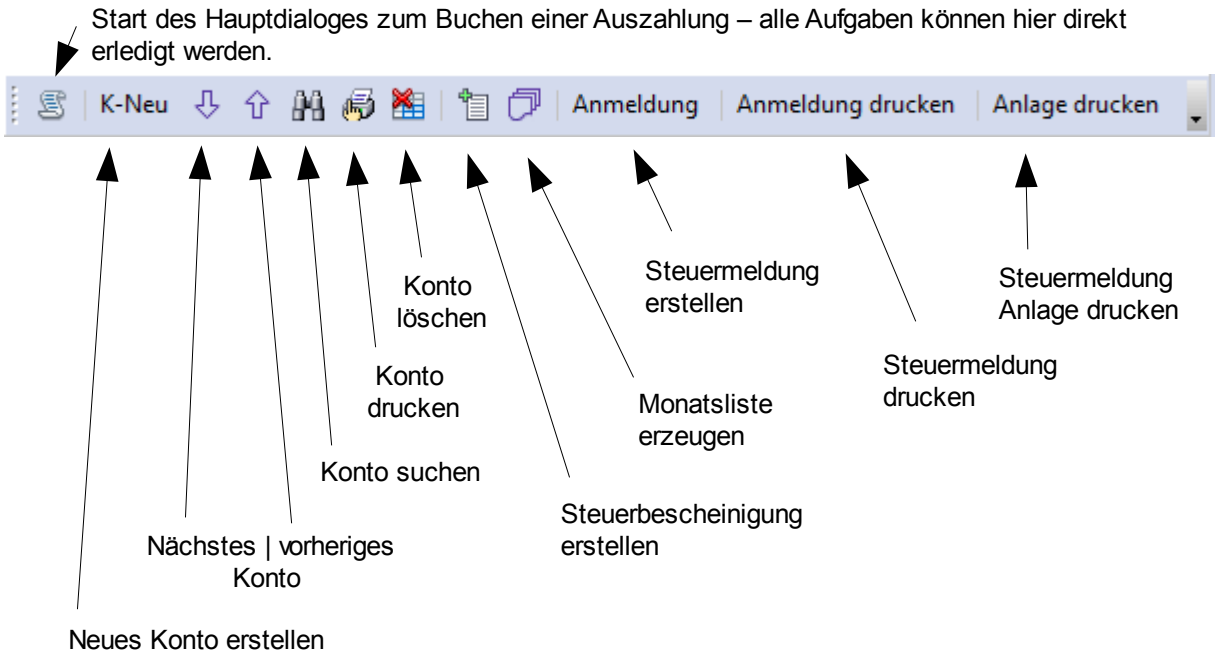


Für den/die Nutzer/in gibt es nun jede Menge Vorteile: Über die Listbox kann er/sie das passende Konto suchen – die Detaildaten werden gleich angezeigt, können aber hier nicht verändert werden. Gleichzeitig wird das Konto im Hintergrund auch angezeigt.

Jetzt können alle Daten eingegeben werden – das Datum kann gewählt werden, die Steuersätze sind vorgegeben. Die Brutto-Netto-Rechnung erfolgt automatisch, je nachdem, welchen Wert man eingibt, und eine Plausibilitätsüberprüfung ist auch gleich mit eingebaut.

Über die Checkboxen können auch gleich die weiterführenden Arbeiten erledigt werden – inklusive der Eingabe der zusätzlichen Daten für die Steuerbescheinigung.

Andere Aufgaben werden – wie bisher – über eigene Makros gestartet. Dafür gibt es eine eigene Symbolleiste, die alle Aufgaben auflistet:



Beispiel einer Symbolleiste.

Für die Programmierung bedeutet es nun, dass der Dialog zusätzlich existiert – der bisherige Weg also auch noch möglich ist.

Zum Aufruf des Dialoges (Start des Gesamtvorgangs):

```
'/** MAK160_BuchungszeileEingeben
'*****************************************
'* @kurztext startet den Dialog zur Eingabe einer Buchungszeile
'* Die Prozedur startet den Dialog zur Eingabe einer Buchungszeile
'* Prozedur trägt die Buchungszeile in das Konto (entweder das aktive oder ein gewähltes)
'* ein und startet auf Wunsch auch die Monatsliste sowie die Steuerbescheinigung.
'*  Verknüpft mit dem Icon "Buchung" der Symbolleiste
'*****************************************
'*/
Sub MAK160_BuchungszeileEingeben
  dim oTab as variant, sName as string, iZeile as long
  dim oSel as variant, aKtoNamen()

  MAK160_init.MAK160_Maininit    'Initialisierungen
  REM Prüfen, ob Cursor in einem Konto steht - dann Namen extrahieren
  oSel = thisComponent.CurrentSelection
  oTab = thisComponent.getSheets().getbyName(MAK160_TABKTO)
  iZeile = MAK160_tools.MAK160_checkSelection(oSel, oTab, true)
  if NOT (iZeile = -1) then  'in einem Konto
    iZeile = Int(iZeile/MAK160_KTOZEILEN) * MAK160_KTOZEILEN +2  'Zeile des Namens
    sName = oTab.getCellByPosition(1, iZeile).string
```

```
  end if
  aKtoNamen = MAK160_Tools.MAK160_getKontenNamen()  'Namen der vorhandenen Konten
  REM Dialog erzeugen
  oDlg = CreateUnoDialog(dialogLibraries.MAK160_AS.dlg_buch)
  REM Dialog initialisieren
  with oDlg
    .model.title = oDlg.model.title & " - " & MAK160_VERSION
    .getControl("lst_kto").model.stringItemList = aKtoNamen
    .getControl("lst_kto").selectItem(sName, true)   'aktuelles Konto evtl. markieren
    REM Kontodaten einlesen
    if iZeile > -1 then
      .getControl("txt_kto").text = sName
      .getControl("txt_knr").text = oTab.getCellByPosition(1, iZeile-1).string
      .getControl("txt_ort").text = oTab.getCellByPosition(1, iZeile+2).string & " / " & _
                                    oTab.getCellByPosition(1, iZeile+3).string
    end if
    .getControl("lst_monat").model.stringItemList = aMonate
    .getControl("lst_est").model.stringItemList = MAK160_tools.MAK160_getListeSteuersatz()
    .getControl("lst_est").selectItem("15,00", true)
    if thisComponent.getSheets().hasbyName(MAK160_TABSTSATZ) then
      .getControl("num_soli").value =
thisComponent.getSheets().getbyName(MAK160_TABSTSATZ).getCellRangeByName("F2").value
    end if
    REM Steuerbescheinigung
    .getControl("cbo_art").model.stringItemList = MAK160_tools.MAK160_getListeTaetigkeiten()
    .getControl("lst_sb").model.stringItemList = MAK160_tools.MAK160_getListeSachbearbeiter()
    .getControl("lst_sb").selectItemPos(0,true)   'ersten Eintrag vorwählen
  end with

  oDlg.execute()

  if bNeuesKto then MAK160_Konto.MAK160_DialogboxKontoStarten  'neues Konto Dialog
End Sub
```

Die wesentlichen Punkte:

- Basisinitialisierung in einer eigenen Routine (`MAK160_init.MAK160_Maininit`) – Laden der DialogLibraries, der Tools-Bibliothek und der Initialisierung der Monatsnamen-Listen (array).
- Prüfen der aktuellen Markierung – steht der Cursor in einem Konto, so wird dieses als Vorgabe in den Dialog eingetragen (wahrscheinlicher Arbeitsgang – Benutzerhilfe).
- Konten-Liste (Namen) extrahieren. Dies geschieht einmalig über eine Suchfunktion:

```
'/** MAK160_getKontenNamen()
'*******************************************
'* @kurztext extrahiert die vorhandenen Kontennamen
'* Die Funktion extrahiert die vorhandenen Kontennamen und liefert eine Liste zurück
'*
'* @return   aListe as array   Liste der Kontennamen
'*******************************************
'*/
function MAK160_getKontenNamen()
  dim oTab as variant, iZeile as long
  dim aListe(), nStep as integer, n%, i as long, s, j
```

```
  oTab = thisComponent.getSheets().getbyName(MAK160_TABKTO)
  nStep = 50   'Step des Arrays
  redim aListe(nStep)
  n = 1 : i = 0
  iZeile = 2    'Index der Zeile mit dem Kontonamen (erstes Konto)
  do until oTab.getCellByPosition(1, iZeile).getType() = 0
    aListe(i) = oTab.getCellByPosition(1, iZeile).string
    i = i + 1
    iZeile = iZeile + MAK160_KTOZEILEN
    if i = n * nStep then  'Array vergrößern
      n = n + 1
      redim Preserve aListe(n * nStep)
    end if
  loop
  if i > 0 then redim Preserve aListe(i-1)

  MAK160_getKontenNamen = aListe()
end function
```

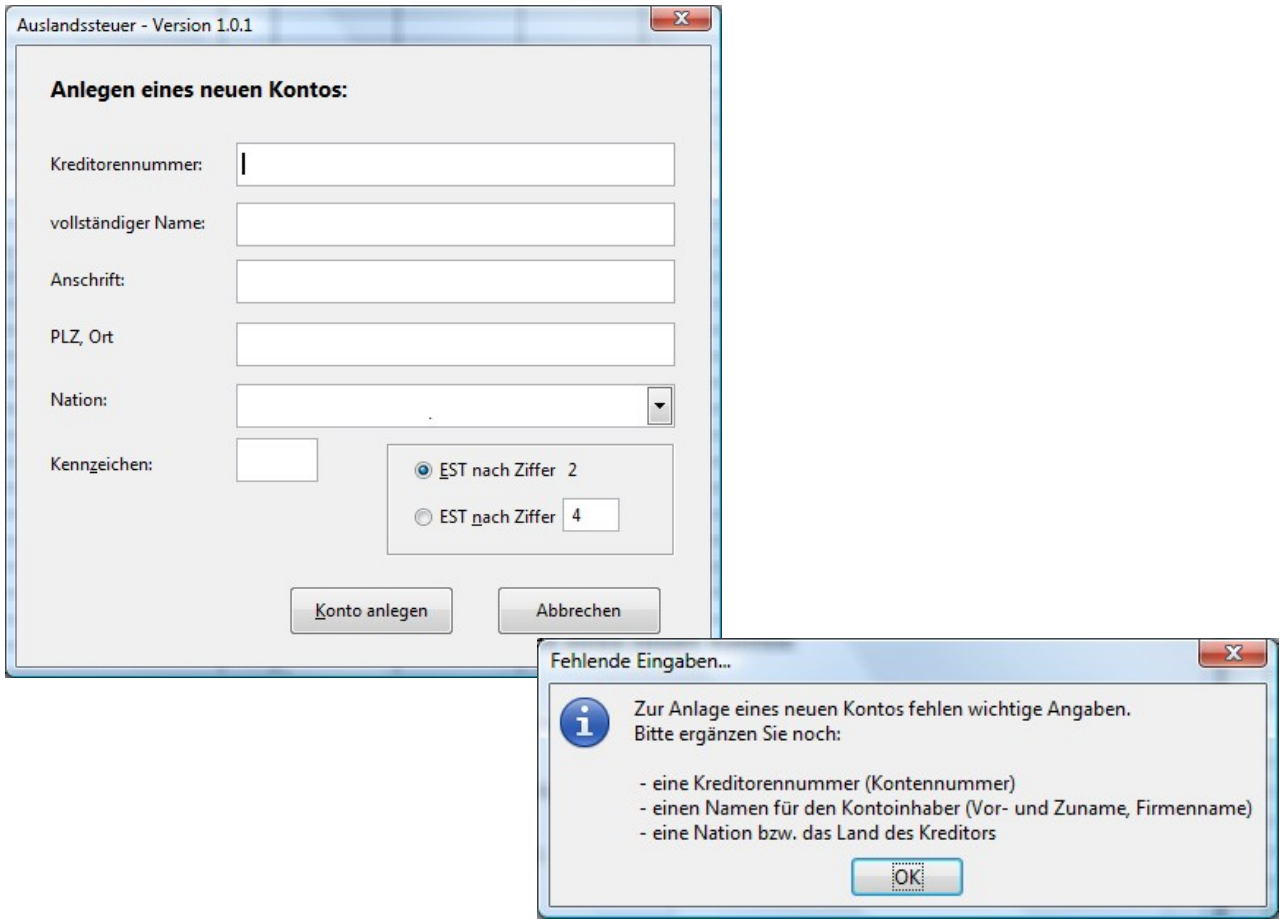
- Die Anzahl der Kontenzeilen pro Konto ist als Konstante fixiert (`MAK160_KTOZEILEN`).
- Dialog erzeugen und Daten entsprechend eintragen.
- Dialog ausführen.

Alle Aktionen des Dialoges werden über die Buttons und über eigene Routinen direkt ausgeführt.

Der Button „Neu“ neben dem Konto setzt das Flag `bNeuesKto` , beendet den Buchungsdialog und startet dann einen eigenen Dialog zum Erstellen eines neuen Kontos.

Anlegen eines neuen Kontos

Auch das Anlegen eines neuen Kontos kann exemplarisch als Beispiel dienen, wie eine effektive Fehlerkorrektur erstellt werden kann. Der Dialog wird entweder über den Button Neu des Buchungsdialoges oder über das entsprechende Icon der Benutzersymbolleiste aufgerufen. Er stellt alle benötigten Eingabefelder zur Verfügung:



Mit Klick auf den Button „Konto anlegen“ startet die entsprechende Routine, die zunächst prüft, ob die Muss-Daten ausgefüllt wurden – ansonsten bricht die Bearbeitung des Anlegens mit einer Fehlermeldung ab – der Dialog bleibt aber offen. Bereits getätigte Eintragungen sind nicht weg, der/die Benutzer/in kann nachbessern und es erneut versuchen.

```
'/** MAK160_KontoAnlegen()
'*******************************************
'* @kurztext legt ein neues Konto an und trägt die Daten ein
'* Die Funktion legt ein neues Konto an und trägt die Daten ein
'* Dieses passiert in der Tabelle "Konten", und zwar unter das letzte Konto
'* Verknüpft mit dem Button "Konto anlegen" des Dialoges
'*******************************************
'*/
Sub MAK160_KontoAnlegen()
  dim sFtxt as string, bFFlag as boolean, oTab as variant, bEndeFlag as boolean
  dim sAnschrift as string, sOrt as string, sNation as string, sNz as string
  dim sKNr as string, iEstZiffer as integer, sName as string
  dim iZeile as long, oZBereich as variant, oQBereich as variant
  dim n%

  REM Prüfen, ob Mussfelder ausgefüllt sind
  sFtxt = "Zur Anlage eines neuen Kontos fehlen wichtige Angaben." & chr(13) & _
          "Bitte ergänzen Sie noch:" & chr(13) & chr(13)
  if trim(oDlg.getControl("txt_krnr").text) = "" then  'Kontonummer
```

```
    sFtxt = sFtxt & " - eine Kreditorennummer (Kontennummer) " & chr(13)
    bfflag = true
  end if
  if trim(oDlg.getControl("txt_name").text) = "" then  'Namensprüfung
    sFtxt = sFtxt & " - einen Namen für den Kontoinhaber (Vor- und Zuname, Firmenname) " &
chr(13)
    bfflag = true
  end if
  if trim(oDlg.getControl("cbo_nation").text) = "" then  'Länderprüfung
    sFtxt = sFtxt & " - eine Nation bzw. das Land des Kreditors " & chr(13)
    bfflag = true
  end if
  if bFFlag then
    msgbox (sFtxt, 64, "Fehlende Eingaben...")
    exit sub
  end if

  ... 'hier geht der Code dann weiter...
```

Wichtig auch hier: zunächst werden alle Fehler gesammelt – hier direkt im Fehlertext-String – dann erst erfolgt eine gesammelte Ausgabe und der Stopp der Prozedur.

Sind keine Fehler mehr enthalten, wird das Konto angelegt. Die Daten des Dialoges (Eingaben) werden zunächst zwischengespeichert, dann wird überprüft, ob

- eine Kontentabelle überhaupt schon existiert (könnte ja das erste Konto sein!)
- ein Konto dieses Namens bereits existiert – dann Rückfrage, ob dieses ersetzt oder zusätzlich angelegt werden soll.

Wird es angefügt, kommt es auch gleich an die passende Stelle – alphabetisch korrekt eingeordnet.

```
... 'Code von weiter oben wird fortgesetzt ..
  REM Prüfen, ob es eine Tabelle Konten gibt - sonst erzeugen
  if not ThisComponent.getSheets().hasbyname(MAK160_TABKTO) then   'Tabelle erzeugen
    ThisComponent.getSheets.CopyByName(MAK160_TABKTOMUSTER, MAK160_TABKTO, 0)
    iZeile = 0     'Startzeile  = Zeile 1 Index 0
    oTab = ThisComponent.getSheets().getbyname(MAK160_TABKTO)
  else     'Tabelle besteht, neues Kontomuster unten anfügen
    oTab = ThisComponent.getSheets().getbyname(MAK160_TABKTO)
    REM Suchen, wo das konto inkommt - entweder ans Ende oder eingeordnet alphabetisch
    iZeile = 2   'Zeile 3 jedes Kontos ist Prüfzeile
    Do until oTab.getCellByPosition(1, iZeile).getType() = 0
      REM Prüfen, ob der aktuelle Eintrag (Name) bereits existiert - dann danach einfügen
      if lcase(trim(oTab.getCellByPosition(1, iZeile).getString())) = lcase(trim(sName)) then
        n = msgbox ("Ein Konto mit dem Namen """ & trim(sName) & """ besteht bereits." &
chr(13) & _
                    "Soll dennoch ein weiteres Konto angelegt werden?", 3 + 32 + 512,
"Kontonamen gleich...")
        if n = 2 then
          exit sub   'Abbruch, - zurück zum Dialog
        elseif n = 7  then  'Abbruch, Dialog Ende
          odlg.endExecute()
          exit sub
        elseif n = 4  then  'ja - neues Konto anlegen
          iZeile = iZeile - 2 + MAK160_KTOZEILEN
```

```
          bEndeFlag = true
          exit do
        end if

      elseif lcase(trim(oTab.getCellByPosition(1, iZeile).getString())) > lcase(trim(sName))
then  'Einfügeposition davor
        iZeile = iZeile - 2 'zwei Zeilen zurück
        bEndeFlag = true
        exit do
      end if
      iZeile = iZeile + MAK160_KTOZEILEN    'nächstes Konto
    loop
    if NOT bEndeFlag then iZeile = iZeile - 2  'Korrektur, falls am Ende eingefügt wird

    REM neues leeres Konto einfügen an Position iZeile
    REM dazu zunächst "Platz" schaffen, Anzahl Zeilen einfügen
    REM Zeilen werden vor der benannten Indexzeile eingefügt
    oTab.getRows().insertByIndex(iZeile, MAK160_KTOZEILEN)
    oZBereich = oTab.getCellByPosition(0, iZeile).CellAddress
    oQBereich = thisComponent.sheets.getByName(MAK160_TABKTOMUSTER).getCellRangeByPosition(0,
0, 11, MAK160_KTOZEILEN-1).RangeAddress
    wait(100)
    oTab.CopyRange(oZBereich, oQBereich)
  end if

    REM Dialog schließen
    oDlg.endexecute()
    REM jetzt Daten eintragen
    with oTab
      .getCellByPosition(1, iZeile+1).string = sKnr         'Kreditorennummer/Bez
      .getCellByPosition(1, iZeile+2).string = sName        'Name
      .getCellByPosition(1, iZeile+3).string = sAnschrift   'Anschrift
      .getCellByPosition(1, iZeile+4).string = sOrt         'Ort und PLZ
      .getCellByPosition(1, iZeile+5).string = sNation      'Nation
      .getCellByPosition(1, iZeile+6).value = iEstZiffer    'EstNr
      .getCellByPosition(3, iZeile+6).string = sNz          'Nationen-Kürzel
      thisComponent.getCurrentController().select(.getCellByPosition(1, iZeile+2))
    end with

end sub
```

Zum Einfügen wird zunächst Platz geschaffen, dann wird ein definiertes Zellmuster (untergebracht in einer versteckten Tabelle) genau dort eingefügt.

Zum Schluss werden noch die Daten eingetragen und das Konto sichtbar auf dem Bildschirm platziert.

Viele der hier geschilderten Details hat der/die Benutzer/in vorher manuell erledigt – und wusste, was er/sie tat. Per Code müssen die Abläufe genauer geplant und auch Eventualitäten berücksichtigt werden, die im Arbeitsablauf (auch in dessen Beschreibung) in der Regel nicht vorkommen (zum Beispiel das Fehlen der Tabelle „Konten“ – beim Jahreswechsel zum Beispiel oder weil sie absichtlich/versehentlich gelöscht wurde.

Wichtig beim Umstellen von Altmakros:

Normalerweise sind Makros „gewachsen“, also Stück für Stück programmiert worden, immer dann, wenn ein Ablauf zu oft wiederholt werden musste.

Aus solch einem „Makro-Sammelsurium“ eine Applikation zu machen, erfordert einen völlig anderen Blick. An dieser Stelle sollte der (logische) Arbeitsablauf im Vordergrund stehen – nicht die bisherige Ablaufpraxis. Jede Applikation muss deutlich fehlerresistenter sein und je stärker der/die Benutzer/in „geleitet“ wird, um so mehr muss daran gedacht werden, was passiert, wenn er/sie anders reagiert oder agiert. Der/Die Benutzer/in ist nicht mehr in der Lage, kleinere Unstimmigkeiten oder Fehler selbst auszubügeln – noch wird er/sie die Bereitschaft dazu mitbringen.

Das Leitmotiv einer Datenbank-Applikation auf der Basis von Calc muss also lauten:

Alle Aktionen werden in Dialogen durchgeführt, die direkte Eingabe von Werten/Daten oder die Veränderung von Zeilen-/Spaltenstrukturen sind unbedingt zu vermeiden. Die Vorteile einer Calc-Applikation liegen sicher in der einfachen Aufbereitung von Druckausgaben – entsprechende Tabellenblätter können vordefiniert (Layout), zur Laufzeit mit Daten/Inhalten gefüllt und dann einfach ausgedruckt werden.

Für die „normale“ UI-Schnittstelle – in der Regel ein Dialog – und für die eigentliche interne Verarbeitung der Daten spielt es hingegen keine große Rolle, ob die Bewegungsdaten in Calc direkt oder in einer Datenbank gespeichert werden.

Auch für Datenbanken gibt es effektive Standardzugriffsroutinen, der Programmier-Mehraufwand kommt in er Regel auch von zusätzlichen Features.

9 Applikationen verteilen

Während Applikationen, deren Code komplett in einer Datei gespeichert wird, für die Verteilung bestens geeignet sind, trifft dies für Programmerweiterungen nicht zu.

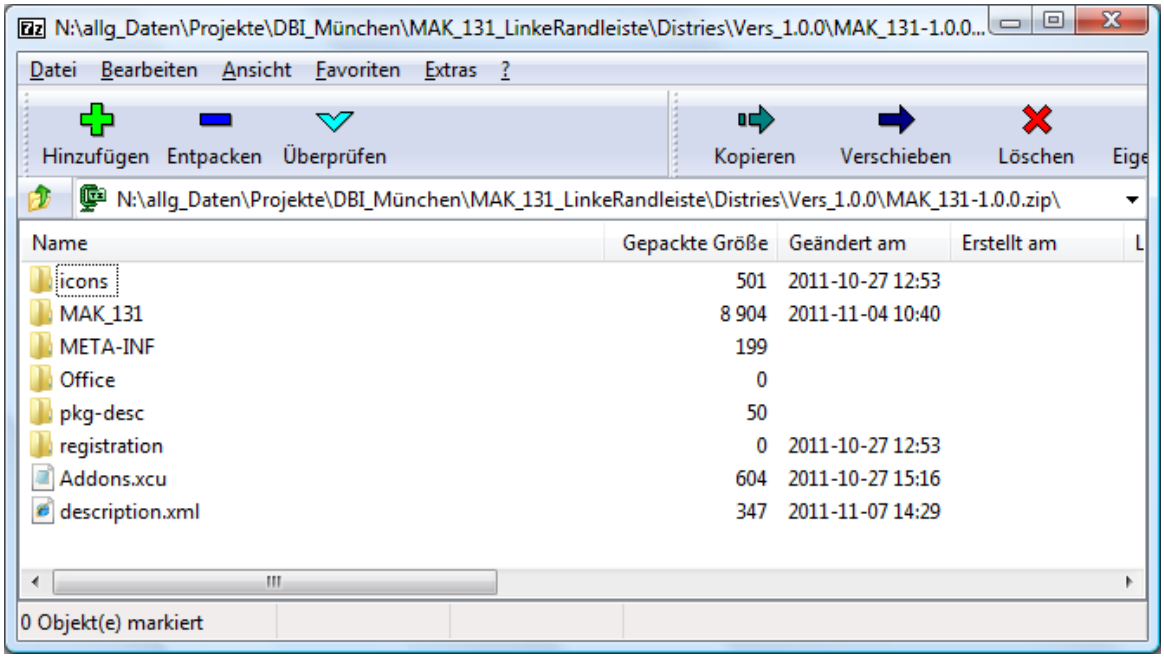
Diese werden als Extensions verteilt und können dann entweder manuell am Arbeitsplatz-Rechner installiert oder per Software-Verteilung ausgerollt werden. Für die korrekte Einbindung in die Office-Suite ist das Programm „unopgk“, Teil des Hauptprogramms, zuständig.

Üblicherweise wird die Dateierweiterung *.oxt direkt mit diesem Programm während der Installation von OpenOffice.org verknüpft, so dass ein Doppelklick auf eine so benannte Extension diese direkt installiert.

9.1 Extensions

Extensions sind reine Zip-Archive, die aber eine bestimmte Struktur aufweisen müssen.

Im folgenden Bild ist einmal die minimale Struktur abgebildet – am Beispiel von MAK_131:



Mit die wichtigste Datei ist die description.xml sowie – damit direkt im Zusammenhang zu sehen – das Verzeichnis META-INF, das genau eine Datei enthält: manifest.xml.

Diese beiden Dateien sind fix und werden intern ausgewertet – und auf deren Basis wird die Extension installiert.

9.1.1 description.xml

Bleiben wir beim Beispiel: die Datei description.xml enthält die wesentlichen Informationen zur Extension an sich – und die Informationen für das Programm **unopgk**, welches die Installation vornimmt.

Der Prozess ist dabei wie folgt: Aus der description.xml wird zunächst der eindeutige Name der Extension ausgelesen (Tag „identifier“). Dieser wird nun in der internen Datenbank unter den installierten Extensions gesucht – ist er bereits vorhanden, dann wird die Versionsnummer (Tag „version“) ausgelesen und mit der bereits installierten Version verglichen. Je nach Ergebnis des Vergleichs (gleich, größer oder kleiner) erhält der/die Benutzer/in eine entsprechende Meldung und kann die Installation abbrechen.

Als nächstes wird der Knoten „dependencies“ ausgewertet – falls vorhanden. Hier wird beispielsweise eine minimale OOo-Version von 3.0.1 vorausgesetzt. Dies wird nun intern verglichen und ist die installierte Version niedriger, erhält der/die Benutzer/in erneut eine Meldung – die Installation wird dann aber nicht durchgeführt bzw. die Erweiterung ist nicht nutzbar.

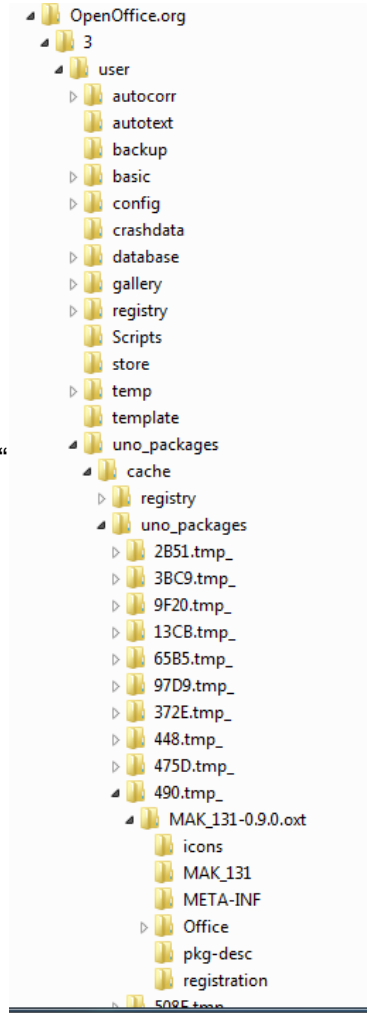
Schließlich wird der Knoten „registration“ ausgewertet und – falls dort entsprechende Tags übergeben wurden – werden nun die entsprechenden Dialoge mit den Texten (meist Lizenztexte, die bestätigt werden müssen) angezeigt. In Abhängigkeit der Benutzereingabe wird nun entweder die Installation beendet oder fortgesetzt.

```
<?xml version='1.0' encoding='UTF-8'?>
<description
 xmlns="http://openoffice.org/extensions/description/2006"
 xmlns:dep="http://openoffice.org/extensions/description/2006"
 xmlns:xlink="http://www.w3.org/1999/xlink">
    <identifier value="de.muenchen.allg.dbi.mak_131_lrl"/>
    <version value="1.0.0"/>
    <dependencies>
        <OpenOffice.org-minimal-version value="3.0.1" dep:name="OpenOffice.org 3.0.1"/>
    </dependencies>
    <publisher>
        <name xlink:href="http://www.muenchen.de" lang="de">Landeshauptstadt München</name>
    </publisher>

    <display-name>
        <name lang="de-DE">MAK_131 Linke Randleiste</name>
    </display-name>
    <registration>
    </registration>
</description>

```

Sind die ersten Hürden „genommen“, werden die Dateien des Zip-Archives in das Benutzerprofil (oder bei Installation mit dem Parameter --shared auch in das Programmverzeichnis) kopiert und die Extension wird intern registriert. Das Zielverzeichnis ist dabei wie folgt aufgebaut (Beispiel Benutzerinstallation):

<Pfad Benutzerprofil>/user/uno_packages/cache/uno_packages/123F.tmp_/<ExtensionFileName.oxt>/

Als Beispiel hier die Extension MAK_131, noch in der Version 0.9.0:



Die Verzeichnisse mit der Endung *.tmp_ werden zufällig, aber eindeutig vom Installationsprogramm erzeugt und intern mit registriert. In dem Verzeichnis der Extension sind dann alle Dateien enthalten, die auch im Zip-Archiv vorhanden waren.

Es ist aber nicht gesagt, dass diese Dateien auch alle genutzt werden. Die vom Programm benötigten Dateien sind identifiziert und spezifiziert in der Datei „manifest.xml“ im Unterverzeichnis META-INF.

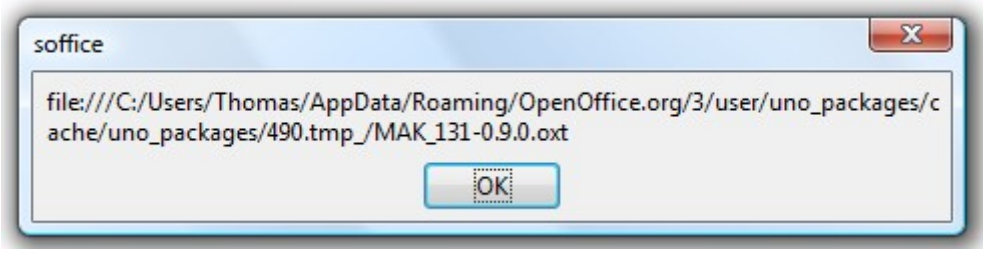
Dateien oder Verzeichnisse, die dort nicht spezifiziert werden, werden von OpenOffice.org ignoriert und nicht genutzt (Ausnahmen: die „Pflichtdateien“ wie eben erwähnt sowie die Datei „addon.xcu“, die im Rootverzeichnis optional vorhanden sein kann, diese wird immer ausgewertet).

Es ist also durchaus möglich, in der Extension zusätzliche Dateien unterzubringen, die entweder durch das Programm selbst (also das Makro selbst) genutzt werden oder lediglich als Information dort vorhanden sind (zum Beispiel Versionshinweise, Anleitungen etc.).

Auch Hilfedateien zum Beispiel in der Form von PDF-Dokumenten lassen sich – auch in eigenen Verzeichnissen – hier bequem unterbringen. Der Vorteil: Der Speicherpfad ist immer bekannt und existiert im System. Zumindest das Benutzerverzeichnis ist nie schreibgeschützt – Prozesse des Benutzers / der Benutzerin können also sowohl lesend als auch schreibend darauf zugreifen. Einzige Einschränkung: Der Pfad kann im Vorfeld nicht eindeutig fixiert werden – lediglich der relative Pfad innerhalb des Installationsverzeichnisses der Extension.

Doch lässt sich der Pfad ganz einfach mit entsprechenden Services auslesen:

```
function GetExtensionInstPfad
  Dim sService as string, sExtensionIdentifier as string, sPackageLocation as string
  dim oPackageInfoProvider as variant

  sExtensionIdentifier = "de.muenchen.allg.dbi.mak_131_lrl"

  sService = "com.sun.star.deployment.PackageInformationProvider"
  oPackageInfoProvider = GetDefaultContext.getValueByName("/singletons/" & sService)
  sPackageLocation = oPackageInfoProvider.getPackageLocation(sExtensionIdentifier)

  msgbox sPackageLocation

End function
```



9.1.2 manifest.xml

Die Datei „manifest.xml“ im Unterverzeichnis META-INF dient OOo dazu, die unterschiedlichen benötigten Verzeichnisse und Dateien eindeutig zu identifizieren:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <manifest:manifest>
3   <manifest:file-entry manifest:full-path="MAK_131/" manifest:media-type=
   "application/vnd.sun.star.basic-library"/>
4   <manifest:file-entry manifest:full-path="pkg-desc/pkg-description.txt" manifest:media-type=
   "application/vnd.sun.star.package-bundle-description"/>
5   <manifest:file-entry manifest:full-path="Addons.xcu" manifest:media-type=
   "application/vnd.sun.star.configuration-data"/>
6  </manifest:manifest>
```

Erkennbar sind:

- das Basic-Code-Verzeichnis
- der Pfad und der Name der Beschreibungsdatei (diese wird im Extensionmanager zur Erweiterung angezeigt)
- der Pfad und der Dateiname zu der Konfigurationsdatei.

Konfigurationsdateien werden beim Programm-Start (von OOo) nacheinander eingelesen und verarbeitet. Die Reihenfolge ist dabei wie folgt:

1. Konfigurationsdateien der OOo-Programm-Installation
2. Konfigurationsdateien der OOo-Benutzer-Installation (User-Verzeichnis)
3. Konfigurationsdateien der Extension-Installationen (ob es dabei ebenfalls eine Reihenfolge gibt, kann ich nicht mit Sicherheit sagen, vermutlich wird die Datenbank nacheinander abgearbeitet).

Diese Reihenfolge ist insofern wichtig, als jede Folgekonfigurationsdatei die bisherigen Einstellungen überschreibt! Die letzte gewinnt – und ist dann gültig.

Konfigurationen werden typischerweise mit der addon.xcu übergeben.

9.1.3 addon.xcu

```
<?xml version='1.0' encoding='UTF-8'?>
<oor:component-data xmlns:oor="http://openoffice.org/2001/registry" xmlns:xs=
"http://www.w3.org/2001/XMLSchema" oor:name="Addons" oor:package="org.openoffice.Office">
 <node oor:name="AddonUI">
  <node oor:name="OfficeToolbarMerging">
   <node oor:name="LinkeRandleiste_MAK131.OfficeToolBar" oor:op="replace">
    <node oor:name="T1" oor:op="replace">
     <prop oor:name="MergeToolBar">
      <value>standardbar</value>
     </prop>
     <prop oor:name="MergePoint">
      <value>.uno:PrintPreview</value>
     </prop>
     <prop oor:name="MergeCommand">
      <value>AddAfter</value>
     </prop>
     <prop oor:name="MergeFallback">
      <value>AddLast</value>
     </prop>
     <prop oor:name="MergeContext">
      <value/>
     </prop>
     <node oor:name="ToolBarItems">
      <node oor:name="m1" oor:op="replace">
     <prop oor:name="Context" oor:type="xs:string">
       <value>com.sun.star.text.TextDocument</value>
     </prop>
     <prop oor:name="URL" oor:type="xs:string">
       <value>macro:///MAK_131.MAK131_Start.MAK131_StartLRL</value>
     </prop>
        <prop oor:name="ImageIdentifier" oor:type="xs:string">
       <value/>
     </prop>
     <prop oor:name="Title" oor:type="xs:string">
       <value>Linke Randleiste einfügen/aktualisieren/entfernen</value>
     </prop>
     <prop oor:name="Target" oor:type="xs:string">
       <value>_self</value>
     </prop>
      </node>
     </node>
     </node>
```

In einer „addon.xcu“ werden typischerweise eigene Symbolleisten definiert oder Ergänzungen/Umbenennungen der bestehenden Symbol- und Menüleisten. Es handelt sich dabei ebenfalls um eine xml-Datei – die Tags sind in der Dokumentation hinreichend beschrieben. Im Beispiel wird in die Standard-Symbolleiste von Writer ein zusätzliches Icon eingefügt, und zwar nach dem Icon Seitenansicht:Das Icon selbst wird im weiteren Verlauf der Datei spezifiziert – in einem eigenen Knoten:

Das Icon selbst (die Grafik) befindet sich im Unterordner „/icons/“ in der Extension. Die Spezifikation erfolgt hier – und ist auch ausreichend. Dieses Verzeichnis muss nicht in der manifest.xml angemeldet sein.

Die Verbindung zu der Einfügestelle erfolgt über die URL, hier also den Marko-Befehl.

```
<node oor:name="Images">
 <node oor:name="mic.de.lhm.mak131.images" oor:op="replace">
  <prop oor:name="URL" oor:type="xs:string">
   <value>macro:///MAK_131.MAK131_Start.MAK131_StartLRL</value>
  </prop>
  <node oor:name="UserDefinedImages">
   <prop oor:name="ImageSmallURL" oor:type="xs:string">
    <value>%origin%/icons/lhmlrl.png</value>
   </prop>
   <prop oor:name="ImageSmallHCURL" oor:type="xs:string">
    <value>%origin%/icons/lhmlrl.png</value>
   </prop>
  </node>
 </node>
</node>
</oor:component-data>
```

Mit der Konfigurationsdatei können umfangreiche Änderungen an der UI vorgenommen werden – es können auch bestehende Icons mit neuen Programmbefehlen verknüpft werden, um so eigene Routinen an Stelle der integrierten zu setzen.

Aber Achtung!

Nicht alle Aufrufe sind wirklich „abfangbar“. Neben den Icons und den Menü-Befehlen gibt es oft noch interne Dialoge, die dann dennoch den Originalbefehl ausführen. Beispiel: „Datei Öffnen“ – neben dem Icon und dem Menü-Befehl „Datei/Öffnen...“ kann der/die Benutzer/in eine Datei auch über „Datei/Dokumentenvorlage“ → und dort über den Button „Datei“ öffnen – und der Aufruf ist nicht abfangbar über die Konfigurationsdatei!

9.1.4 Basic-Verzeichnis

Im Basic-Verzeichnis schließlich ist Ihre programmierte Basic-Bibliothek komplett enthalten – der Verzeichnisname ist der Name der Bibliothek. Die Bibliothek beinhaltet alle Module als eigenständige Dateien sowie alle Dialoge als XML-Dateien. Dazu kommen die Steuerungsdateien dialog.xlb und script.xlb, die natürlich korrekt aufgebaut sein müssen (siehe hierzu auch Kapitel 4.4.3).

Ist die Extension installiert und schaut man in den Ordner des Basic-Verzeichnisses, erscheint dort zusätzlich eine Datei „RegisteredFlag“. Die Datei ist leer und lediglich für OOo interessant – so wird dargestellt, dass die (Basic-)Extension in die Dateien script.xlc und dialog.xlc aufgenommen wurde und somit in der IDE angezeigt und bearbeitet werden kann.

9.2 Extension erzeugen

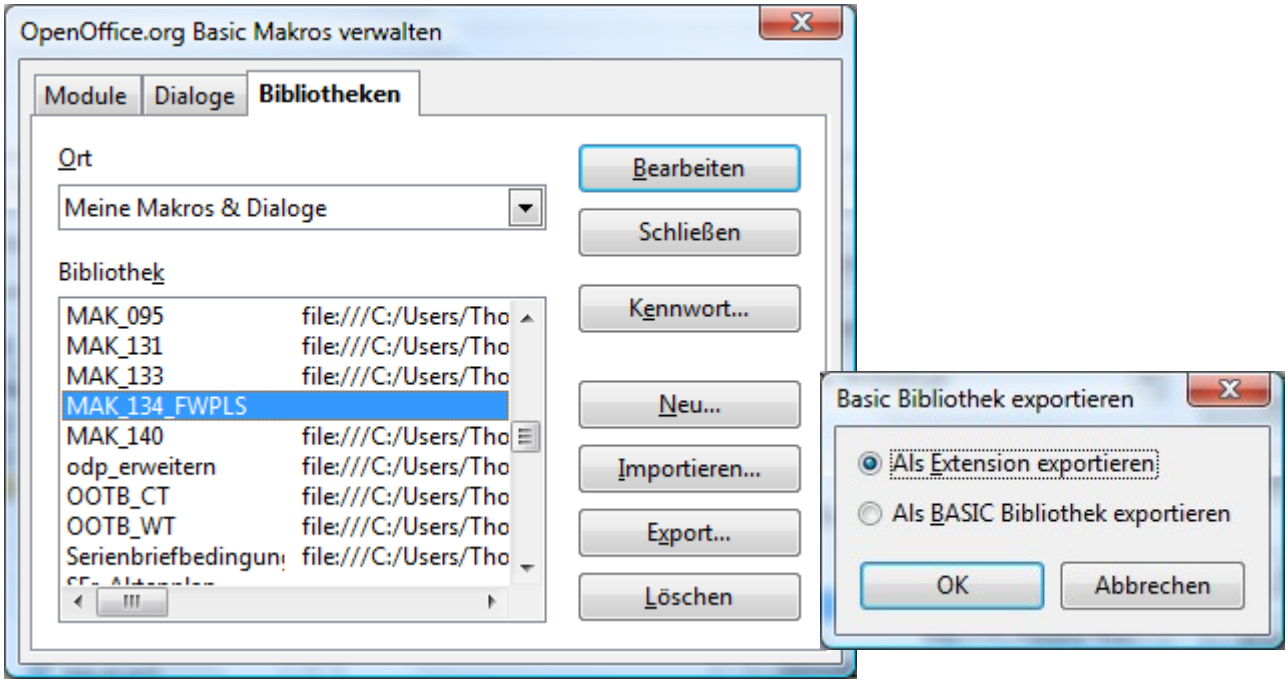
Nachdem nun klar ist, wie eine Extension aufgebaut ist, ist es einfach möglich, eine solche auch zu erzeugen. Alle xml-Files können manuell aufgebaut werden – was natürlich eine „Heidenarbeit“ darstellt und nicht wirklich sinnvoll erscheint.

Ein gangbarer Weg ist folgender:

Die Entwicklung der Extension erfolgt auf einem Entwicklungsrechner. Dort wird zunächst eine neue (Basic-)Bibliothek mit dem gewünschten Namen angelegt – dann in der Bibliothek die entsprechenden Module und Dialoge. Alle Tests werden direkt hier durchgeführt. Läuft alles zur Zufriedenheit, dann kann die Bibliothek als Extension exportiert werden:

9.2.1 Bibliothek als Extension exportieren

Im Verwaltungsdialog (Extras/Makros/Makros verwalten/OOo Basic... → Verwalten → Bibliotheken) kann nun eine Bibliothek ausgewählt werden und dann über den Button Export aus der aktuellen Struktur heraus exportiert werden:



Sie haben dabei die Wahl zwischen einer „Extension“ und einer „BASIC-Bibliothek“. Wählen Sie die „Extension“, so wird ein Zip-Archiv mit der Bibliothek erstellt und als *.oxt Datei abgespeichert. Diese Extension ist dann aber eine „Rumpf-Extension“ – sie lässt sich zwar installieren, fügt dann aber lediglich die Bibliothek auf dem Zielrechner hinzu.

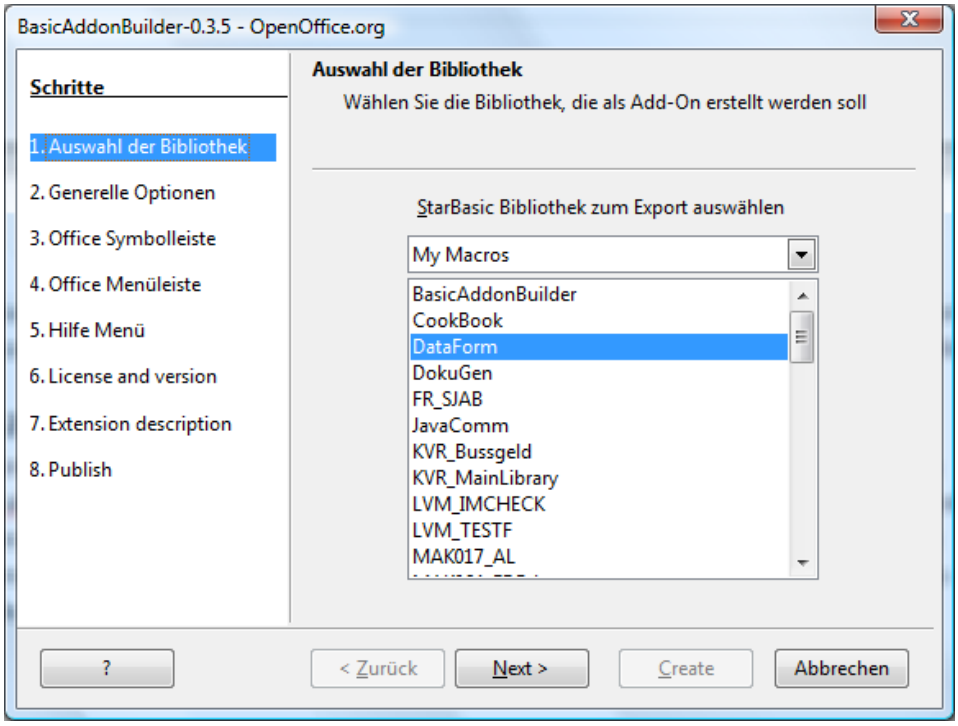
Alle anderen Konfigurations- und Steuerdateien müssen nun zusätzlich erstellt und der Extension beigefügt werden.

Wählen Sie im übrigen „als BASIC-Bibliothek“, dann wird ein Verzeichnis erstellt (mit dem Namen der Bibliothek) und darin befinden sich alle Module und Dialoge – eben so, wie sie diese auch in der Zip-Datei vorfinden – jetzt allerdings unkomprimiert. Dies ist ein sehr guter Weg, die Daten zu sichern, und es sollte regelmäßig durchgeführt werden!

Doch zurück zur Extension. Da durch den Export lediglich eine „Minimalversion“ einer Extension erzeugt wird, ist der Aufwand noch ziemlich hoch, daraus eine „gute“ Erweiterung zu erstellen.

9.2.2 BasicAddonBuilder

Einfacher geht es mit dem BasicAddonBuilder, selbst eine Extension, die dann dialoggeführt auf der Basis einer Basic-Bibliothek eine gute „Rumpf-Extension“ erzeugt – inklusive Beschreibungsdatei, Menü- und Symbolleisten sowie sonstiger Extras. „Rumpf“-Extension deswegen, weil auch diese Datei nicht perfekt ist und in vielem noch angepasst werden muss. Doch die Schritte sind dann einfach.



Der BasicAddonBuilder führt in klaren Schritten durch die einzelnen Positionen, die zu einer Extension gehören. Im Bild sieht man schon die aktivierten Optionen „Office Symbolleiste“, „Office Menüleiste“ sowie „Hilfe Menü“.

Im Ergebnis wird eine Extension erzeugt, die bereits eine addon.xcu sowie eine description.xml enthält, ebenso eine Paketbeschreibung, die Basic-Bibliothek und entsprechende Office-Addon-Dateien.

Hat man eine solche Extension erst einmal erzeugt, kann diese nun in eine Zip-Datei umbenannt und entpackt werden – jetzt können die Dateien einzeln bearbeitet und angepasst und schließlich wieder der Zip-Datei hinzugefügt werden. Dadurch wird die Struktur nicht verändert, die Extension funktioniert nach wie vor.

Folgende Änderungen müssen typischerweise nachträglich vorgenommen werden:

- Ergänzung der description.xcu um die Tags <publischer>, <display-name> und < dependencies>.
- Eventuell Anpassung der Versionsnummer und des Identifiers (in der description.xcu).
- Wichtig: Änderung des Dateiformates auf „UTF-8“. Der Basic-Addon-Builder erzeugt lediglich ACSII-Dateien – spätestens bei Umlauten in Namen versagen diese Dateien! Muss in allen Dateien durchgeführt werden, die nachträglich geändert werden!
- Ändern und Anpassen der Paketbeschreibung.

Zusätzlich können natürlich eigene Dateien integriert und Verzeichnisse erstellt werden.

Auf diese Art und Weise lässt sich schnell und mit geringem Aufwand eine gute Extension auf der Basis einer bereits vorhandenen Bibliothek erstellen.

9.2.3 Update

Eine bereits erstellte Extension kann genauso einfach erneuert werden – wenn beispielsweise Fehler im Code bereinigt wurden.

Dazu nimmt man die alte Extension, öffnet diese in einem Zip-Programm (die *.oxt Datei muss vorher meist in *.zip umbenannt werden) und löscht nun das Verzeichnis mit den Basic-Makros (Bibliotheksname = Verzeichnisname).

Dann wird die geänderte Bibliothek, wie weiter oben beschrieben, als „Basic-Bibliothek“ exportiert – und dieses Verzeichnis dann der Zip-Datei hinzugefügt. Da sich der Bibliotheksname kaum geändert hat, ist keine weitere Anpassung nötig.

Sinnvoll und notwendig ist natürlich, auch die Versionsnummer in der description.xml um eins zu erhöhen – und somit ein Update zu ermöglichen.

Damit wäre ein Update aber auch schon erledigt.

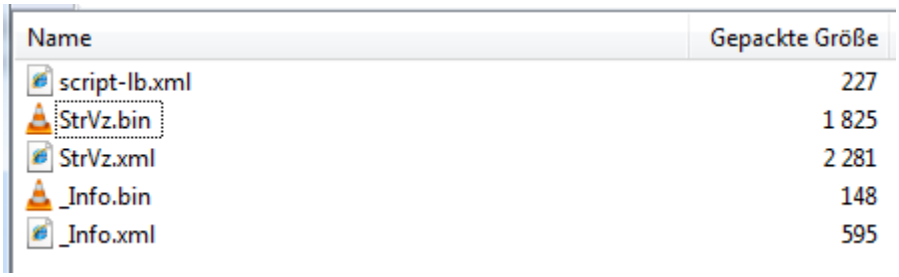
Es empfiehlt sich also, auf dem Entwicklungsrechner oder im Repository neben der Extension auch eine entpackte Version vorzuhalten und zu pflegen. Dann sind Ergänzungen und Updates schnell und einfach zu realisieren.

9.3 Bibliotheken verschlüsseln

Werden Bibliotheken verschlüsselt (also mit einem Kennwort versehen), werden die Dateien in einen Binär-Code umgewandelt und sind im Klartext nicht mehr zu lesen.

Den Makro-Vorschriften der LHM zufolge müssen Dokumentenmakros verschlüsselt werden. Die Verschlüsselung zieht nun einige Konsequenzen nach sich.

Aus jeder Modul-Datei werden nun zwei, einmal eine *.xml-Datei sowie eine dazu korrespondierende *.bin Datei. Beide sind in einem Editor im Klartext nicht mehr lesbar.



Name	Gepackte Größe
script-lb.xml	227
StrVz.bin	1 825
StrVz.xml	2 281
_Info.bin	148
_Info.xml	595

Zwar können die Makros problemlos ausgeführt werden (sie werden sowieso binär benötigt – normalerweise aber werden sie erst zur Laufzeit kompiliert), aber alle direkten Schreib- und Lesezugriffe auf die Bibliothek (Module) führen jetzt ins Leere.

Beispiel:

Die Funktion GetMarkoVersionsNummer, die die aktuelle Versionsnummer aus dem _Info-Modul extrahiert und diese dann beispielsweise in Dialog-Kopfzeilen schreibt, versagt bei verschlüsselten Bibliotheken.

```
'/** GetMakroVersionsNummer
'*************************************************************************.
'* @kurztext Liefert die Versionsnummer des Makros (aus dem Info-Block)
'* Diese Funktion liefert die Versionsnummer des Makros (aus dem Info-Block)
'*
'* @param1 sBibliothek as string   Name der Bibliothek  (Makro muss in der aktuellen Bibliothek
sein)
'*
'* @return sVersion as string   die eingestellte Versionsnummer, oder "---", falls keine
vorhanden
'*************************************************************************
'*/
function GetMakroVersionsNummer(sBibliothek as string)
  dim aCode()
  dim sCodeZeile as string
  dim i%

  if basicLibraries.hasByname(sBibliothek) then
    if basicLibraries.getByname(sBibliothek).hasByName("_Info") then
      aCode = split(basicLibraries.getByname(sBibliothek).getByName("_Info"), chr(10))
      for i = 0 to uBound(aCode)
        sCodeZeile = aCode(i)
        if left(lcase(sCodeZeile),12) = lcase("'*  Version:") then
          GetMakroVersionsNummer = trim(mid(sCodeZeile, 13))
```

```
            exit function
          end if
        next
      end if
    end if
    GetMakroVersionsNummer = "---"
End function
```

Der Grund ist einfach erklärt: Der Text des Moduls ist ja kein Klartext mehr – die Vergleichskriterien können nicht gefunden werden.

Im Fall der Verschlüsselung muss also die Versionsnummer als Konstante abgelegt und bei einem Update regelmäßig mit angepasst werden.

Gleiches gilt auch für andere direkten Manipulationen der Modul- und Dialogtexte. So funktioniert auch die Applikation DokuGen (Erzeugung der automatischen Dokumentation) nicht, wenn die Bibliothek verschlüsselt ist.

10 Praktische Anwendung Datenbank-Frontend

Ein letzter Punkt soll hier noch kurz erwähnt werden: Datenbank-Frontend.

10.1 Grundsätzliche Gedanken DB-Frontend

Wird mit Basic ein Frontend für Datenbanken realisiert, so lassen sich die folgenden Punkte aus der Erfahrung zusammenfassen:

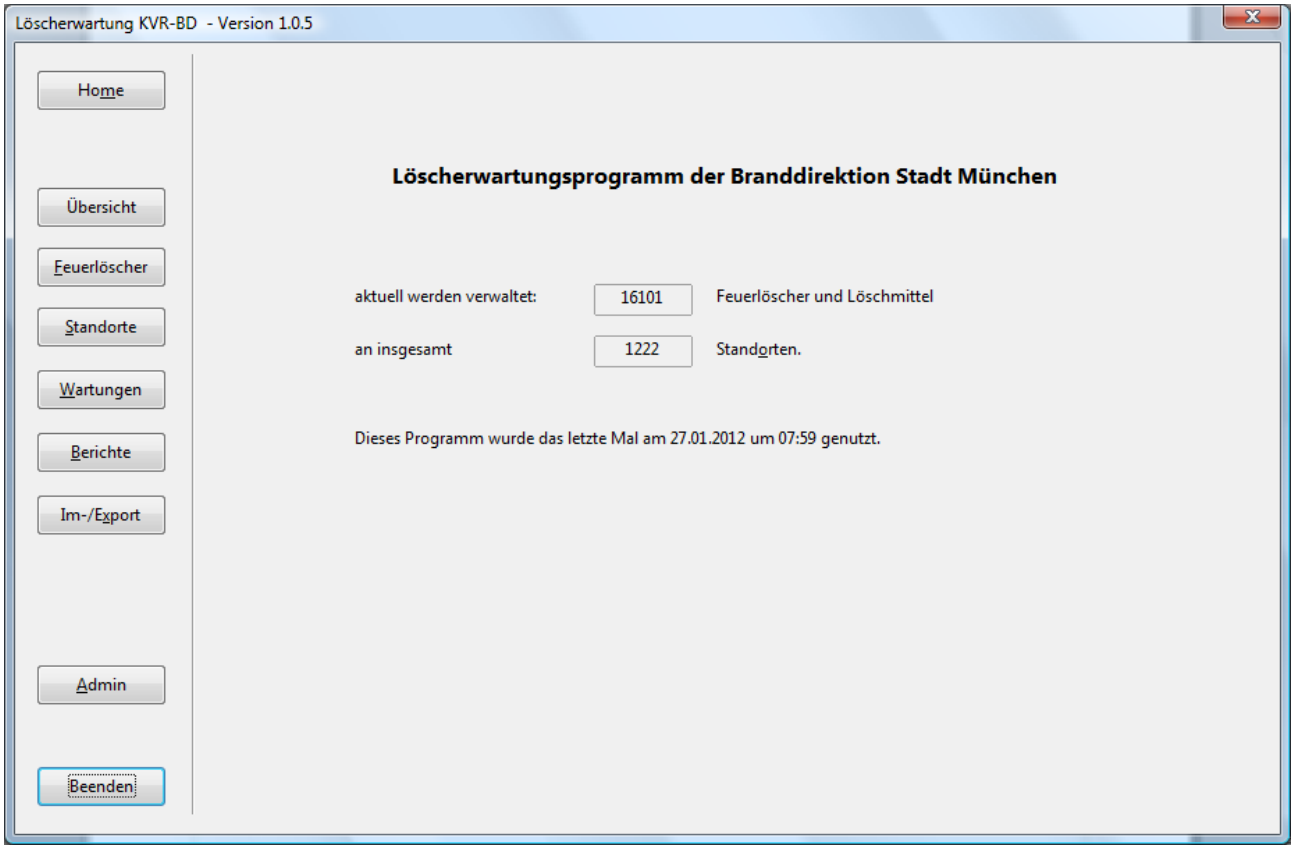
- Nutzen Sie keine Formulare, außer für sehr einfach strukturierte Anwendungen. Zwar ist der Programmieraufwand für Formulare deutlich geringer und wichtige Steuerelemente wie „nächster Datensatz“, „vorheriger Datensatz“ und so weiter sind bereits fertig integriert, doch ist die Flexibilität deutlich eingeschränkt – und passt selten. Intern wird das Formular durch ein Resultset repräsentiert, alle Änderungen müssen nun am Resultset vorgenommen werden – nicht in der DB selbst. Und das bedeutet nun erheblichen zusätzlichen Programmieraufwand für Zusatzfunktionen.
- Das Frontend sollte einen umfassenden Rahmen besitzen – also quasi den Hauptdialog, der dann auch die komplette Steuerung (Menü-Befehle) beinhaltet. Dies kann als eigenständiges Fenster erfolgen oder mit Hilfe des Dialogeditors.
- Datenbank-Zugriffe sind immer SQL-Befehle, diese variieren aber von DB-System zu DB-System. Sinnvollerweise werden diese also alle zusammengefasst in einem eigenen Modul – und könnten dann dort bei Bedarf (Änderung des DB-Systems) geändert werden.
- Datenbank-Verbindungen sollten immer nur kurzfristig aufgebaut werden, und nach Durchführung des SQL-Befehls wieder geschlossen werden.

- Dialogen fehlt das in Formularen verfügbare „Tabellen-Grid-Kontrollelement“ – was aber gerade bei Listendarstellungen stark vermisst wird. Ab der Version 3.3 ist es theoretisch vorhanden, die Einbindung ist aber noch „trickreich“ und nicht über den Dialog-Editor möglich. Tabellarische Listendarstellungen müssen also simuliert werden (siehe hierzu auch Kapitel 6.8.4), wobei dies durchaus eine akzeptable Lösung darstellt.
- Datenbank-Funktionen können gut gekapselt und wiederverwendet werden. Das Code-Repository beinhaltet alle wichtigen Funktionen (DB-Zugriffe, Abfragen, Insert und Delete, einfache Steuerungsbefehle etc.).
- Alle DB-Applikationen arbeiten intern mit den Primärschlüssel-Werten – also den Tabellen-ID-Werten. Dadurch werden Datensätze identifiziert und angesprochen. Primärschlüssel sollten somit „Autowerte“ sein, ohne die Möglichkeit des Benutzereingriffs. Wird eine benutzerdefinierte Zählung dennoch benötigt bzw. gewünscht, sollte über eine zusätzliche Spalte nachgedacht werden – die dann nicht Primärschlüssel ist.
- Teilweise ist es möglich – und manchmal auch nötig – Berechnungen und Zusammenstellungen von Listen aus vielen unterschiedlichen Tabellen über DB-Views zu realisieren und somit Rechenleistung an die DB-Engine abzugeben.
- Bei DB-Applikationen muss man sich rechtzeitig Gedanken machen, wie eine Ausgabe aussehen soll (außer Bildschirmdarstellung). Berichte in Calc oder Writer wären denkbar. In jedem Fall aber ist die Programmierung der (Druck-)Ausgabe immer aufwendig. Im Repository gibt es einen universellen Ausgabe-Code – der kann individuell angepasst werden.

10.2 Beispiel einer Realisierung

Das folgende Beispiel zeigt exemplarisch die Realisierung eines Frontends zur „Feuerlöscherwartung“. Es handelt sich dabei um das Makro „MAK_133“ – Details und Dokumentation können dort nachgelesen werden.

Mit dem Start der Applikation wird der Hauptdialog initialisiert:



Die Startseite beinhaltet lediglich Informationen (hier statistische Daten), die Navigation – entsprechend den Aufgaben – befindet sich auf der linken Seite.

Realisiert ist der Dialog über die Step-Eigenschaft – jeder Aufgabe ist ein eigener Step des Dialoges zugeteilt. Die linke Navigationsleiste ist immer gleich und sichtbar.

Die Buttons besitzen einfache Namen (so etwas wie cmd_1 bis cmd_n) und sind alle mit der selben Funktion verbunden (Ereignis: Beim Auslösen). Die Aufgabe ist nun zunächst zu ermitteln, welcher Button gedrückt wurde (ist über die Fokus-Eigenschaft ermittelbar), und dann entsprechend den passenden Step zu aktivieren.

```
'/** MAK133_dlgButton
'*************************************************************************
'* @kurztext schaltet die entsprechende Step-Eigenschaft des Dialoges ein
'* Diese Funktion schaltet die entsprechende Step-Eigenschaft des Dialoges
'* entsprechend der Aufgabe ein.
'* ist verbunden mit allen Funktionsbuttons der Navigationsleiste (außer abbrechen)
'*
'*************************************************************************
'*/
Sub MAK133_dlgButton
  dim oDBVerb as variant, sFormName as string
  dim oForm as variant
  dim i%, a(), a2(), sTxt as string

  REM aktiven Button suchen
```

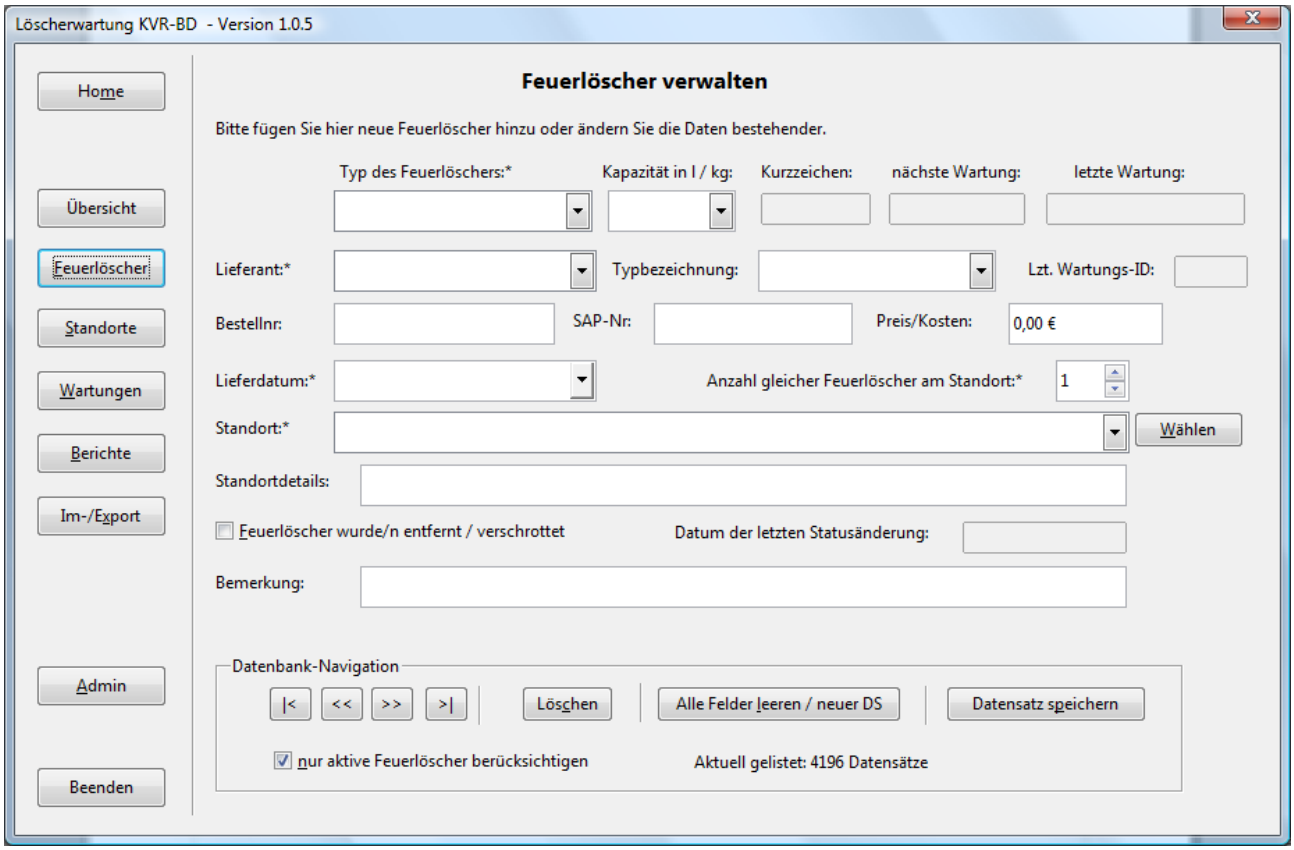
```
  for i = 1 to 8
    if oDlg.getControl("cmd_" & i ).hasFocus() then exit for
  next i
  REM Prüfen, ob Änderungen vorhanden
  if bChangeflag then
    'if MAK133_Helper1.MAK133_ChangeAbbruch then exit sub
  end if

  Select case i
    case 1   'Start
      REM Listen aktualisieren
      oDlg.getControl("lbl_1fl").text = clng(db_tools.getSingleValue(MAK133_SQL1.Step1_SQL001))
      oDlg.getControl("lbl_1ort").text =
clng(db_tools.getSingleValue(MAK133_SQL1.Basis_SQL002("DISTINCT ORT", MAK133_TabFL)))
      oDlg.model.step = i
    case 2   'Wartung
     MAK133_Step2.MAK133_Step2_Start
    case 3   'Feuerlöscher verwalten
      Mak133_Step3.Mak133_Step3_Start
      oDlg.model.step = i
    case 4 'Standorte
      Mak133_Step4.Mak133_Step4_Start
      oDlg.model.step = i
    case 5  'Administration
      Mak133_Step7.MAK133_Step5_Start
      oDlg.model.step = i
    case 6    'Preise
      Mak133_Step6.Mak133_Step6_Start
      oDlg.model.step = i
    case 7   'Berichte
      oDlg.getControl("num_7id").value = 0  'einziges Eingabefeld zurücksetzen
      oDlg.getControl("num_7id").text = ""
      oDlg.model.step = i
    case 8   'Import/Export
      msgbox ("Diese Funktion ist noch nicht implementiert!", 64, "Leider noch nicht
fertig...")
    case else
      oDlg.model.step = 1

  end select

end sub
```

Erkennbar ist hier noch etwas: Jeder Step (Aufgabe) hat meist zunächst eine Initialisierungssequenz – dabei werden zum Beispiel die Listboxen mit Daten gefüllt, Eintragungen vorgenommen etc. Diese Funktionen werden zuerst aufgerufen, erst dann wird auf den Step umgeschaltet.

Die jeweilige Aufgabe zeigt dann alle nötigen Felder an – teilweise nur Information, teilweise offen für Eingaben oder Änderungen. Auch benötigte Navigationselemente werden hier angezeigt (für jeden Step, auch wenn die verknüpften Funktionen wieder identisch sind – für gleiche Funktionstasten auf unterschiedlichen Steps – und die Unterscheidung wieder über die Fokuseigenschaft läuft).

Gerade bei mehrstufigen Dialogen ist es daher sinnvoll, den Namen der Kontrollelemente nicht nur die entsprechend den Makrorichtlinien geforderte Vorsilbe mitzugeben, sondern zusätzlich eine Nummer entsprechend ihrer Step-Eigenschaft (z.B. „cmd_3next“ – Button „Next“ auf Step 3).

Die folgende Abbildung zeigt nun eine „Aufgabe“:



Typischerweise werden Combo-Boxen verwendet, deren Daten mit den bisherigen Einträgen gefüllt sind. Das bietet dem/der Benutzer/in einmal die Möglichkeit, einen schon bestehenden Eintrag auszuwählen, aber auch einen neuen einzugeben. Die Zuweisung der Liste sollte per Code erfolgen – nicht durch direkte Bindung des Kontrollelementes an eine Datenbank-Abfrage. Dadurch gewinnt der/die Programmierer/in mehr Flexibilität und kann auf verbundene Eingaben besser reagieren. Hier beispielsweise: Der Inhalt der Liste „Typbezeichnung“ wird jeweils angepasst nach Eingabe/Wahl des Lieferanten.

In der Regel erfolgt die Listenauswahl über einen SQL-Befehl – zum Beispiel:

```
'-- Liste aller DS für Aufbereitung
function Step3_SQL007(sFID as string, optional sFilter as string)
  dim sSQL as string

  sSQL =  "SELECT DISTINCT BEZ FROM " & MAK133_TabFL & " WHERE IDFA = '" & sFID & "'"
  if NOT isMissing(sFilter) then  sSQL = sSQL & " AND TYP = '" & sFilter & "'"
  sSQL = sSQL & " ORDER BY BEZ ASC"
```

```
   Step3_SQL007 = sSQL
end function
```

Doppelbezeichnungen werden vermieden und die Liste wird sortiert. Filter (z.B. Lieferanten) können optional übergeben werden.

Eine typische Arbeitsfläche für den/die Benutzer/in (Dialogstep) beinhaltet sowohl Eingabe- als auch Informationsfelder.

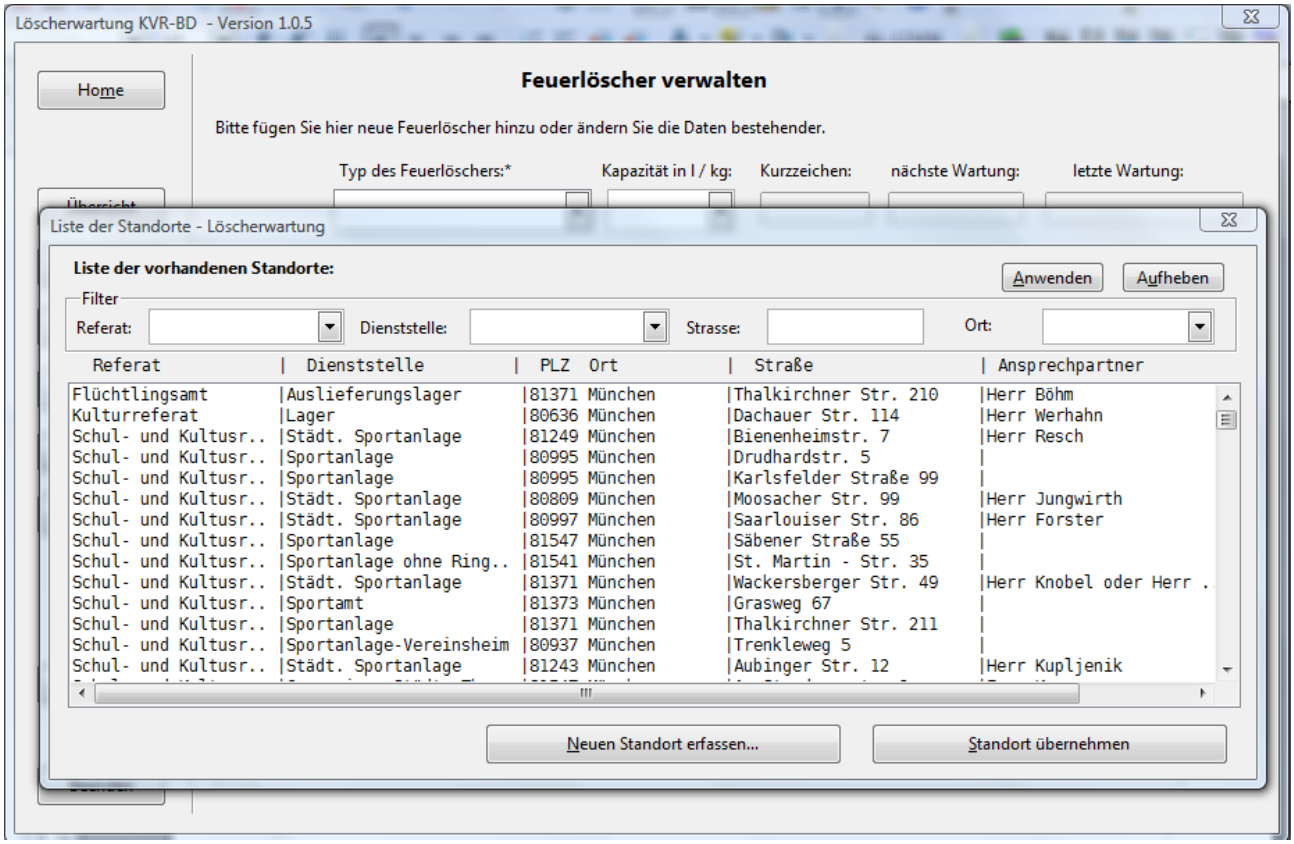
Eingabefelder sollten gekennzeichnet sein, wenn es sich um „Pflicht-Eingabefelder“ handelt (hier gekennzeichnet durch einen *) – und natürlich muss dann auch eine Prüfung stattfinden, bevor der Dialog verlassen oder der Datensatz gespeichert wird!

Informationsfelder müssen für den/die Benutzer/in erkennbar anders gestaltet sein, so dass er/sie gar nicht erst auf die Idee kommt, Änderungen dort vornehmen zu können. Realisierbar ist dies beispielsweise durch Textfelder, deren Eigenschaft „Nur lesen“ auf „Ja“ gestellt wird und die einen andersfarbigen Hintergrund erhalten, oder durch Labelfelder, die per se nicht beschreibbar sind, die dann aber der Optik von Eingabefeldern angepasst werden sollten und ebenfalls eine andere Hintergrundfarbe erhalten sollten.

Unter Linux ist die 2. Möglichkeit immer vorzuziehen, da die aktuell verwendeten Desktop-GUI-Renderer des LiMUX-Builds Hintergrundfarben der Eingabefelder nicht wiedergeben und somit keine optische Unterscheidung zwischen möglicher und nicht möglicher Eingabe gegeben ist. Bei Label-Feldern hingegen ist dies erkennbar.

Nicht immer lassen sich auf einem Step des Dialoges alle gewünschten Informationen sinnvoll unterbringen – dann empfiehlt es sich, mit Zusatzdialogen zu arbeiten.

Im dargestellten Fall kann beispielsweise der Standort aus einer eigenen Liste gewählt werden – dies ist ein eigenständiger Dialog, der vor dem Hauptdialog ausgeführt wird:



Ein solcher Zusatzdialog bietet beispielsweise Filtermöglichkeiten und zusätzliche Informationen.

Wichtig bei allen Dialogen:

Achten Sie darauf, dass ein Dialog nur einmal gestartet werden kann! Der Mehrfachstart (technisch möglich) führt mit Sicherheit zum Absturz, da die Zugriffe auf Kontrollelemente und die Funktionsaufrufe nun nicht mehr eindeutig sind!

Ein Doppelstart ist schnell ausgelöst: Der/Die Benutzer/in klickt auf einen Button – intern beginnt eine Berechnung, eine kurze Verzögerung tritt ein, der/die Benutzer/in ist sich nicht mehr sicher, ob er/sie überhaupt ausgelöst hat und klickt erneut. Eine wirksame Verhinderung sieht dann wie folgt aus:

```
public bMr_list as boolean  'Flag, Auswahldialog gestartet

'*/
Sub MAK133_StartAuswahldlgStandort

  REM Prüffunktion, ob Makro läuft
  if bMR_List then  'Ende, da bereits ein Dialog geöffnet ist
    msgbox ("Ein Standort-Auswahldialog ist bereits geöffnet!" & chr(13) & _
            "Ein zweiter Start ist nicht möglich", 16, "Doppelstart...")
    exit sub
  else
```

```
    bMR_List = true    'Makro läuft Flag setzen
  end if

REM hier folgt der Code des Auswahldialoges

  bMr_list = false   'Makro läuft Flag wieder löschen
end sub
```

Die Flags müssen allerdings global definiert sein – nur dann funktioniert dies korrekt.

Zusatzdialoge können auch „schwebend“ realisiert werden und bleiben dann offen für den/die Benutzer/in – er/sie kann regelmäßig Informationen erhalten oder Auswahlen treffen. Zu schwebenden Dialogen siehe auch Kapitel 6.3, doch auch hier gilt: Ein Doppelstart muss vermieden werden!

Bei Dialogen mit mehreren Steps – wie es in jeder DB-Applikation der Fall ist – denken Sie daran: Der Dialog wird lediglich zum ersten Start hin initialisiert – also so ausgebildet, wie im Erstellprozess vorgesehen. Alle Kontrollelemente sind im Hauptspeicher vorhanden mit all ihren Eigenschaften (auch Inhalt), und zwar unabhängig vom aktuellen Step und der Sichtbarkeit der Elemente.

Hat der/die Benutzer/in z.B. in der Aufgabe „Feuerlöscher“ – hier Step 3 – Daten eingegeben, wechselt nun auf Step 5, so verbleiben die Inhalte in den Feldern des Steps 3 – sie werden weder automatisch gespeichert noch gelöscht. Schaltet der/die Benutzer/in zurück, sieht er/sie wieder die bisherigen Eingaben. Dies kann manchmal gewünscht sein, steht oft aber auch dem erwarteten Ergebnis entgegen (neue Aufgabe – leere Eingabemaske). In diesem Fall muss der/die Programmierer/in dafür sorgen, dass die Felder beim Umschalten geleert werden.

10.2.1 Tab-Reihenfolge

Während bei Formularen die Tab-Reihenfolge (also die Fokus-Reihenfolge der Elemente, wenn der/die Benutzer/in die Tab-Taste nutzt) leicht über eine Liste realisiert werden kann, ist dies bei Dialogen leider nicht möglich. Zwar besitzt jedes einzelne Kontrollelement eine „Tab-Eigenschaft“ mit zwei Ausprägungen: „Tabstop“ und „Aktivierungsreihenfolge“, welche die Tab-Reihenfolge bestimmen, doch gibt es keine „echte“ Übersicht. „Tabstop“ wird beim Anlegen der Kontrollelemente automatisch gesetzt (Eingabefelder und Buttons erhalten ein „Ja“, Labelfelder z.B. ein „Nein“), die Aktivierungsreihenfolge entspricht der Reihenfolge der Erzeugung des Kontrollelementes und wird automatisch fortgesetzt.

Aktivierungsreihenfolgenummern bekommen aber auch die Elemente, die gar nicht eingebunden werden (Labelfelder, grafische Elemente und so weiter) – es ist also schwierig, aus der Nummer direkt eine Aktivierung auszulesen.

Hier bleibt nur die manuelle Neunummerierung aller gewünschten Elemente nach Abschluss aller Layout-Phasen. Doch auch das hat so seine „Tücken“. Mit jeder vergebenen Nummerierung werden sofort alle bisherigen Nummern neu berechnet und automatisch

angepasst (eine Nummer darf nicht zweimal vorkommen!) – und schon kann die gewünschte Reihenfolge erneut nicht stimmen.

Hier ist also ein mehrfacher Zyklus notwendig – gerade bei Applikationen mit mehreren 100 Kontrollelementen eine zeitraubende Aktion.

10.3 Zusammenfassung Datenbank-Frontend-Applikation

Ein paar Worte zum Abschluss:

OOo-Basic als Frontend für Datenbanken ist ein einfacher Schritt, schnell eine praktikable Lösung als Applikation zu erhalten. Die Nutzung von Writer oder Calc als Ausgabemedium erweist sich zusätzlich als Vorteil.

Je umfangreicher aber die Aufgaben werden, um so aufwendiger muss die Programmierung ausfallen – die Performance sinkt dann in gleichem Maß. Besitzt eine Applikation zum Beispiel 30 Steps (Dialogoberflächen und 500 Kontrollelemente), so müssen alle diese intern zunächst gerendert und dann initialisiert werden, bevor der Dialog tatsächlich sichtbar wird.

Auch viele Listboxen in einem Step mit direkter Verbindung zur Datenbank „bremsen“ den Aufbau des Ergebnisses.

Ich denke, für kleinere Projekte ist diese Lösung ideal, ab einer gewissen Größe und Komplexität jedoch ist Basic nicht mehr die beste Wahl.