ELO Java Client Scripting

[Stand: 29.04.2016 | Programmversion: 10.00.000]

Der ELO Java Client bietet umfangreiche Möglichkeiten der Anpassung und Erweiterung per Scripting. Er kann sowohl extern über eine COM-Schnittstelle, als auch intern über JavaScript Schnittstelle angesprochen werden. Damit ist es möglich Funktionen des Clients an spezielle Anforderungen anzupassen. Arbeitsabläufe können automatisiert und erweitert werden. Hier finden Sie eine Übersicht der Funktionalitäten dieser beiden Schnittstellen.

Dieses Dokument liefert die grundlegenden Informationen zur Scripting Schnittstelle, erklärt deren Funktionsweise und beinhaltet Übersichten, welche die internen Bezeichnungen des Scripting den passenden Namen in der Oberfläche zuordnen.

Die exakte Dokumentation der Schnittstellenobjekte liegt als JavaDoc vor und ist hier nicht wiedergegeben. Dies ist eine Sammlung von HTML-Seiten, welche für jede Klasse die verfügbaren Methoden beschreibt. Dort ist auch angegeben, ab welcher Client-Version die jeweiligen Klassen und Methoden zur Verfügung stehen.

Inhalt

1 Technischer Überblick Internes Scripting 4
1.1 Laden der Skripte 4
1.2 Skripte in Bearbeitung 4
1.3 Java Client Scripting Base 4
1.4 Debugger 5
1.5 Variablen 6
1.5.1 Variablen in einer Methode 6
1.5.2 Variablen in einem Skript 7
1.5.3 Skriptübergreifende Variablen 7
1.6 Include 8
1.7 Lokalisierte Texte 10
2 Events 11
2.1 Basisfunktionen 11
2.2 Rückgabewert bei Start Events 12
2.3 Weitere Events 13
2.4 Event mit Parametern 13

3 Schaltflächen in der Multifunktionsleiste (ScriptButtons) 15
3.1 Events 15
3.2 Icon 15
3.3 Name 16
3.4 Position im Client 16
3.4.1 Mehrere ScriptButtons definieren 17
3.5 Zusätzliche Bänder in der Multifunktionsleiste 17
3.6 Zusätzliche Tabs in der Multifunktionsleiste 19
3.6.1 Kontext-Tabs 19
3.7 (De)Aktivierung eines ScriptButtons 20
3.7.1 Immer aktiv 20
3.7.2 Explizite Aktivierung 21
3.7.3 Koppelung an eine Basisfunktion 21
3.7.4 Aktivierung anhand einer Regel 21
4 Scripting Objekte 23
4.1 Client-Objekte 23
4.2 Klassenhierachien 26
4.2.1 Einträge in den Funktionsbereichen 26
4.2.2 Funktionsbereiche/Sichten 26
4.2.3 Verschlagwortung 27
4.2.4 Buttons 27
4.3 Java Objekte 28
4.3.1 Enumeration 28
5 Meldungen und Dialoge 29
5.1 Rückmeldung (FeedbackMessage) 29
5.2 Einfache Meldung (InfoBox) 29
5.3 Warnhinweis (AlertBox) 31
5.4 Ja-/Nein-Fragen (QuestionBox) 31
5.5 Abfrage einer Bezeichnung (InputBox) 32
5.6 Auswahl im Dateisystem (FileDialog) 33
5.7 Dokument oder Ordner im Archiv auswählen (TreeSelectDialog) 35
5.8 Benutzerauswahl (UserSelectionDialog) 36

5.9 Berechtigungen (PermissionsDialog) ... 37
5.10 Auswahl-Dialog (CommandLinkDialog) ... 38
5.11 Komplexe Dialoge (GridDialog) ... 39
5.11.1 Größe der Spalten und Zeilen ... 41
5.11.2 Dialog-Komponenten ... 44
6 Scripting mit den Funktionsbereichen ... 46
7 Hintergrundprozesse ... 48
7.1 Serverprozesse ... 48
7.2 Hintergrundprozesse im Client ... 49
7.2.1 Beispiel: JPEGs archivieren ... 49
7.2.2 Beispiel: Fotos archivieren ... 52
8 ActiveX Objekte ... 56
9 Weitere Beispiele ... 57
9.1 Anzahl der Druckvorgänge zählen ... 57
9.2 Automatisierte Archivablage ... 57
10 Externes Scripting ... 59
10.1 COM Server ... 59
11 Anhang ... 60
11.1 Ribbon-Positionen ... 60
11.2 Basisfunktionen ... 63
12 Index ... 72

1 Technischer Überblick Internes Scripting

Das interne Scripting des ELO Java Client benutzt JavaScript / ECMAScript auf Basis der Mozilla Rhino Engine. Die Skript-Dateien (kurz Skripte) liegen als Dokumente in einem speziellen Register des ELO-Archivs (siehe Scripting Base).

Das Scripting funktioniert über Events, welche der Java Client an definierten Programmzuständen sendet (siehe Events). Innerhalb der Skripte werden Funktionen aufgerufen. Die Zuordnung einer Funktion zu einem Event erfolgt dabei über Namenskonvention: Es wird die zum Event gleichlautende Methode in jedem der für den Client vorhandenen Skripte gestartet. Alle vom Client aufgerufenen Methoden beginnen mit „elo". Dieses Präfix sollte daher nicht für andere Funktionen benutzt werden.

Die Reihenfolge in welcher die Skripte abgearbeitet werden ist nicht eindeutig festgelegt. Es muss daher bei der Skriptentwicklung von einer zufälligen Reihenfolge ausgegangen werden.

1.1 Laden der Skripte

Die Skripte werden beim Start des Clients automatisch vom Server geladen und ausgeführt. Die globalen Variablen können zum Start des Clients angelegt werden und bleiben dann innerhalb des Skripts bis zum Beenden des Clients erhalten. Dies kann dazu benutzt werden um Werte zwischen den einzelnen Funktionen des Skripts auszutauschen.

Möchten Sie während des Betriebs des Clients die Skripte neu laden, zum Beispiel um ein geändertes Skript auszuprobieren, so ist dies über das Tastenkombination <STRG>+<ALT>+R möglich. Es werden anschließend alle Skripte erneut vom Server geladen.

1.2 Skripte in Bearbeitung

Sind Skripte durch den Benutzer ausgecheckt und befinden sich somit bei ihm in Bearbeitung, dann lädt der Client die Datei aus dem lokalen Bereich *In Bearbeitung* statt aus dem Archiv. Hiermit ist es auf einfache Weise möglich, ein Skript zunächst lokal zu entwickeln oder zu verändern und erst später über das einchecken im Scripting Base bei allen Benutzern zu aktivieren.

1.3 Java Client Scripting Base

Alle Skripte sind normale Dokumente im ELO-Archiv. Sie müssen in einem speziellen Ordner liegen. Das Setup von ELOprofessional / ELOenterprise 2011 legt diesen Ordner als „Java Client Scripting Base" an. Dieser Ordner ist definiert durch die GUID **(E10E1000-E100-E100-E100-E10E10E10E11)**. Andere Merkmale (wie Kurzbezeichnung oder Typ) sind unwichtig, sie könnten den Ordner also auch umbenennen und er funktioniert weiterhin. Im Rahmen dieser Dokumentation wird dieser Ordner kurz als **Scripting Base** bezeichnet wird.

Im Scripting Base können beliebig viele Skripte als Dokumente abgelegt werden. Dabei können mit den üblichen Zugriffsrechte in ELO (ACL) die Skripte einzelnen Benutzern oder Gruppen zugeordnet werden. Der Java Client lädt beim Start nur genau die Skripte, auf welche der angemeldete Benutzer Leserechte hat.

Der Name (Kurzbezeichnung) und Dateityp eines Skriptes spielt keine Rolle, er darf aber nicht mit einem der reservierten Wörter „ScriptButton", „lib_" oder „text_" anfangen.

Ab Version 9.00.000 müssen die Dateien der Skripte die Dateiendung .js haben, damit sie als Skripte angesehen, geladen und ausgewertet werden. Die Dateiendung einer Dokumentversion wird in der Versionsgeschichte des Dokuments angezeigt.

Ab Version 9.01.000 werden Skripte in Unterordnern des Java Client Scripting Base ebenfalls eingelesen. Damit ist es möglich umfangreichere Scripting Lösungen, die aus mehreren Dateien bestehen, geordnet im Archiv darzustellen. Es wird nur die erste Ebene an Unterordnern eingelesen, der Inhalt von Ordnern in Ordnern wird weiterhin ignoriert.

1.4 Debugger

Ein Rhino **JavaScript-Debugger** kann per Option oder Tastenkombination <ALT>+<STRG>+D ein- und ausgeschaltet werden. Hiermit ist ein gezieltes Debugging von Internen Skripten des Java Client möglich.

Im folgenden Bild sieht man den Debugger auf Zeile 10 des Script *Dialog Demo* stehen. In Zeile 11 ist ein Breakpoint eingetragen. Breakpoints können durch einfachen Klick auf den Rand mit der Zeilennummer gesetzt und wieder entfernt werden. Zur Anzeige eines der vom Client geladenen Skripte wählen Sie dieses aus dem Menü *Window* aus.

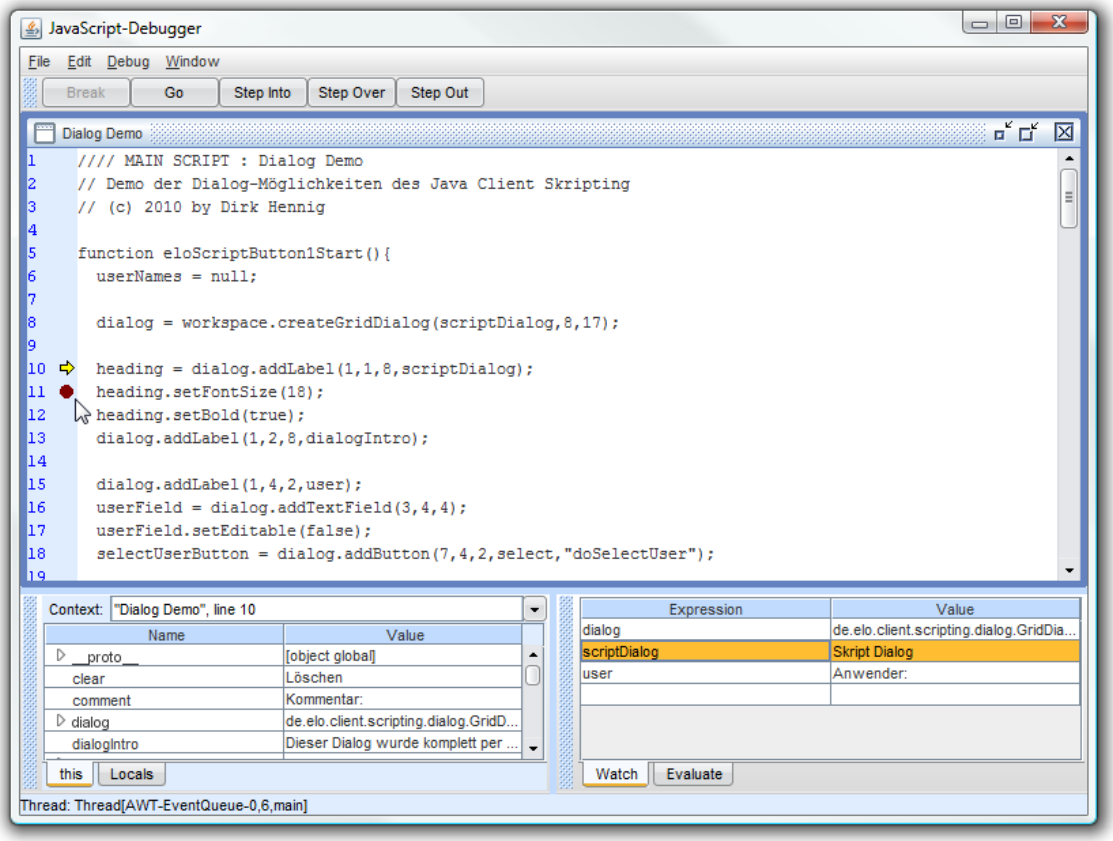


Abb. 1: JavaScript-Debugger; Breakpoint

Beachten Sie: Breakpoints in Dialogen führen zu einem Deadlock, welcher sowohl den Client als auch den Debugger hängen lassen. Es hilft dann nur ein externes Beenden des Clients.

1.5 Variablen

Innerhalb von Skripten können Variablen mit unterschiedlichen Geltungsbereichen definiert werden.

1.5.1 Variablen in einer Methode

Soll eine Variable nur innerhalb einer Methode des Scripts zur Verfügung stehen, so muss diese mit einem vorangehenden **var** bei der ersten Verwendung deklariert werden.

Bitte beachten Sie dabei, dass JavaScript keine Block-Bildung per Klammerebenen berücksichtigt. Im folgenden Beispiel steht die Variable *tag*, welche innerhalb eines *if*-Blocks definiert wird auch außerhalb des Blockes zur Verfügung.

Beispiel:

```
function test1(){
  var monat = "Januar";
  // Ausgabe im Info-Dialog funktioniert
  workspace.showInfoBox( "Monat 1", monat );
}

function test2(){
  // Erzeugt Fehlermeldung:
  // Reference Error: "monat" is not defined.
  workspace.showInfoBox( "Monat 2", monat );
}

function test3(){
  if (true) {
    var tag = 12;
  }
  // Zeigt 12 an !
  workspace.showInfoBox( "Tag", tag );
}
```

1.5.2 Variablen in einem Skript

Alle ohne *var* direkt angegebenen Variablen sind global innerhalb eines Skripts.

Beispiel:

```
local = 0;

function eloScriptButton1Start() {
  local++;
  workspace.showInfoBox( "Info 1", " local counter: " + local );
}

function getScriptButtonPositions() {
  return "1,home,new";
}
```

1.5.3 Skriptübergreifende Variablen

Variablen, welche in allen Skripten innerhalb des Clients verfügbar sein sollen, müssen mit den Namespace globalScope vorangestellt bekommen.

Beispiel:

```
function count() {
  globalScope.counter++;
  workspace.showInfoBox( "Info 1", "counter=" + globalScope.counter );
}
```

1.6 Include

Sie können andere Skript-Dateien in ein Skript importieren. Der Inhalt der importierten Datei wird dabei über dem Inhalt der Skriptdatei eingefügt. Es handelt sich hierbei ein echtes Zusammenkopieren vor der Ausführung des Skripts. Wenn Sie also eine Skript-Datei mit einer Variablen a in zwei andere Skripte importieren, handelt es sich um zwei verschiedene Variablen a in den beiden Skripten. Im Debugger wird in der komplette zusammenkopierte Skript-Text angezeigt.

Beachten Sie: Die Schreibweise hat sich mit der Java Client Version 8.01.000 geändert.

In allen Versionen setzen Sie statt „NAME" die Kurzbezeichnung des Skript-Dokuments im Scripting-Base-Ordner ein. Ab Version 8.01.000 werden Skripte deren Kurzbezeichnung mit lib_ beginnt auch nicht mehr als normales Skript geladen – es lässt sich somit zwischen Bibliotheken und den eigentlichen Skripten unterscheiden.

Bis Version 8.01.000:

```
/**
* @include NAME
*/
```

Ab Version 8.01.000 (einschließlich):

```
//@include NAME
```

Der Name muss hierbei mit lib_ beginnen, damit die Datei eingebunden und nicht als eigenes Skript geladen wird.

Beispiel:

Diese Funktion ist in einem Skript mit der Kurzbezeichnung *lib_InfoDialog* im *Scripting Base* abgelegt:

```
function openInfo(){
  workspace.showInfoBox( "Info", "This is an INFO dialog." );
}
```

Ein zweites Skript *IncludeDemo* im *Scripting Base* könnte dann so aussehen:

```
// Demo-Script for include statement

//@include lib_InfoDialog

function eloScriptButton1Start(){
  openInfo();
}

function getScriptButton1Name(){
  return "Open Info Demo";
}

function getScriptButtonPositions(){
  return "1,home,new";
}
```

Nach einem Kommentar steht oben das Include für die andere Datei. Deren Methode *openInfo* kann somit im Script benutzt werden.

Im Debugger sieht man die zusammenkopierten Textinhalte der beiden Skriptdateien:

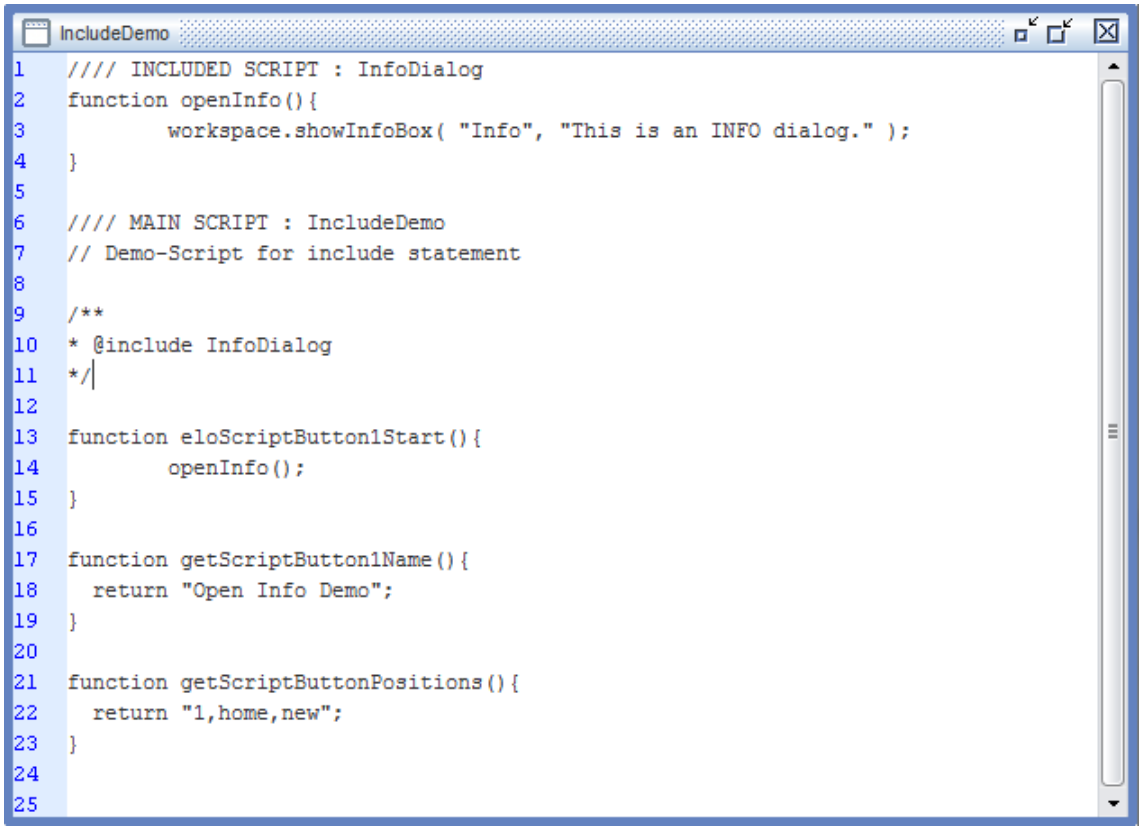


Abb. 2: Zusammenkopierte Skripte im Debugger

1.7 Lokalisierte Texte

Der Java Client unterstützt ab Version 8.03.000 Textdateien mit lokalisierten Texten für das Scripting. Dazu werden Dateien mit den lokalisierten Texten Scripting Base abgelegt. Diese Dateien für die **Default-Sprache** werden folgendermaßen benannt:

text_<Name>

Die Kurzbezeichnung muss mit dem Präfix text_ anfangen, damit das Dokument als Textdatei erkannt wird. Dahinter folgt als <Name> eine frei definierbare Bezeichnung über die aus dem Scripting auf genau diese Datei zugegriffen werden kann.

Für **weitere Sprachen** können zusätzliche Textdateien abgelegt werden, diese bekommen dann die Sprache für welche sie gedacht sind zusätzlich in die Kurzbezeichnung:

text_<Name>_<Sprache>

Die Platzhalter <Sprache> müssen dabei durch einen ISO 639-1 Sprachkürzel ersetzt werden, also zum Bespiel DE für Deutsch, EN für Englisch, FR für Französisch.

Die Textdatei und muss als UTF-8 gespeichert sein und ist intern wie eine Windows INI-Datei bzw. eine Java Properties-Datei aufgebaut. In einer Zeile steht ein Texteintrag, bestehend aus einem Bezeichner für den Eintrag, einem Gleichzeichen und dem Text. Eine sehr einfache Datei könnte zum Beispiel so aussehen:

```
MyDialogTitle=Alterseingabe
HelpText=Bitte geben Sie ihr Alter ein.
```

Das Auslesen eines lokalisierten Textes erfolgt mit Hilfe der Methode *getText(String ressourceName, String textID)* in der Klasse *UtilsAdapter*.

Beispiel:

Es soll der lokalisierte Wert des Begriffs *MyDialogTitle* aus einer der Textdatei mit dem Namen *DemoText* ausgelesen werden.

```
utils.getText( “DemoText“, “MyDialogTitle“ );
```

Wenn der gesuchte Begriff in der Textdatei für die aktuell aktive Sprache des Java Clients nicht existiert oder es keine passende Datei für die aktuelle Sprache gibt, dann versucht der Java Client den Begriff aus der Textdatei mit der Default-Sprache auszulesen. Falls der gesuchte Begriff auch in dort nicht gefunden wird, wird der Begriff von der Methode einfach zurück geliefert, im Beispiel oben also *MyDialogText*.

2 Events

2.1 Basisfunktionen

Die Basisfunktionen des Java Clients sind dessen grundlegende Funktionen, welche sich auf die Menüs, Kontextmenüs und Toolbars konfigurieren lassen bzw. sich in der Multifunktionsleiste des Java Clients befinden. Zu jeder dieser Basisfunktion gibt es zwei Events: Start und End. Dazwischen liegt die Ausführung der Funktion. Entsprechend der Namenkonvention ergeben sich die Events dabei folgendermaßen.

elo<Name>Start

elo<Name>End

Beim Anklicken des Drucker-Icons in der Toolbar wird die Funktion „Print“ ausgeführt. Dabei wird zunächst im Scripting das Event *eloPrintStart* gesendet. Der Java Client sucht in allen geladenen Skripten nach Funktionen mit diesem Namen und startet sie in einer zufälligen Reihenfolge. Danach wird die eigentliche Basisfunktion Drucken des Java Client ausgeführt. Abschließend wird das Event *eloPrintEnd* gesendet und wieder wird nach passenden Funktionen in den Skripten gesucht und diese ausgeführt.

Eine tabellarische Übersicht der Basisfunktionen finden Sie im Anhang. Dabei ist die Funktion mit ihrer Schreibweise in der Multifunktionsleiste angegeben.

Beispiel: Funktionsaufrufe zählen

In diesem Beispiel wird ein Skript erstellt, das mitzählt, wie oft die Funktion „Dokument-Versionen“ aufgerufen wird. Der interne Name dieser Funktion lautet DocVersionsDialog (siehe Dokumentation Java Client Scripting), das entsprechende Event beim Ende der Funktion lautet daher eloDocVersionsDialogEnd. Dieses benutzen wir nun, um per Skript die Aufrufe zu zählen und einen Info-Dialog mit dem aktuellen Zählerstand anzuzeigen.

```
// Zähler initialisieren
cnt = 0;

// wird nach der Funktion "Dokument-Versionen" aufgerufen
function eloDocVersionsDialogEnd(){
  // Zähler um 1 erhöhen
  cnt++;

  // Info-Dialog anzeigen
  Workspace.showInfoBox( "Zähler", cnt );
}
```

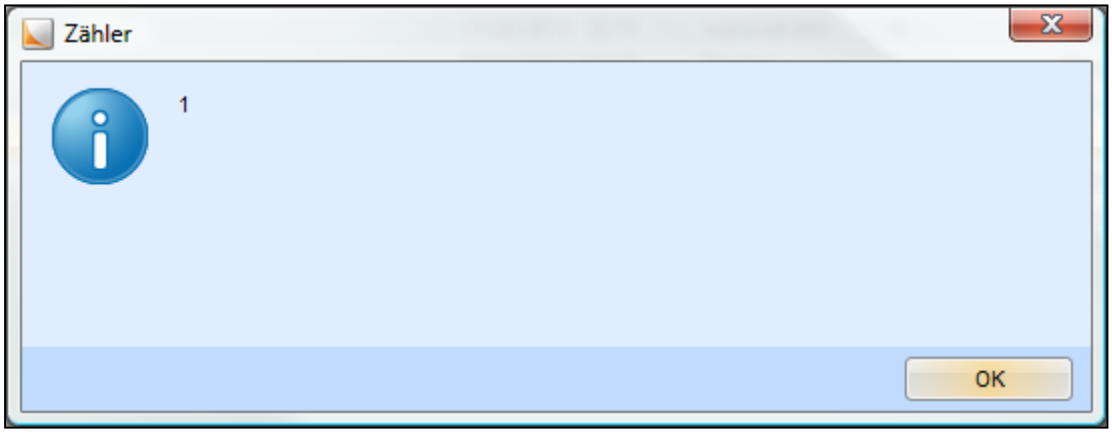


Abb. 3: Info-Box mit Zählerstand

In diesem Beispiel sieht man, wie über die Namenskonvention eine Skript-Funktion an ein Event angehängt wird. Der Zähler zeigt auch, dass jedes Skript im Java Client während der ganzen Laufzeit des Clients „lebt": Es wird beim Start des Clients geladen und sofort werden die Befehle auf der obersten Ebene ausgeführt – in diesem Fall wurde der Zähler *cnt* auf 0 initialisiert. Solche internen Variablen bleiben während der ganzen Laufzeit des Clients erhalten, bis er beendet wird oder die Skripte per STRG+ALT+R neu geladen werden.

2.2 Rückgabewert bei Start Events

Bei diesem Ablauf haben die Start-Funktionen (das ist jede Funktion, deren Name auf "Start" endet) eine besondere Eigenschaft: Liefert eine Start-Funktion eine negative Zahl als Rückgabewert, so wird die Ausführung der Funktion und des Scripting ausgelassen, bis wieder eine Start-Funktion aufgerufen wird. Auf diese Weise kann also eine eigentlich gestartete Aktion des Java Client durch das Scripting abgebrochen werden.

Beispiel:

Dieses Skript zeigt beim Starten der Druckfunktion einen Dialog, welcher nachfragt, ob wirklich gedruckt werden soll. Der Nachfrage-Dialog wird über *showQuestionBox* des Objekts *workspace* realisiert. Diese Funktion zeigt dem Benutzer eine Ja/Nein-Frage an. Welchen Button der Benutzer im Dialog geklickt hat, kann über den Rückgabewert ermittelt werden: Er ist ein logisches *true* bei einem Klick auf *Ja*, ein logisches *false* bei einem Klick auf *Nein*. Bei einem Mausklick auf *Ja* funktioniert die Druckfunktion wie gewohnt, bei einem Klickt auf *Nein* wird das Drucken abgebrochen mit dem Rückgabewert *-1* abgebrochen.

```
function eloPrintStart() {
  var yesPressed = workspace.showQuestionBox( "Frage",
    "Möchten sie das Dokument wirklich drucken?" ) ) {

  // Funktion abbrechen, wenn nicht OK geklickt wurde.
  if (!yesPressed) {
    return -1;
  }
}
```

Bei einem Start-Event können Sie nicht nur eine Nachfrage anzeigen, sondern natürlich auch kompliziertere Logik einbauen. Wird am Ende immer eine -1 zurückgegeben, sieht der Benutzer nicht mehr die originale Funktion, sondern nur noch das, was das Scripting darstellt. Auf diese Weise können Sie, wenn gewünscht, Standardfunktionen des Java Client durch eigene Funktionalitäten ersetzen. Beachten Sie dabei aber, dass die Aktivierung und Deaktivierung der Funktion nach wie vor dem Originalverhalten entspricht, zum Beispiel ist die Funktion *Dokument drucken* nicht aktiv, wenn ein Ordner ausgewählt ist.

2.3 Weitere Events

Zusätzliche zu Basisfunktionen und Script-Schaltflächen sind weitere Events an besonderen Stellen des Clients definiert. Auch hier ist das Verhalten der Startfunktion auf negative Rückgabewerte analog zu den Client-Funktionen, wenn ein Start-Event definiert ist (der Name endet auf „Start“). Diese Events sind in der JavaDoc in Enum SimpleScriptEvent in der Klasse ScriptEvents beschrieben. In der Klasse finden sich als Interface definiert auch komplexere Events, welche mit Parametern die notwendigen Objekte übergeben.

2.4 Event mit Parametern

Beispiel: eloInsertDocument

Im folgenden Beispiel soll bei dem Event *eloInsertDocumentStart* ein Dialog angezeigt werden, welcher dem Benutzer die Möglichkeit gibt, die Datei nicht an der gerade ausgewählten Archivposition, sondern in einem per Skript vordefinierten Standard-Ordner im Archiv abzulegen. Der Pfad zu dem Standardordner muss dafür im Archiv bereits vorhanden sein – ein automatisches Anlegen des Pfades wäre auch möglich, ist in diesem Beispiel aber nicht enthalten.

```
// Standardort für neue Dokumente
var NEW_DOC_FOLDER = "¶Neue Dokumente";

// Verschlagwortungsmaske für neue Dokumente
var NEW_DOC_MASK = "Freie Eingabe";

function eloInsertDocumentStart( mode, file, targetId ) {
  var ok = workspace.showQuestionBox( "An Standardort ablegen?",
    "<html><body><h3>An Standardort ablegen?</h3>"
    + "Das Dokument wird dann am Standardort für neue Dokumente, statt an
der aktuell ausgewählten Archivposition abgelegt.</body></html>" );
  if (ok){
    var newDocFolder = archive.getElementByArcpath( NEW_DOC_FOLDER );
    var sord = newDocFolder.prepareDocument( NEW_DOC_MASK );
    newDocFolder.addDocument( sord, file.getPath() );
    workspace.setFeedbackMessage( "Dokument an Standardort archiviert");
    return -1;
  }
}
```

Abb. 4: Frage-Box

Zuerst wird dem Benutzer die Frage per *workspace.showQuerstionBox* gestellt. Wenn er auf *Ja* klickt, liest das Skript den Zielordner in die Variable *newDocFolder* ein. Für das neue Dokument wird eine Verschlagwortung mit der vorbestimmten Maske vorbereitet, hierzu dient die Methode *prepareDocument*. Dann wird das Dokument per *addDocument* des Zielordners archiviert. Anschließend wird dem Benutzer eine Mitteilung angezeigt und als letztes der normale Ablagevorgang per *return -1* unterbunden.

Abhängig von der Auswahl befindet sich das Dokument danach in *Neue Dokumente* (bei *Ja*) oder (bei *Nein*) an dem Ziel des Drop bzw. in dem Ordner, aus dem der Benutzer *Datei einfügen* aufgerufen hat.

3 Schaltflächen in der Multifunktionsleiste (ScriptButtons)

Neben den Basisfunktionen des Java Clients gibt es die Möglichkeit Schaltflächen ohne eigene Funktion der Multifunktionsleiste (Ribbon) hinzuzufügen. Dies sind die sogenannten Script-Schaltflächen. Obwohl der Name *ScriptButton* sich auf die Multifunktionsleiste bezieht, können diese – genau wie die Basisfunktionen - auch als Einträge in der Schnellstartleiste und den Kontextmenüs verwendet werden.

Jede Script-Schaltfläche ist mir einer Nummer gekennzeichnet. Es stehen die Nummern 0 bis 999, also tausend unterscheidbare Script--Schaltflächen zur Verfügung. Die in der Menü- und Toolbarkonfiguration angezeigt Menge kann zur besseren Übersicht per Option begrenzt werden. Voreingestellt ist der Wert 10.

3.1 Events

Die Namen der von den Skript-Schaltflächen erzeugten Events folgt der Namenskonvention der Basisfunktionen und besteht ebenfalls aus einem Start und End. Auch hier kann ein negativer Rückgabewert eines Start-Skriptes die Ausführung der End-Skripte unterbinden.

eloScriptButton<nr>Start

eloScriptButton<nr>End

3.2 Icon

Die Icons, mit welchen die Script-Schaltflächen angezeigt werden, sind genau wie die Skripte Dokumente im Scripting Base. Auch hier wird wieder eine Namenskonvention benutzt: Alle Dokumente im Scripting Base deren Name oder Kurzbezeichnung mit ScriptButton anfängt, werden als solche Icons behandelt. Diese werden beim Start des Java Client zusammen mit den Skripten geladen und benutzt. Für die Script-Schaltfläche 1 müsste also ein passendes Icon mit dem Namen *ScriptButton1* vorhanden sein. Als Format sollte eine PNG-Grafik, 32 x 32 Pixel groß mit transparentem Hintergrund verwendet werden. Ein Satz an Templates und Beispielen ist bei ELO verfügbar.

Sollte zu einem im Java Client konfigurierten Skript-Schaltfläche kein Icon im Scripting Base vorhanden sein, so wird ein einheitliches Standard-Icon benutzt.

Es ist möglich eine zusätzliche, kleine 16 x 16 Pixel Version des Icons zu hinterlegen, welches dann z.B. in der Schnellstartleiste verwendet wird. Die Kurzbezeichnung des kleinen Icons entspricht dem Namen des großen mit einem angehängten „_16“. Für den ersten ScriptButton also bspw. „ScriptButton1_16“.

Ab Version 9.02 können alternativ zu PNG-Grafiken auch ICO-Dateien verwendet werden. Durch mehrere Ebenen unterschiedlicher Größen können die Icons dann besser für höhere DPI Einstellungen skaliert werden. Auch hier wird mit „_16“ zwischen kleinen und großen Icons unterschieden.

3.3 Name

Die Script-Schaltflächen können nicht nur in der Multifunktionsleiste liegen, sondern auch per Konfiguration in die Schnellstartleiste oder ein Kontextmenü gelegt werden. Da hier jeweils neben dem Icon immer auch ein Text angezeigt wird, ist es wichtig, der Schaltfläche auch einen Namen zu geben.

Der Client ermittelt den Namen für eine Skript-Schaltfläche durch Aufruf einer speziellen Methode im Skript.

getScriptButton<nr>Name

Ist eine solche Methode vorhanden, wird deren Rückgabewert als Name verwendet. Wenn zu einer Schaltfläche in mehreren Skripten Namen vorhanden sind, so werden diese mit Kommas getrennt hintereinander aufgelistet.

3.4 Position im Client

Die Position in der Multifunktionsleiste (Ribbon) wird analog zum Namen direkt im Script angegeben werden. Die hierfür im Script zu implementierende Methode trägt den Namen **getScriptButtonPositions.**

Ihr Rückgabewert muss folgendes Format haben:

<ScriptButtonNummer>,<Ribbon-Tab>,<Ribbon-Band>

Ribbon-Tab und Ribbon-Band sind dabei fest definierte Bezeichner. Unter einem *Ribbon-Tab* versteht man eine der Leisten, welche über die Tab-Reiter am oberen Rand des Ribbon erreichbar sind. Ein *Ribbon-Band* ist eine Gruppe von Funktionen innerhalb eines Ribbon-Tabs. Eine Auflistung aller im Client vorhandenen Tabs und Bänder findet sich im Anhang.

Beispiel:

```
// Funktion für den SkriptButton1
function eloScriptButton1Start(){
  workspace.showInfoBox("Info ", "Meine Funktion 1");
}

// Definition der Position in der Multifunktionsleiste
// ButtonNummer,Tab,Gruppe
function getScriptButtonPositions(){
  return "1,home,new";
}

// Name der Funktion
function getScriptButton1Name(){
  return "Meine Funktion 1";
}
```

Nach dem Programmstart finden Sie den SkriptButton1 in der Multifunktionsleiste. Die Skript-Schaltflächeist dabei immer so konfiguriert, dass er mit großem Icon und Text sichtbar ist. Er wird voreingestellt am hinteren Ende der Gruppe positioniert.

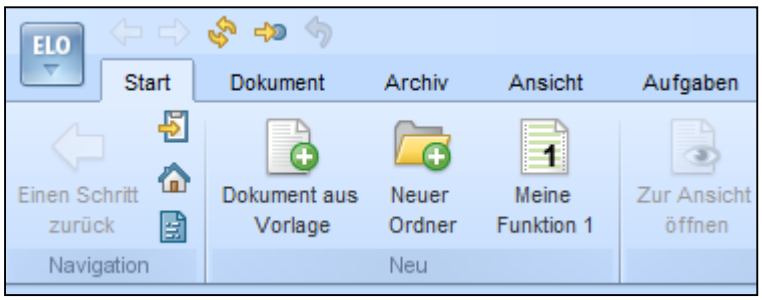


Abb. 5: Neue Skript-Schaltfläche

3.4.1 Mehrere ScriptButtons definieren

Sollen mehrere Skript-Schaltflächen eingeblendet werden, so müssen diese getrennt durch ein Semikolon aufgezählt werden.

Beispiel:

Um die Skript-Schaltfläche 2 im Tab *Start* in das Band *Ansicht* und der Schaltfläche 10 im Tab *Dokument* in das Band *Ausgabe* einzublenden, muss die Methode folgende Information zurückgeben:

```
function getScriptButtonPositions() {
   return “2,home,view;10,document,print”;
}
```

3.5 Zusätzliche Bänder in der Multifunktionsleiste

Wenn die vorhandenen Bänder (auch „Gruppen“ genannte) thematisch nicht zu den neuen, per Scripting bereitgestellten Funktionen passen, dann ist es sinnvoll, zusätzliche Bänder einzufügen. Hierzu ist wiederum eine Methode mit dem Namen **getExtraBands** vorgesehen. Sie muss die zusätzlichen Bändern zurückgeben und benutzt dabei eine ähnliche Syntax wie die Zuordnung der Skript-Schaltflächen.

Der Rückgabewert muss folgendes Format haben:

<Ribbon-Tab>,<Position>,<Name des neuen Bands>

Die Position ist eine positive Zahl, welche angibt, wo im Ribbon-Tab das neue Band eingefügt werden soll. Die Standard-Bänder des Java Client belegen die 10er-Positionen: Im Tab *Start* findet sich das Band *Navigation* also an Position 10, das Band *Neu* an Position 20 und so weiter, bis zum Band *Löschen* an Position 80. Alle Positionen zwischen den bereits vorhandenen Bändern können für zusätzliche Bänder benutzt werden. Mit den Position 0 bis 9 können als Bänder ganz am linken Rand eingefügt werden. Die Positionen 81 bis 89 wären im Tab *Start* also am rechten Rand. Zwischen *Neu* und *Ansicht* liegen die Positionen 21 bis 29.

Auch bei den Bändern können mehrere Angaben mit einem Semikolon getrennt aufgezählt werden.

Beachten Sie: Leere Bänder werden nicht angezeigt. Sie müssen einem Band also Skript-Schaltflächen hinzufügen, damit dieses sichtbar wird. Dies funktioniert auf die gleiche Weise wie bei den Standard-Bändern.

Beispiel:

Das folgende Skript fügt ein Band *Beispiel* hinter dem Band *Neu* in den Ribbon-Tab *Start* ein. In dieses Band werden dann die Skript-Schaltflächen 5 und 6 eingefügt.

```
// Definition der neuen Gruppe
// Tab,Position,Gruppe
function getExtraBands(){
  return "home,21,Demo";
}

// Definition der Position in der Multifunktionsleiste
// ButtonNummer,Tab,Gruppe
function getScriptButtonPositions(){
  return "1,home,Demo";
}
```

Wenn noch ein passendes Icon hinterlegt wurde, sieht das Ergebnis nach Neustart des Clients dann so aus:

Abb. 6: Neue Gruppe auf der Multifunktionsleiste

3.6 Zusätzliche Tabs in der Multifunktionsleiste

Das Anlegen zusätzliche Leisten funktioniert analog zum Anlegen weiterer Bänder (siehe den vorangehenden Abschnitt) über die Methode **getExtraTabs.**

Der Rückgabewert muss folgendes Format haben:

<Position>,<Name des neuen Tabs>

Die Position ist wieder eine positive Zahl, welche angibt, wo im Ribbon das neue Band eingefügt werden soll. Die Standard-Bänder des Java Client belegen die 10er-Positionen: *Start* liegt also auf Position 10, *Dokument* auf Position 20, *Archiv* auf 30, *Ansicht* auf 40 und *Workflow* auf 50. Die kontextabhängigen Tabs folgen danach: *Postboxtools / Archivieren* auf 60, *Suchtools / Recherchieren* auf 70 und *Zwischenablage* auf 80. Alle Positionen zwischen den bereits vorhandenen Tabs können für zusätzliche Tabs benutzt werden. Mit den Position 1 bis 9 können als Tabs ganz am linken Rand eingefügt werden, mit 91 bis 99 ganz am rechten Rand.

Auch bei den Tabs können mehrere Angaben mit einem Semikolon getrennt aufgezählt werden.

Bitte beachten Sie, dass leere Tabs nicht angezeigt werden. Sie müssen Bänder und Skript-Schaltflächen hinzufügen, damit dieses sichtbar wird.

3.6.1 Kontext-Tabs

Neben den immer sichtbaren Tabs der Multifunktionsleiste gibt es auch spezielle Tabs, welche nur in einem besonderen Kontext sichtbar werden, z.B. das Tab *Archivieren* im Kontext *Postboxtools*, welches im Funktionsbereich *Postbox* sichtbar wird. Ein weiteres Beispiel ist der Tab *Kopieren / Einfügen* im Kontext *Zwischenablage*, welcher nach dem Aufruf der Funktion *Kopieren* sichtbar wird.

Um ein per Scripting definierten Tab nicht immer, sondern nur in einem Kontext anzuzeigen, wird der passende Kontext zusammen mit dem Tab definiert, indem er als dritter Parameter angegeben wird:

<Position>,<Name des neuen Tabs>,<Name des Kontexts>

Im folgenden Beispiel wird ein Kontext *Kalendertools* mit dem Tab *Verwalten* definiert:

```
function getExtraTabs(){
  return "62,Verwalten,Kalendertools";
}
```

Der Kontext kann dann einem zusätzlichen Funktionsbereich (ExtraView) zugewiesen werden. Im Folgenden ist dies ein neuer per Scripting erstellter Funktionsbereich *Kalender*.

```
view = workspace.addView( "Kalender", false, myComponent, null );
view.setContextTaskGroup( "Kalendertools" );
```

Beispiel:

Das folgende Skript fügt einen neuen Tab *Workshop* hinter dem Tab *Archiv* in die Multifunktionsleiste ein. Diesem wird die Gruppe *Demo* mit dem *SkriptButton 1* hinzugefügt.

```
// Definition des neuen Tab
// Position,Tab
function getExtraTabs(){
  return "31,Workshop";
}

// Definition der Position in der Multifunktionsleiste
// ButtonNummer,Tab,Gruppe
function getScriptButtonPositions(){
  return "1,Workshop,Demo";
}

// Definition der neuen Gruppe
// Tab,Position,Gruppe
function getExtraBands(){
  return "Workshop,21,Demo";
}
```

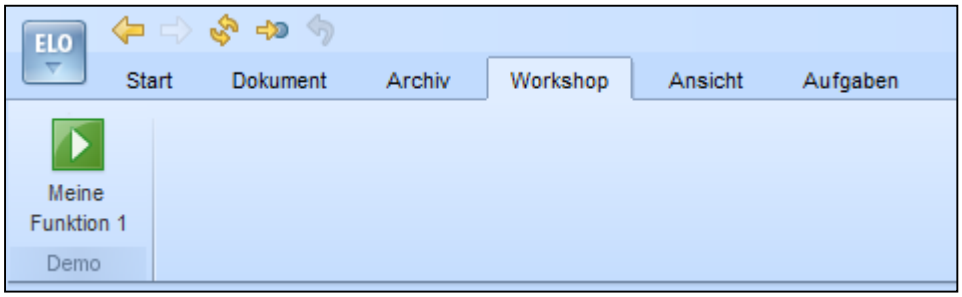


Abb. 7: Neuer Registertab

3.7 (De)Aktivierung eines ScriptButtons

Für die Aktivierung eines ScriptButtons gibt es verschiedene Möglichkeiten:

3.7.1 Immer aktiv

Ohne weitere Einstellungen sind die ScriptButtons standardmäßig immer aktiviert. Es muss dann innerhalb der Funktionalität des Skripts überprüft werden, ob die aktuelle Auswahl im Client für die Funktion des Skripts geeignet ist. Wenn dies nicht der Fall ist, sollte eine InfoBox den Benutzer darüber informieren, welche Auswahl benötigt wird.

Solch eine zusätzliche Prüfung der aktuellen Auswahl kann auch mit den folgenden Aktivierungen kombiniert werden um komplizierte Fälle abzudecken.

3.7.2 Explizite Aktivierung

Über den **WorkspaceAdapter** kann mit der Methode **setScriptButtonEnabled** eingestellt werden, ob ein ScriptButton aktiv oder inaktiv sein soll. Diese Möglichkeit der absoluten Festlegung des Zustand der Schaltfläche eignet sich für Bereiche, in denen passende Selektionsevents vorhanden sind, z.B. per Skript erstellt zusätzliche Funktionsbereiche.

Beispiel:

Der ScriptButton 102 soll inaktiv sein:

```
workspace.setScriptButtonEnabled( 102, false );
```

3.7.3 Koppelung an eine Basisfunktion

Für die meisten ScriptButtons in den normalen Funktionsbereichen besteht die einfachste Lösung darin, die Aktivierung an eine vorhandene Basisfunktion zu koppeln. Der ScriptButton wir dann analog zu der Schaltfläche der Basisfunktion abhängig von der aktuellen Auswahl im Client aktiviert und deaktiviert.

Beispiel:

Der ScriptButton 5 soll genauso aktiviert werden, wie die Basisfunktion *Einchecken*:

```
workspace.setScriptButtonEnabled( 5, "CheckIn" );
```

3.7.4 Aktivierung anhand einer Regel

Die Aktivierung eines Scriptbuttons anhand der aktuellen Auswahl im Client kann auch unabhängig von einer Basisfunktion mit einer Regel definiert werden. Dabei wird angegeben, bei welcher Selektion die Funktion aktiv sein soll. Zur Auswahl stehen: Dokument, Ordner, Wiedervorlage, Workflow, Aktivität und eine Auswahl ob eine Mehrfachselektion möglich sein soll. Die Methode dafür befindet sich im WorkspaceAdapter:

```
setScriptButtonEnabled( int nr, boolean document, boolean folder, boolean
reminder, boolean workflow, boolean activity, boolean multiselect )
```

Bitte beachten Sie, dass es sich bei der Mehrfachauswahl auch um eine Mischung der einzelnen erlaubten Auswahlarten handeln kann.

Beispiel:

Der ScriptButton 201 soll nur bei der Auswahl genau eines Dokuments aktiv sein, ScriptButton 202 für alle Arten von Aufgaben, auch bei Mehrfachselektion.

```
workspace.setScriptButtonEnabled( 201, true, false, false, false, false,
false );
workspace.setScriptButtonEnabled( 202, false, false, true, true, true, true
);
```

4 Scripting Objekte

4.1 Client-Objekte

Die Steuerung der Client-Oberflächen durch das Scripting erfolgt über definierte Objekte, welche in der Laufzeitumgebung der Skripte zur Verfügung gestellt werden. Neben der allgemeinen Programmoberfläche (workspace) ist eine Einteilung anhand der Funktionsbereiche (checkout, intray, search...) vorhanden. Wichtige zusätzliche Objekte, wie zum Beispiel der Verschlagwortungsdialog werden ebenfalls direkt angeboten (indexDialog). Im Folgenden sind die vorhandenen Objekte kurz beschrieben.

Die komplette Dokumentation der Klassen und ihrer Methoden liegt als JavaDoc vor.

4.1.1.1 workspace

Das Haupt-Fenster des Java Client ist der Workspace. Es ist gleichzeitig die Basis für alle seine Komponenten und stellt grundlegende Funktionen bereit.

Klasse: WorkspaceAdapter

4.1.1.2 archiveViews

Dieses Objekt ermöglicht den Zugriff auf die verschiedenen im Java Client vorhandenen Archivansichten in Form von Objekten der Klasse ArchiveViewAdapter. In diesen können dann ArchivElemente (Dokumente, Ordner) selektiert und auf diese zugegriffen werden.

Klasse: ArchiveViews

4.1.1.3 intray

Dieses Objekt dient dem Zugriff auf den Funktionsbereich „Postbox“.

Klasse: IntrayAdapter

4.1.1.4 checkout

Dieses Objekt dient dem Zugriff auf den Funktionsbereich „In Bearbeitung“.

Klasse: CheckoutAdapter

4.1.1.5 tasks

Dieses Objekt ermöglicht den Zugriff auf die verschiedenen im Java Client vorhandenen Aufgabenansichten in Form von Objekten der Klasse TasksViewAdapter. In diesen können dann ArchivElemente (Dokumente, Ordner) selektiert und auf diese zugegriffen werden.

Klasse: TasksViews

4.1.1.6 clipboard

Dieses Objekt dient dem Zugriff auf den Funktionsbereich „Klemmbrett“.

Klasse: ClipboardAdapter

4.1.1.7 searchViews

Dieses Objekt ermöglicht den Zugriff auf die verschiedenen im Java Client vorhandenen Suchansichten in Form von Objekten der Klasse SearchViewAdapter. Es können auch neue Suchansichten erzeugt werden. In einer Suchansicht kann eine Suche ausgeführt und mit deren Ergebnissen gearbeitet werden.

Klasse: SearchViews

4.1.1.8 archive

Das Archiv-Objekt ermöglicht den direkten Zugriff auf das ELO Archiv. Hiermit kann direkt auf Archiv-Elemente (Dokumente und Strukturelemente) zugegriffen werden, ohne diese in einer Archivansicht sichtbar zu machen.

Klasse: ArchiveAdapter

4.1.1.9 dialogs

Über dieses Objekt können für das Scripting vorbereitete Dialoge des Clients erreicht werden.

Klasse: DialogsAdapter

4.1.1.10 components

Dieses Objekt ermöglicht die Erstellung von speziellen Komponenten, zum Beispiel eines GridPanel.

Klasse: ComponentsAdapter

4.1.1.11 indexDialog

Dieses Objekt dient der Steuerung des Verschlagwortungsdialogs. Es erlaubt das Setzen von Ablagemasken und Verschlagwortungswerten.

Klasse: IndexDialogAdapter

4.1.1.12 preview

Dieses Objekt ermöglicht die Steuerung der Dokumentenvorschau. Diese kann aktiviert, deaktiviert, erneuert oder auf ein bestimmtes Dokument gesetzt werden.

Klasse: PreviewAdapter

4.1.1.13 utils

Eine Sammlung von Hilfsfunktionen.

Klasse: UtilsAdapter

4.1.1.14 clientInfo

In diesem Objekt finden sich verschiedene wichtige Verbindungsinformationen, welche als Parameter für den IX bei Verwendung des ix-Objekts benötigt werden. Bei Verwendung des ixc-Objekts entfällt es.

Klasse: **de.elo.ix.client.ClientInfo**

4.1.1.15 ix

Dieses Objekt ermöglicht den direkten Zugriff auf die Funktionen des ELO Indexservers (IX). Die Funktionen des IX benötigen dann noch zusätzlich das clientInfo-Objekt. Eine einfachere Möglichkeit ist die Verwendung des ixc-Objektes.

Interface: **de.elo.ix.client.IXServicePortIF**

Beispiel:

```
edi = ix.createSord(clientInfo, 0, 1, ixConst.getEDIT_INFO().getMbAll());
```

4.1.1.16 ixc

Das IX-Connection-Objekt vereinfacht die Verwendung der IX-Funktionen im Java Client ab 7.00.004. Es bietet genau den Funktionsumfang wie das ix-Objekt, allerdings kommen alle Funktionsaufrufe ohne das clientInfo-Objekt aus. Dieses ist bei den ix-Aufrufen der erste Parameter und entfällt bei den ixc-Aufrufen.

Beispiel:

```
edi = ixc.createSord(0, 1, ixConst.getEDIT_INFO().getMbAll());
```

4.1.1.17 ixConst

In diesem Objekt finden sich verschiedene Konstanten, welche als Parameter für den IX benötigt werden.

Klasse: **de.elo.ix.client.IXServicePortC**

4.1.1.18 log

Zur Ausgabe von debug- und Fehlerinformationen kann dieses Log4J-Logger-Objekt benutzt werden. Die Informationen werden dann mit in das log des Java Client geschrieben und ermöglichen eine Betrachtung des Skripts im Kontext des Java Clients.

Beispiele:

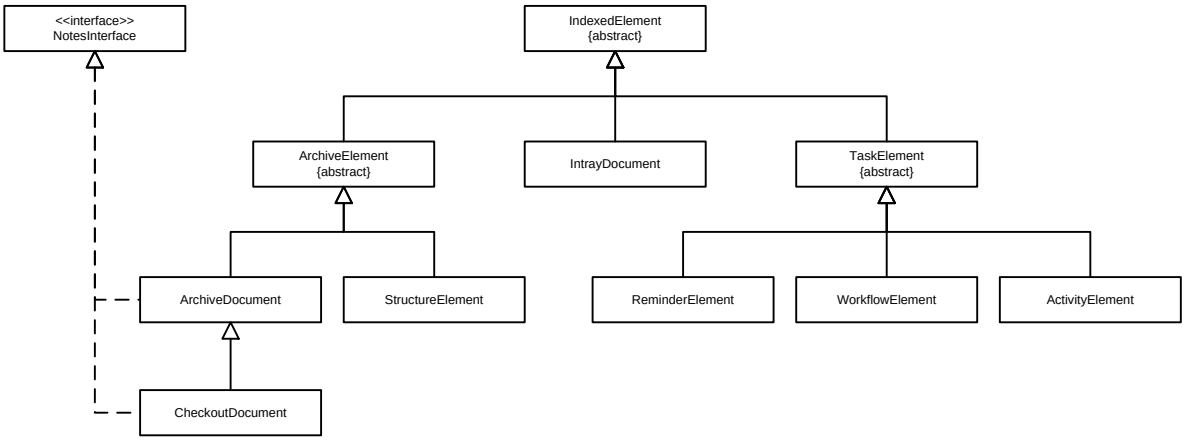
```
log.info( "Funktion gestartet" );
log.debug( "Technische Details: FctNo=123" );
log.warn( "Fehler beim Laden", exception );
```

4.2 Klassenhierachien

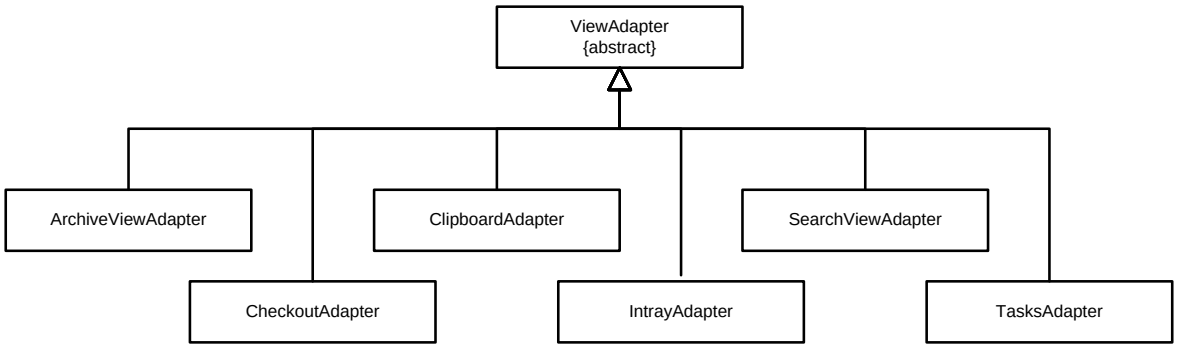
Von den Client-Objekten der Funktionsbereiche werden die Ordner, Dokument etc. als Objekte gekapselt zurückgegeben. Diese implementieren sinnvolle Methoden auf diesen Elementen und ermöglichen somit ein einfaches Arbeiten ohne tiefergehende Kenntnisse der IX-Schnittstelle. Die Objekte sind in objektorientierte Klassenhierarchien eingebunden, welche in den unten stehenden Schaubildern dargestellt sind.

Die komplette Schnittstellenbeschreibung ist als JavaDoc vorhanden.

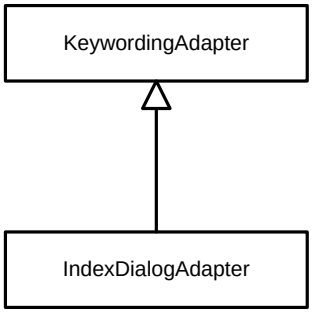
4.2.1 Einträge in den Funktionsbereichen

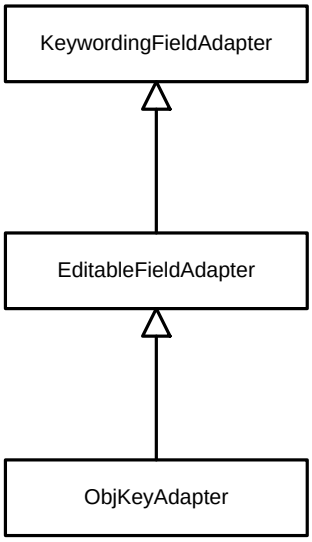


4.2.2 Funktionsbereiche/Sichten

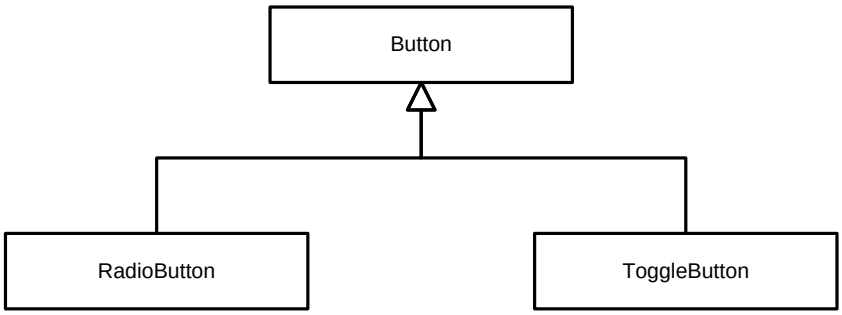


4.2.3 Verschlagwortung





4.2.4 Buttons



4.3 Java Objekte

Die API des Internen Scripting arbeitet mit Java-Objekten, welche über die Rhino-Engine in JavaScript bereitgestellt werden. Alle Klassen die von Java Client für das Interne Scripting bereitgestellt werden sind in der Java Client Scripting JavaDoc beschrieben, die Klassen der Indexserver-Schnittstelle in dessen JavaDoc. Darüber hinaus tauchen auch einige Basis-Klasse der Sprache Java in der Schnittstellen auf. Diese sollen hier kurz beschrieben werden. Genauere Informationen bieten die Oracle Java™ Platform, Standard Edition 6 API Specification JavaDoc und Literatur über Java.

4.3.1 Enumeration

Eine Enumeration ist ein einfacher Aufzählungstyp. Er findet zum Beispiel in den Funktionsbereichen Anwendung: Dort liefert zum Beispiel *getAllSelected()* alle aktuell selektierten Einträge als eine Enumeration zurück.

Um mit einer Enumeration zu arbeiten, benötigt man nur dessen zwei Methoden:

hasMoreElements()

liefert einen boolean zurück: True wenn es weitere Einträge gibt, false wenn nicht.

nextElement()

liefert den nächsten Eintrag, wobei auch der erste Eintrag hiermit geholt wird.

Beispiel:

Eine Schleife über alle selektierten Dokument der Postbox sieht dann so aus:

```
var selection = intray.getAllSelected(); //Enumeration<IntrayDocument>

while( selection.hasMoreElements() ) {
  var document = selection.nextElement(); // IntrayDocument

  // hier kann nur etwas mit dem einzelnen Dokument gemacht werden
}
```

5 Meldungen und Dialoge

Meldungen und Dialoge lassen sich mit dem Internen Scripting leicht erzeugen. Standard-Dialoge für einfache Mitteilungen können direkt verwendet werden. Komplexere Dialoge lassen sich anhand eines Tabellen-Layouts zusammenbauen.

5.1 Rückmeldung (FeedbackMessage)

Die einfachste Form einer Mitteilung ist eine Rückmeldung an den Benutzer in Form einer kleinen Einblendung im Kopfbereich des Clients. Solche Rückmeldungen sind dafür gedacht, dem Benutzer einen Hinweis zu geben, wenn eine Funktion erfolgreich durchgeführt wurde, das Ergebnis für den Benutzer aber nicht sofort sichtbar ist. Ein gutes Beispiel hierfür ist das Starten eines Workflows: Der Benutzer sieht nicht, dass der Workflows gestartet wurde.

Solche Rückmeldungen dürfen nur als kleine Hinweis auf erfolgreiche Aktionen benutzt werden. Für Fehlermeldungen und auch wichtige Hinweise muss immer ein Dialog benutzt werden.

Es kann immer nur eine Rückmeldung angezeigt werden, werden schnell hintereinander mehrere gesetzt, ist für den Benutzer nur die letzte sichtbar. Außerdem ist zu beachten, dass die Abarbeitung der Skripte synchron mit der Oberfläche läuft. Es ist daher nicht möglich innerhalb einer langlaufenden Routine mehrere Rückmeldungen anzuzeigen, die Oberfläche aktualisiert sich nur am Ende und zeigt daher nur die letzte Meldung.

```
workspace.setFeedbackMessage( "Workflow wurde gestartet" );
```

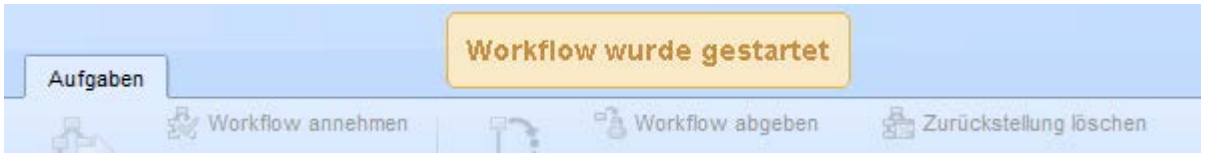


Abb. 8: Rückmeldung im Client

5.2 Einfache Meldung (InfoBox)

Der einfachste Dialog im Java Client ist die InfoBox, ein Dialog mit Titel, Text, einem festen Info-Symbol und einer Schaltfläche „OK“ zum Schließen. Das folgende Beispiel zeigt den Aufruf und die Parameter der InfoBox:

```
workspace.showInfoBox( "Titel", "Text" );
```

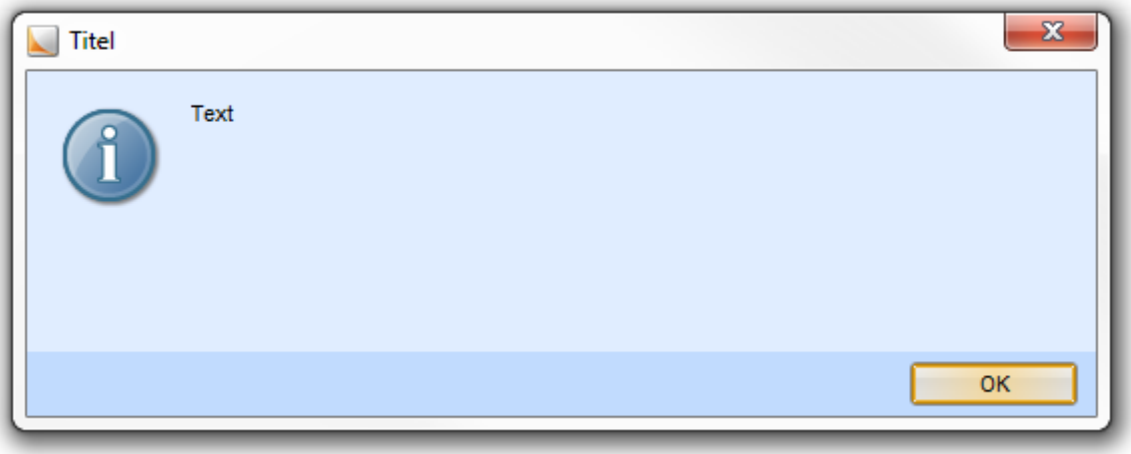


Abb. 9: Schlichte Info-Box

Im Java Client wird in solchen Dialogen die wichtigste Information üblicherweise in größerer, fetter Schrift hervorgehoben. Im Scripting erreichen Sie dies durch die Verwendung von HTML-Text, der Haupttexte sollte als Überschrift 3, also mit dem HTML-Tag **<h3>** formatiert sein. Das folgende Beispiel zeigt einen Meldung, welche nach dem Druckern eines Dokuments erscheinen könnte, wenn der Druckauftrag an eine Zentrale Druckstelle geschickt wurde:

```
workspace.showInfoBox( "Dokument drucken", "<html><h3>Die gedruckte Datei
wird Ihnen per Hauspost zugestellt.</h3></html>" );
```

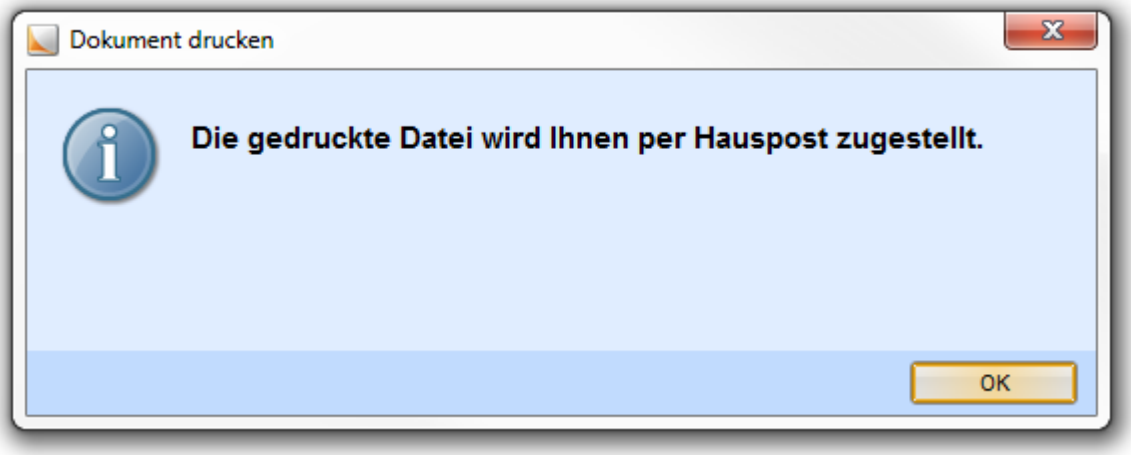


Abb. 10: Info-Box mit Überschrift

5.3 Warnhinweis (AlertBox)

Für Warnungen gibt es einen weiteren einfachen Dialog, die AlertBox. Sie wird genauso verwendet wie die InfoBox, erscheint allerdings mit einem Warn-Dreieck als Symbol. Im Beispiel bekommt der Benutzer einen Hinweis, dass sein Druckkontigent aufgebraucht ist. Dabei wird der Haupttext mit einer zusätzlichen Erklärung ergänzt. Diese ist in zwei Absätze unterteilt, welche durch einen doppelten erzwungenen HTML-Zeilenumbruch **
** erzeugt werden.

```
workspace.showAlertBox( "Dokument drucken", "<html><h3>Druckkontigent
aufgebraucht</h3>Sie können keine weiteren Dokumente drucken, weil Ihr
Druckkontigent für diesen Monat aufgebraucht ist.<br><br>Nächsten Monat
können Sie wieder drucken.</html>" );
```

Abb. 11: Warnhinweis mit Überschrift und Erklärung

5.4 Ja-/Nein-Fragen (QuestionBox)

Für einfache Fragen an den Benutzer, die dieser mit *Ja* oder *Nein* beantworten kann, ist die QuestionBox gedacht. Auch hier wird per Dialog wieder ein Text angezeigt. Der Haupttext sollte dabei die Frage sein, welche der Benutzer dann per Klick auf die Schaltfläche *Ja* oder *Nein* beantwortet. Dieser Dialog liefert als **Rückgabewert** eine **boolean** Variable. Sie hat den Wert *true*, wenn *Ja* angeklickt wurde und *false*, wenn *Nein* angeklickt oder der Dialog per X geschlossen wurde. Im folgenden Beispiel wird der Benutzer vor dem Drucken gefragt, ob er wirklich drucken möchte. Der Rückgabewert wird benutzt, um die Standardfunktion abzubrechen. Dies ist bei den Start-Events des Java Client per negativen Rückgabewert möglich.

```
// Nachfragen, ob das Dokument wirklich gedruckt werden soll
function eloPrintStart() {
  var wirklichDrucken = workspace.showQuestionBox( "Dokument drucken",
"<h3>Wirklich drucken?</h3>Um Papier zu sparen und die Umwelt zu schonen,
sollten Dokumente nur in Ausnahmefällen ausgedruckt werden." );
  if (!wirklichDrucken) {
    // Drucken abbrechen
    return -1;
  }
}
```

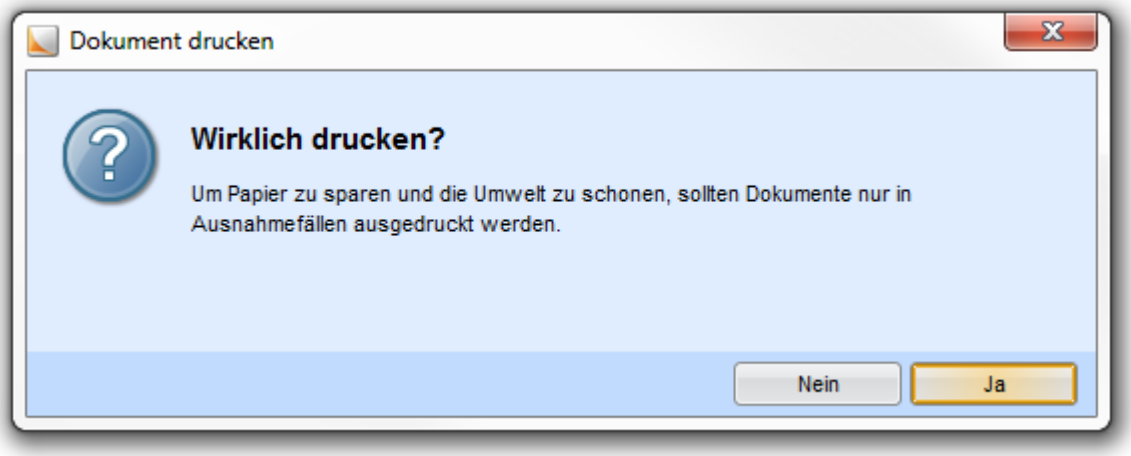


Abb. 12: Frage-Box mit Überschrift und Erklärung

5.5 Abfrage einer Bezeichnung (InputBox)

Um einen einzelnen Text, zum Beispiel einen Namen abzufragen, kann die InputBox benutzt werden. Eine Minimal- und Maximallänge kann angegeben werden. Die InputBox eignet sich auch für eine Passworteingabe, bei der die Eingabe unleserlich dargestellt wird.

Das folgende Beispiel zeigt einen Dialog zur Eingabe einen Namens beim Umbenennen an.

```
var oldName = "Beispielname";
var newName = workspace.showInputBox( "Umbenennen", "Geben Sie bitte den
neuen Namen ein.", oldName, 1, 100, false, -1 );
```

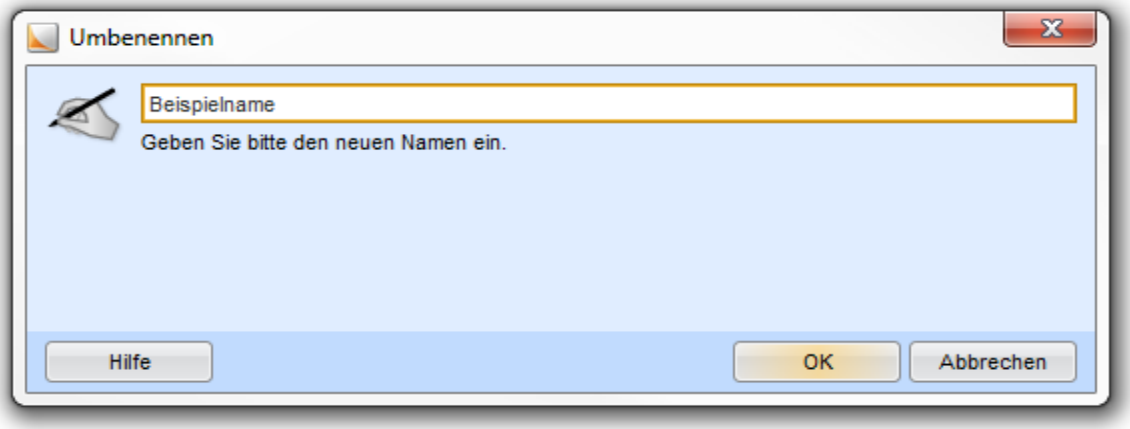


Abb. 13: Box mit Eingabefeld

5.6 Auswahl im Dateisystem (FileDialog)

Für die Auswahl von Dateien oder Ordnern im Dateisystem des Betriebssystems steht der entsprechende Standard-Dialog auch im Scripting zur Verfügung. Die vom Benutzer im Dialog ausgewählten Dateien / Ordner werden als File-Objekte zurückgegeben, bei einem Abbruch des Dialogs ist der Rückgabewert null.

```
var saveMode = false; // Modus „Laden“
var selectOnlyFiles = true; // Nur Dateien
var selectionPath = ""; // Keine Vorauswahl
var files = workspace.showFileChooserDialog( "Datei einfügen", saveMode,
selectOnlyFiles, selectionPath );
if (files != null) {
  // Dateien verarbeiten
}
```

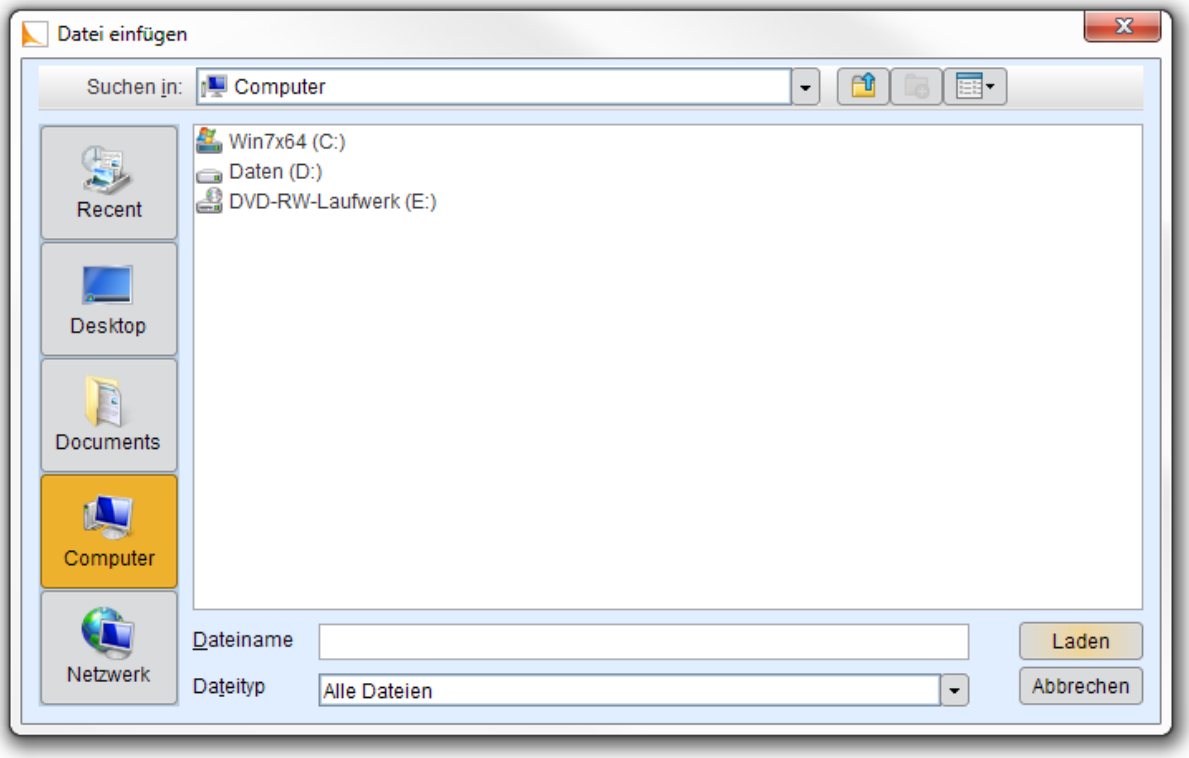


Abb. 14: Dialog für Dateien aus dem Dateisystem

5.7 Dokument oder Ordner im Archiv auswählen (TreeSelectDialog)

Um ein Dokument oder einen Ordner im Archiv auszuwählen kann der TreeSelectDialog benutzt werden. Er wird im Java Client zum Beispiel für die Auswahl bei der Archivablage aus der Postbox benutzt. Dafür sind auch die Favoriten gedacht, welche bei der Verwendung im Scripting ein- oder ausgeblendet werde können. Weiterhin kann angegeben werden, ob Dokumente oder Ordner oder beides als Auswahl erlaubt sind und welcher Teil des Archivs angezeigt werden soll. Wie immer müssen die Texte im Dialog belegt werden.

Das folgende Beispiel ermöglicht die Auswahl eines Ordners aus dem ganzen Archiv oder den Favoriten:

```
var title = "Ablegen";
var text = "Suchen Sie bitte das passende Archivziel aus.";
var rootId = 1;
var docs = false;
var folders = true;
var favorites = true;
var targetId = workspace.showTreeSelectDialog( title, text, rootId, docs,
folders, favorites );
```

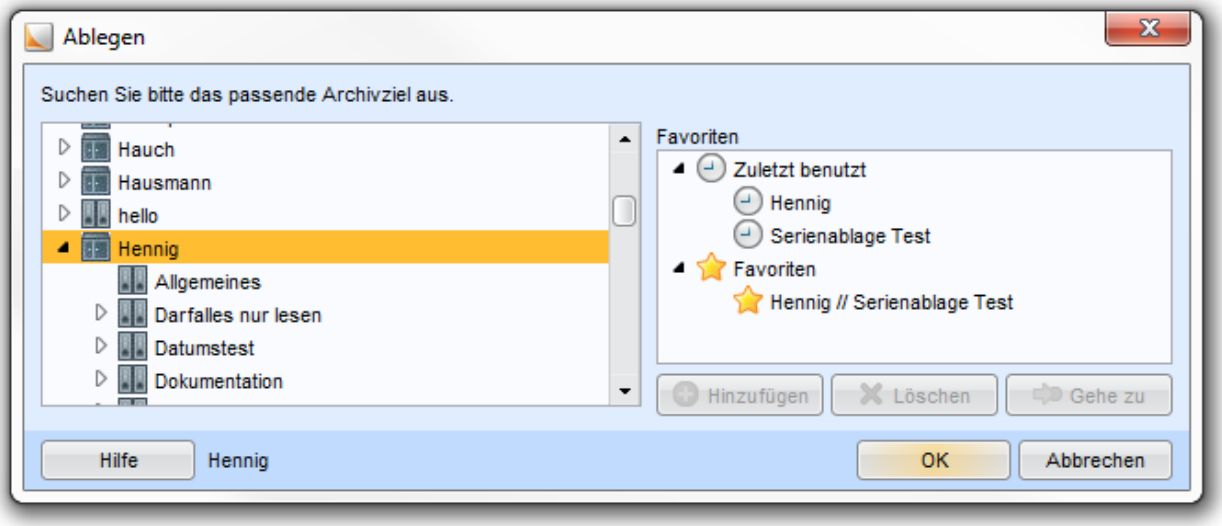


Abb. 15: Dialog mit Baumansicht

5.8 Benutzerauswahl (UserSelectionDialog)

Für die Auswahl von Benutzern und/oder Gruppen dient der UserSelectionDialog. Es kann angegeben werden, ob Benutzer und/oder Gruppen gewünscht sind und wie viele minimal und maximal ausgewählt werden sollen.

Das folgende Beispiel zeigt einen Dialog in dem 1 bis 5 Benutzer oder Gruppen ausgewählt werden können:

```
var multiselect = true;
var min = 1;
var max = 5;
var allowUsers = true;
var allowGroups = true;
var selectedUsers = workspace.showUserSelectionDialog( multiselect, min,
max, allowUsers, allowGroups);
```

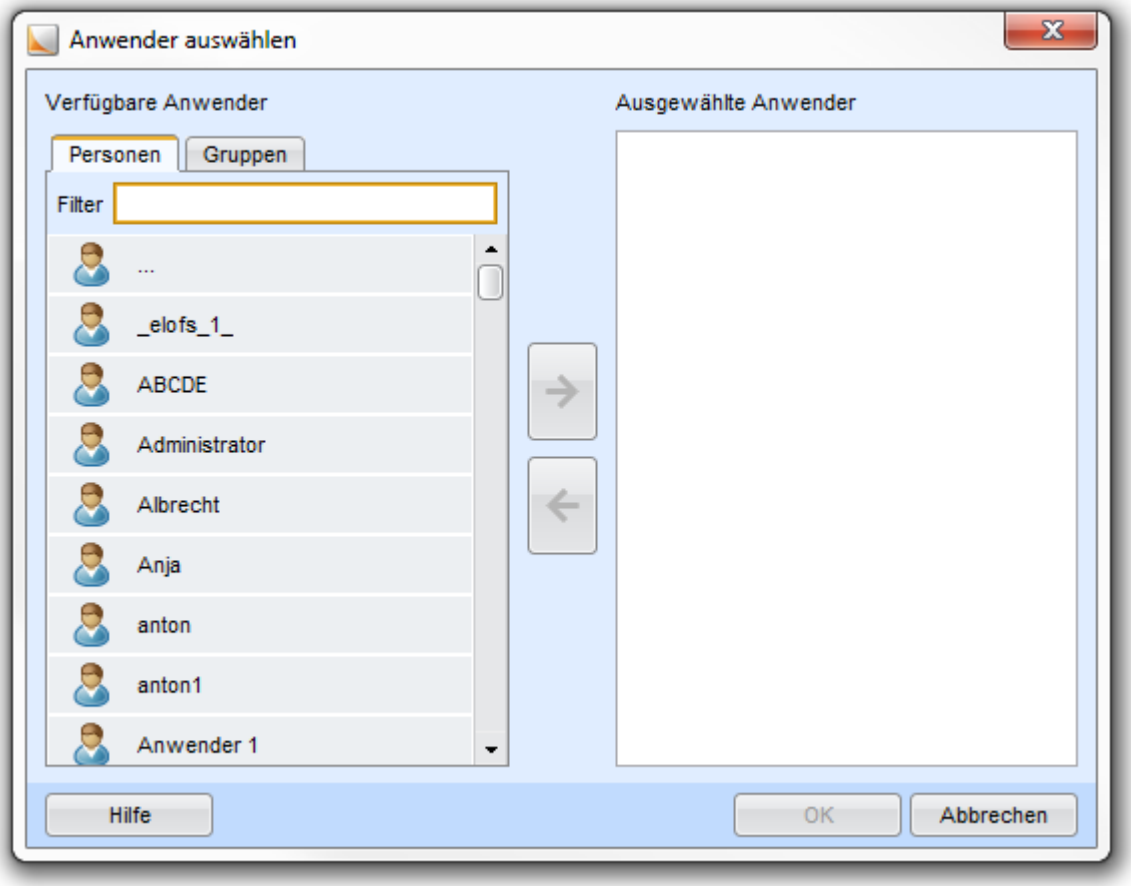


Abb. 16: Dialog mit Benutzerauswahl

5.9 Berechtigungen (PermissionsDialog)

Der PermissionsDialog kann mit einer Vorbelegung der Berechtigungen angezeigt werden, er eignet sich damit sowohl zum Erstellen als auch zum Bearbeiten von Berechtigungen.

Das folgende Beispiel zeigt einen Dialog zum Einstellen der Berechtigungen ohne vordefinierte Berechtigungen.

```
var title = "Berechtigungen bearbeiten";
var oldPermissions = [];
var newPermissions = workspace.showPermissionsDialog( title, oldPermissions
);
```

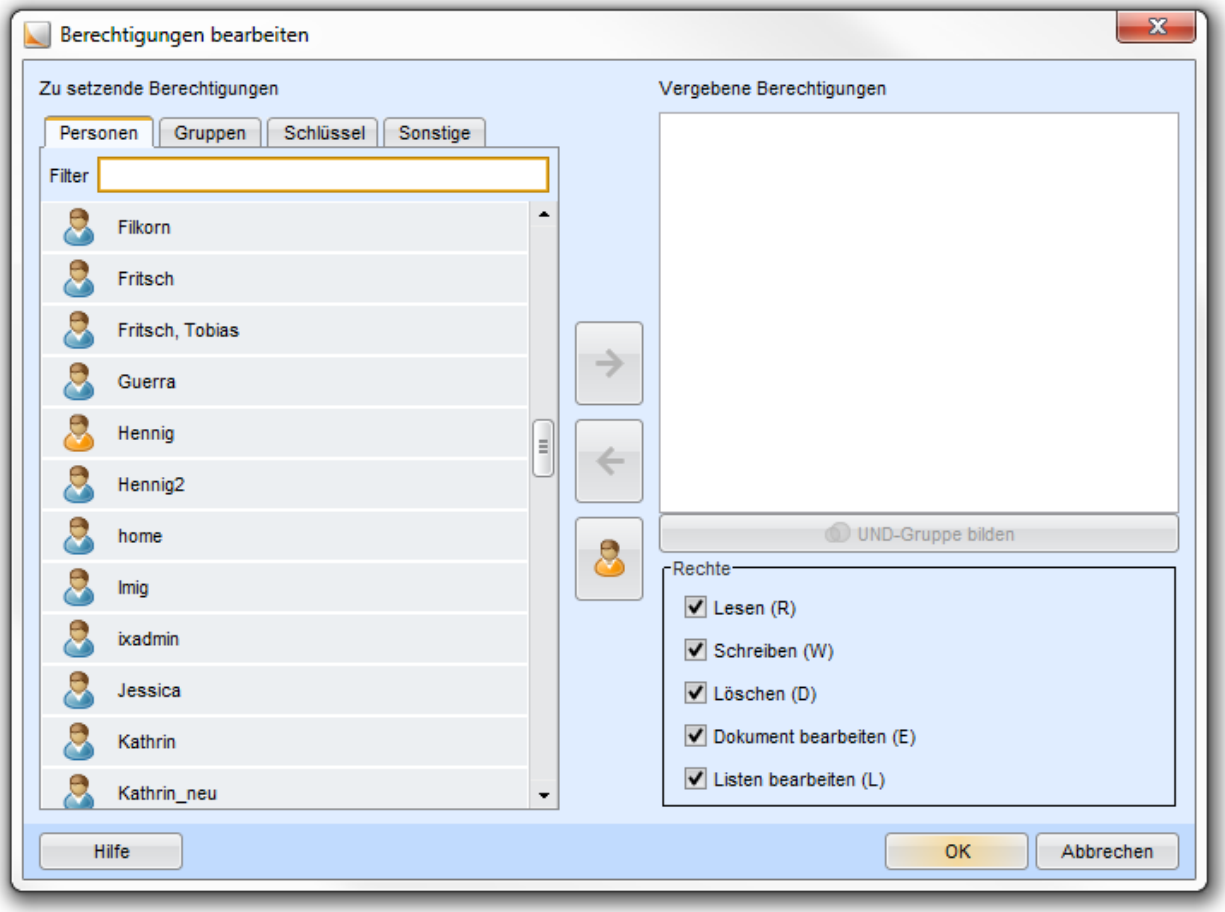


Abb. 17: Dialog mit Berechtigungen

5.10 Auswahl-Dialog (CommandLinkDialog)

In vielen Situationen soll der Benutzer zwischen zwei oder mehr Möglichkeiten wählen können und zusätzliche die Möglichkeit haben die Aktion abzubrechen. Hierfür wird im Java Client ein Auswahl-Dialog benutzt, welcher auch im Scripting zur Verfügung steht. Die Auswahl wird dabei in Form von flachen Schaltflächen mit einem Icon dargestellt. Die übliche bzw. sichere Möglichkeit sollte dabei immer ganz oben stehen und wird im Dialog vorausgewählt.

Das folgende Beispiel ist für eine Funktion gedacht die einen Workflow startet und normalerweise nur einmal aufgerufen werden soll, in besonderen Fällen aber auch mehrmals benutzt werden kann. Der Dialog zeigt zwei Möglichkeiten an.

```
importClass(Packages.de.elo.client.scripting.constants.CONSTANTS);

var dlgTitel = "Workflow starten";
var dlgHeader = "Bereits ein Workflow vorhanden";
var dlgText = "Es wurde bereits ein Workflow zu diesem Eintrag gestartet.
Bitte wählen Sie:";
var optionNames = [ "Vorhandenen Workflow anzeigen", "Neuen Workflow
starten" ];
var optionDescriptions = [ "Es wird kein neuer Workflow gestartet.", "Die
beiden Workflows laufen parallel." ];
var option = workspace.showCommandLinkDialog( dlgTitel,  dlgHeader,
dlgText, CONSTANTS.DIALOG_ICON.QUESTION, optionNames, optionDescriptions,
null );
```

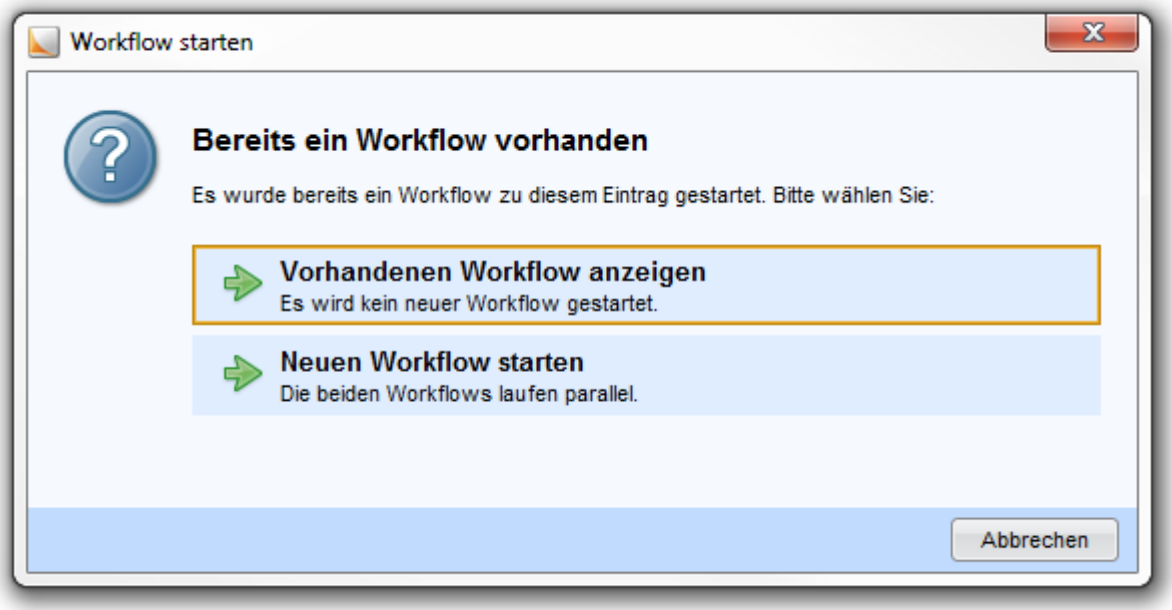


Abb. 18: Dialog mit Auswahlmöglichkeiten

Der Aufruf liefert die Auwahl in Form einer Nummer zurück, wobei die Möglichkeiten von oben nach unten durchnummeriert sind. „Abbrechen“ wird mit einer -1 gekennzeichnet.

5.11 Komplexe Dialoge (GridDialog)

Für umfangreichere Dialoge steht der GridDialog zur Verfügung. Er bietet ein Tabellenraster in dem die Dialogelemente angeordnet werden. Die Anzahl der Spalten und Zeilen wird beim Erstellen des Dialogs angegeben:

workspace.createGridDialog(<Titel>, <Spalten>, <Zeilen>)

Als Dialogelemente stehen zur Verfügung: Button, CheckBox, ComboBox, Label, List, RadioButton, TextArea, TextField und ToggleButton. Außerdem können Java Objekte vom Typ Component hinzugefügt werden. Dabei wird jeweils die Position und Ausdehnung im Tabellenraster angegeben.

Das folgende Beispiel erstellt einen Dialog, in dem ein Hinweis an einen anderen Benutzer eingegeben werden kann. Der Dialog soll es ermöglichen, den Hinweis zu betiteln, einen Benutzer und eine Geheimhaltungsstufe einzustellen und einen längeren Hinweistext einzugeben.

```
dialog = workspace.createGridDialog("Neuer Nachrichtenordner",10,11);

heading = dialog.addLabel(1,1,10,"Neuer Nachrichtenordner");
heading.setFontSize(18);
heading.setBold(true);

dialog.addLabel(1,2,1,"Kurzbezeichnung:");
titleField = dialog.addTextField(2,2,9);
titleField.text = "Neuer Nachrichtenordner";
titleField.addChangeEvent( "checkFields" );

dialog.addLabel(1,3,1,"Von:");
dialog.addLabel(2,3,7,workspace.userName);

dialog.addLabel(1,4,1,"Für:");
userField = dialog.addTextField(2,4,7);
userField.setEditable(false);
selectUserButton = dialog.addButton(9,4,2,"Auswählen","doSelectUser");

dialog.addLabel(1,5,1,"Geheimhaltung:");
var prioOpts = ["Normal","Geheim","Streng geheim"];
combo = dialog.addComboBox(2,5,7,prioOpts,false);

dialog.addLabel(1,6,10,"Kommentar:");
textArea = dialog.addTextArea(2,6,9,5);
textArea.addChangeEvent( "checkFields" );

dialog.addButton(2,11,2,"Löschen","doClear");
```

```
var okButtonPressed = dialog.show();
```

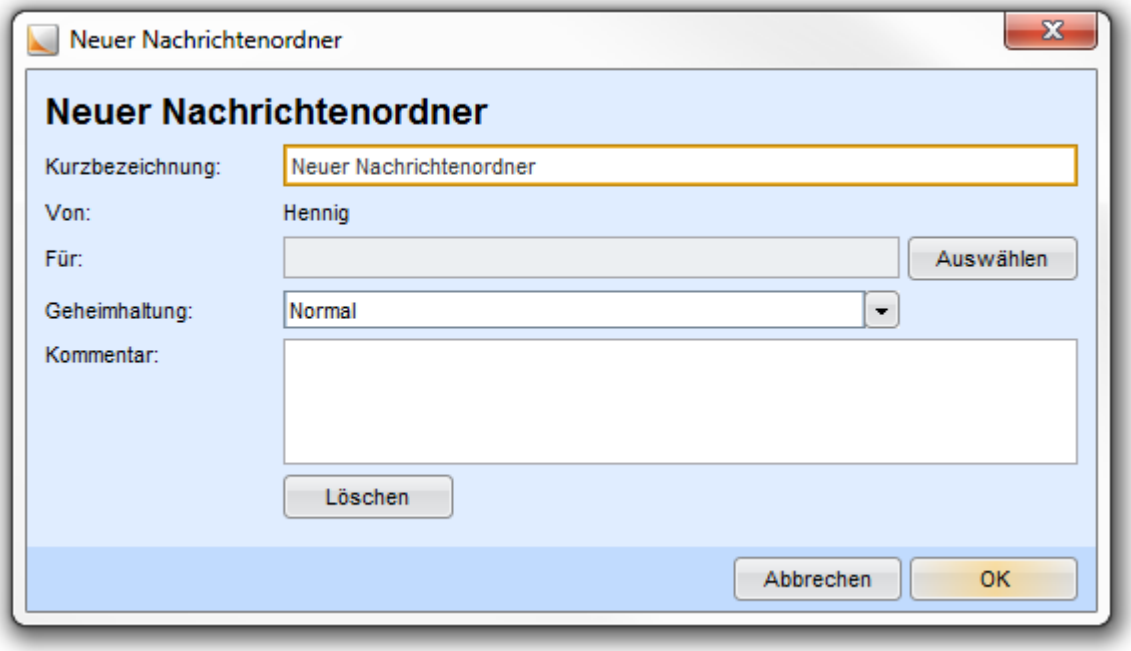


Abb. 19: Komplexer Dialog (GridDialog)

5.11.1 Größe der Spalten und Zeilen

Die Breite der Spalten wird so aufgeteilt, das jede Spalte die Breite bekommt, welche für seinen Inhalt notwendig ist. Sollte danach noch Platz vorhanden sein, wird dieser gleichmäßig auf alle Spalten verteilt.

Die Zeilen haben eine vordefinierte Höhe welche wenn notwendig vergrößert wird, damit der Inhalt reinpasst. Überschüssiger Platz wird als freie Fläche unter den Zeilen dargestellt.

Beispiel:

Der folgende Dialog stellt eine Auswahl von Standorten dar. Der Benutzer überträgt darin einen vorhandenen Standorte aus der linken Liste in die rechte Auswahlliste. Zusätzlich gibt es unten eine Option „Standortleiter benachrichtigen.

```
var dialog = workspace.createGridDialog( "Standort-Auswahl", 5, 7 );
var grid = dialog.gridPanel;

grid.addLabel( 1,1,2, "Vorhande Standorte:" );
var listVorhanden = grid.addList( 1,2,2,5 );
listVorhanden.setData( [ "Berlin", "Hamburg", "Madrid", "New York",
"Paris", "Rom", "Stuttgart", "Tokio"] );

grid.addLabel( 4,1,2, "Auswahl:" );
grid.addList( 4,2,2,5 );

grid.addButton( 3,3,1, "hinzufügen >", "xxx" );
grid.addButton( 3,5,1, "< entfernen", "xxx" );
grid.addCheckBox( 1,7,5, "Standortleiter benachrichtigen", true );

dialog.show();
```

Der von diesem Skript erzeugt Dialog sieht dann in verschiedenen Größen so aus:

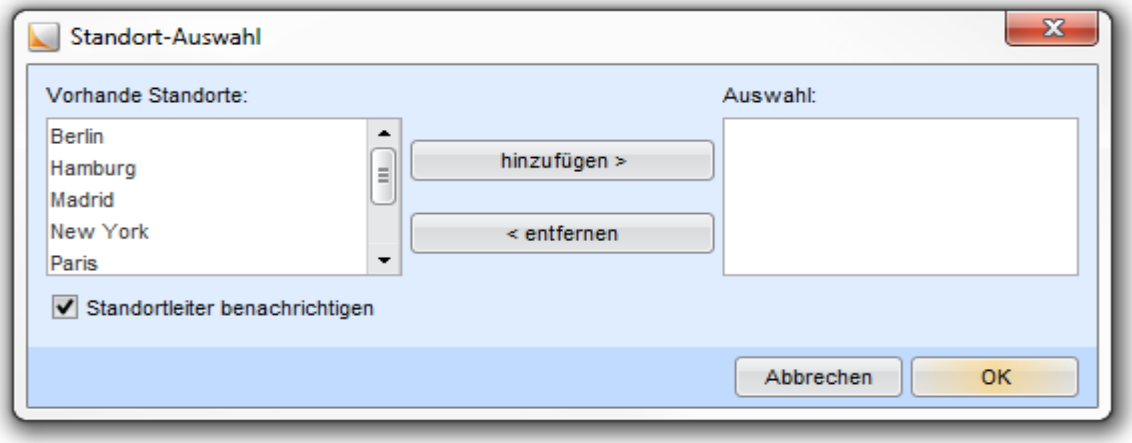


Abb. 20: Einfacher Grid-Dialog

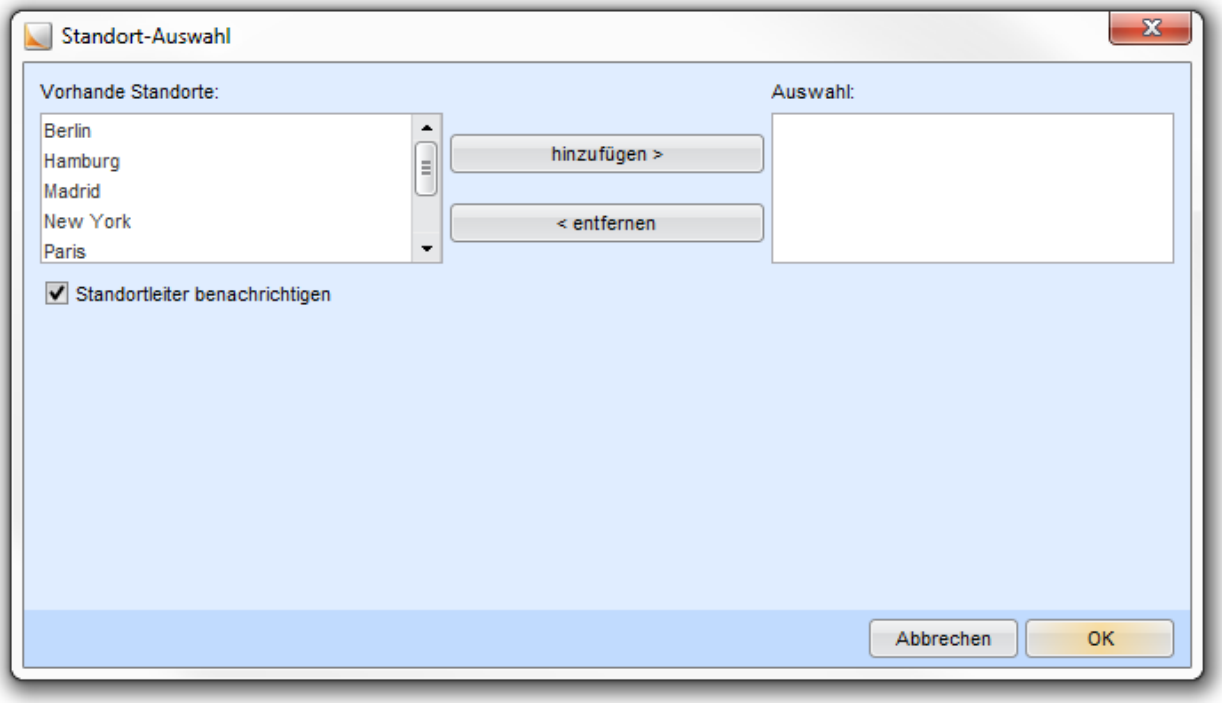


Abb. 21: Vergrößerter Grid-Dialog mit statischen Elementen

Wie man sieht, wachsen die Spalten und Zeilen bei größerem Dialog wie oben beschrieben. Die vorhandene Fläche wird nicht sinnvoll ausgenutzt.

Ab Version 8.04.000 des Java Clients gibt es die Möglichkeit, im GridDialog anzugeben, welche Spalten und Zeilen mit Vergrößerung der Dialogfläche wachsen soll. Dazu ist nur eine einzelne Zeile zwischen createGridDialog und dialog.show notwendig:

```
grid.setGrowing( [ 1,4 ], [ 2,6 ] );
```

Mit der Festlegung der Spalten 1 und 4 und der Zeilen 2 und 6 als wachsend, sieht der Dialog nun wesentlich besser aus; vor allem in der großen Version wird der vorhandene Platz besser genutzt.

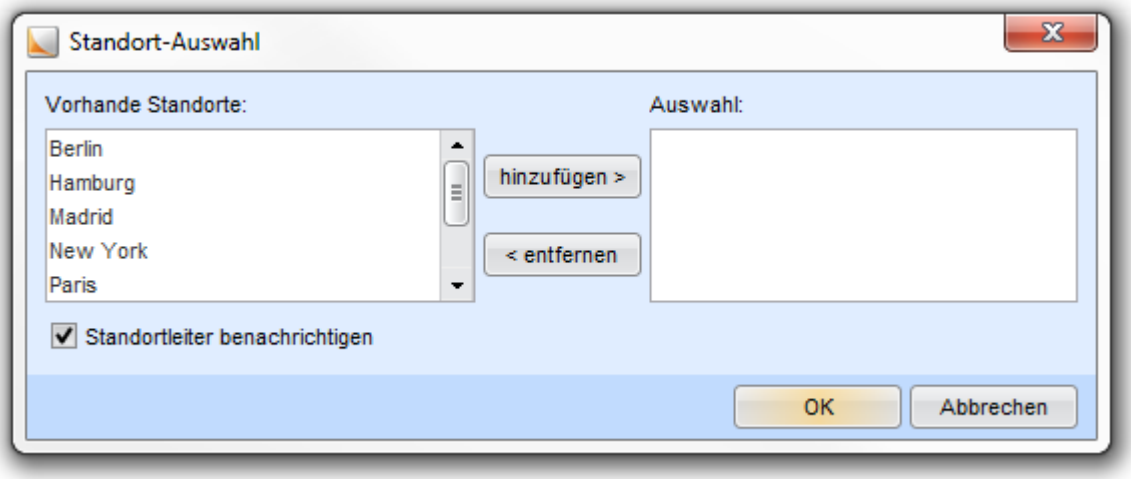


Abb. 22: Grid-Dialog mit dynamisch wachsenden Elementen; hier: Kleine Version

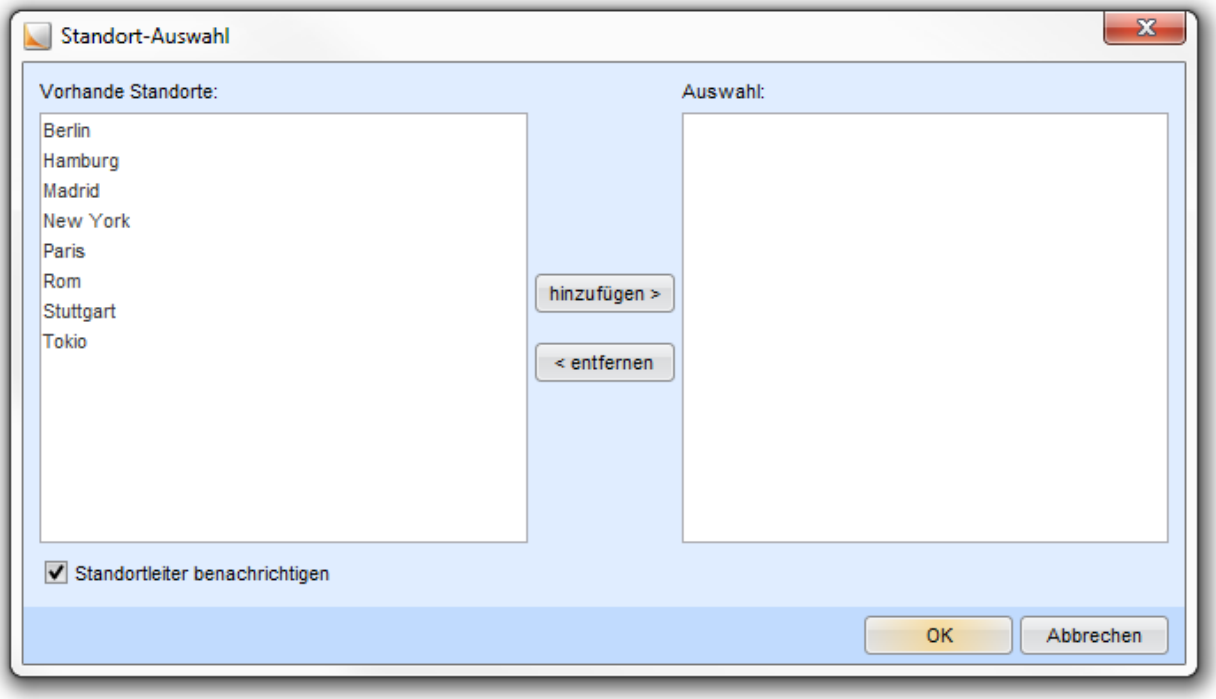


Abb. 23: Grid-Dialog mit dynamisch wachsenden Elementen; hier: Große Version

5.11.2 Dialog-Komponenten

Innerhalb eines GridPanel können verschiedene Komponenten verwendet werden. Die folgende Übersicht zeigt einen Dialog mit den möglichen Komponenten und ihrem Namen.

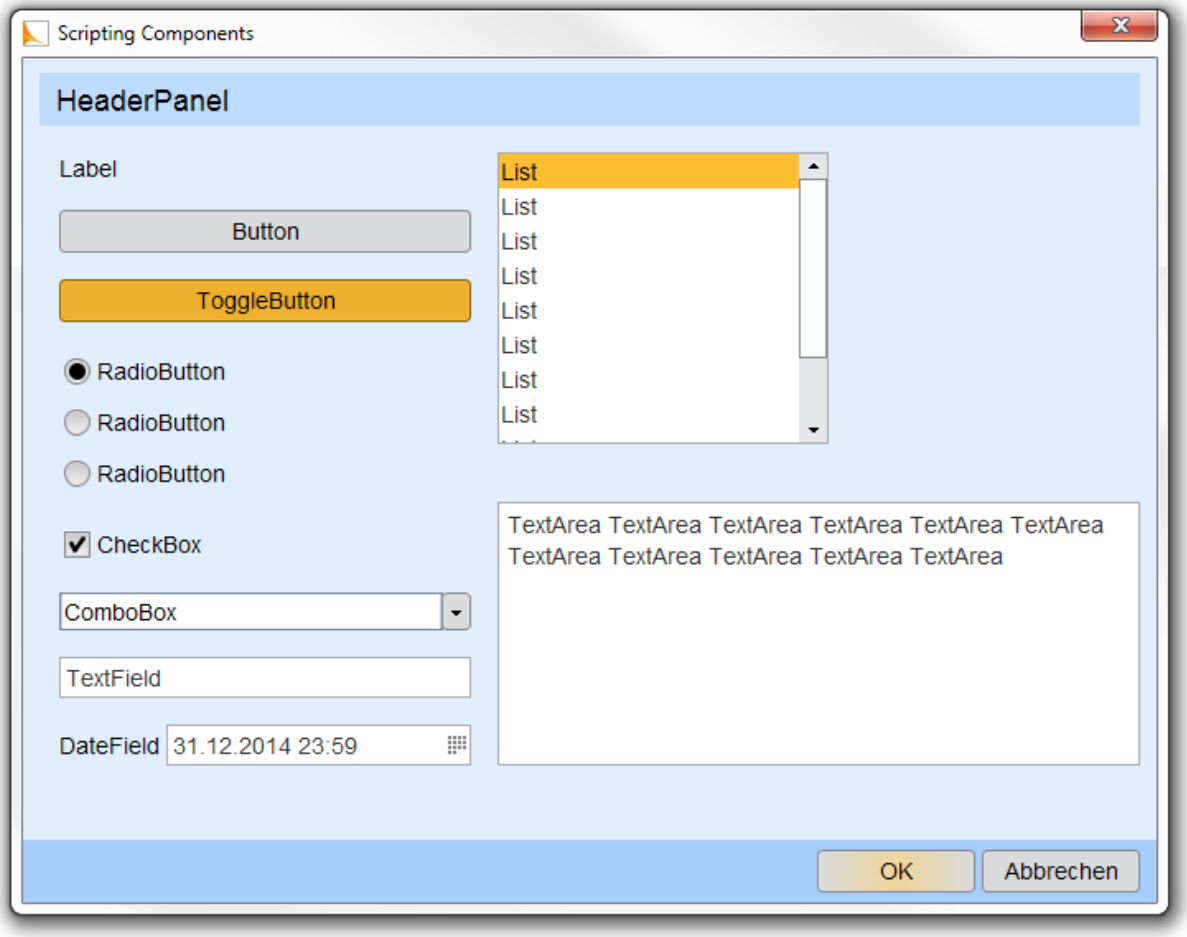


Abb. 24: Beispiel für die unterschiedlichen Komponenten

```
var dialog = workspace.createGridDialog("Scripting Components", 6, 20);
var panel = dialog.gridPanel;
panel.setGrowing( [3,5,6], [20] );

panel.addHeaderPanel( 1, 1, 6, "HeaderPanel" );

panel.addLabel(2, 3, 2, "Label");

panel.addButton(2, 5, 2, "Button", "");

var toggleButton = panel.addToggleButton(2, 7, 2, "ToggleButton", "",
"toggleGroup");
toggleButton.setSelected( true );
```

```
var radioButton1 = panel.addRadioButton(2, 9, 2, "RadioButton", "",
"radioGroup");
radioButton1.setSelected( true );
panel.addRadioButton(2, 10, 2, "RadioButton", "", "radioGroup");
panel.addRadioButton(2, 11, 2, "RadioButton", "", "radioGroup");

panel.addCheckBox(2, 13, 2, "CheckBox", true);

panel.addComboBox(2, 15, 2, ["ComboBox","ComboBox","ComboBox"], false);

var textField = panel.addTextField(2, 17, 2);
textField.text = "TextField";

panel.addLabel(2,19,1,"DateField");
var dateField = panel.addDateField(3, 19, 1);
dateField.setIsoDate("20141231235959");

var list = panel.addList(5, 3, 1, 8);
list.setData(
["List","List","List","List","List","List","List","List","List","List","Lis
t"] );
list.setSelected( "List" );

var textArea = panel.addTextArea(5, 12, 2, 8);
textArea.text = "TextArea TextArea TextArea TextArea TextArea TextArea
TextArea TextArea TextArea TextArea TextArea";

var okPressed = dialog.show()
```

6 Scripting mit den Funktionsbereichen

Die Funktionsbereiche des Java Clients sind im internen Scripting als Objekte vorhanden. Damit ist es etwa möglich in Skript-Funktionen zu berücksichtigen, in welchem Funktionsbereich sich der Benutzer befindet und welche Dokumente oder Ordner er gerade selektiert hat.

Funktionsbereich	Scripting-Objekt	Klasse
Archiv (mehrfach)	archiveViews	ArchiveViews
Postbox	intray	IntrayAdapter
Klemmbrett	clipboard	ClipboardAdapter
Aufgaben	tasks	TasksAdapter
In Bearbeitung	checkout	CheckoutAdapter
Suchen (mehrfach)	searchViews	SearchViewAdapter

In den Funktionsbereichen *Archiv*, *Aufgaben* und *Suche* gibt es mehrere Ansichten, daher sind hier die *ArchiveViewAdapter*, *TasksViewAdapter* und *SearchViewAdapter* nicht direkt erreichbar, sondern es muss über die entsprechende Sammlung der Ansichten zugegriffen werden.

Beispiel: Manuelle Dublettenprüfung in der Postbox

Im folgenden Beispiel wird in die Postbox eine manuelle Dublettenprüfung eingebaut. Ein zusätzlicher Button in *Postboxtools – Archivieren – Ablage* überprüft dann per Knopfdruck alle aktuell in der Postbox selektierten Dokumente.

```
function getScriptButtonPositions(){
  return "4,intray,archiving";
}

function getScriptButton4Name(){
  return "Dublettenprüfung";
}
```

```
function eloScriptButton4Start(){
  var selDocs = intray.getAllSelected();
  var dubFound = false;
  while (selDocs.hasMoreElements()) {
    var intrayDoc = selDocs.nextElement(); // JC IntrayDocument
    if (intrayDoc.isDoublet()) {
```

```
      workspace.showInfoBox("Dublettenprüfung",
        "<html>Dublette gefunden: <b>" + intrayDoc.getName() );
      }
    }

  if (!dubFound){
    workspace.setFeedbackMessage( "Keine Dubletten gefunden" );
  }
}
```

Wenn eine Dublette gefunden wird, erscheint ein Dialog, welcher diese nennt:

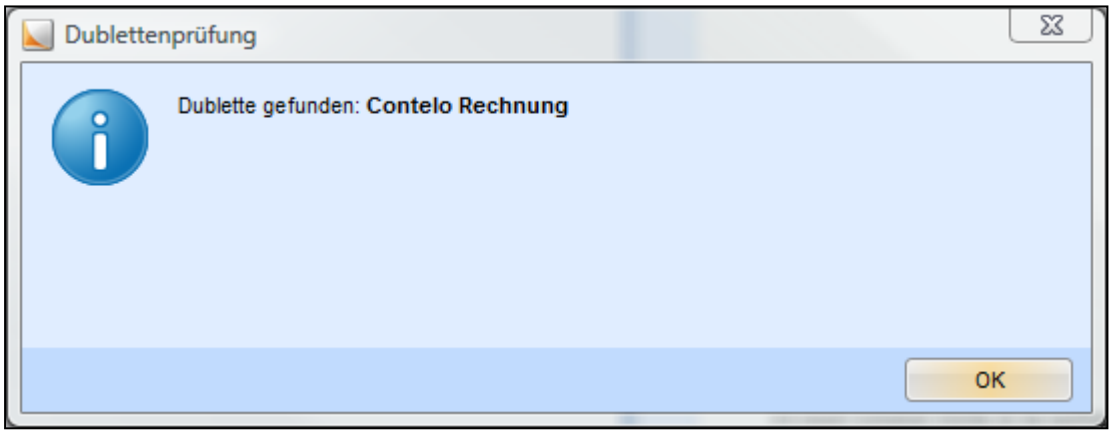


Abb. 25: Info-Box 'Dublettenprüfung'

Wenn alle Dokument überprüft und keine einzige Dublette gefunden wurde, wird eine spezielle Rückmeldung angezeigt, damit für den Benutzer ersichtlich ist, dass die Prüfung überhaupt stattgefunden hat:

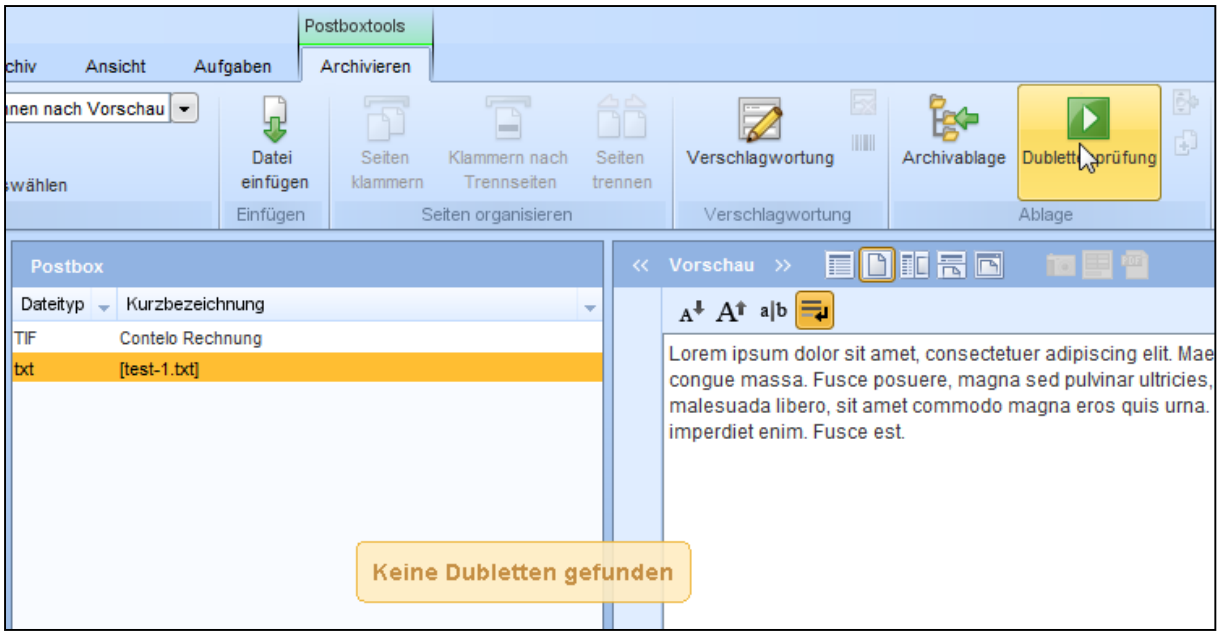


Abb. 26: Rückmeldung 'Keine Dubletten gefunden'

7 Hintergrundprozesse

Der Java Client beherrscht Multitasking und führt eine Reihe von Funktionen im Hintergrund aus, darunter zum Beispiel Import, Export, Löschen im Archiv, Dauerhaft entfernen. Die „Prozessübersicht“ zeigt diese Hintergrundprozesse des Clients an und bietet einen Zugriff auf ein vom Prozess geschriebenes Protokoll. Außerdem können manche Hintergrundprozesse über diesen Dialog auch abgebrochen werden.

Im Scripting können Hintergrundprozesse verwendet werden, um langlaufende Prozesse durchzuführen, ohne dass die Oberfläche des Java Clients hängt. Der Benutzer kann also weiterarbeiten, während eine aufwändige Funktionalität im Hintergrund weiterläuft.

7.1 Serverprozesse

Wenn der Prozess vorhandene Einträge im Archiv bearbeiten soll, ist es sinnvoll, diese Logik auf der Serverseite auszuführen. Einerseits kann der Benutzer dann den Client beenden, auch wenn der Prozess noch nicht fertig ist, andererseits ist dann auch eine Bearbeitung mit administrativen Rechten möglich. Außerdem ist die Performance besser und die Lösung funktioniert auch mit andere ELO Clients.

Prozesse des Indexservers können per Scripting über dessen Schnittstelle gestartet werden. Damit diese auch nach dem Start für den Benutzer sichtbar sind, sollten Sie in die Prozessübersicht des Java Clients eingetragen werden, hierfür ist **WorkspaceAdapter.listServerProcess** vorgesehen. Wenn ein Serverprozess auf diese Weise in die Prozessübersicht aufgenommen, dann kann ein evtl. vom Server geschriebenes Protokoll auch direkt dort eingesehen werden.

Beispiel:

Im folgenden Beispiel wird die Serverfunktion „Dauerhaft entfernen“ gestartet und in die Prozessübersicht eingetragen.

```
var options = new DeleteOptions();
ixc.cleanupStart( options );
var jobState = ixc.cleanupState();
workspace.listServerProcess( "Dauerhaft entfernen", jobState.jobGuid, "%1
Einträge verarbeitet" );
```

7.2 Hintergrundprozesse im Client

Wenn mit größeren Mengen lokalen Daten oder Dateien gearbeitet werden soll, ist eine Abarbeitung auf der Serverseite nicht möglich und eine Hintergrundprozess im Client sinnvoll. Dazu muss die Routine für den Hintergrundprozess in einer Skriptfunktion stehen. Diese kann dann per **workspace.startBackgroundProcess** im Hintergrund gestartet werden. Sie wird gleichzeitig in die Prozessübersicht des Clients eingetragen. Das zurückgegebene Objekt der Klasse **BackgroundJob** ermöglicht das Anlegen eines Protokolls.

Achtung: Die Oberfläche des Java Clients wartet nicht auf die Abarbeitung der Routine, der Benutzer kann also im Client weiterarbeiten. Es darf daher aus der im Hintergrund laufenden Routine die Oberfläche nicht ferngesteuert und keine Dialoge geöffnet werden. Das Setzen einer Selektion, Springen in einen anderen Funktionsbereich oder auch die Anzeige eines Fehlerdialogs würden den Benutzer in seiner Arbeit stören und könnten zu Fehleingaben führen. Achten Sie also sehr genau darauf, welche Methoden der Scripting API Sie verwenden und fangen Sie alle Exceptions die auftreten könnten mit try-catch-Blöcken innerhalb Ihres Skripts sinnvoll ab.

7.2.1 Beispiel: JPEGs archivieren

Im folgenden Beispiel werden alle JPEG aus einem Ordner im Dateisystem in ELO archiviert. Dem Benutzer wird ein Dialog zur Auswahl des Ordners angezeigt, die Archivablage der Bilder erfolgt dann im Hintergrund.

Die Prozessübersicht zeigt dann den Zustand der Archivierung:

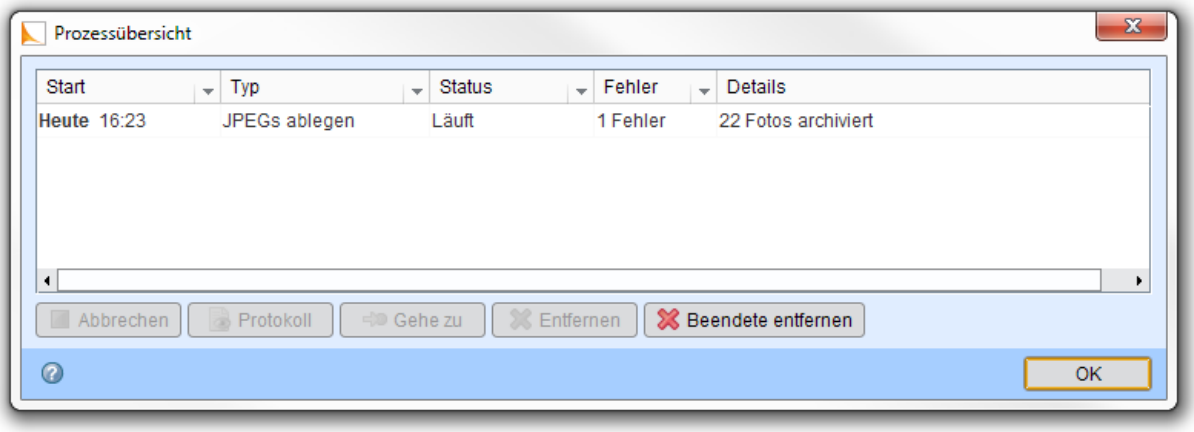


Abb. 27: Hintergrundprozess; Bilddateien archivieren

In der Prozessübersicht kann auch ein Protokoll aufgerufen werden, welches Details zu den einzelnen Dateien enthält und genauere Fehlermeldungen bietet:

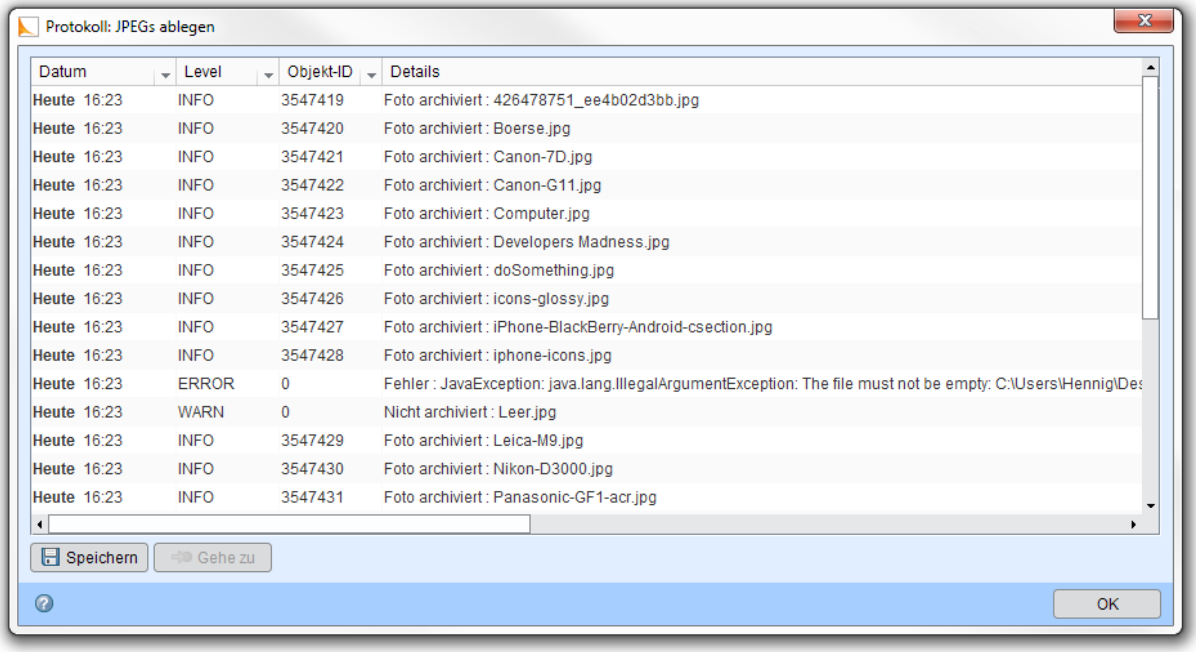


Abb. 28: Protokoll

```
var FCT_NAME = "JPEGs ablegen";
var MASK_NAME = "Freie Eingabe";

function archiveJPEGs(){
  // Bilderordner auswählen
  folders = workspace.showFileChooserDialog( FCT_NAME, false, false, "" );
  if (folders == null) return;

  targetFolder = workspace.activeView.firstSelected;
  importProcess = workspace.startBackgroundProcess( FCT_NAME,
"processJPEGs" );
}

function processJPEGs(){
  fileCount = 0;
  archivedCount = 0;
  var files = folders[0].listFiles();
  for( i=0; i<files.length; i++ ){
    if (importProcess.isStopped()){
      // Abbruchmöglichkeit des Dialogs "Prozessübersicht" unterstützen
      break;
    }
    var file = files[i];
    var filename = file.name;
    var filenameUC = filename.toUpperCase();
    if (filenameUC.endsWith("JPG") || filenameUC.endsWith("JPEG")) {
```

```
      archiveFile(filename, file);
    }
  }
  // Archivierung beendet
  importProcess.setFinished();
}

function archiveFile( filename, file ) {
  fileCount++;
  try {
    // Verschlagwortung vorbereiten
    var sord = targetFolder.prepareDocument( MASK_NAME );
    sord.name = "Foto "
    if (fileCount < 100) sord.name += "0";
    if (fileCount < 10) sord.name += "0";
    sord.name += fileCount;

    // Archivablage mit Protokoll
    var archiveDocument = targetFolder.addDocument( sord, file );
    archivedCount++;
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.INFO, "Foto
archiviert : " + filename, archiveDocument.sord.id );
  } catch (exception) {
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.ERROR, "Fehler
: " + exception );
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.WARN, "Nicht
archiviert : " + filename );
  }

  // Status aktualisieren
  importProcess.setStatus( archivedCount + " Fotos archiviert" );
}
```

7.2.2 Beispiel: Fotos archivieren

Dieses Beispiel baut die Funktionalität des vorigen noch weiter aus. Es wird jetzt ein Dialog angezeigt, welcher neben der Auswahl des Ordners auch eine Auswahl der Dateitypen erlaubt. Außerdem wird der Fortschritt direkt im Dialog angezeigt, der Prozess kann von hier abgebrochen und auch das Protokoll aufgerufen werden. Da es sich um einen nichtmodalen Dialog handelt, ist trotzdem ein paralleles Weiterarbeiten möglich.

Gegenüber dem vorigen Beispiel dient der Dialog dazu, dem Benutzer den Vorgang besser sichtbar und kontrollierbar zu machen. Er fordert beim Aufruf zur Ordnerauswahl auf:

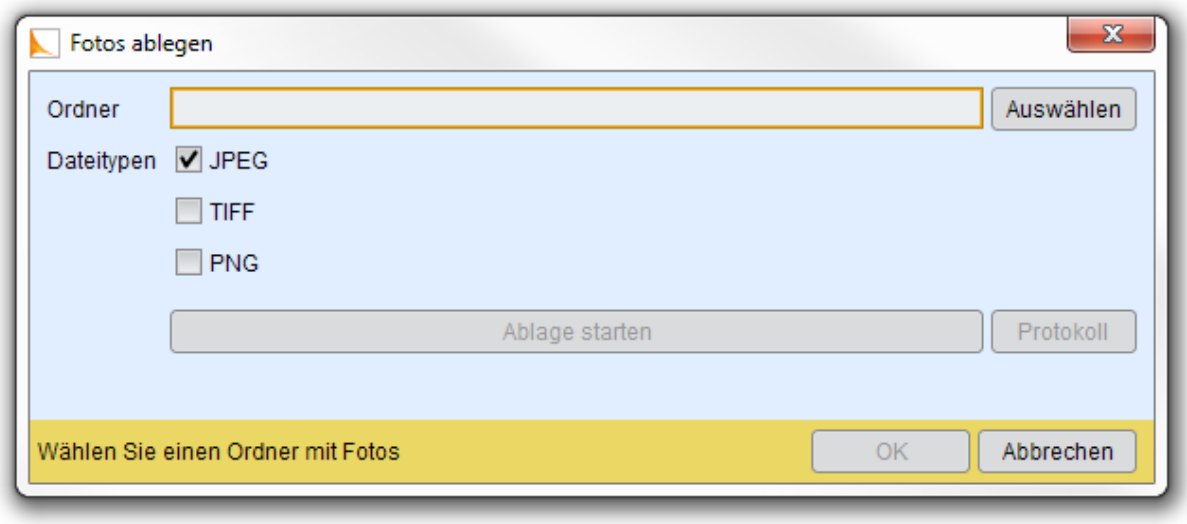


Abb. 29: Dialog mit Handlungsaufforderung

Wenn die Archivierung gestartet wurde, erscheint ein Fortschrittsbalken und der aktuelle Status wird in der Statuszeile des Dialogs angezeigt:

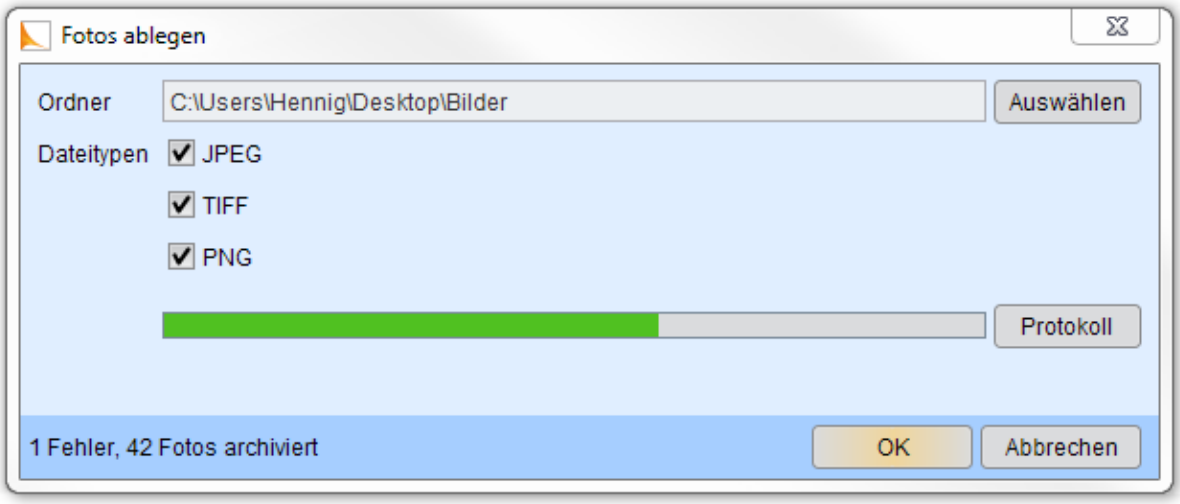


Abb. 30: Dialog mit Fortschrittsbalken

```
var FCT_NAME = "Fotos ablegen";
var MASK_NAME = "Freie Eingabe";

function getScriptButton125Name() {
  return FCT_NAME;
}

function getScriptButtonPositions() {
  return "125,archive,insert";
}

function eloScriptButton125Start(){
  if (!checkSelection()) {
    workspace.showInfoBox( FCT_NAME, "<html><b>Bitte wählen Sie einen
Zielordner im Archiv.</b></html>" );
    return;
  }

  targetFolder = workspace.activeView.firstSelected;

  dialog = workspace.createGridDialog( FCT_NAME, 4, 7 );
  var panel = dialog.gridPanel;
  panel.setGrowing( [2], [] );

  panel.addLabel( 1, 1, 1, "Ordner " );
  pathField = panel.addTextField( 2, 1, 2 );
  pathField.editable = false;
  panel.addButton( 4, 1, 1, "Auswählen", "selectPictureFolder" );

  panel.addLabel( 1, 2, 1, "Dateitypen " );
  jpegCheckbox = panel.addCheckBox( 2, 2, 3, "JPEG", true );
  tiffCheckbox = panel.addCheckBox( 2, 3, 3, "TIFF", false );
  pngCheckbox = panel.addCheckBox( 2, 4, 3, "PNG", false );

  startButton = panel.addButton( 2, 6, 2, "Ablage starten",
"startPictureImport" );
  startButton.enabled = false;
  progressBar = panel.addProgressBar( 2, 6, 2 );
  progressBar.visible = false;
  protocolButton = panel.addButton( 4, 6, 1, "Protokoll",
"showPictureImportProtocol" );
  protocolButton.enabled = false;

  finishedLabel = panel.addLabel( 2, 7, 2, "Ablage der Fotos beendet" );
  finishedLabel.bold = true;
  finishedLabel.visible = false;

  dialog.setStatusYellow( "Wählen Sie einen Ordner mit Fotos" );
  dialog.show( "", "cancelPictureImport" );
}
```

```
function checkSelection(){
  if (workspace.activeView.selectionCount != 1) return false;
  if (workspace.activeView.firstSelected.isStructure()) return true;
  return false;
}

function selectPictureFolder(){
  folders = workspace.showFileChooserDialog( FCT_NAME, false, false, "" );
  if (folders != null) {
    pathField.setText( folders[0].getPath() );
    startButton.enabled = true;
    dialog.setStatusNormal( "" );
  }
}

function startPictureImport(){
  importProcess = workspace.startBackgroundProcess( FCT_NAME,
"processPictures" );
  startButton.visible = false;
  progressBar.visible = true;
}

function cancelPictureImport(){
  importProcess.stop();
}

function showPictureImportProtocol(){
  importProcess.showProtocol();
}

function processPictures(){
  imageCount = 0;
  archivedCount = 0;
  errorCount = 0;
  var files = folders[0].listFiles();
  for( i=0; i<files.length; i++ ){
    if (importProcess.isStopped()){
      break;
    }

    var file = files[i];
    var filename = file.name;
    var filenameUC = filename.toUpperCase();
    if (checkFileFormat(filenameUC)) {
      archiveFile(filename, file);
    }

    var percent = 100 * i / files.length;
    progressBar.setPercentage( percent );
  }
```

```
  importProcess.setFinished();
  progressBar.setPercentage( 100 );
  finishedLabel.visible = true;
}

function checkFileFormat(filenameUC){
  if (jpegCheckbox.isChecked() && (filenameUC.endsWith("JPG") ||
filenameUC.endsWith("JPEG"))) return true;
  if (tiffCheckbox.isChecked() && (filenameUC.endsWith("TIF") ||
filenameUC.endsWith("TIFF"))) return true;
  if (pngCheckbox.isChecked() && filenameUC.endsWith("PNG")) return true;

  return false;
}

function archiveFile( filename, file ) {
  imageCount++;

  try {
    // Verschlagwortung vorbereiten
    var sord = targetFolder.prepareDocument( MASK_NAME );
    sord.name = "Foto "
    if (imageCount < 100) sord.name += "0";
    if (imageCount < 10) sord.name += "0";
    sord.name += imageCount;

    // Archivablage
    var archiveDocument = targetFolder.addDocument( sord, file );
    archivedCount++;
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.INFO, "Foto
archiviert : " + filename, archiveDocument.sord.id );
  } catch (exception) {
    errorCount++;
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.ERROR, "Fehler
: " + exception );
    importProcess.addProtocolEntry( CONSTANTS.PROTOCOL_LEVEL.WARN, "Nicht
archiviert : " + filename );
  }

  protocolButton.enabled = true;
  var message = archivedCount + " Fotos archiviert";
  importProcess.setStatus( message );
  if (errorCount == 0) {
    dialog.setStatusNormal( message );
  } else {
    dialog.setStatusNormal( errorCount + " Fehler, " + message );
    finishedLabel.text = "Ablage der Fotos mit Fehler beendet";
    finishedLabel.setColor( 200, 0, 0 );
  }
}
```

8 ActiveX Objekte

Unter Windows können ActiveX-Objekte aus dem Scripting heraus angesprochen werden. Hierzu verwendet der Client die **Jacob** Java to Com Bridge. Die dafür notwendigen Klassen müssen über einen **JavaImporter** in das Script integriert werden.

Beispiel:

Das folgende Beispiel öffnet Microsoft Excel mit einer neuen Tabelle und trägt in diese einige Werte ein.

```
var importNames = JavaImporter();
importNames.importPackage(Packages.com.ms.com);
importNames.importPackage(Packages.com.ms.activeX);
importClass(Packages.com.jacob.activeX.ActiveXComponent);
importClass(Packages.com.jacob.com.Dispatch);
importClass(Packages.com.jacob.com.Variant);

function openExcel(){
  var xl = new ActiveXComponent("Excel.Application");
  Dispatch.put(xl,"Visible",1);
  var workbooks = Dispatch.get(xl, "Workbooks").toDispatch();
  var workbook = Dispatch.get( workbooks, "Add" ).toDispatch();
  var sheet = Dispatch.get( workbook, "ActiveSheet" ).toDispatch();
  var cell = Dispatch.call(sheet, "Cells", 1,1).toDispatch();
  Dispatch.put(cell, "Value", "123");
}

function eloScriptButton1Start(){
   openExcel();
}

function getScriptButton1Name(){
  return "Open Excel";
}

function getScriptButtonPositions(){
  return "1,home,new";
}
```

9 Weitere Beispiele

9.1 Anzahl der Druckvorgänge zählen

Dieses Skript zählt die Anzahl der durchgeführten Druckvorgänge in einer globalen Variablen. Dieser Wert kann dann über den ScriptButton6 in einer Info-Box ausgegeben werden.

```
counter = 0;

// Ausführungen des Druckens zählen
function eloPrintEnd() {
  counter++;
}

// Anzahl Druckaufrufe in Info-Box ausgeben.
function eloScriptButton6Start() {
  workspace.showInfoBox( "Zähler", "Drucken wurde " + counter
    + " mal ausgeführt." );
}
```

9.2 Automatisierte Archivablage

Dieses Skript legt die in der Postbox (Intray) selektierten Dokumente im Archiv im Strukturelement mit der ID 4180 ab. Vorher wird der vorgegebene Name um den Text „ (Skriptablage)" erweitert, die Ablagemaske auf die Maske mit der MaskId 19 geändert und drei Zeilen der Verschlagwortung gesetzt: Zeile 0 mit dem Wert „Test-0“, Zeile 2 mit dem Wert „Test-2“ und die (erste) Zeile mit dem Namen „Anzeige“ mit dem Wert „Test-Anzeige“. Zusätzlich werden Informationen zum Debuggen über den Log4J-Logger ausgegeben.

```
// Legt die im InTray selektierten Dokumente mit überarbeiteter
// Verschlagwortung Strukturelement mit ID 4180 ab.
function eloScriptButton3Start() {
  log.debug( "Skriptablage gestarted" );
  docs = intray.getSelected();
  while ( docs.hasMoreElements() ){
   log.debug( "more elements" );
    doc = docs.nextElement();
   log.debug( "doc : " + doc.getFilePath() );
   doc.setName( doc.getName() + " (Skriptablage)");
   doc.setMaskId( 19 );
   doc.setObjKeyValue( 0, "Test-0" );
   doc.setObjKeyValue( "Anzeige", "Test-Anzeige" );
   doc.setObjKeyValue( 2, "Test-2" );
   doc.saveSord();
   doc.insertIntoArchive( 4180, "1", "Skriptablage" );
  }
 log.debug( "done" );
}
```

10 Externes Scripting

Der ELO Java Client kann auch von außen per Scripting ferngesteuert werden.

Diese Möglichkeit besteht nur unter Microsoft Windows wenn der Client per Setup installiert wurde.

10.1 COM Server

Der Java Client wird als COM Server in Microsoft Windows registriert. Er kann dann von allen COM-fähigen Skriptsprachen angesprochen werden, zum Beispiel Visual Basic. Die ELO Office Makros für den Windows Client benutzen genau diese Methodik.

Die genaue Beschreibung der Vorhandenen Funktionen entnehmen sie der JavaDoc zu der Klasse **EloComServer**.

Beispiel in Visual Basic:

```
' COM-Objekt erzeugen
set Elo = CreateObject("jniwrapper.elocomserver")

' Client starten und am Archiv anmelden
Elo.login "Archiv","http://eloserver:8080/ixArchiv/ix","Admin","elo"

' Anzeige des Archivnamens in einem Hinweisdialog
MsgBox Elo.getArchiveName

' Den Client zum Eintrag mit der ObjektID 20 springen lassen
Elo.gotoObjectId(20)
```

11 Anhang

11.1 Ribbon-Positionen

Name in der Oberfläche		Positionsangabe
Tab	Band / Gruppe	
Start		home
	Navigation	navigation
	Neu	new
	Ansicht	view
	Bearbeiten	edit
	Verschlagwortung	indexing
	Versenden	send
	Zwischenablage	clipboard
	Löschen	delete
Dokument		document
	Erweitern	extend
	Ausgabe	print
	Externe Links	docurl
	Überprüfen	verify
	Randnotiz	notes
	Dateianbindung	attachment

Archiv		archive
	Ansicht	view
	Link	link
	Information	information
	Struktur	structure
	Einfügen	insert
	Export / Import	export
	Verwaltung	control
	Papierkorb	recyclebin
Ansicht		view
	Funktionsbereiche	functionalAreas
	Archivanzeige	archiveView
	Darstellung	display
	Miniaturansichten	thumbnails
	Ordnen	order
	Vergleichen	compare
	Tabellen	tables
Aufgaben		
	Neu	start
	Bearbeiten	edit
	Organisieren	organize

Vertretung	substitution
Übersichten	information
Anzeige	view
Recherchieren (Suche)	search
Verschlagwortung	indexing
Filter	filters
Verlauf	-
Favoriten	-
Bereich	searcharea
Neu	new
Ansicht	view
Archivieren (Postbox)	intray
Scannen	scan
Einfügen	insert
Seiten organisieren	organizepages
Verschlagwortung	indexing
Ablage	archiving
Übergabe	transfer
Löschen	delete

11.2 Basisfunktionen

Funktion	Name
Abgelegt von (iSearch Filter)	AddSearchFilterOwner
Ablagedatum (iSearch Filter)	AddSearchFilterFilingDate
Ad-hoc-Workflow	StartAdHocWorkflow
Aktualisieren	RefreshView
Aktualisieren	RefreshSidebar
Allgemeine Randnotiz	InsertNormalNote
Als Standardregister speichern	CreateStandardRegister
Ansicht bearbeiten	RenameActiveView
Ansicht löschen	DeleteActiveView
Ansichten verwalten	ManageViews
Archivablage	InsertIntoArchive
Archivanfang	Home
Auf das Klemmbrett legen	AddToClipboard
Aus dem Suchergebnis entfernen	RemoveFromSearchResult
Aus Volltext entfernen	RemoveFromFulltext
Auschecken und bearbeiten	CheckOut
Automatische Ablage	AutomatedArchiving
Barcode-Erkennung	ExtractBarcodes
Baum	ShowVirtualTree

Funktion	Name
Baumansichten	SelectVirtualTree
Baumansichten bearbeiten	ConfigureVirtualTree
Bearbeiter (iSearch Filter)	AddSearchFilterEditor
Beenden	Quit
Berechtigungen auflisten	PermissionLists
Berechtigungen setzen	SetPermissions
Checksumme prüfen	Checksum
Datei einfügen	InsertFile
Datei speichern unter	SaveFileAs
Dateianbindung hinzufügen	AddAttachment
Dateianbindung löschen	DelAttachment
Dateianbindung speichern unter	SaveAttachment
Dateianbindung zur Ansicht öffnen	ActivateAttachment
Datum (iSearch Filter)	AddSearchFilterDate
Dauerhaft entfernen	DeletePermanent
Dokument aus Vorlage	NewDocument
Dokument bearbeiten	EditDocument
Dokument drucken	Print
Dokument faxen	FaxDocument
Dokument scannen	ScanMultipage

Funktion	Name
Dokument versenden	SendMail
Dokument versenden	SendMailFromSidebar
Dokumentenänderung verwerfen	DeleteDocumentChanges
Dokumentendateien verschieben	MoveDocumentFilingPath
Dokumentversionen	DocVersionsDialog
Dynamischer Ordner	SaveSearchAsFolder
Einchecken	CheckIn
Einen Schritt vor	GoForward
Einen Schritt zurück	GoBackward
Eintrag verschieben	MoveElement
Einträge zählen	CountArchiveElements
Erweitern / reduzieren	ReduceExpandGrouping
Eskalationen	ShowWorkflowTimeEscalations
Export	ExportDialog
Externen Link erstellen	ShareDocumentUrl
Favorit hinzufügen	SaveSearchFavorite
Favoriten verwalten	ManageSearchFavorites
Formulardesigner	ShowFormularDesigner
Fristverlängerung Workflow	SetFlowTimeExtension
Funktionsbereich Archiv aufrufen	ShowArchiveView

Funktion	Name
Funktionsbereich Aufgaben aufrufen	ShowTasksView
Funktionsbereich Bearbeitung aufrufen	ShowCheckoutView
Funktionsbereich Klemmbrett aufrufen	ShowClipboardView
Funktionsbereich Mein ELO aufrufen	ShowMyEloView
Funktionsbereich Postbox aufrufen	ShowIntrayView
Funktionsbereich Suche aufrufen	ShowSearchView
Gehe zu	Goto
Gehe zu	GotoFromSidebar
Gelöschte Einträge einblenden	ShowDeletedElements
Gruppenaufgaben	ShowGroupTasks
Gruppierung	DisplayClustering
Hilfe	Help
Import	ImportDialog
In Volltext aufnehmen	InsertIntoFulltext
Indexfeld (iSearch Filter)	AddSearchFilterIndexfield
Kacheln	DisplayTiles
Klammern (Trennseiten)	StapleBySeparatingPages
Konfiguration	EditOptions
Konfiguration - Technische Voreinstellungen	ELOacOnly_TechnicalConfig
Kopie einfügen	PasteId

Funktion	Name
Kopieren	CopyId
Link Drag and Drop	EcdDragAndDrop
Liste (Ansicht)	DisplayList
Löschen	Delete
Maske (iSearch Filter)	AddSearchFilterMask
Miniaturansichten	ShowThumbnails
Neue Ansicht	AddView
Neue Version	AppendNewVersion
Neue Version laden	NewVersion
Neuer Ordner	AddStructure
Neues Fenster öffnen	Login
Nur aktueller Ordner	SearchInCurrentFolder
Nutzer-Feedback	UserFeedback
Objekttyp (iSearch Filter)	AddSearchFilterType
Passwort ändern	ChangePassword
PDF-Konvertierung	CreatePdfDocument
Permanente Randnotiz	InsertStampNote
Persönliche Randnotiz	InsertPersonalNote
Prozessübersicht	ShowThreadMonitor
Referenz erstellen	ReferenceElement

Funktion	Name
Replikationskreise definieren	EditReplicationSets
Replikationskreise zuordnen	AssignReplicationSets
Report zum Eintrag	ShowReport
Rückgängig	Undo
Scanner auswählen	SelectScanner
Scan-Profile	ScanProfilesDialog
Schriftfarbe	ChangeColor
Seiten anfügen	AppendDocument
Seiten klammern	MergeTiffs
Seiten scannen	ScanSinglepage
Seiten trennen	SplitMultipageTiff
Serienablage	DirectInsertIntoArchive
Signatur erstellen	CreateSignature
Signatur prüfen	CheckSignature
Sortierung (Archiv)	ChangeSortOrder
Sortierung (Ansicht)	DisplaySorting
Speichern als ELO-Link	SaveElementAsEcd
Sperre entfernen	RemoveLock
Standardregister einfügen	InsertStandardRegister
Tabelle (Ansicht)	DisplayTable

Funktion	Name
Tabelle in Zwischenablage	ExportSelectedTableElements
Tabellenspalten wiederherstellen	RestoreTableDefaults
Teilbaum komplett öffnen	OpenSubtree
Teilbaum komplett öffnen	OpenSubtreeFromSidebar
TIFF-Konvertierung	CreateTiffDocument
Über das Programm	AboutDialog
Übersetzungstabelle	ShowTranslationsTable
Übersicht externe Links	DocumentUrlOverview
Übersicht Überwachungen	ElementObservationOverview
Übersicht Wiedervorlagen	ReminderOverview
Übersicht Workflows	WorkflowOverview
Veränderungen überwachen	AddElementObservation
Verfallsdokumente löschen	DeleteDelDateElements
Verlinkung	EditLinks
Verschlagwortung	IndexDialog
Verschlagwortung drucken	PrintIndex
Verschlagwortung durchsuchen	DoSearch
Verschlagwortung löschen	DeleteIndex
Versenden als ELO-Link	SendEcdMail
Versenden als PDF	SendAsPdfDocument

Funktion	Name
Versenden in andere Postbox	SendInOtherIntray
Versionen der Dateianbindung	AttachmentVersions
Vertreter einsetzen	SetSubstitution
Vertretung übernehmen	ApplySubstitution
Vertretungsaufgaben	ShowSubstitutionTasks
Visueller Vergleich	CompareDocuments
Volltextinhalt anzeigen	ShowFulltextContent
Vom Klemmbrett entfernen	RemoveFromClipboard
Vorlagen	NavigateToTemplatesFolder
Vorschau-Dokument erstellen	CreatePreviewDocument
Vorschau-Profile verwalten	ManagePreviewProfiles
Wiederherstellen	RestoreElements
Wiedervorlage	NewReminder
Wiedervorlage ändern	EditReminder
Workflow abgeben	ReleaseWorkflowNode
Workflow annehmen	AcceptWorkflow
Workflow anzeigen	EditFlow
Workflow delegieren	DelegateWorkflowNode
Workflow starten	NewFlow
Workflow weiterleiten	ConfirmFlow

Funktion	Name
Workflow zurückgeben	ReturnWorkflowNodeToGroup
Workflow zurückstellen	DeferWorkflow
Workflows zum Eintrag	WorkflowsOfElement
Workflow-Vorlagen bearbeiten	EditFlowTemplate
Zur Ansicht öffnen	Open
Zurückstellung löschen	CancelDeferredWorkflow

12 Index

Abfrage einer Bezeichnung 32
ActiveX 56
Aktivierung 20
AlertBox 31
Archivablage 57
Archivansichten 23
archive 24
archiveViews 23
Auswahl-Dialog 38
BackgroundJob 49
Basisfunktionen 11
Benutzerauswahl 36
Berechtigungen 37
Breakpoint 5
checkout 23
clientInfo 25
clipboard 23
Com 56
COM Server 59
CommandLinkDialog 38
components 24
Dateisystem 33
Debugger 5
Dialoge 29
Dialogs 24
Dokument auswählen 35
Dokumentenvorschau 24
ECMAScript 4
Einfache Meldung 29
EloComServer 59
Enumeration 28
Events 4, 11, 15
Explizite Aktivierung 21
ExtraView 19
FeedbackMessage 29
FileDialog 33
Funktionsbereich 19, 23
getExtraBands 17
getExtraTabs 19
getScriptButtonPositions 16
Globale Variablen 4
GridDialog 39
GridPanel 44
Größe der Spalten 41
Größe der Zeilen 41
Hintergrundprozesse 48
Icon 15
Include 8
IndexDialog 24
InfoBox 29
InputBox 32
intray 23
ix 25
ixc 25
ixConst 25
Ja-/Nein-Fragen 31
Java Objekte 28
JavaDoc 1
JavaScript 4
Klassendiagramm
  Buttons 27
  Einträge in den Funktionsbereichen 26
  Funktionsbereiche/Sichten 26
  Verschlagwortung 27
Klassenhierachie 26
Komplexe Dialoge 39
Komponenten 44
Kontext 19
Laden 4
log 25
Log4J 25
Lokalisierte Texte 10
Meldungen 29
Ordner auswählen 35
PermissionsDialog 37
Position 18, 19
Postbox 57
Präfix 4
preview 24
Protokoll 48, 50
Prozessübersicht 48, 49
QuestionBox 31
Reihenfolge 4
Ribbon 17, 19
Rückgabewert 12
Rückmeldung 29
Scripting Base 4
Scripting Objekte 23
Script-Schaltflächen 15
searchViews 24
Serverprozesse 48
Setup 59
Skripte in Bearbeitung 4
tasks 23
Tastenkombination 4, 5
TreeSelectDialog 35
UserSelectionDialog 36
Variablen 6
Warnhinweis 31
Weitere Events 13
workspace 23
Zusätzliche Bänder 17

Zusätzliche Tabs 19