

ELO Java Client Scripting

[Stand: 08.04.2013 | Programmversion: 8.04.000]

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.

Die exakte Dokumentation der Schnittstellenobjekte liegt als JavaDoc vor und ist hier nicht wiedergegeben.

Inhalt

1	Technischer Überblick Internes Scripting	3
1.1	Laden der Skripte	3
1.2	Skripte in Bearbeitung	3
1.3	Java Client Scripting Base	3
1.4	Debugger	4
1.5	Variablen	5
1.6	Include	6
1.7	Lokalisierte Texte	8
2	Events	10
2.1	Basisfunktionen	10
2.2	Rückgabewert bei Start Events	10
2.3	Weitere Events	11
3	Schaltflächen in der Multifunktionsleiste (ScriptButtons)	12
3.1	Events	12
3.2	Icon	12
3.3	Name	13
3.4	Position im Client	13
3.5	Zusätzliche Bänder im Ribbon	14
3.6	Zusätzliche Tabs im Ribbon	15

4	Scripting Objekte	17
4.1	Client-Objekte	17
4.2	Klassenhierarchien	20
4.3	Java Objekte	22
5	Meldungen und Dialoge	23
5.1	Rückmeldung (FeedbackMessage)	23
5.2	Einfache Meldung (InfoBox)	24
5.3	Warnhinweis (AlertBox)	25
5.4	Ja-/Nein-Fragen (QuestionBox)	26
5.5	Abfrage einer Bezeichnung (InputBox)	27
5.6	Dokument oder Ordner auswählen (TreeSelectDialog)	28
5.7	Anwenderauswahl (UserSelectionDialog)	29
5.8	Berechtigungen (PermissionsDialog)	30
5.9	Auswahl-Dialog (CommandLinkDialog)	31
5.10	Komplexe Dialoge (GridDialog)	32
6	ActiveX Objekte	36
7	Beispiele	37
7.1	Anzahl der Druckvorgänge zählen	37
7.2	Automatisierte Archivablage	37
8	Externes Scripting	38
8.1	COM Server	38
9	Anhang	39
9.1	Ribbon-Tabs	39
9.2	Ribbon-Bänder / -Gruppen	40
9.3	Basisfunktionen	42
10	Index	47

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 Tastaturkommando <STRG>+<ALT>+R möglich. Es werden anschließend alle Skripte erneut vom Server geladen.

1.2 Skripte in Bearbeitung

Sind Skripte durch den Anwender 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 Anwendern 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.

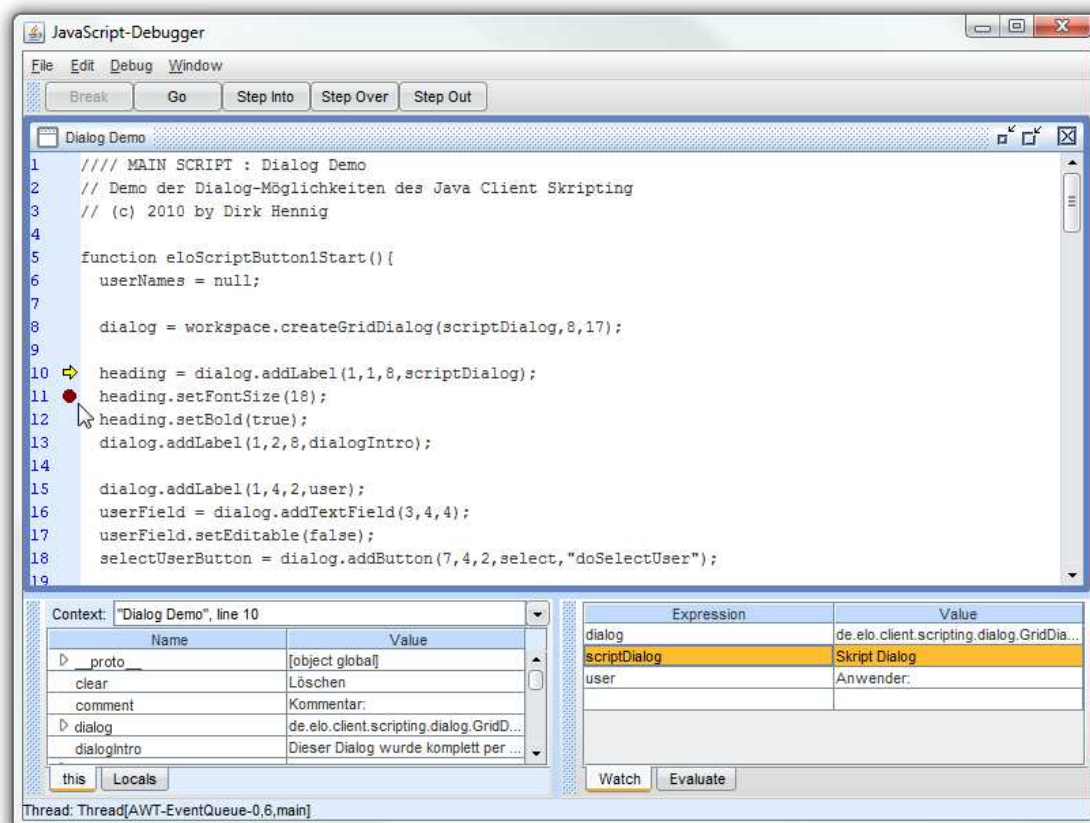
Im Scripting Base können beliebig viele Skripte als Dokumente abgelegt werden. Dabei können mit den üblichen Zugriffsrechte in ELO (ACL) die Skripte einzelnen Benutzern oder Gruppen zugeordnet werden. Der Java Client lädt beim Start nur genau die Skripte, auf welche der angemeldete Benutzer Leserechte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

1.4 Debugger

Ein Rhino **JavaScript-Debugger** kann per Option oder Tastaturbefehl <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.



Bitte beachten Sie, dass Breakpoints in Dialogen zu einem Deadlock führen, 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.

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 );
}
```

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";
}
```

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 zusammengekopierte Skript-Text angezeigt.

Beachten Sie, dass sich die Schreibweise mit der Java Client Version 8.01.000 geändert hat.

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-Skript for include statement

//@include lib_InfoDialog

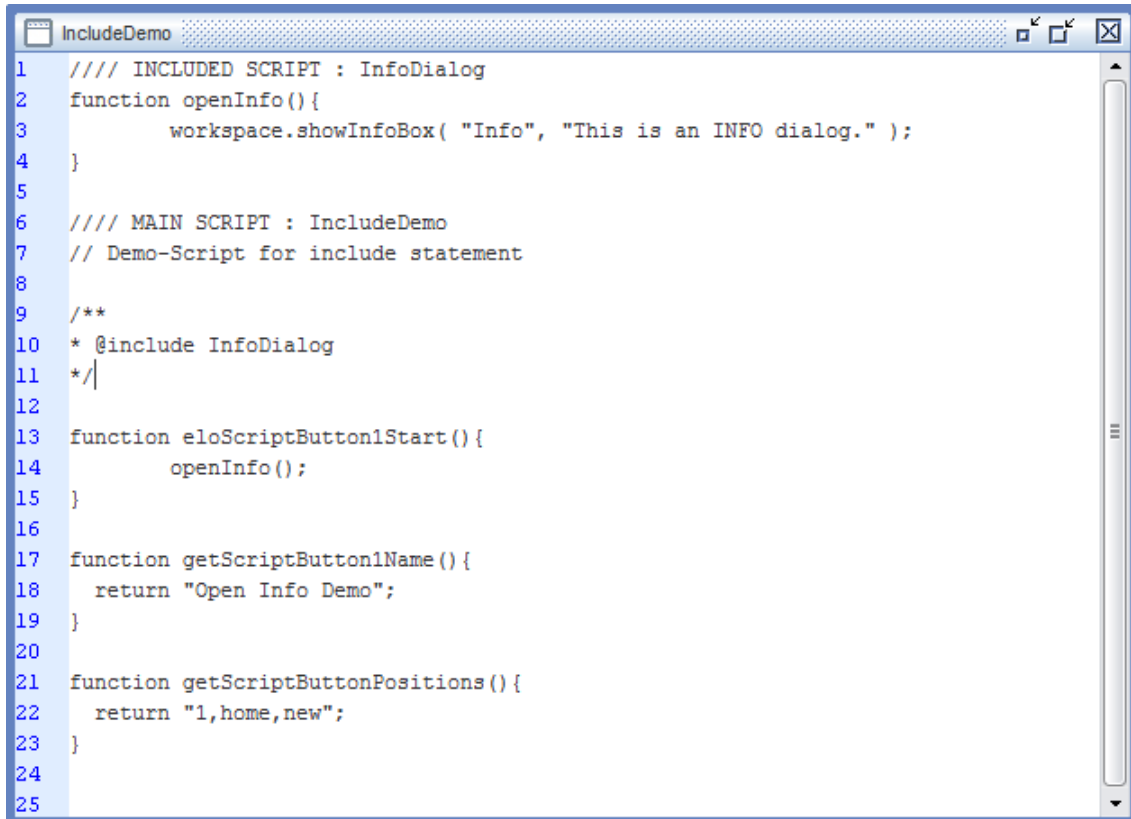
function eloScriptButton1Start(){
    openInfo();
}

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 Skript benutzt werden.

Im Debugger sieht man die zusammenkopierten Textinhalte der beiden Skriptdateien:



```
1  //// INCLUDED SCRIPT : InfoDialog
2  function openInfo(){
3      workspace.showInfoBox( "Info", "This is an INFO dialog." );
4  }
5
6  //// MAIN SCRIPT : IncludeDemo
7  // Demo-Script for include statement
8
9  /**
10 * @include InfoDialog
11 */
12
13 function eloScriptButton1Start(){
14     openInfo();
15 }
16
17 function getScriptButton1Name(){
18     return "Open Info Demo";
19 }
20
21 function getScriptButtonPositions(){
22     return "1,home,new";
23 }
24
25
```

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 Datei 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 Beispiel DE für deutsch, EN für Englisch, FR für Französisch.

Die Textdatei 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 nachfolgend. Dabei ist die Funktion mit ihrer Schreibweise in der Multifunktionsleiste angegeben.

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 des Druckfunktion einen Dialog, welcher nachfragt, ob wirklich gedruckt werden soll. Klickt der Nutzer dann *Nein*, so wird das Drucken abgebrochen. Bei einem Mausklick auf *Ja* funktioniert die Druckfunktion wie gewohnt.

```
function eloPrintStart() {  
    if ( !workspace.showQuestionBox( "Frage",  
        "Möchten sie das Dokument wirklich drucken?" ) ) {  
        return -1;  
    }  
}
```

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.

3 Schaltflächen in der Multifunktionsleiste (ScriptButtons)

Neben den Basisfunktionen des Java Clients gibt es die Möglichkeit Schaltflächen ohne eigene Funktion der Toolbar hinzuzufügen. Dies sind die sogenannten Script-Schaltflächen. Obwohl der Name „Script-Button“ sich auf die Toolbar bezieht, können diese – genau wie die Basisfunktionen – auch als Einträge in den Menüs und Kontextmenüs verwendet werden.

Jede Script-Schaltfläche ist mit einer Nummer gekennzeichnet. Es stehen die Nummern 0 bis 999, also tausend unterscheidbare Script-Schaltflächen zur Verfügung. Die in der Menü- und Toolbarkonfiguration angezeigte 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 folgen 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 in der Toolbar 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 „ScriptButton1_16“.

3.3 Name

Die Script-Schaltflächen können nicht nur als Icon in der Toolbar liegen, sondern auch per Konfiguration in das Hauptmenü oder ein Kontextmenü gelegt werden. Da hier keine Icons angezeigt werden, ist es wichtig, der Schaltfläche auch einen Namen zu geben. Dieser wird auch als Tooltip in der Toolbar oder als Text unter dem Icon in der Multifunktionsleiste angezeigt.

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

Bei der „klassischen“ Ansicht mit Hauptmenü und Toolbar müssen die Script-Schaltflächen per Konfiguration in ein Menü oder eine Toolbar eingetragen werden.

Wird die Multifunktionsleiste (Ribbon) verwendet, dann kann die Position in der Leiste 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-Reiten 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.

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 im Ribbon

Wenn die im Ribbon vorhandenen Bänder 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 Bänder 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.

Bitte beachten Sie, dass leere Bänder nicht angezeigt werden. 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 Ribbon werden dann die Skript-Schaltflächen 5 und 6 eingefügt.

```
function getExtraBands(){  
    return "home,21,Beispiel";  
}  
  
function getScriptButtonPositions(){  
    return "5,home,Beispiel;6,home,Beispiel";  
}
```

3.6 Zusätzliche Tabs im Ribbon

Das Anlegen zusätzliche Leisten im Ribbon 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.

Beispiel:

Das folgende Skript fügt einen neuen Tab *Voting* hinter dem Tab *Workflow* in das Ribbon ein. Diesem werden zwei Bänder hinzugefügt: *Start* und *Bewertung*. In *Start* kommt die Skript-Schaltfläche 10 mit dem Namen *Voting starten*, in *Bewertung* kommen Skript-Schaltfläche 11 *Gut* und 12 *Schlecht*.

```
function getExtraTabs(){
    return "41,Voting";
}

function getExtraBands(){
    return "Voting,10,Start;Voting,20,Bewertung";
}

function getScriptButtonPositions(){
    return "10,Voting,Start;11,Voting,Bewertung;12,Voting,Bewertung ";
}

function getScriptButton10Name(){
    return "Voting starten";
}

function getScriptButton11Name(){
    return "Gut";
}

function getScriptButton12Name(){
    return "Schlecht";
}
```


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.

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**

archiveViews

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

Klasse: **ArchiveViews**

intray

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

Klasse: **IntrayAdapter**

checkout

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

Klasse: **CheckoutAdapter**

tasks

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

Klasse: **TasksAdapter**

clipboard

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

Klasse: **ClipboardAdapter**

searchViews

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

Klasse: **SearchViews**

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**

dialogs

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

Klasse: **DialogsAdapter**

components

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

Klasse: **ComponentsAdapter**

indexDialog

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

Klasse: **IndexDialogAdapter**

preview

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

Klasse: **PreviewAdapter**

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**

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());
```

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());
```

ixConst

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

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

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:

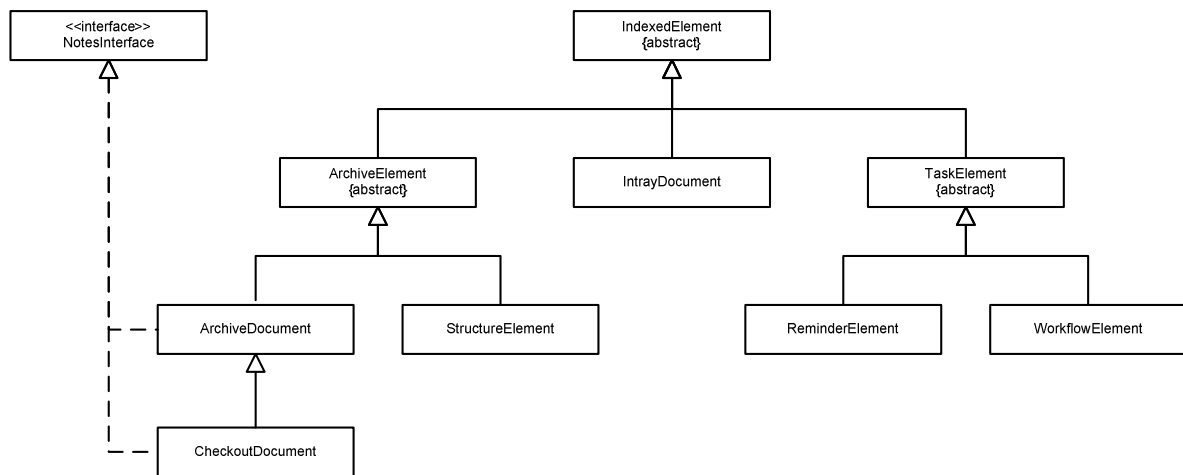
```
log.info( "Funktion gestartet" );  
log.debug( "Technische Details: FctNo=123" );  
log.warn( "Fehler beim Laden", exception );
```

4.2 Klassenhierarchien

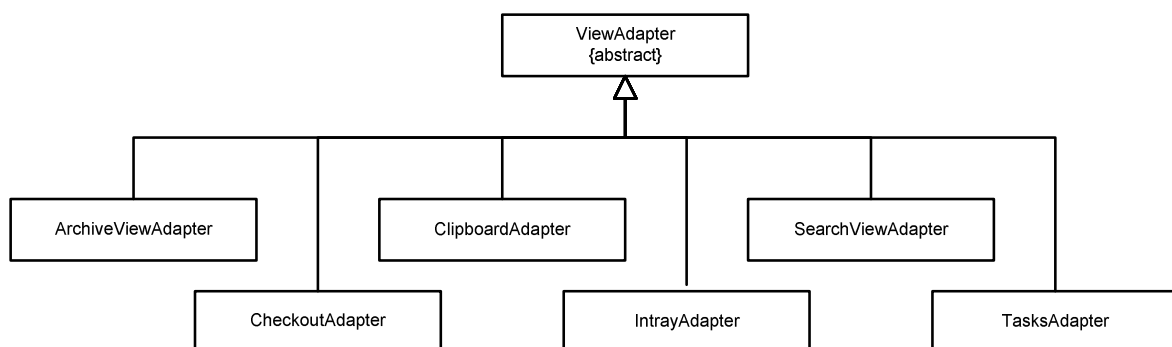
Von den Client-Objekten der Funktionsbereiche werden die Ordner, Dokument etc. als Objekte gekapselt zurückgegeben. Diese implementieren sinnvolle Methoden auf diesen Elementen und ermöglichen somit ein einfaches Arbeiten ohne tiefergehende Kenntnisse der IX-Schnittstelle. Die Objekte sind in objektorientierte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

Die komplette Schnittstellenbeschreibung ist als JavaDoc vorhanden.

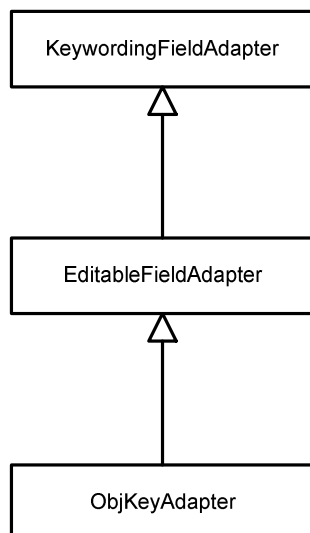
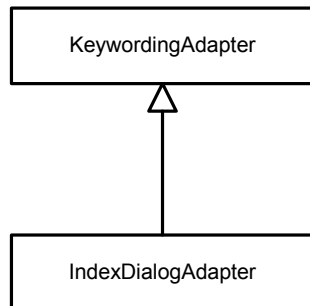
Einträge in den Funktionsbereichen



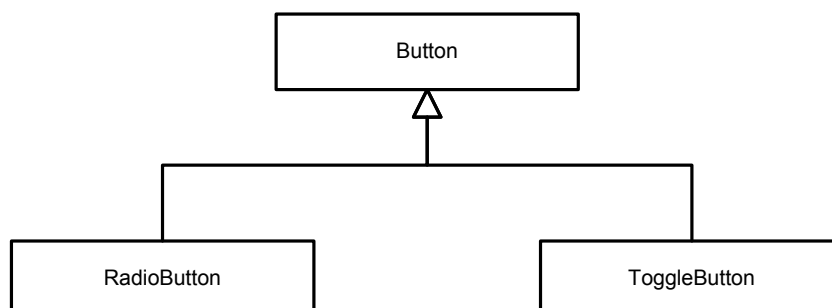
Funktionsbereiche/Sichten



Verschlagwortung



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.

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

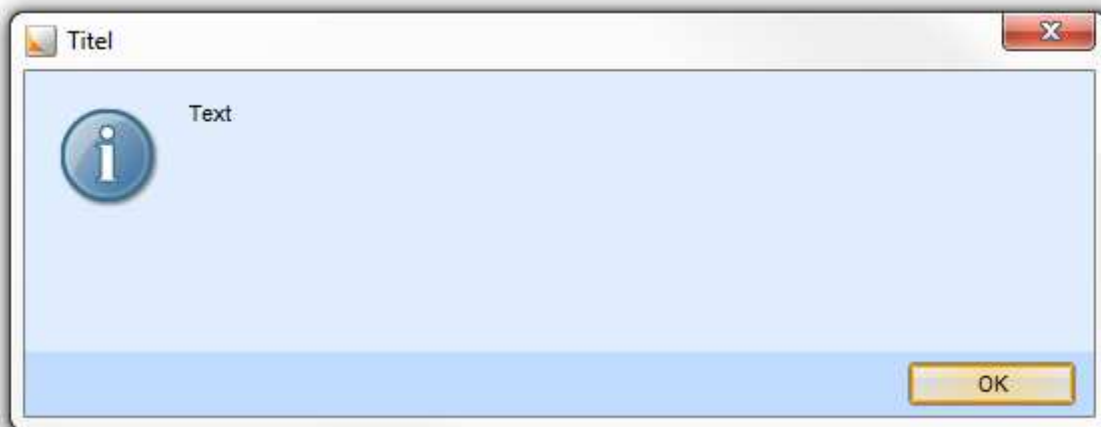
```
workspace.setFeedbackMessage( "Workflow wurde gestartet" );
```



5.2 Einfache Meldung (InfoBox)

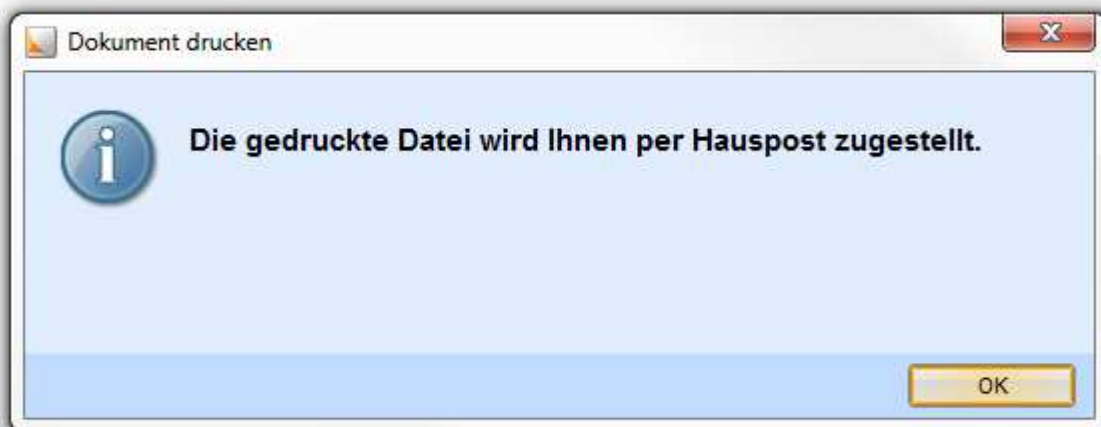
Der einfachste Dialog im Java Client ist die InfoBox, ein Dialog mit Titel, Text, einem festen Info-Symbol und einer Schaltfläche „OK“ zum Schließen. Das folgende Beispiel zeigt den Aufruf und die Parameter der InfoBox:

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



Im Java Client wird in solchen Dialogen die wichtigste Information üblicherweise in größerer, fatter Schrift hervorgehoben. Im Scripting erreichen Sie dies durch die Verwendung von HTML-Text, der Haupttext sollte als Überschrift 3, also mit dem HTML-Tag **<h3>** formatiert sein. Das folgende Beispiel zeigt eine Meldung, welche nach dem Drucken 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>" );
```



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 Anwender einen Hinweis, dass sein Druckkontingent 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>Druckkontingent  
aufgebraucht</h3>Sie können keine weiteren Dokumente drucken, weil Ihr  
Druckkontingent für diesen Monat aufgebraucht ist.<br><br>Nächsten Monat  
können Sie wieder drucken.</html>" );
```



5.4 Ja-/Nein-Fragen (QuestionBox)

Für einfache Fragen an den Anwender, 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 Anwender 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 Anwender vor dem Drucken gefragt, ob er wirklich drucken möchte. Der Rückgabewert wird benutzt, um die Standardfunktion abubrechen. 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;
    }
}
```

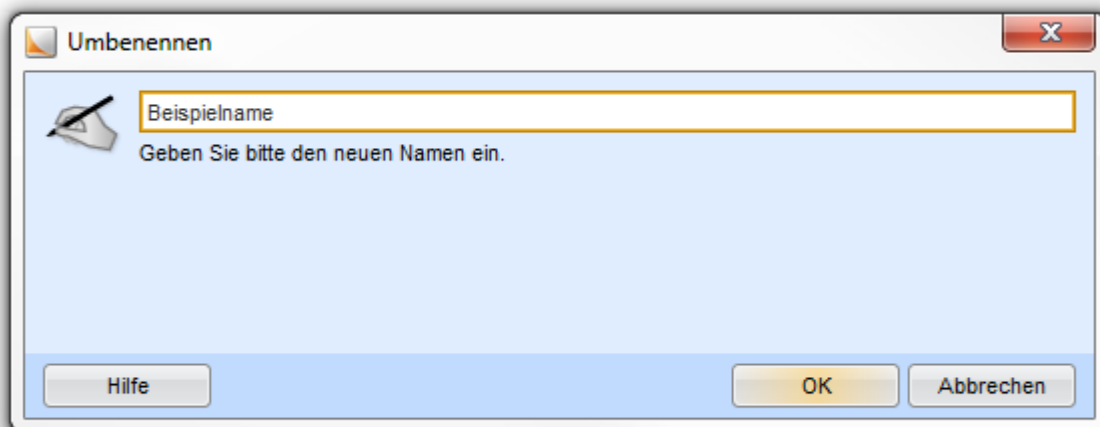


5.5 Abfrage einer Bezeichnung (InputBox)

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 eines Namens beim Umbenennen an.

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

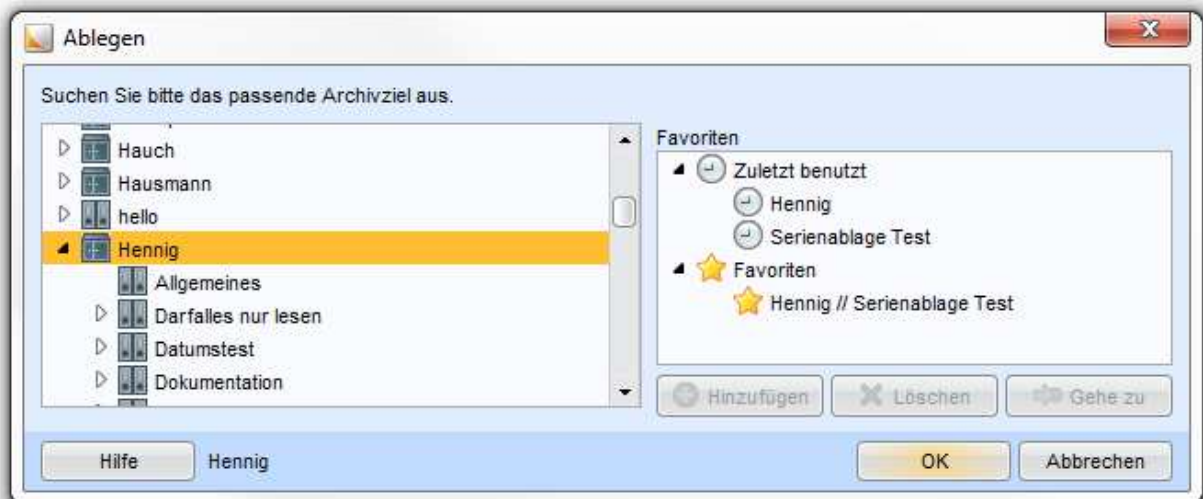


5.6 Dokument oder Ordner auswählen (TreeSelectDialog)

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 werden 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 );
```

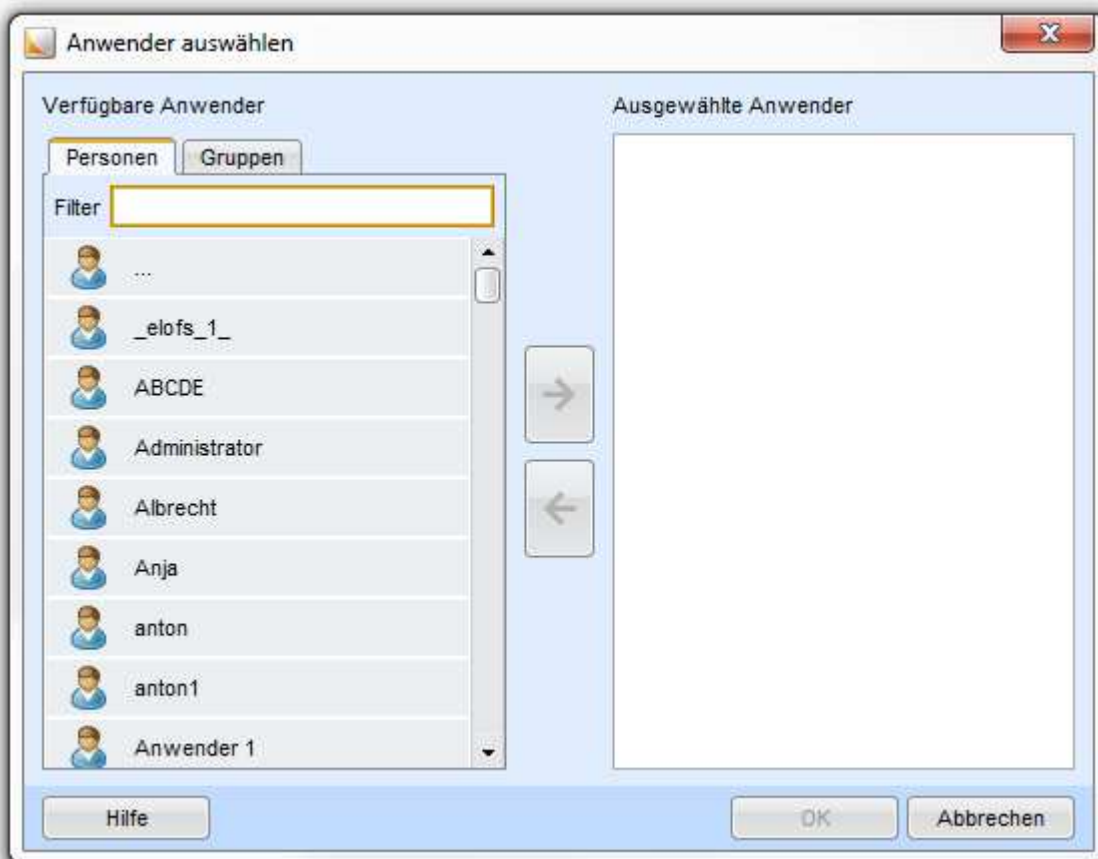


5.7 Anwenderauswahl (UserSelectionDialog)

Für die Auswahl von Anwendern und/oder Gruppen dient der UserSelectionDialog. Es kann angegeben werden, ob Anwender 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 Anwender 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);
```

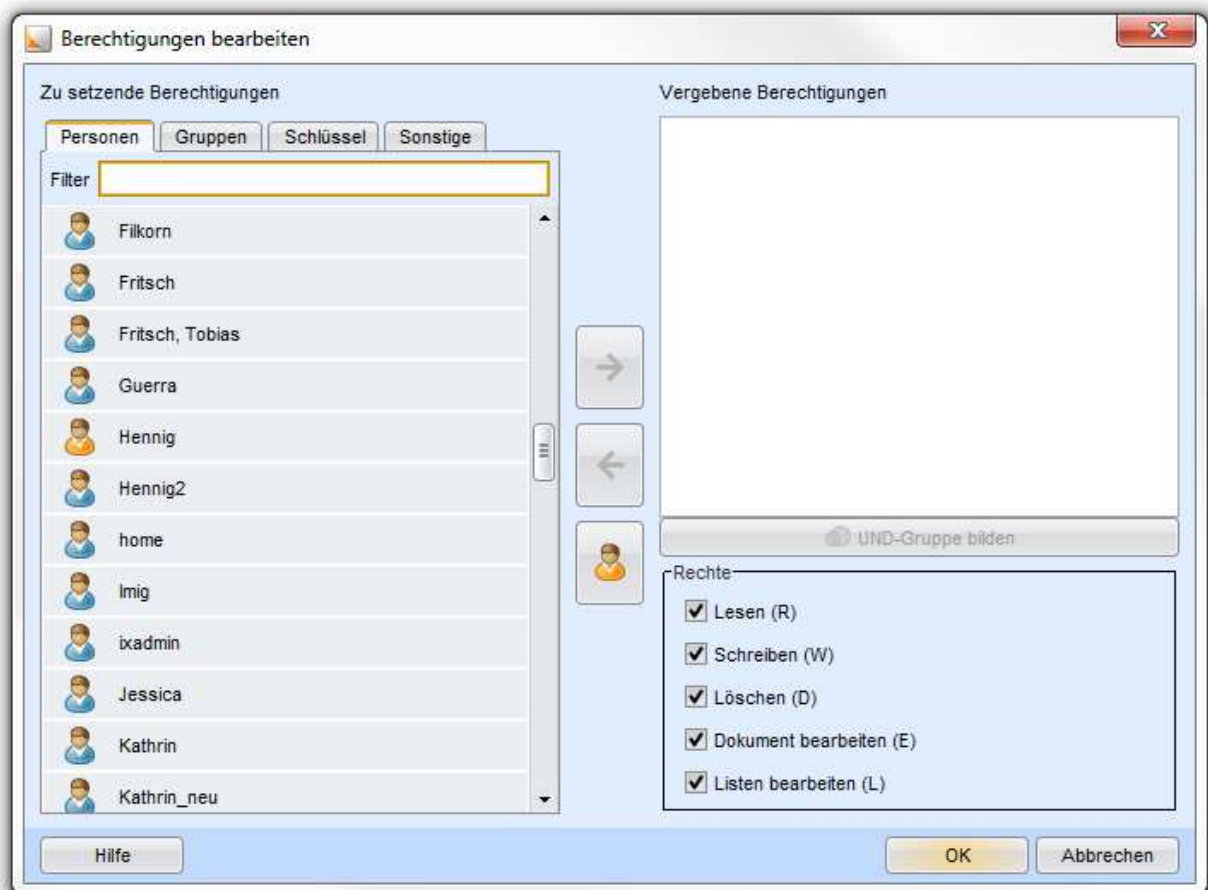


5.8 Berechtigungen (PermissionsDialog)

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  
);
```



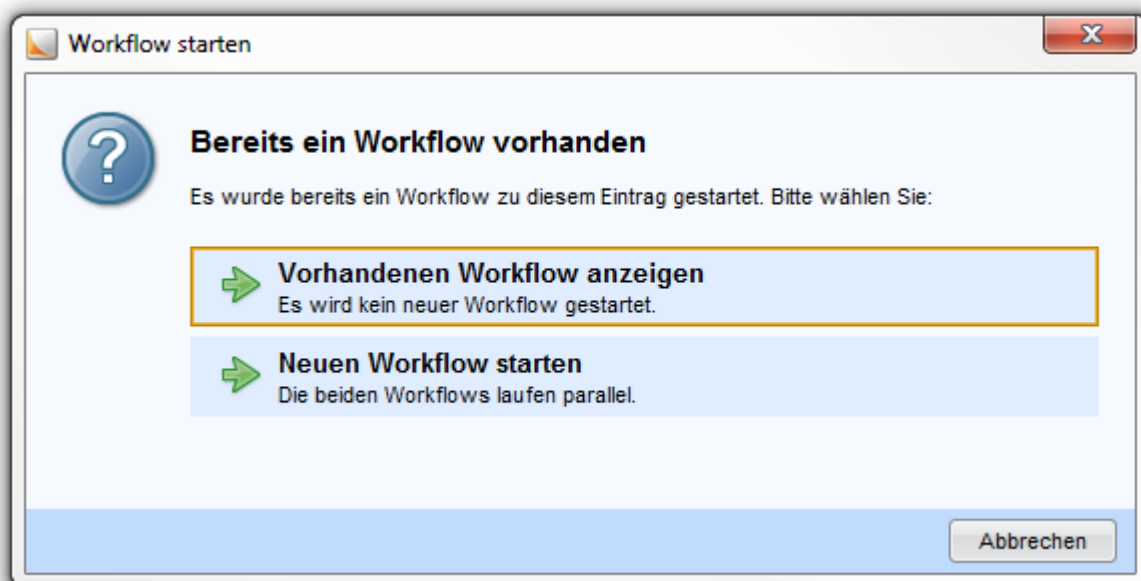
5.9 Auswahl-Dialog (CommandLinkDialog)

In vielen Situationen soll der Anwender zwischen zwei oder mehr Möglichkeiten wählen können und zusätzliche die Möglichkeit haben die Aktion abubrechen. 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 );
```



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

5.10 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 Anwender 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:");
textField = dialog.addTextField(2,2,9);
textField.text = "Neuer Nachrichtenordner";
textField.addChangeEvent( "checkFields" );

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

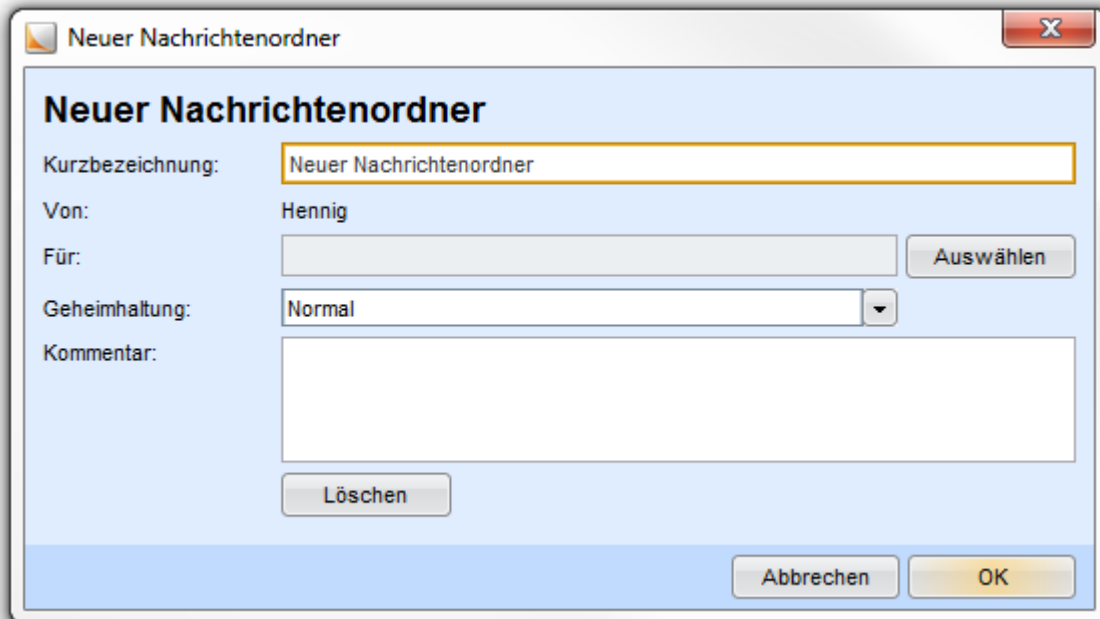
dialog.addLabel(1,4,1,"Für:");
userField = dialog.addTextField(2,4,7);
userField.setEditable(false);
selectUserButton = dialog.addButton(9,4,2,"Auswählen","doSelectUser");

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();
```

Größe der Spalten und Zeilen

Die Breite der Spalten wird so aufgeteilt, dass 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 Anwender überträgt darin einen vorhandenen Standort 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;

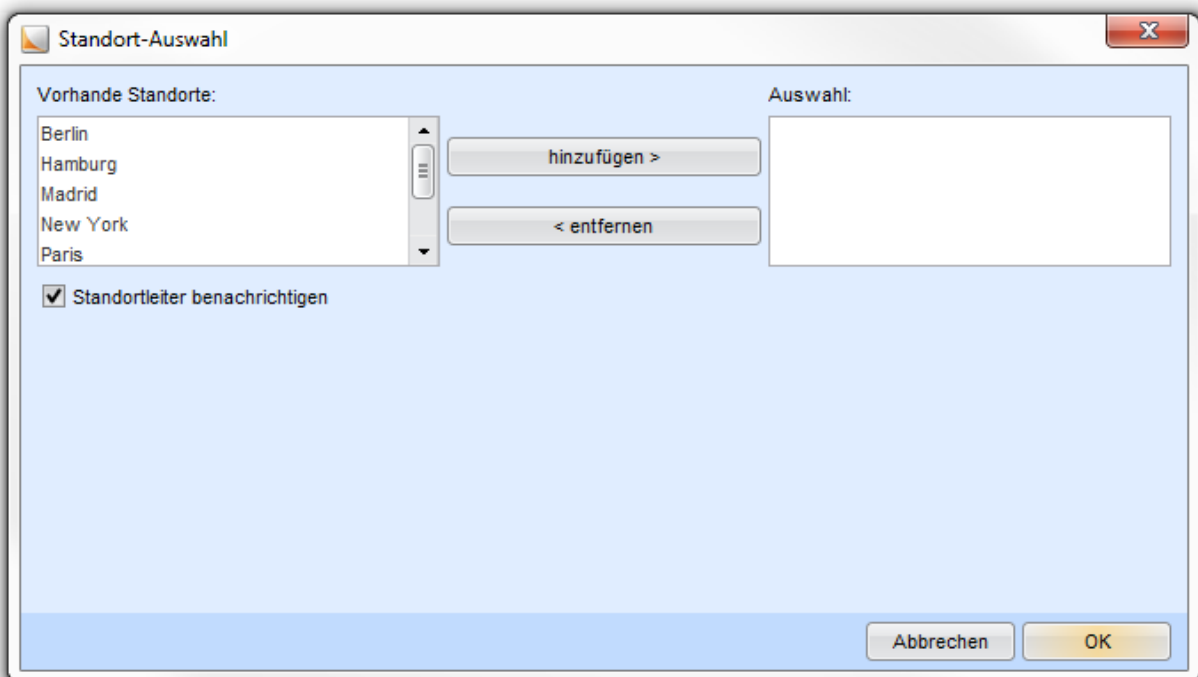
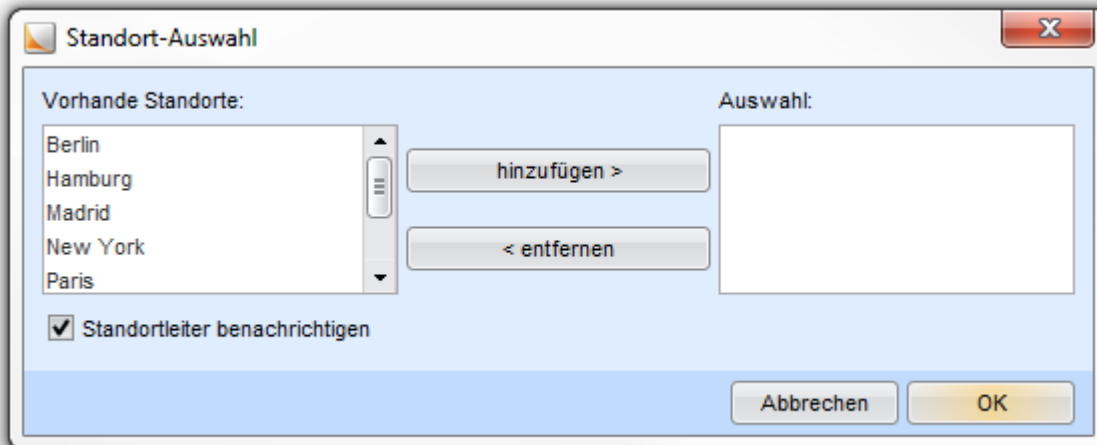
grid.addLabel( 1,1,2, "Vorhandene Standorte:" );
var listVorhanden = grid.addList( 1,2,2,5 );
listVorhanden.setData( [ "Berlin", "Hamburg", "Madrid", "New York",
"Paris", "Rom", "Stuttgart", "Tokio" ] );

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:

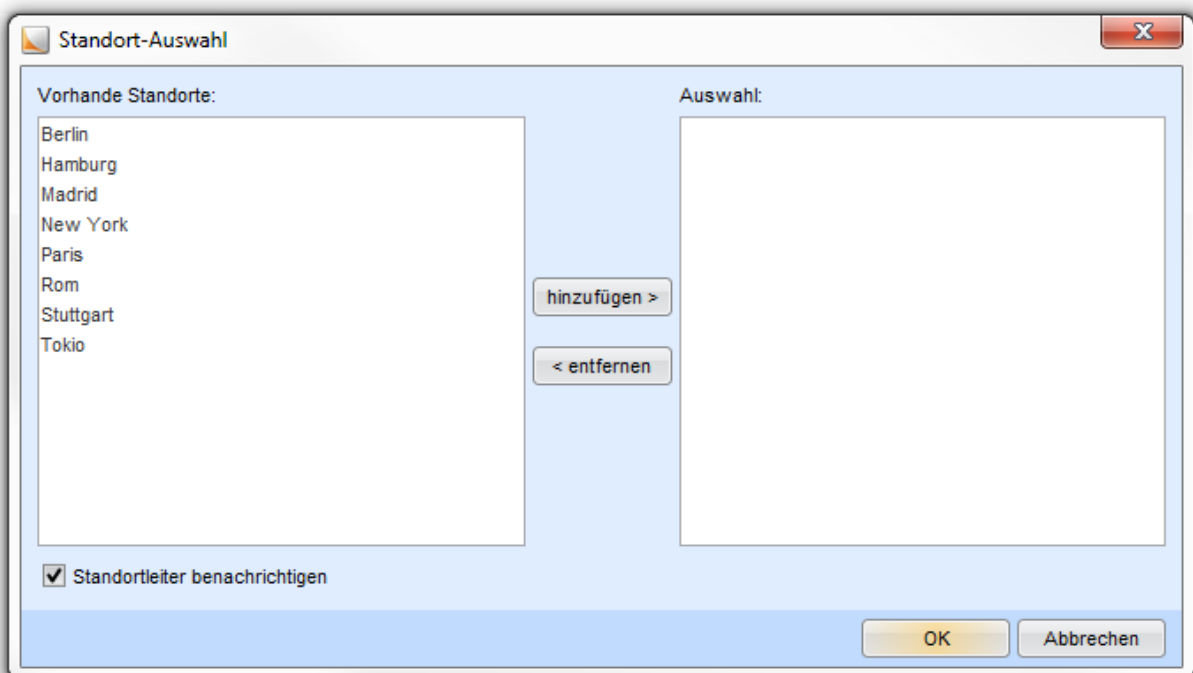
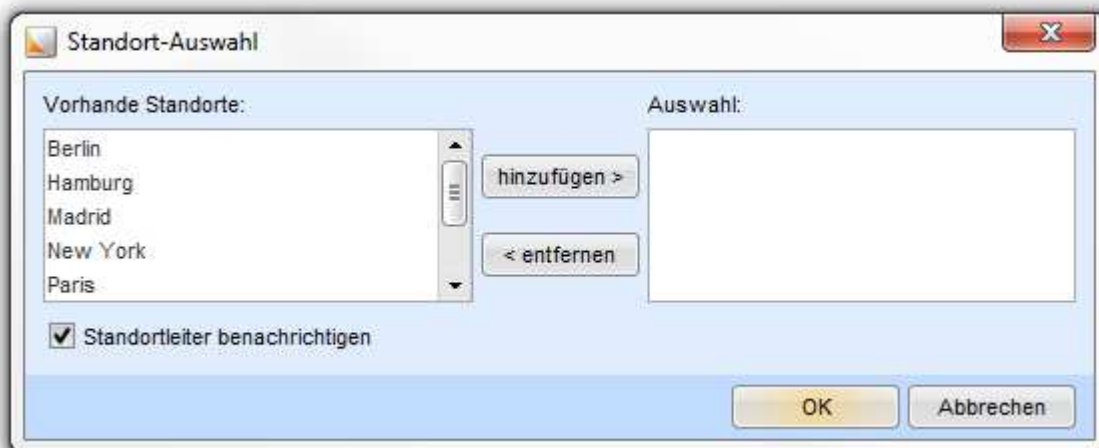


Wie man sieht, wachsen die Spalten und Zeilen bei größerem Dialog wie oben beschrieben. Die vorhandene Fläche wird nicht sinnvoll ausgenutzt.

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.



6 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";
}
```

7 Beispiele

7.1 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." );
}
```

7.2 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 gestartet" );
    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" );
}
```

8 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.

8.1 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)
```

9 Anhang

9.1 Ribbon-Tabs

Name in der Oberfläche	Ribbon-Tab
Ansicht	view
Archiv	archive
Dokument	document
Postboxtools - Archivierung	intray
Start	home
Suchtools - Recherchieren	search
Aufgaben	workflow

9.2 Ribbon-Bänder / -Gruppen

Name in der Oberfläche	Ribbon-Band
Ablage	archiving
Ansicht	view
Anzeige	view
Archivanzeige	archiveview
Ausgabe	print
Bearbeiten	edit
Bereich	searcharea
Darstellung	display
Dateianbindungen	attachment
Einfügen	insert
Erweitern	extend
Export / Import	export
Filter	filters
Funktionsbereiche	workareas
Information (Archiv)	information
Link	link
Löschen	delete
Miniaturansichten	thumbnails
Navigation	navigation
Neu	new
Neu (Aufgaben)	start
Ordnung	order
Organisieren	organize
Papierkorb	recyclebin
Randnotiz	notes
Scannen	scan
Seiten organisieren	organizepages
Struktur	structure

Name in der Oberfläche	Ribbon-Band
Suche	search
Tabellen	tables
Überprüfen	verify
Übersichten (Aufgaben)	information
Vergleichen	compare
Verschlagwortung	indexing
Versenden	send
Vertretung	substitution
Verwaltung	control
Zwischenablage	clipboard

9.3 Basisfunktionen

Funktion	Name
Abgelegt von (iSearch Filter)	AddSearchFilterOwner
Ablagedatum (iSearch Filter)	AddSearchFilterFilingDate
Ad-hoc-Workflow	StartAdHocWorkflow
Aktive Workflows anzeigen	ShowActiveWorkflows
Aktuelle Ansicht löschen	DeleteActiveView
Allgemeine Randnotiz	InsertNormalNote
Als Standardregister speichern	CreateStandardRegister
Ansicht aktualisieren	RefreshView
Ansicht umbenennen	RenameActiveView
Archivablage	InsertIntoArchive
Archivanfang	Home
Archivansicht hinzufügen	AddArchiveView
Archiveinträge zählen	CountArchiveElements
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
Baumansichten	SelectVirtualTree
Baumansichten bearbeiten	ConfigureVirtualTree
Beenden	Quit
Berechtigungslisten anzeigen	PermissionLists
Checksumme prüfen	Checksum
Datei einfügen	InsertFile
Datei speichern unter	SaveFileAs
Dateianbindung hinzufügen	AddAttachment

Funktion	Name
Dateianbindung löschen	DelAttachment
Dateianbindung speichern unter	SaveAttachment
Dateianbindung zur Ansicht öffnen	ActivateAttachment
Datum (iSearch Filter)	AddSearchFilterDate
Dokumentenänderungen verwerfen	DeleteDocumentChanges
Dokument aus Vorlage	NewDocument
Dokument bearbeiten	EditDocument
Dokument drucken	Print
Dokument faxen	FaxDocument
Dokument scannen	ScanMultipage
Dokument versenden	SendMail
Dokument-Versionen	DocVersionsDialog
Einchecken	CheckIn
Einen Schritt vor	GoForward
Einen Schritt zurück	GoBackward
Einfügen	PasteId
Eintrag verschieben	MoveElement
Erweitern / reduzieren	ReduceExpandGrouping
Eskalationen	ShowWorkflowTimeEscalations
Export	ExportDialog
Favorit hinzufügen	SaveSearchFavorite
Favoriten verwalten	ManageSearchFavorites
Gehe zu	Goto
Gruppenaufgaben	ShowGroupTasks
Gruppierung	DisplayClustering
Hilfe	Help
Import	ImportDialog
Indexfeld (iSearch Filter)	AddSearchFilterIndexfield
In Volltext aufnehmen	InsertIntoFulltext
Kacheln (Ansicht)	DisplayTiles

Funktion	Name
Klammern nach Trennseiten	StapleBySeparatingPages
Konfiguration...	EditOptions
Kopieren	CopyId
Link Drag and Drop	EcdDragAndDrop
Liste (Ansicht)	DisplayList
Löschen	Delete
Maske (iSearch Filter)	AddSearchFilterMask
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
Replikationskreise definieren...	EditReplicationSets
Replikationskreise zuordnen	AssignReplicationSets
Report anzeigen	ShowReport
Rückgängig	Undo
Scanner auswählen	SelectScanner
Scan-Profile	ScanProfilesDialog
Schriftfarbe	ChangeColor
Seiten anfügen	AppendDocument
Seiten klammern	MergeTiffs
Seiten scannen	ScanSinglepage

Funktion	Name
Seiten trennen	SplitMultipageTiff
Signatur erstellen	CreateSignature
Signatur prüfen	CheckSignature
Serienablage	DirectInsertIntoArchive
Sortierung (Ansicht)	DisplaySorting
Sortierung (Archiv)	ChangeSortOrder
Speichern als Link	SaveElementAsEcd
Sperre entfernen	RemoveLock
Standardregister einfügen	InsertStandardRegister
Suchansicht hinzufügen	AddSearchView
Tabelle (Ansicht)	DisplayTable
Tabelle in Zwischenablage	ExportSelectedTableElements
Tabellenspalten wiederherstellen	RestoreTableDefaults
Teilbaum komplett öffnen	OpenSubtree
TIFF Konvertierung	CreateTiffDocument
Über das Programm...	AboutDialog
Übersetzungstabelle...	ShowTranslationsTable
Übersicht Überwachungen	ElementObservationOverview
Übersicht Workflows	WorkflowOverview
Veränderungen überwachen	AddElementObservation
Verlinkung	EditLinks
Verschlagwortung	IndexDialog
Verschlagwortung drucken	PrintIndex
Verschlagwortung durchsuchen	DoSearch
Verschlagwortung löschen	DeleteIndex
Versenden als Link	SendEcdMail
Versenden als PDF	SendAsPdfDocument
Versionen der Dateianbindung	AttachmentVersions
Vertreter einsetzen	SetSubstitution
Vertretung übernehmen	ApplySubstitution

Funktion	Name
Vertretungsaufgaben	ShowSubstitutionTasks
Visueller Vergleich	CompareDocuments
Vorlagen	NavigateToTemplatesFolder
Vorschau-Dokument erstellen	CreatePreviewDocument
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
Workflow zurückgeben	ReturnWorkflowNodeToGroup
Workflow zurückstellen	DeferWorkflow
Workflowdauer verlängern	SetFlowTimeExtension
Workflows zum Eintrag	WorkflowsOfElement
Workflow-Vorlagen bearbeiten...	EditFlowTemplate
Zur Ansicht öffnen	Open
Zurückstellung löschen	CancelDeferredWorkflow

10 Index

A

Abfrage einer Bezeichnung 27
ActiveX 36
AlertBox 25
Anwenderauswahl 29
Archivablage 37
Archivansichten 17
archive 18
archiveViews 17
Auswahl-Dialog 31

B

Basisfunktionen 10
Berechtigungen 30

C

checkout 17
clientInfo 19
clipboard 18
Com 36
COM Server 38
CommandLinkDialog 31
components 18

D

Debugger 4
Dialoge 23
Dialogs 18
Dokument auswählen 28
Dokumentenvorschau 18

E

ECMAScript 3
Einfache Meldung 24
EloComServer 38
Enumeration 22
Events 3, 10, 12

F

FeedbackMessage 23
Funktionsbereich 17

G

getExtraBands 14
getExtraTabs 15
getScriptButtonPositions 13

Globale Variablen 3
GridDialog 32
Größe der Spalten 33
Größe der Zeilen 33

I

Icon 12
Include 6, 8
IndexDialog 18
InfoBox 24
InputBox 27
in tray 17
ix 19
ixc 19
ixConst 19

J

Ja-/Nein-Fragen 26
Java Objekte 22
JavaScript 3

K

Klassendiagramm
 Buttons 21
 Einträge in den Funktionsbereichen 20
 Funktionsbereiche/Sichten 20
 Verschlagwortung 21
Komplexe Dialoge 32

L

Laden 3
log 19
Log4J 19

M

Meldungen 23

O

Ordner auswählen 28

P

PermissionsDialog 30
Position 14, 15
Postbox 37
Präfix 3
preview 18

Q

QuestionBox 26

R

Reihenfolge 3

Ribbon 14, 15

Rückgabewert 10

Rückmeldung 23

S

Scripting Base 3

Scripting Objekte 17

Script-Schaltflächen 12

searchViews 18

Setup 38

Skripte in Bearbeitung 3

T

tasks 17

TreeSelectDialog 28

U

UserSelectionDialog 29

V

Variablen 5

W

Warnhinweis 25

Weitere Events 11

workspace 17

Z

Zusätzliche Bänder 14

Zusätzliche Tabs 15