die modellwerkstatt

# Das MoWare Kompendium

Sommer 2014 / MoWare 2.0 für MPS 3.0

Anleitung zum Chefprogrammierer
von Geschäftsanwendungen

Daniel Stieger / Wolfgang Messner
modellwerkstatt.org

<u>Versionen:</u>

* Ver 0.1 - April 2014, initial Version
* Ver 0.2 - Mai 2014, Permissions Anpassungen
* Ver 1.0 - Mai 2014, Dank an Karoline Geißler für s Korrekturlesen

**Inhalt**

**Abbildungen**

## 1. Ein erstes Beispiel

Als erstes Beispiel wollen wir Belege aus einer einfachen Datenbanktabelle am Bildschirm anzeigen. Für dieses Kapitel haben wir den Beleg Lieferaviso gewählt, der unter anderem auch über ein Datumsfeld "Lieferdatum" verfügt. Anhand von diesem Datumsfeld soll der Endanwender suchen können. Er soll bei diesem Beispiel zuerst ein Belegdatum von / bis in einer Eingabemaske angeben (Seite 1), dann sollen alle Lieferaviso, dessen Lieferdatum in diesen Zeitbereich fallen, in einer Liste angezeigt werden (Seite 2). In den nächsten beiden Abbildungen ist das gewünschte Ergebnis zu sehen.

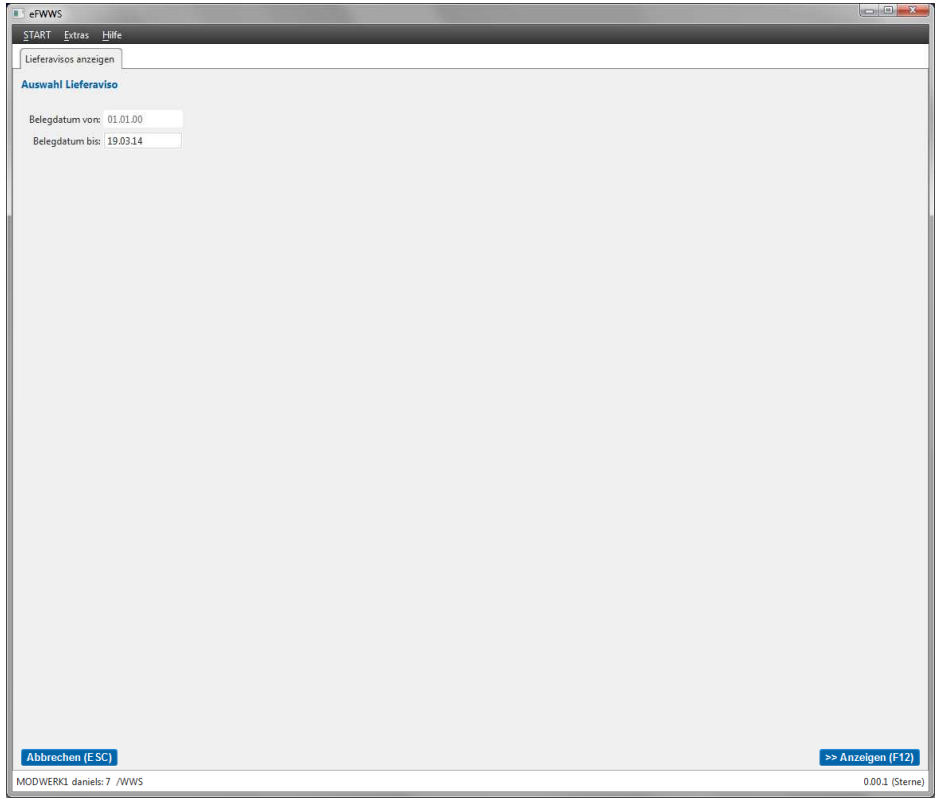



Abbildung 1: Suchmaske
Maske zur Eingabe des Datums, anschließend mit Schaltfläche ">> Anzeigen (F12)" zur Ergebnisliste.

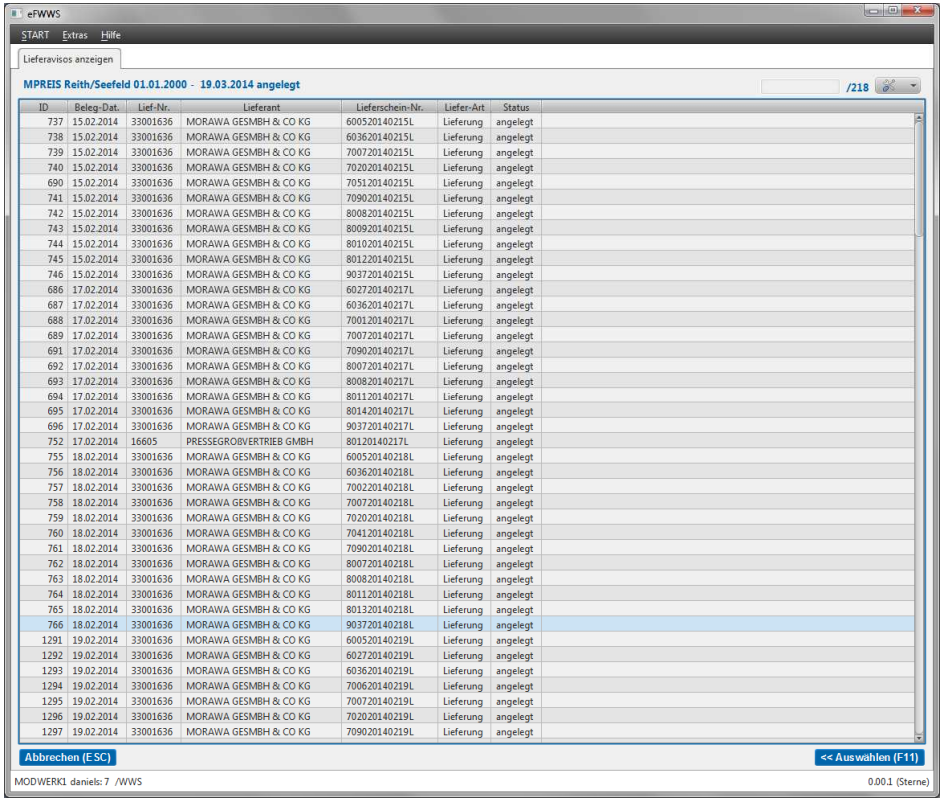



Abbildung 2: Ergebnisliste
Ergebnisliste auf Seite 2 mit mehreren Lieferaviso. Mit Schaltfläche "<< Auswählen(F11)" zurück zur Datum-Eingabemaske.

Um diese Anforderungen mit MoWare umsetzen zu können, sind mehrere Konzepte notwendig. Eine zentrale Rolle nimmt die Entität Lieferaviso ein, die den Beleg selbst modelliert. Ein Kommando "Lieferaviso suchen" bildet den Ablauf Datumseingabe und Ergebnisliste ab. Wie in den Abbildungen zu erkennen, sind für dieses Kommando zwei Seiten notwendig. Zuvor muss allerdings auch die Schnittstelle zur Datenbank modelliert werden. Wie ist ein Lieferaviso in der Datenbank abgebildet (Mapping) und wie kann auf die Datenbank zugegriffen werden (Repository), anders formuliert: Wie können Lieferaviso von der Datenbank geladen werden?

Wenden wir uns erst der Modellierung des Lieferavisos zu. Wie in der nachfolgenden Abbildung zu erkennen ist, wird das Lieferaviso als Entität mit diversen Properties (Datenfelder), einem Konstruktor, einer einfachen Methode (addPos) und zwei Stati angelegt. Eine Entität bezeichnet "etwas, das existiert, ein Seiendes, einen konkreten oder abstrakten Gegenstand" (Wiki). Eine Entität muss zwingend immer einen eindeutigen Schlüssel - meist einen Datenbankschlüssel - ausweisen. Im Fall dieses Belegs wurde ein Integer Datenfeld "id" als Schlüssel gewählt (Option KEY). Dann folgen weitere Datenfelder, auf die nicht im Detail eingegangen werden soll. Beim Vorgangsort handelt es sich um eine Referenz auf eine weitere Entität, bei LocalDate um ein einfaches Datumfeld, bei string um eine Bezeichnung und beim Dokumentenstatus um einen Status - wie der Name bereits sagt. Ein Status kann nur definierte Ausprägungen annehmen. Sie wurden weiter unten im Beleg festgelegt. Schließlich enthält das Lieferaviso auch Positionen, die im Rahmen dieses Beispiels nicht verwendet werden. Die Positionen bleiben daher leer.

Bei den Datenfeldern kann der Anwendungsentwickler jeweils eine Short- und eine Long-Description, sowie ein Format angeben. Long-Descriptions werden als Bezeichnungen in Eingabeformularen angezeigt, wenn also viel Platz verfügbar ist; Short-Descriptions bspw. als Spaltenüberschriften in Tabellen. Auf diese Weise ist gewährleistet, dass Beschriftung und Darstellung in der Anwendung einheitlich erfolgt. Das Format muss grundsätzlich nicht spezifiziert werden, da Voreinstellungen festegelegt sind. Nur in Sonderfällen ist das Format zusätzlich anzugeben.

Auch bei Status Definitionen, bspw. beim Dokumentenstatus, sind Short- und Long-Descriptions anzugeben. Neben dem Wert, der auf der Datenbank gespeichert wird (meist kurz aber nicht selbsterklärend), wird die Bezeichnung in Eingabeformularen und Tabellen angezeigt. In der Dokumentationsspalte kann zusätzlich noch ein längerer Text hinterlegt werden, der in der Oberfläche zur jeweiligen Ausprägung als Tooltip angezeigt wird.

Neben der Entität Lieferaviso wurde für dieses Beispiel auch ein ViewObject "AuswahlBelegAnzeige" modelliert. Bei der AuswahlBelegAnzeige handelt es sich nicht um eine Entität, da diese Auswahl nicht tatsächlich als Dokument oder Beleg existiert. Sie hat in diesem Sinne eher temporären Charakter, wird einmal verwendet und dann verworfen. ViewObjects werden als temporärer Datenspeicher - primär für die Anzeige - verwendet. Es ist daher auch kein eindeutiger Schlüssel festzulegen.

Neben anderen Datenfeldern verfügt die "AuswahlBelegAnzeige" auch über ein Datum "vonDat" und ein "bisDat". Die weiteren Felder werden in diesem einfachen Einführungsbeispiel nicht verwendet. Sie bieten grundsätzlich Speicherplatz für weitere Auswahlkriterien. Interessanterweise wurde die toString() Methode überschrieben. Eine Zeichenkette wird mit einem StringBuilder zusammengestellt, so dass der aktuelle Inhalt des ViewObject's - die aktuelle Auswahl - einfach ausgegeben werden kann.

Entity LieferAviso extends <no superclass>

```
// LieferAviso angelehnt an EDI Standard, ist DokumentStatus verschieden von ProzessStatus        //
   TODO: Systembweite eindeutige Belegnummer, dadurch möglich: entstanden durch Vorgängerbeleg Nr.
```

properties of Entity:

| Type | Name | Short Desc | Long Desc | Format | Options | Documentation |
|---|---|---|---|---|---|---|
| int | id | "ID" | "ID" | | KEY | ... |
| VorgangsOrt | lieferant | ... | "Lieferant" | ... | ... | // TODO: GLN fuer Partei zusaetzlich in Entity // aufnehmen? Nachweis zum Zeitpunkt des erzeugens bestimmte GLN |
| VorgangsOrt | besteller | ... | "Besteller" | ... | ... | // Meist Zentrale. Filiale bestellt bei Zentrale, die // ergänzt und bestellt bei Lieferant Bei MPREIS ist Besteller immer Rechnungsempfänger |
| VorgangsOrt | warenEmpfaenger | ... | "Warenempfaenger" | ... | ... | // Optional: Nur ausgefuellt, wenn // sich Besteller und Warenempfaenger unterscheider |
| LocalDate | ausstellDatum | "Beleg-Dat." | "Belegdatum" | ... | ... | // Datum vom Lieferschein/ bzw. Bestellung // |
| LocalDate | lieferDatum | "Liefer-Dat." | "Lieferdatum" | ... | ... | // Geplante Anlieferungsdatum // |
| string | lsNummer | "Lieferschein-Nr." | "Lieferscheinnummer" | ... | ... | ... |
| Lieferart | lieferArt | "Liefer-Art" | "Liefer-Art" | ... | ... | // TODO: gegenwärtig einfach aus EH2000 uebernommben // |
| DokumentenStatus | dokumentStatus | "Status" | "Dokumentstatus" | ... | ... | ... |
| list<LieferAvisoPos> | pos | ... | ... | ... | ... | ... |

object identity (overwrite equals() method):

<use standard hash>

members:

```
public LieferAviso() {
  <no statements>
}
public void addPos(LieferAvisoPos p) {
  this.pos.add(p);
}
```

status declarations:

status DokumentenStatus default=

| Name | DB value | Short label | Long label | Documentation |
|---|---|---|---|---|
| Angelegt | AN | angelegt | angelegt | <no doc> |
| Bearbeitet | BE | bearbeitet | bearbeitet | <no doc> |
| Abgeschlossen | AB | abgeschlossen | abgeschlossen | <no doc> |
| Fehler | FE | fehler | fehler | <no doc> |

status Lieferart default=

| Name | DB value | Short label | Long label | Documentation |
|---|---|---|---|---|
| Normal | L | Lieferung | Lieferung | <no doc> |
| Nach | NL | Nachlieferung | Nachlieferung | <no doc> |
| Ersatz | EL | Ersatzlieferung | Ersatzlieferung | <no doc> |
| VN | VN | (unbekannt) | (unbekannt) | <no doc> |

Abbildung 3: Lieferaviso

ViewObject AuswahlBelegAnzeige

// View für die Anzeige Aviso Auswahl //

properties of ViewObject:

| Type | Name | Short Desc | Long Desc | Format | Options | Documentation |
|---|---|---|---|---|---|---|
| VorgangsOrt | filiale | ... | "Filiale" | ... | ... | // View für die Filial Auswahl // |
| VorgangsOrt | lieferant | ... | "Lieferant" | ... | ... | // View für die Lieferanten Auswahl // |
| LocalDate | datVon | "BelegDatVon" | "Belegdatum von" | ... | ... | // Beleg Datum von // |
| LocalDate | datBis | "BelegDatBis" | "Belegdatum bis" | ... | ... | // Beleg Datum bis // |
| DokumentenStatus | docStatus | "Status" | "Status" | ... | ... | ... |

object identity (overwrite equals() method):
<use standard hash>

members:

```
public AuswahlBelegAnzeige() {
  <no statements>
}
@Override
public String toString() {
  StringBuilder b = new StringBuilder();
  b.append(filiale.bezeichnung);
  if (lieferant != null) {
    b.append(" / " + lieferant.bezeichnung);
  }
  b.append(" " + UserEnvironmentInformation.dateOnlyFormatter.print(datVon) + " - ");
  b.append(" " + UserEnvironmentInformation.dateOnlyFormatter.print(datBis) + " ");
  if (docStatus != null) {
    b.append(this.docStatus#Meta.getLongText(docStatus));
  }
  b.toString();
}
```

status declarations:
<< ... >>

Abbildung 4: AuswahlBelegAnzeige

Nachdem die Datenobjekte modelliert wurden, können wir uns direkt der Oberfläche widmen. Alle Oberflächenelementen werden als Forms abgebildet.
Die Gestaltung beschränkt sich im Wesentlichen auf eine Ergebnistabelle (Tabelform) und ein Suchformular (DelegateForm). Die folgenden Abbildungen zeigen dieses beiden Formulare.

```
FormContainer FCAuswahlBeleg (AuswahlBelegAnzeige as boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["1*"]
  label: "Auswahl Lieferaviso"
<no command trigger>

DelegateForm DFAuswahlBeleg (AuswahlBelegAnzeige as boundObject loaded with selected: AuswahlBelegAnzeige.<property>)
  col(|): ["-1"]
  label: <no lable>
  LocalDateDelegate f1 setProperty(boundObject.datVon), setEnabled(false) ;
  LocalDateDelegate f2 setProperty(boundObject.datBis) ;

  <no onLoad>
  <no onStore>
  <no onValidate>
```

Abbildung 5: Suchformular

Definition des Suchformulars als DelegateForm mit einfachen Editoren für LocalDate

```
FormContainer FCLieferAvisoListView (list<LieferAviso> with boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["1*"]
  label: <no heading>
<no command trigger>

TableForm TFAvisoKoepfe (list<LieferAviso> with boundObject loaded with selected: <class>.<property>)
  label: <no heading>
  select first: false
  <advanced selections>

  MenuButton " "  (image: <no imageString> )
    on trigger run LieferAvisoProc.Lieferaviso bearbeiten(getSelected(LieferAviso.class))
      view for page: AusgAviso = LADetail
    hk: UNDEFINED

    on trigger run Wareneingang.Wareneingang aus Lieferaviso(null, getSelected(LieferAviso.class).id)
      view for page: Standard = LBDetail
    hk: UNDEFINED



  setProperty(boundObject.id) setWidth(60) <overwrite label> <overwrite format> ;
  setProperty(boundObject.ausstellDatum) setWidth(80) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferant.partei#Key) setWidth(80) "Lief-Nr." <overwrite format> ;
  setProperty(boundObject.lieferant.bezeichnung) setWidth(200) "Lieferant" <overwrite format> ;
  setProperty(boundObject.lsNummer) setWidth(120) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferArt) setWidth(80) <overwrite label> <overwrite format> ;
  setProperty(boundObject.dokumentStatus) setWidth(80) <overwrite label> <overwrite format> ;
```

Abbildung 6: ErgebnisTabelle

Definition der Ergebnistabelle (TableForm) mit 7 Spalten und zwei CommandTrigger, die das in der Tabelle markierte LieferAviso bearbeiten.

Das Suchformular DFAuswahlBeleg ist sehr einfach gestaltet. Im Wesentlichen wurden im DelegateForm zwei Eingabefelder (Delegates) festgelegt, wobei das datVon gesperrt wurde (setEnabled(false)). Das Formular erwartet ein AuswahlBelegAnzeige Objekt und stellt dieses unter dem Variablennamen "boundObject" zur Verfügung. Die beiden Eingabefelder werden direkt an datVon und datBis des boundObject gebunden - daher der Name. Offensichtlich muss bei der Verwendung dieses Formulars dann ein AuswahlBelegAnzeige Objekt übergeben werden. Das Formular zeigt die aktuellen Daten aus dem Objekt an und speichert Änderungen durch den Benutzer in das Objekt zurück. Das Suchformular selbst ist in ein FormContainer eingebettet. Er bildet einen äußeren Rahmen mit einer Überschrift. Grundsätzlich werden mit FormContainer Layoutspezifikationen vorgenommen. In diesem Fall lediglich "1*", d.h. maximale Größe anwenden.

Die Ergebnistabelle ist analog aufgebaut. Die TabelForm TFAvisoKoepfe erwartet eine Liste von Lieferaviso, wobei ein einzelnes Objekt Lieferaviso wieder unter dem Variablennamen boundObject zur Verfügung steht. Die Spalten der Tabelle werden einfach mit setProperty() festgelegt. Wie im Formular werden die Bezeichnungen direkt aus der Objektdefinition (Short- /Longdescription) übernommen. Auf die Beschreibung eines MenuButton soll an dieser Stelle nicht eingegangen werden - dazu später mehr. Die Tabelle ist wiederum in einen FormContainer eingebettet, der an dieser Stelle allerdings auch keine besonderen Layouteigenschaften spezifiziert.

Was in diesen Oberflächenkonzepten bisher nicht modelliert wurde, sind die Schaltflächen für Anzeigen und Auswählen (siehe Abbildung 1: Suchmaske und Abbildung 2: Ergebnisliste). Diese müssen in einem Kommando festgelegt werden, das meist auch die Geschäftslogik aufnimmt. In den folgenden Abbildungen ist das entsprechende Kommando dargestellt.

```
command 'Lieferavisos anzeigen' in process LieferAvisoProc with LieferAviso doc  OR null (set proc doc first)!

command parameter:
<< ... >>

local variables:
AuswahlBelegAnzeige auswahlBelegAnzeige


<docu>


command settings:
command type:               SEARCH_VIEW (new session)
enabled if:                 <cond>
question on external abort: <msg>
title addon:                <msg>
command icon:               HafinaDefaults.ICON_SEARCH

command init:
func()->void {
  LocalDate actDate = new_LocalDateFromServer();
  auswahlBelegAnzeige = {AuswahlBelegAnzeige with <doc>
    datBis : actDate.plusWeeks(1)
    datVon : 1.1.2000
  };
}


command pages:
page 'SearchView'
  form bound to list< AuswahlBelegAnzeige > as  form
  dynamic page title: <title>

  page init:
    pageLoadFunc()->Object {
      auswahlBelegAnzeige;
    }

  set scopes for page:
    pageSetScopesFunc()->void {
      <no statements>
    }


  page conclusions:
    conclusion '>> Anzeigen'  (enabled if: <cond>)
      request 'save data' from page form: save hotkey: NEXT
      func()->void {
        page ListView (run page init)
      }
```

Abbildung 7: Kommando Lieferavisos anzeigen

```
page 'ListView'
  form bound to list< LieferAviso > as  form
  dynamic page title: auswahlBelegAnzeige.toString()

  page init:
    pageLoadFunc()->Object {
      list<LieferAviso> lieferAvisos = call EingangsBelegRepo.
          findeLieferAviso(auswahlBelegAnzeige.datVon, auswahlBelegAnzeige.datBis) ;

      lieferAvisos;
    }

  set scopes for page:
    <no scopeConceptFunc>


  page conclusions:
    conclusion '<< Auswählen'  (enabled if: <cond>)
      request 'save data' from page form: save hotkey: BACK
      func()->void {
        page SearchView (run page init)
      }


FINAL_OK_CONCLUSION:
  <func>

  check process, then: DO_NOT_COMMIT_SESSION
  notification: <msg>
  selection(s)/update(s) on parent: <no finalSelection>


FINAL_CANCEL_CONCLUSION: // do revert first
  <func>


EXCEPTION_CONCLUSION: exception // do revert first
  <func>
```

Abbildung 8: Kommando Lieferavisos anzeigen (Fortsetzung)

Auf den Screenshots eingangs dieses Kapitels, ist im Reiter bereits zu erkennen, dass das Kommando "Lieferavisos anzeigen" ausgeführt wird. Das Kommando selbst beinhaltet nur sehr wenig Geschäftslogik, da lediglich die Ergebnisliste zu verwalten ist. Beginnend von oben nach unten kann folgendes festgehalten werden:

- Das Kommando ist dem Prozess "LieferAvisoProc" zugeordnet - mehr zu Prozessen später. Grundsätzlich muss jedes Kommando einem Prozess zugeordnet werden, der das Kommando steuert (Freigaben, Rechte etc.). Jeder Prozess verwaltet ein Prozessdokument. In diesem Beispiel verwaltetet der Prozess LieferAvisoProc ein LieferAviso, das auch im Kommando als Variable doc zur Verfügung steht. Die gelbe Markierung im Kommando zeigt an, dass die Variable doc evtl. nicht gesetzt wurde und daher null sein könnte. Bei diesem Kommando ist die Variable sicher null, da es sich um eine Suche handelt und kein einzelnes Lieferaviso beim Aufruf vorliegt.

- Der Anwendungsentwickler könnte im Abschnitt "command parameter" zusätzliche Parameter für das Kommando definieren, die beim Start des Kommandos übergeben werden müssten.

- Im Abschnitt "local variables" kann der Entwickler lokale Variablen deklarieren, die er innerhalb des Kommandos verwendet - in diesem Beispiel das ViewObjekt AuswahlBelegAnzeige, das an mehreren Stellen referenziert wird.

- Bei <docu> könnte ein mehrzeiliger Text hinterlegt werden, der in der Applikation als Tooltip zu dem Kommando angezeigt wird.

- Im Abschnit "command settings" sind mehrere Einstellungen zusammengefasst. Unter anderem wird dort der Kommand-Typ (SEARCH_VIEW für Suchen) oder eine Freigabebedingung, die für den Start des Kommandos erfüllt sein muss, spezifiziert. Jedes Kommando kann vom Benutzer abgebrochen werden. "Question on external abort" gibt dann die Frage an, die dem Benutzer vor dem Abbruch noch gestellt wird (z.B. "Haben Sie ihre Änderungen gespeichert?"). In der Benutzeroberfläche ist das Kommando grundsätzlich über den Namen "Lieferavisos anzeigen" zu finden. Mit "Title addon" kann der Titel des Tabs zusätzlich erweitert werden. Last but not least - auch ein Icon für das Kommando kann festgelegt werden.

- Die im Abschnitt "command init" angegebene Funktion wird beim Starten des Kommandos ausgeführt. Alle Operationen in dieser Funktion sollten grundsätzlich wenig Zeit beanspruchen, da in der Oberfläche keine Sanduhr/Busy-Indicator angezeigt wird. In diesem Beispiel wird das Serverdatum abgefragt und eine Instanz des AuswahlBelegAnzeige Objektes erzeugt. Die verwendete Builder-Syntax erleichtert dabei die Lesbarkeit.

- Als nächstes folgen zwei Seiten, die "SearchView" und die "ListView". Nachdem die "command init" Funktion ausgeführt wurde, wird als erstes die SearchView ausgeführt. Diese Seite benötigt zusätzlich ein Formular, das mit einer Liste von AuswahlBelegAnzeige Objekten arbeiten kann (hier offensichtlich nur eines). Die "page init" ist analog der "command init", wird aber immer dann ausgeführt, wenn eine Seite gestartet oder neu geladen (reload) wird. In diesem Beispiel wird nur das AuswahlBelegAnzeige Objekt auf die Benutzeroberfläche gespielt. Auf scopes werden wir später noch genau eingehen. Sie dienen der Berechnung von Auswahlmöglichkeiten (z.B. für Dropdown-Felder).

- Conclusions modellieren nun die eigentlichen Schaltflächen. Sie geben damit an, wie eine Seite beendet werden kann. Eine Abbrechen-Schaltfläche wird immer automatisch jeder Seite hinzugefügt. Sie bricht immer das gesamte Kommando ab und nicht nur die aktuelle Seite. Im Beispiel wird zusätzlich eine ">> Anzeigen"

Schaltfläche mit Hotkey "Next" der Seite angefügt. Falls der Endanwender diese Conclusion wählt, soll die Seite "ListView" gestartet werden.

- Die "ListView" benötigt ein Formular, das eine Liste von LieferAviso darstellen kann. Die Liste von LieferAviso wird von der Datenbank geladen, wobei dazu eine Methode des EingangsBelegRepo (Repository) mit den Parametern datVon und datBis aufgerufen wird. Das Ergebnis wird durch die "page init" Funktion an die Benutzeroberfläche weitergegeben. Ferner wird auch noch ein dynamischer Seitentitel berechnet. Die toString() Methode des AuswahlBelegAnzeige ViewObject's wird aufgerufen (siehe Abbildung 2: Ergebnisliste). Als Conclusion steht neben dem Abbrechen auch noch "<< Auswählen" bereit, mit dem wieder auf die "SearchView" Seite verzweigt wird.

- Die FINAL_OK_CONCLUSION, FINAL_CANCEL_CONCLUSION und EXCEPTION_CONCLUSION wurden nicht implementiert.

  FINAL_OK_CONCLUSION wird ausgeführt, wenn das Kommando erfolgreich beendet wird,
  FINAL_CANCEL_CONCLUSION, wenn das Kommando abgebrochen wird und
  FINAL_EXCEPTION_CONCLUSION, falls eine technische Exception auftritt.

  Die so genannten Final Conclusions sind vor allem im Zusammenhang mit dem Bearbeiten von Daten von Bedeutung. So müssen bspw. zwingend in der FINAL_OK_CONCLUSION Speichervorgänge auf die Datenbank vorgemerkt werden.

Um dieses Beispiel abzuschließen fehlen noch drei Konzepte. Zum einen haben wir den Prozess noch nicht dargestellt, zum anderen fehlt die Repository Methode um Lieferaviso zu laden und schließlich müssen wird das Kommando noch in einer Applikation eintragen. Auf den Prozess wollen wir später im Detail nochmals eingehen. An dieser Stelle genügt der einfache, in Abbildung 9 abgebildete Prozess. Grundsätzlich handelt es sich dabei um eine Zustandsmaschine, die den Zustand des Datenfeldes dokumentStatus im Lieferaviso prüft. Dieser Status umfasst die Ausprägungen angelegt, bearbeitet, abgeschlossen und Fehler. Unabhängig vom Zustand dieses Status kann das Kommando "Lieferavisos anzeigen" immer ausgeführt werden, daher ist es im Abschnitt "creators and state-independent commands" angeführt. Der Endanwender kann das Kommando daher ohne Einschränkungen immer starten.

```
process 'LieferAvisoProc' using LieferAviso doc process-status-field is dokumentStatus

<docu>


creators and state-independent commands:
// open a session in those commands, command will be available in every state !
on trigger [<cndt>] Lieferavisos anzeigen -> < >


states:
state Angelegt: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>


state Bearbeitet: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>


state Abgeschlossen: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>


state Fehler: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>
```

Abbildung 9: Prozess mit Lieferavisos anzeigen Kommando

Wenden wir  uns der Methode findeLieferaviso() des EingangsBelegRepo Repository zu. Datenbankabfragen sind nur in Repository Methoden möglich. Es wird unterschieden, ob eine Repository Methode Entitäten ReadOnly oder Read/Write lädt. Werden Entitäten nur ReadOnly geladen, können Sie nicht verändert und gespeichert werden. Für die Lieferaviso Suchanzeige wird gerade ein ReadOnly Ladevorgang benötigt. Die Funktion ist in Abbildung 10 dargestellt.

```
model repository EingangsBelegRepo


<doc>


//
READONLY list<LieferAviso> findeLieferAviso(LocalDate von, LocalDate bis) {
  list<LieferAviso> result = null;

  result = MapLieferAviso <join> where({~lieferaviso,  =>
      lieferaviso.lieferDatum >= von && lieferaviso.lieferDatum <= bis  }) ReadOnly;

  result;
}
```

Abbildung 10: Repository Methode

```
persistence description for
LieferAviso, LieferAvisoPos


map LieferAviso as MapLieferAviso
  on table "FWWS_LIEFERAVISO" options OPTIMISTIC_LOCK {
  int id -> "KEY_LAVISO" KEY, AUTOID("S_LIEFERAVISO")
  ref VorgangsOrt lieferant: int id -> "REF_LIEFERANT"
  ref VorgangsOrt besteller: int id -> "REF_BESTELLER"
  ref VorgangsOrt warenEmpfaenger: int id -> "REF_WARENEMPFAENGER"
  LocalDate ausstellDatum -> "DAT_AUSSTELLUNG" <opt>
  LocalDate lieferDatum -> "DAT_LIEFERUNG" <opt>
  string lsNummer -> "NUM_LS" <opt>
  status lieferArt -> "COD_LIEFERART" <opt>
  status dokumentStatus -> "COD_DOK_STATUS" <opt>
}
```

Abbildung 11: Mapping Lieferaviso

In der rechten Abbildung ist das Mapping MapLieferAviso zu erkennen. Es formalisiert die Abbildung der LieferAviso Entität auf die Datenbanktabelle FWWS_LIEFERAVISO, die mit "optimistic locking" geschützt wird (Versionierung von Datensätzen). In der Repository Methode dient das Mapping auch als Referenz, um die eigentliche Abfrage durchzuführen. MapLieferAviso gibt eine Liste von Aviso zurück, die durch einer where() Bedingung eingeschränkt wird.  Die resultierenden LieferAviso werden ReadOnly geladen. Außer dieser Repository Methode sind im Rahmen dieses Beispiels keine weiteren Funktionen notwendig. Es fehlt lediglich noch das Konzept Applikation.

Das Konzept der Applikation verbindet die vorgestellten Konzepte zu einer gemeinsamen Anwendung. In diesem Beispiel wird die DemoConfigWwws als Konfiguration verwendet, in der diverse Datenbank- und Java-Einstellungen vermerkt sind. Es kann eingestellt werden, ob ein Anmelde-Bildschirm beim Starten der Anwendung angezeigt wird. Benutzername und Password (als Integer-Hashcode der Eingabe) werden dann der startup() Funktion übergeben. Liefert startup() true zurück, kann die Applikation gestartet werden.

Im Start-Menü der Applikation wurde ein Trigger modelliert. Der Endanwender kann daher das Kommando "Lieferavisos anzeigen" aus dem Prozess LieferAvisoProc mit dem Parameter null für das Prozessdokument starten. Damit kann über das Dokument kein Status geprüft werden und im Kommando steht auch kein Prozessdokument als Parameter zur Verfügung. Der Trigger definiert, welche Formulare für die Seiten verwendet werden sollen. Die beiden Formular wurden eingangs dieses Beispiels erstellt, die Seiten selbst sind im Kommando modelliert. Ein Hotkey zum Starten wird nicht zugewiesen.

Alle Anforderungen des Beispiels wurden vollständig mit Modellkonzepten umgesetzt. Kein zusätzlicher Java Code ist an dieser Stelle erforderlich.

```
Application eFWWS

configuration: DemoConfigWws

version information: 0.00.1 (Sterne)

show login screen: false

startup(username, password)->boolean {
  true;
}

'start' menu definition:
  on trigger run LieferAvisoProc.Lieferavisos anzeigen(null)
    view for page: SearchView = FCAuswahlBeleg
    view for page: ListView = FCLieferAvisoListView
  hk: UNDEFINED

'extras' menu definition:
  << ... >>
```

Abbildung 12: Applikation für erstes Beispiel

# 2. Einführung

Statt einer Einleitung sind wir im ersten Kapitel direkt in ein erstes Beispiel gestartet. Nun werden in dieser MoWare Beschreibung alle verwendeten Konzepte nochmals im Detail beschrieben.

In diesem Abschnitt werden die bereits bekannten Konzepte um weitere ergänzt und nochmals zusammengefasst. Im darauf folgenden Kapitel 3 wird speziell auf Datenstrukturen und Datenfelder eingegangen. Daran schließt sich eine umfassende Erklärung der Datenbankschnittstelle durch Repositories an. In Kapitel 5 werden Konzepte zur Modellierung von Geschäftslogik diskutiert. Vor allem das Kommando und der Prozess stehen im Vordergrund. Anschließend werden in Kapitel 6 Konzepte zur Gestaltung von Oberflächen vorgestellt. Kapitel 7 schildert typische Anwendungsfälle. Dabei wird das bestehende Wissen vertieft, neue Konzepte kommen nicht mehr hinzu. Kapitel 8 widmet sich dem Drucken mit MoWare, Kapitel 9 den BatchJobs, die auf einem Server ständig wiederkehrende Aufgaben (Jobs) abarbeiten. Das letzte Kapitel stellt die wichtigsten  Entwicklungswerkzeuge von MoWare vor, die in der Programmierumgebung integriert wurden.

## *Die Laufzeit- und Entwicklungsumgebung*

Grundsätzlich ist die komplette Entwicklungs- und Laufzeitumgebung in Java realisiert. Gerade für den Enterprise-Bereich bietet Java umfangreiche Plattformen, Bibliotheken und Werkzeuge an. Als Entwicklungsumgebung wird das Jetbrains Meta Programming System verwendet. Es handelt sich dabei um einen Projektional-Editor mit Projektverwaltung und GIT Integration (Sourcecode Verwaltung). Mehr Informationen zu MPS sind unter http://www.jetbrains.com/mps/concepts/ zu finden. Als Build-Umgebung wird gegenwärtig ant verwendet. Alle Anwendungen werden im Moment als Java-FX Executables mit MSI Installer deployed. Aktuelle Build-Scripts können von bestehenden Projekten übernommen und angepasst werden.

Wichtigste HotKeys, mit denen man sich in der Entwicklungsumgebung vertraut machen sollte: CRTL-N, CRTL-R, CRTL-P

## *Der MoWare Stack*

MoWare ist eine Sammlung von Sprachen und Laufzeitumgebungen, die die Erstellung von Geschäftsanwendungen vereinfachen und beschleunigen. Einen ersten Überblick über die Idee hinter dem Stack gibt der Artikel "Adieu teure Software-Enwicklung" (Javamagazin Vol. 3/4 2014).

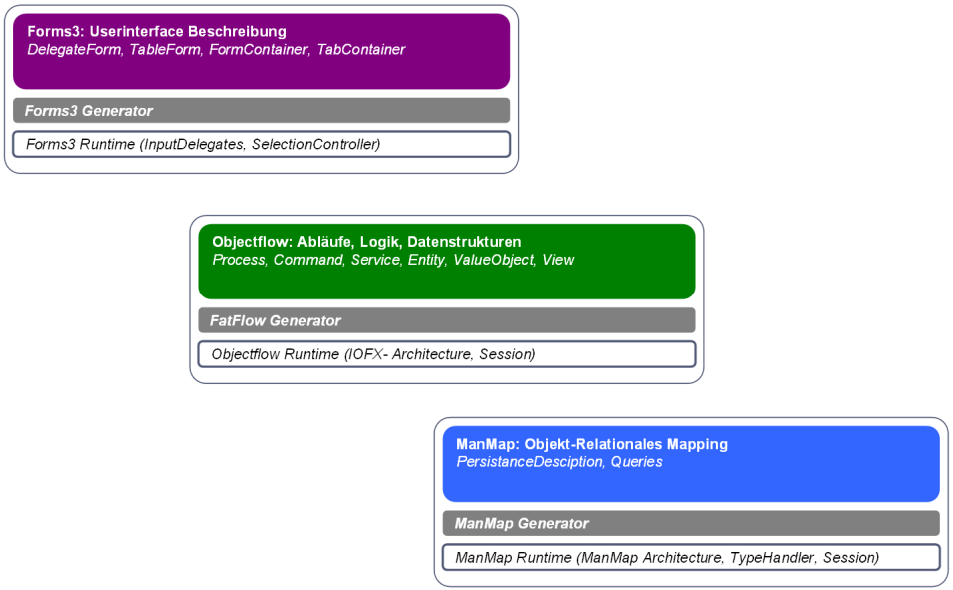



Abbildung 13: Sprachen Forms3 / Objectflow und Manmap

Der MoWare-Stack setzt sich im wesentlichen aus drei Sprachen zusammen (siehe Abbildung), mit denen ERP-ähnliche Applikationen aufgebaut werden können. Wir bezeichnen diese als dokumentenzentrierte Anwendungen, da die Erstellung und Verwaltung von Dokumenten im Vordergrund steht. Die Idee hinter den Sprachen lässt sich in 4 Punkte fassen:

1. Die Sprachen unterstützen die Entwicklung von Applikationen mit User-Interface und Server BatchJobs. Applikationen sind durch Dritte einfacher zu verstehen und zu warten, da die wesentlichen Applikationsgrundstrukturen standardisiert wurden. Standardisiert sind bspw. Konzepte wie Kommandos, Prozesse und Entitäten.

2. Die Sprachen unterstützen einen Entwicklungsprozess, der mit einfachen, aber konkreten Use-Cases startet, die später wiederum als Test-Szenarien herangezogen werden. Test-Driven Software Development wird dadurch aktiv unterstützt. Ohne Test-Szenarien wäre langfristig keine Qualitätssicherung möglich, was den Wechsel von Infrastruktur erheblich erschwert.

3. Die Sprachen bilden Applikationen unabhängig von den verwendeten Laufzeitumgebungen und Frameworks ab. Für Anwender sind verwendete technische Frameworks (z.B. User-Interface Framework) nicht von Belang - sie konzentrieren sich auf infrastruktur-neutrale Applikationslogik.

4. In Zukunft können Codegeneratoren die Applikationen auf verschiedene Laufzeitumgebungen übersetzen. Neben klassischen Desktop-Applikationen sind Web-Oberflächen und Mobile-Apps mögliche Zielumgebungen.

Neben den 3 Sprachen sind unterschiedliche Generatoren verfügbar. So ist mit fxforms ein Generator für Java FX verfügbar, mit fatflow ein Generator der Prozesse und Kommandos für Fat-Clients erzeugt. In Zukunft werden weitere Zielplattformen und Laufzeitumgebungen unterstützt.

## *Modelle im MPS*

Eine Applikation greift im MPS auf Konzepte dieser Sprachen zurück. Sie muss innerhalb von MPS zwingend mit Modellen strukturiert werden. Ein Modell entspricht in etwa einem Unterverzeichnis. Mehrere Komponenten wie Kommandos, Repositories oder Entitäten und ViewObjects können in einem Modell zusammengefasst werden. Modelle sollten untereinander möglichst wenig Abhängigkeiten aufweisen, so dass sie einzeln wieder verwendet werden können. Modelle sollte einen logischen Bereich erfassen, allerdings sollten sie gleichzeitig nicht zu umfangreich werden. Gegenwärtig werden Modelle folgendermaßen gebildet:

| | |
|---|---|
| <Modellname>UI | UI Modelle enthalten lediglich Forms-Konzepte (DelegateForm, FormContainer, TableForm etc.). Unabhängig von den Kommandos könnten die Oberflächenkonzepte ausgetauscht werden (z.B. für mobile Anwendungen). Gegenwärtig werden allerdings nur Standard ERP-ähnliche Konzepte unterstützt. Mit sogenannten Virtual-Foldern kann auch innerhalb der Modelles nochmals strukturiert werden. |
| <Modellname>DATA | DATA Modelle enthalten Entitäten, ValueObjects, ViewObjects, PersistanceDescriptions und Repositories. Gerade die Datenobjekte sind für Schnittstellen als Rückgabe Objekte oder Parameter von besonderer Bedeutung. Um Abhängigkeiten gering zu halten und eine Wiederverwendung über Applikationen hinweg zu ermöglichen, werden diese Konzepte in eigenen Modellen festgehalten. |
| <Modellname>APP | Enthält alle Konzepte, die speziell für eine bestimmte Applikation notwendig sind. Dazu gehören vor allem diverse Konfigurationen und das Applikations-Konzept, welches eine Startup Funktion, das Start-Menu, das Extras-Menu und die aktuell verwendete Konfiguration bestimmt. |
| <Modellname>PROC | Enthält alle Konzepte die Abläufe und Services betreffen, vor allem Kommandos und Prozesse. Meist wird ein Prozess pro Modell verwendet und alle zugehörigen Kommandos im selben Modell verwaltet. |

### *Übersicht über die wichtigsten Konzepte in MoWare*

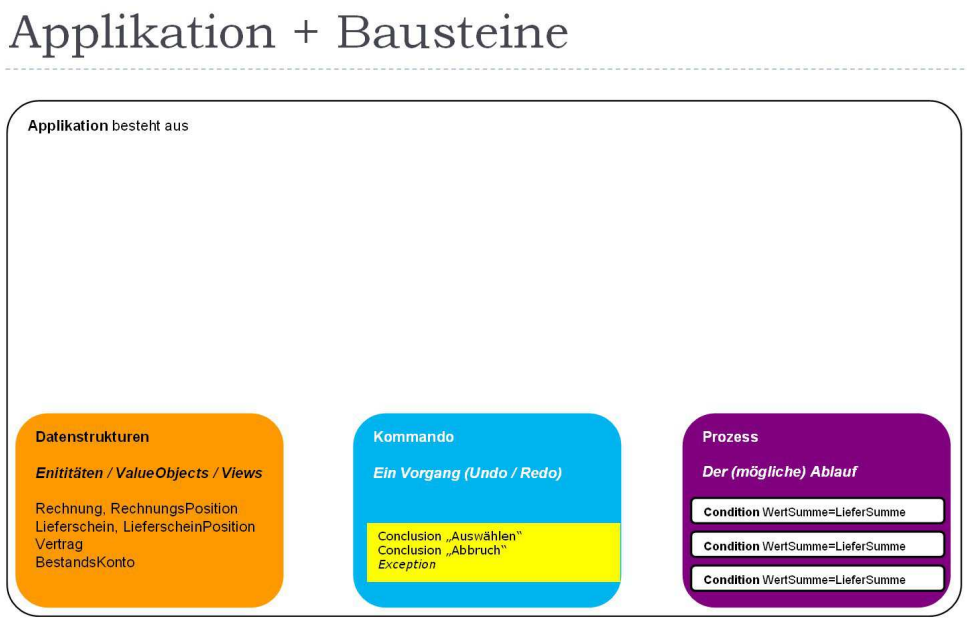



Abbildung 14: Datenstrukturen, Kommando und Prozess

Eine Geschäftsanwendung ist grundsätzlich aus Datenstrukturen, mehreren Kommandos und Prozessen aufgebaut. Die Modellierung geeigneter Datenstrukturen um Belege, Dokumente und Beziehungen abzubilden, ist wohl die schwierigste Aufgabe bei der Erstellung einer Anwendung. MoWare bietet als Datenstrukturen

- Entitäten: Objekte, die existieren und auch einen eindeutigen Schlüssel benötigen,
- ValueObjects: Objekte, die nicht selbst existieren; nur deren Wert zählt, und
- ViewObjects: Objekte, die nur temporären Charakter haben und bspw. nicht auf die Datenbank gespeichert werden können

Kommandos bilden zum einen Geschäftslogik ab, zum andern können sie - müssen aber nicht - mit Benutzeroberflächen interagieren. Sie sind dadurch gekennzeichnet, dass sie erfolgreich beendet oder abgebrochen werden. Entweder wird das gesamte Kommando erfolgreich ausgeführt und damit all dessen beabsichtigte Änderungen an Datenstrukturen, oder keine einzelne der Änderungen wird vollzogen. Kommandos lassen sich in diesem Sinne auch rückgängig machen.

MoWare kann mit Hilfe von Prozess und Bedingungen auch Unternehmensprozesse abbilden. Hinter dem Begriff des Prozesses befindet sich ein Konzept, das in der Informatik üblicherweise eher als Zustandsmaschine bezeichnet wird. Jeder Unternehmensprozess lässt sich anhand eines Prozess-Dokuments (immer eine Entität) und einem Prozessstatus (Status in dieser Entität) modellieren. Mit Bedingungen lassen sich Zustandsübergänge modellieren, die entweder automatisch geprüft werden (auto) oder nach Benutzerinteraktionen (on trigger). Kommandos müssen einem Prozess zugeordnet werden. Sie bilden die notwendige Geschäftslogik des Prozesses ab.

In Abbildung 15: Kommando und Datenstrukturen ist das Verhältnis zwischen Kommandos und Datenstrukturen nochmals zusammengefasst. Kommandos können Tätigkeiten abbilden, da sie nicht nur Geschäftslogik, sondern auch Interaktionen mit der Benutzeroberfläche aufnehmen. Sie werden immer erfolgreich beendet (ok conclusion) oder abgebrochen (cancel conclusion). Es kann auch ein technischer Fehler zum Abbruch führen (exception conclusion). Kommandos verändern die Datenstrukturen. Sind alle Änderungen abgeschlossen - meist Änderungen durch mehrere Kommandos - so werden alle veränderten Datenstrukturen in einem Arbeitsschritt auf der Datenbank persisiert (checkin).

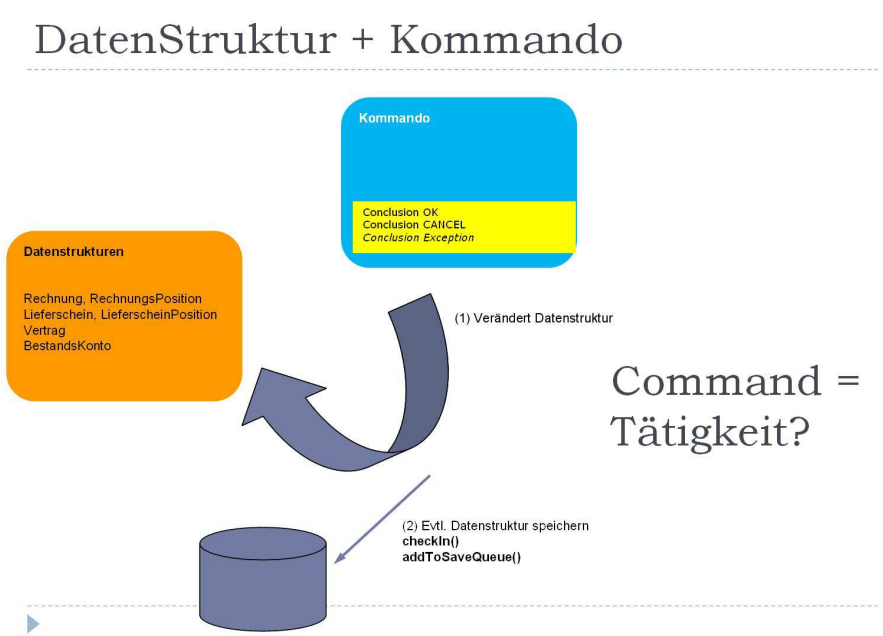



Abbildung 15: Kommando und Datenstrukturen

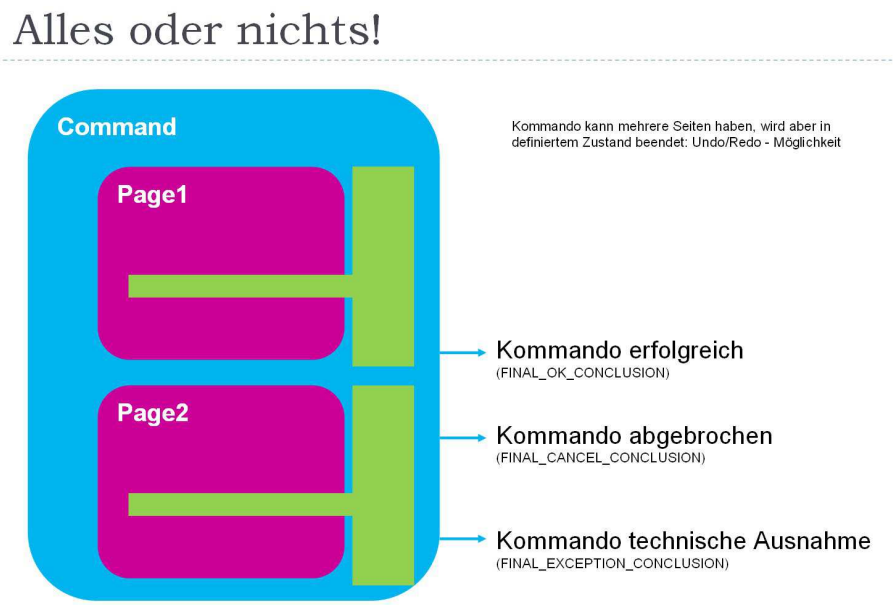



Abbildung 16: Kommando mit Seiten

Ein Kommando kann keine, eine oder mehrere Seiten für Benutzerinteraktionen enthalten.

## Applikation + Bausteine

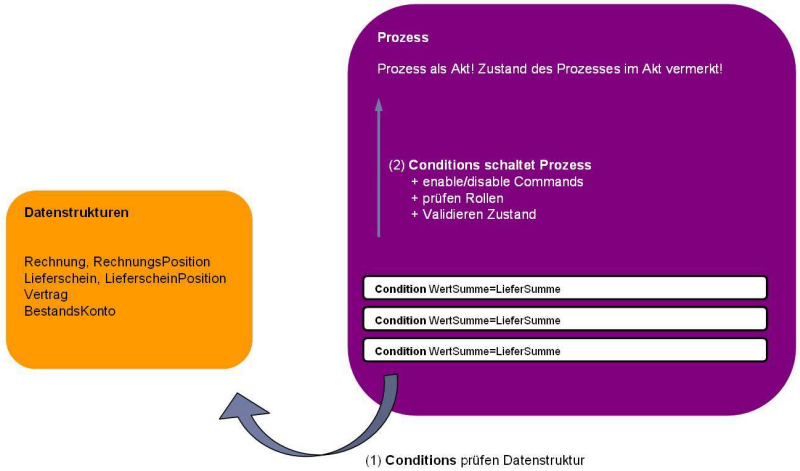



Abbildung 17: Prozess und Datenstrukturen

Jeder Prozess befindet sich zu jedem Zeitpunkt in einem bestimmten Zustand. Bedingungen prüfen relevante Datenstrukturen und wechseln den Zustand des Prozesses - entweder automatisch oder nach der erfolgreichen Ausführung eines Kommandos. Bedingungen wechseln aber nicht nur Prozesszustände, sie können auch Kommandos freigeben oder sperren.

Die wesentlichen Bedingungen im Zusammenhang mit Geschäftsprozessen sollten - wenn möglich - im Prozess Konzept auch explizit als Bedingung definiert werden. Auf diese Weise ist auch für Dritte der Prozess und dessen wichtigste Eigenschaften auf den ersten Blick erkennbar.

Neben den in den Schaubildern dargestellten Konzepten sind zusätzlich Services und ModelRepositories von Bedeutung.

**ModelRepositories**

In ModelRepositories werden die Zugriffe auf die Datenbank abgekapselt. In einer baselang.collection ähnlichen Syntax werden Abfragen an die SQL Datenbank formuliert und Ergebnismengen in Entitäten abgepackt. ModelRepositories kapseln somit die Funktionalitäten von klassischen Objektrelationalen-Mappern. Eine Methode in einem ModelRepository persistiert einen kompletten Java-Objektgraphen oder lädt ihn vollständig (d.h. lädt ihn und baut ihn auf).

**Services**

In Services können "Hilfsfunktionen" ausgelagert werden. Primär werden Services von Kommandos aufgerufen. Pro Applikation wird von der Runtime jeweils eine Instanz eines jeden Services verwaltet. Services sind damit "Stateless", d.h. auch gleichzeitig: Klassen-Felder können verwendet werden, sind allerdings applikationsglobal.

Zusammenfassend die wichtigsten Konzepte

| **Konzept** | **Funktion** |
|---|---|
| Entität, ValueObject, ViewObject | Modellierung der Datenstrukturen |
| Command | Manipulation von Datenstrukturen, Steuerung von Dialogen in Oberflächen. Commands sind abgeschlossene Einheiten, die allen Programmpaketen zur Verfügung stehen. Commands werden durch Prozesse verwaltet. |
| Prozess | Zugriffsrechte auf Kommandos, Ablauf, Zustände und Bedingungen |
| ModelRepository | Zugriff auf Datenbank, Laden und schreiben von Datenstrukturen. |
| PersistanceMapping | Beschreibung, wie eine Entität auf eine Tabelle abgebildet wird (Abbildungsvorschrift einer Entitäten auf eine Tabelle). Eine Entität kann durchaus mehrere Mappings verwenden. |
| Service | Einfache Hilfsfunktionen zur Modularisierung. Services sind zustandslos! Keine Instanzfelder verwenden, um Daten zu cachen! Instanzfelder sind primär für Debuggingzwecke (z.B. JMX Informationen), für Konstanten (private final) oder für die Einbindung externer Bibliotheken.<br><br>Services enthalten meist private Hilfsmethoden. In einer SOA Architektur werden Services im Sinne von MoWare nicht öffentlich zur Verfügung gestellt. Es sind die Commands, die public zur Verfügung stehen müssten. |
| Session | Die Session ist ein sehr wichtiges, aber ungewohntes Konzept. Sie begleitet mehrere Arbeitsschritte. Die Session bietet den Kontext, in dem eine Unit-Of-Work geladen, bearbeitet und gespeichert wird. Anschließend wird die Session zerstört. Eine |

| | |
|---|---|
| | Session entspricht in den Oberflächen meist einem Tab im Hauptreiter der Applikation. Die Tabs (Session) haben untereinander keine Beziehung.<br><br>Die Session fungiert auch als First-Level-Cache. Abfragen innerhalb einer Session liefern bei gleichem Schlüssel auch die selbe Entität im Speicher. |
| BatchJob | Batch Jobs werden auf einem Server zyklisch ausgeführt. BatchJobs verarbeiten Daten im “Hintergrund”. Sie können ebenfalls auf Prozesse und Commands zurückgreifen, haben aber kein User-Interface. |

## 3. Datentypen Entitäten, ValueObjects und ViewObjects

Datenstrukturen und -objekte sind wohl wesentlichster Teil von Geschäftsanwendungen. Besonders die geschickte Modellierung von Datenstrukturen kann eine Anwendung lesbar und langfristig einfach wart- und erweiterbar gestalten. In diesem Abschnitt werden die grundlegenden Konzepte zur Modellierung von Datenstrukturen vorgestellt und Anwendungsempfehlungen gegeben.

Wichtigstes Konzept bei der Modellierung von Daten ist sicherlich die Entität. Entitäten sind Objekte, welche nicht durch ihre Eigenschaften, sondern durch ihre Identität definiert sind. Beispielsweise wird eine Person meist als Entität abgebildet. Eine Person bleibt dann dieselbe Person, auch wenn sich ihre Eigenschaften ändern und sie unterscheidet sich von einer anderen Person, auch wenn diese dieselben Eigenschaften hätte. Entitäten werden oft mit Hilfe von eindeutigen Identifikatoren/Schlüssel modelliert. Entitäten weisen meist Beziehungen untereinander auf. So referenziert ein Rechnungskopf mehrere Rechnungspositionen. Eine Rechnungsposition referenziert einen Rechnungskopf. Üblicherweise wird der Großteil der Datenstrukturen als Entitäten modelliert. Daten aus Geschäftsprozessen benötigen meist ein eindeutiges Identifizierungsmerkmal (Schlüssel) und nur Entitäten lassen sich unmittelbar auf der Datenbank persistieren.

MoWare stellt zusätzlich zwei weitere Daten-Objekte zur Verfügung: ValueObjects und ViewObjects. ViewObjects sind einfach zu beschreiben. Sie sind im Wesentlichen völlig analog den Entitäten zu verwenden, sie lassen sich allerdings nicht persistieren. Häufig werden sie verwendet, um in der Applikation temporär Daten zu speichern. Im Einführungsbeispiel (Suche Lieferaviso) wurde ein ViewObject verwendet, um die Suchkriterien zu speichern.

Wie ViewObjects verfügen auch ValueObjects nicht über Identifikatoren. Stattdessen besitzen ValueObjects eine kennzeichnende Eigenschaft. Zwei ValueObjects sind dann gleich, wenn ihre Werte - sprich ihre Datenfelder - gleich sind. Sie können allerdings keine weiteren Objekte referenzieren, d.h. sie können keinerlei Beziehung zu weiteren Objekten aufweisen. Der Entwickler kann ValueObjects verwenden, um für eine Domäne wichtige, primitive Datentypen zu definieren. Jedem der Objekte ist nachfolgend ein Unterkapitel gewidmet.

Es hat sich in der Vergangenheit als nützlich erwiesen, erst die logischen Beziehungen der Objekte (1), dann die Objekte selbst (2) zu modellieren. Die Abbildung auf Datenbanktabellen kann zu einem späteren Zeitpunkt erfolgen. Auch in dieser Anleitung wird erst im nächsten Kapitel auf die Persistierung eingegangen.

Zuerst zu den Beziehungen (1): Objekte können im Allgemeinen einzelne andere Objekte referenzieren, z.B. eine Rechnungsposition bezieht sich auf einen Artikel **(Kardinalität 0 oder 1)**, oder eine ganze Liste von weiteren Entitäten **(Kardinalität 0 ... n)**[1]**.** Im Fall der Entiät Rechnungskopf enthält dieser eine Liste von Rechnungspositionen.

Beziehungen der Entitäten untereinander sind meist auch in Benutzeroberflächen gut zu erkennen. In der nächsten Abbildung ist eine Anwendung zur Rechnungskontrolle dargestellt. Angezeigt wird die Hauptansicht eines Rechnungskontrollaktes. Wie einfach zu erkennen, ist oben der Akt mit div. Datenfeldern angezeigt. In einem Akt befinden

[1] Evtl. müsste die Kardinalität als Option genau festgelegt werden, um beim Laden und Speichern Prüfungen vornehmen zu können. Artikel in Rechnungsposition müsste vermutlich Kardinalität 1 und eben nicht 0 aufweisen. Gegenwärtig nur im Repo mit der get() Operation ausgedrückt.

sich mehrere Rechnungen - im dargestellten Fall 3 Rechnungen. D.h. die Entität RekoAkt enthält neben den Datenfeldern auch eine Liste von Rechnungen (list<Rechnung>). Eine Rechnung enthält wiederum mehrere Rechnungspositionen. Im Screenshot werden im Moment die Positionen der ersten Rechnung angezeigt, da diese markiert wurde (Rechnung 1 blau hinterlegt). Neben Preisen wird in der Tabelle der Rechnungspositionen auch eine Artikelnummer und eine Artikelbezeichnung angezeigt. Diese beiden Felder sind allerdings nicht als int und string in die Position modelliert, sondern die Position enthält eine Referenz auf den entsprechenden Artikel. Die Entität Artikel enthält als int die Artikelnummer - als Identifikator/Schlüssel - und die Artikelbezeichnung als Text.

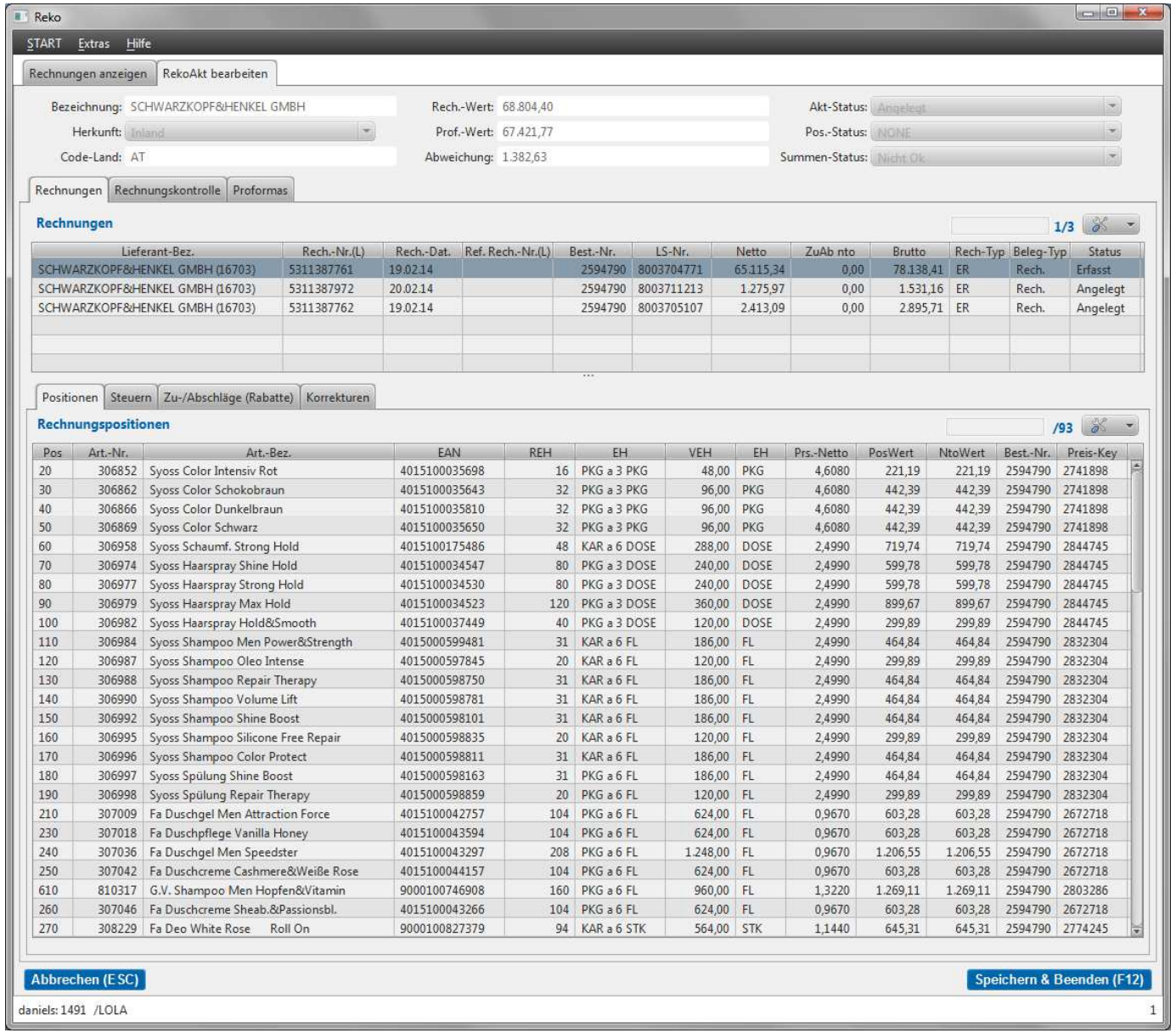



Abbildung 18: Screenshot aus der Applikation zur Rechnungskontrolle

(2) Neben diesen beiden Formen von Beziehungen nehmen Objekte auch Daten auf. Datenfelder von Objekten werden in der MoWare Beschreibung auch Properties oder Eigenschaften der Objekte genannt. Folgende primitive Datentypen stehen zur Verfügung.

| **Datentyp** | **Verwendung** | **Besonderheiten/Erläuterungen** |
|---|---|---|
| BigDecimal | Dezimaler Wert mit Festkomma-Eigenschaften (unendliche Genauigkeit). Wird als standardtyp verwendet, um Werte zu modellieren. | Insbesondere bei der Division ist das Rundungsverhalten anzugeben (z.B. falls eine periodische Zahl als Ergebnis resultiert). Ferner ist zu überlegen, ob während Rechenschritten oder erst bei der letzten Zuweisung gerundet werden soll. Vergleichsoperatoren wie < > oder == wurde auch für BigDecimal mit MoWare implementiert, was in Java unüblich ist. |
| int | Abbildung von ganzzahligen Werten. Häufig verwendet als Identifikator/Schlüssel für Entitäten und als Zähler. | |
| string | Generell zur Modellierung von Zeichenketten verwendet. Per Voreinstellung keine Längenbeschränkung in Entitäten. | Vergleichsoperatoren == und != wurden durch MoWare implementiert (in Java unüblich). |
| LocalDate | Modellierung von Datum-Felder ohne Zeit Merkmal | Vergleichsoperatoren == und != stehen **nicht** zur Verfügung. Häufig werden Vergleiche mit isBefore() und isAfter() vorgenommen. Im Hintergrund wurde dieser Typ durch die joda DateTime Bibliothek implementiert. Alle Funktionalitäten dieser Bibliothek stehen zur Verfügung. Eine umfassende Dokumentation findet sich im Internet. |
| DateTime | Modellierung von Datum-Felder mit Zeit Merkmal | Vergleichsoperatoren == und != stehen **nicht** zur Verfügung. Ebenfalls durch Joda implementiert. |
| Status | Der Datentyp Status wird verwendet, um nominale Daten zu modellieren. Ein Status kann n verschiedene, diskrete Ausprägungen annehmen (z.B. Geschlecht Male/Female). | Status wird durch MoWare umfangreich unterstützt. == und != Operatoren stehen zur Verfügung. Bei Statusausprägungen muss die Datenbankrepräsentation (wie auf der Datenbank abgebildet?), eine kurz und eine Langbezeichnung (wie in Oberflächen dargestellt?) angegeben werden. |
| | | |

## *Die Entität*

Entitäten sind Datenobjekte die in Datenbank-Tabellenzeilen gespeichert werden. Sie müssen daher über einen eindeutigen Schlüssel verfügen. Meist wird als Schlüssel (ID) ein Integerwert verwendet. MoWare unterstützt auch Zeichenketten als Schlüssel. Mit ValueObjects lassen sich zusammengesetzte Schlüssel abbilden. Prototypisch für eine Entität ist bspw. eine Person. Selbst wenn zwei Personen den selben Namen aufweisen, handelt es sich dennoch um unterschiedliche Personen. Weist man ihnen eine eindeutige ID zu, so können sie anhand dieser differenziert werden. Der Identifikator einer Entität bestimmt, wann zwei Entitäten als die selben gelten. In der Praxis kann es allerdings zu Situationen kommen, insbesondere beim Erzeugen neuer Entitäten, in denen die ID noch nicht gesetzt wurde.

Entitäten führen einen ReadOnly, einen Dirty und einen persistierten Zustand:

- Eine Entität wird als Dirty markiert, wenn sie nach dem Laden aus der Datenbank verändert wurde. Im Speicher neu angelegte Entitäten sind immer Dirty. Entitäten im nicht Dirty Zustand werden nicht auf der Datenbank gespeichert, auch wenn MoWare dazu angewiesen wird (dazu später mehr).

- Entitäten im ReadOnly Zustand lösen eine IllegalAccessError() Exception aus, wenn versucht wird, Datenfelder oder Referenzen der Entität zu verändern. ReadOnly-Entitäten können folglich auch nicht in einem Dirty-Zustand vorliegen.

- Wird ein Integer als Schlüssel verwendet, so ist dieser bei nicht-persistierten Entitäten auf 0 gesetzt. Wird die Entität gespeichert, so ist der Schlüssel von 0 verschieden und positiv. Bei Zeichenketten und ValueObjects ist dieser Zustand noch undefiniert.

Jede Java-Klasse implementiert laut Spezifikation die Methode boolean equals(Object o). Die Methode prüft, ob zwei Java Objekte gleich sind. Häufig wird diese Methode überschrieben. Eine eigene Implementierung wird angegeben, um auf inhaltliche Gleichheit zweier Objekte zu prüfen: aObject.equals(bObject) == true. Das aObject ist eine andere Instanz als das bObject, allerdings sind sie inhaltlich gleich.

MoWare überschreibt sowohl die equals(), als auch die hashCode() Methode. Es können jene Properties angegeben werden, die für einen Vergleich heranzuziehen sind. Allerdings ist grundsätzlich “<use standard hash>” bei Entitäten einzustellen. Die Laufzeitumgebung der Session trägt dann dafür Sorge, dass nur eine Instanz der selben Entität (identifiziert über den Schlüssel) im Speicher gehalten wird. Dadurch können unterschiedliche Entitäten direkt über ihre Speicheradresse verglichen werden. Für equals() wird dann unmittelbar der hashCode() herangezogen, der per Java-Voreinstellung als die virtuelle Speicheradresse der Instanz implementiert ist.

Entity Rechnung extends <no superclass>

<doc>

properties of Entity:

| Type | Name | Short Desc | Long Desc | Format | Options | Documentation |
|---|---|---|---|---|---|---|
| int | id | "ID" | "ID" | "0" | KEY | ... |
| int | nrBeleg | "Beleg-Nr." | "Belegnummer" | ... | ... | // interne Belegnummer // |
| LocalDate | datRech | "Rech.-Dat." | "Rechnungsdatum" | "dd.MM.yy" | ... | // Belegdatum des Lieferanten => Rechnungsdatum // |
| LocalDate | datEingang | "Eing.-Dat." | "Eingangsdatum" | "dd.MM.yy" | ... | // Eingangsdatum der Rechnung // |
| LocalDate | datFaellig | "Fäll.-Dat." | "Fälligkeitsdatum" | "dd.MM.yy" | ... | ... |
| string | nrRechnungL | "Rech.-Nr.(L)" | "Rech.-Nr. Lieferant" | ... | ... | // RechnungsNr des Lieferanten // |
| string | refRechnungL | "Ref. Rech.-Nr.(L)" | "Ref. Beleg-Nr. Lieferant" | ... | ... | // Opt. Ref. auf Ext.BelegNr bei GS und Rech-Kürzungen // |
| string | uid | "Uid" | "UID" | ... | ... | // ATU Nummer // zur Doku aus Partei |
| list<RechPos> | rechPos | ... | ... | ... | CONTAINMENT <no businessProperty> | // <lines> // |
| list<ZuAb> | zuAbPos | ... | ... | ... | CONTAINMENT <no businessProperty> | ... |
| list<Steuer> | steuerPos | ... | ... | ... | CONTAINMENT <no businessProperty> | ... |
| list<RechKorrInfo> | rechKorrInfos | ... | ... | ... | CONTAINMENT <no businessProperty> | ... |
| RekoAkt | rekoAkt | "Akt" | "Akt" | "0" | OPPOSITE | ... |
| StatusRechnung | status | "Status" | "Status" | ... | ... | ... |
| BigDecimal | summeNetto | "Netto" | "Summe Netto" | ... | ... | // converted from virtual property // |
| DateTime | zzCreatedAt | ... | ... | ... | ... | // createdAt (auditable by objectflow) // |
| int | zzCreatedAtUid | ... | ... | ... | ... | // createdAt UserId (auditable by objectflow) // |
| DateTime | zzModifiedAt | ... | ... | ... | ... | // modifiedAt (auditable by objectflow) // |
| int | zzModifiedAtUid | ... | ... | ... | ... | // modifiedAt UserId (auditable by objectflow) // |

object identity (overwrite equals() method):
<use standard hash>

members:

```
public Rechnung() {
  <no statements>
}
public void addRechPos(RechPos aPos) {
  aPos.beleg = this;
  this.rechPos.add(aPos);
}
public void addSteuerPos(Steuer aPos) {
  aPos.beleg = this;
  this.steuerPos.add(aPos);
}
public void addZuAbPos(ZuAb aPos) {
  aPos.beleg = this;
  this.zuAbPos.add(aPos);
}
public void addRechKorrInfo(RechKorrInfo info) {
  info.beleg = this;
  this.rechKorrInfos.add(info);
}
```

Abbildung 19: Entität

In der obigen Abbildung wurde eine einfache Rechnungs-Entität modelliert. Neben zahlreichen Datenfeldern und Listen von diversen Positionen enthält die Rechnung auch eine Referenz zum zugehörigen Rekoakt.

- Jedes Feld verfügt über eine Short und Long-Description, die als Textbeschreibungen in der Benutzeroberfläche verwendet werden.
- Das Format muss nicht gesetzt werden, außer es entspricht nicht der Voreinstellung. In der obigen Abbildung wäre kein Format notwendig, da sowohl Datum als auch int ohnedies das angegebene Format aufweisen.
- Die Option KEY weist den Schlüssel aus.
- Die Option OPPOSITE gibt an, dass sich in dem Rekoakt eine Liste von Rechnungen befindet, d.h. folgt man Opposite Referenzen, ergibt sich ein zirkulärer Durchlauf. Opposite-Referenzen müssen daher zwingend ausgewiesen werden.
- Die CONTAINMENT Option gibt an, dass die Listenpositionen logisch mit dem Lebenszyklus der (Vater-) Entität verbunden sind, d.h. wird bspw. die Vater Entität gelöscht, sollten auch die Positionen gelöscht werden (die Rechnung "besitzt" eine Position).

Im Abschnitt object identity wurde nichts gewählt, womit der Default "use standard hash" gilt. Die Session stellt dann sicher, dass jeweils nur eine Entität mit selbem Schlüssel im Speicher instanziert wird. Bei der detaillierten Erläuterungen zur Session wird dieses Verfahren genauer erklärt.

Es wurden im Abschnitt members zahlreiche Methoden angelegt. In diesem Beispiel wird hauptsächlich die Verwaltung von Listenpositionen vereinfacht. Positionen werde dem entsprechenden Listenfeld hinzugefügt, gleichzeitig wird bei der Position die Rückwärtsreferenz zum Vater (Rechnung) gesetzt. *(Nach aktuellem Stand, wird in Positionen keine Rückwärtsreferenz mehr modelliert. Nur ein int Feld wird angelegt, das beim Speichern der Position richtig belegt wird. Dadurch entfallen die dargestellten member-Funktionen. Die einfachen Listen Operationen der MPS collection language genügen).*

Grau hinterlegt ist das Datenfeld summeNetto. Es handelt sich um ein sogenanntes Virtualproperty. D.h. dieses Feld liegt nicht als Datenspeicher vor, sondern es wird bei jedem Zugriff neu berechnet. Die Berechnungsvorschrift ist nicht in der Tabelle hinterlegt, sondern muss im Inspektorfenster nachgeschlagen werden (siehe nachfolgende Abbildung).

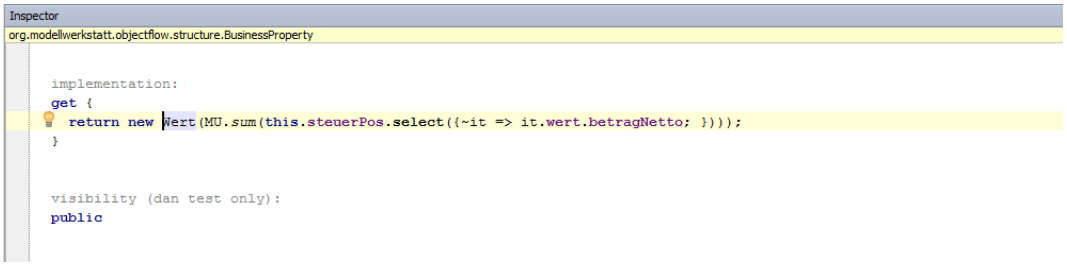
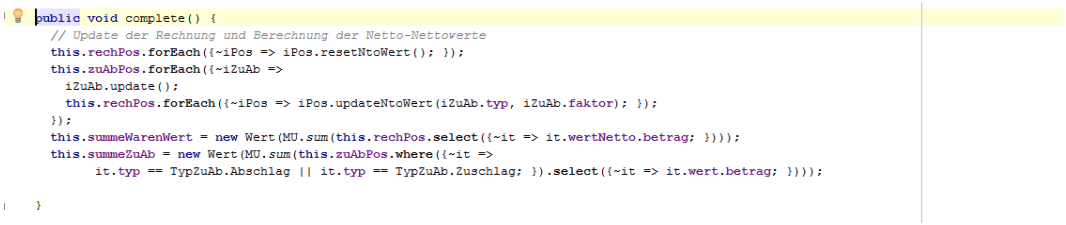


Abbildung 20: Inspektorfenster

Berechnung der Summe über ein ValueObject wert,  das über ein  Property betragNetto verfügt.



```
public void complete() {
    // Update der Rechnung und Berechnung der Netto-Nettowerte
    this.rechPos.forEach({~iPos => iPos.resetNtoWert(); });
    this.zuAbPos.forEach({~iZuAb =>
      iZuAb.update();
      this.rechPos.forEach({~iPos => iPos.updateNtoWert(iZuAb.typ, iZuAb.faktor); });
    });
    this.summeWarenWert = new Wert(MU.sum(this.rechPos.select({~it => it.wertNetto.betrag; })));
    this.summeZuAb = new Wert(MU.sum(this.zuAbPos.where({~it =>
          it.typ == TypZuAb.Abschlag || it.typ == TypZuAb.Zuschlag; }).select({~it => it.wert.betrag; })));

}
```

Abbildung 21:Complete Methode

In der Abbildung der Rechnungsentität wurde aus Platzgründen die complete() Methode und Statusdeklarationen nicht dargestellt. Die complete() Methode wird häufig implementiert, um abschließende Berechnungen mit der Entität durchzuführen, die auch persistiert werden sollen. Verfügen bspw. die Positionen über ein BigDecimal warenWertNetto, so ergibt sich der Gesamtwarenwert aller Positionen aus deren Summe (MU.sum() kann verwendet werden, um eine Liste von BigDecimal zu summieren, siehe S.39). Befindet sich in der Kopfentität ein Feld gesamtWarenWert so könnte dieses Feld in der complete() Methode berechnet werden. Bei Wert handelt es sich um ein ValueObject, das im nachfolgenden Abschnitt modelliert wurde.

In einer Entität können auch Statusdeklarationen vorgenommen werden. Im Einführungsbeispiel ist im Lieferaviso eine derartige Deklaration ersichtlich.

## *Das ValueObject*

Im Vergleich zu Entitäten benötigen ValueObjects keinen eindeutigen Schlüssel. Damit können sich auch nicht direkt auf einer Datenbank persistiert werden. Meist werden sie in Entitäten eingebunden. Zwei ValueObjects sind dann gleich, wenn ihr Inhalt - ihre Werte - übereinstimmen. Prominentes Beispiel für ein ValueObject ist bspw. ein Datum. Es besteht aus 3 int Werten (Tag, Monat, Jahr). Zwei ValueObjekte Datum sind dann gleich, wenn diese drei Werte übereinstimmen. Ein Datum-Objekt verwendet intern keinen Schlüssel. Der Zweck ein Datum-Objekt direkt - also isoliert - in die Datenbank zu speichern (d.h. ohne weitere Daten) wäre fragwürdig. Die Java-Klasse org.joda.DateTime wurden genau im Sinne eines ValueObjects implementiert. Zwei DateTime Klassen sind dann gleich (equals() und hashCode() Methode), wenn die Uhrzeit und das Datum exakt übereinstimmt. ValueObjects können keine Listen oder Referenzen enthalten. Lediglich primitive Datentypen und Status stehen zur Wahl.

Entwickler verwenden das Konzept des ValueObjects um neuen, domänenspezifische primitive Typen zu kreieren. ValueObjects müssen im sogenannten “Immutable Pattern” implementiert werden. Wird ein ValueObject angelegt, so dürfen dessen Werte über den Lebenszyklus des Objektes nicht mehr verändert werden. Stattdessen ist ein neues ValueObject im Speicher zu instanzieren. Dazu stehen die withXXX() Methoden bereit. Sie erstellt eine Kopie des ValueObjects und setzt das entsprechende Datenfeld XXX auf den, als Parameter übergebener Wert. Im Code müssen alle Veränderung mit den withXXX() Methoden oder mit entsprechendem Konstruktor und Parameter erzeugt werden (siehe folgende Abbildung).

DateTime b = a.withDay(12).withMonth(12)

Das Statement erzeugt 2 DateTime Instanzen und weist b die Letzte zu. Für alle Properties in einem ValueObject müssen vom Entwickler withXXX() Funktionen erstellt werden. Moware stellt dafür eine Intention "Create missing methods for proper ValueObject" (ALT+ENTER) zur Verfügung.

Für ValueObjects reicht es nicht aus, die equals() Implementierung in der Voreinstellung "use standard hash" zu belassen. Dies würde dazu führen, dass jedes ValueObjekt jedenfalls von jeder weiteren Instanz verschieden ist. Es müssen im Abschnitt object identity daher alle Werte genannt werden, die beim Vergleich von ValueObjekten zu berücksichtigen sind. Für den Fall des Datum Objektes wären das die drei Felder Tag, Monat und Jahr. Neben der equals() wird dann auch die hashCode() Methode entsprechend korrekt implementiert.

ValueObject Wert

```
// program in immutable style


<doc>
```

properties of ValueObject:

| Type | Name | Short Desc | Long Desc | Format | Options | Documentation |
|---|---|---|---|---|---|---|
| BigDecimal | betrag | ... | ... | ... | ... | ... |
| WertEinheit | einheit | "EH" | "Einheit" | ... | ... | ... |

object identity (overwrite equals() method):

betrag, einheit

members:

```
// watch to program imutable-pattern style
  public Wert() {
    this.einheit = WertEinheit.EUR;
  }
  public Wert(BigDecimal aBetrag) {
    this.betrag = aBetrag;
    this.einheit = WertEinheit.EUR;
  }
  public Wert(BigDecimal aBetrag, WertEinheit aEinheit) {
    this.betrag = aBetrag;
    this.einheit = aEinheit;
  }
  public Wert withBetrag(BigDecimal val) {
    new Wert(val, this.einheit);
  }
  public Wert withEinheit(WertEinheit val) {
    new Wert(this.betrag, val);
  }
  public Wert addBetrag(BigDecimal aBetrag) {
    new Wert(this.betrag.add(aBetrag), this.einheit);
  }
  public Wert subtrectBetrag(BigDecimal aBetrag) {
    new Wert(this.betrag.subtract(aBetrag), this.einheit);
  }
  public Wert multiplyBetrag(BigDecimal aBetrag) {
    new Wert(this.betrag.multiply(aBetrag), this.einheit);
  }
  public Wert divideBetrag(BigDecimal aBetrag) {
    new Wert(this.betrag.divide(aBetrag, 2, BigDecimal.ROUND_HALF_UP), this.einheit);
  }
```

status declarations:

| | | Name | DB value | Short label | Long label | Documentation |
|---|---|---|---|---|---|---|
| status WertEinheit | default= | EUR | EUR | EUR | Euro | <no doc> |

Abbildung 22: ValueObject

MoWare unterstützt aktiv die richtige Handhabung von ValueObjects. So wird bspw. in den Editoren der Benutzeroberfläche (in den Delegates) ein neues ValueObjects erzeugt, sollte ein Wert des Objektes geändert werden. Es wird dann völlig automatisch eine Kopie des Objektes angelegt und das ursprüngliche ValueObject ersetzt.

Ein ValueObject hat weder einen ReadOnly Zustand, noch einen Dirty Zustand. Es kann nicht verändert werden - "immutable". Bei der Implementierung von ValueObjects könnte der Entwickler dennoch Methoden vorsehen, um die Properties des Objektes direkt zu verändern. MoWare führt dazu keine Prüfungen durch. Der Entwickler muss selbst kritisch prüfen. Ein Debugging von derartigen Fehlern ist sicherlich aufwendig.

## *Das ViewObject*

Ein ViewObject ist ein temporärer Datenspeicher, der meist ausschließlich zu Anzeigezwecken verwendet wird. Views sind einfache Objekte, die Werte, Referenzen und auch Listen enthalten können. Im Gegensatz zu Entitäten sind Views allerdings nicht persistierbar. Views definieren häufig in Oberflächen eine eigene Anzeigehierarchie und dienen damit als Bindeglied zwischen Anzeige und persistenter Datenstrukturen. ViewObjekte können in diesem Sinne eine neue Ansicht auf bestehende Datenstrukturen schaffen. Grundsätzlich sind ViewObjekte allerdings sparsam einzusetzen. Neben Entitäten - die ohnedies notwendig sind - müssen ansonsten auch die Views gewartet werden.

## *Optionen für Datenfelder/Properties*

Bei jedem Datenfeld können zusätzliche Optionen hinterlegt werden. So steht in einer Entität die Option KEY für ein Datenfeld zur Verfügung, um dieses Feld als Schlüssel zu markieren. Wie bereits angesprochen müssen Rückwärtsreferenzen mit OPPOSITE markiert werden. Auch die weiteren Optionen sind anzuwenden, um erweiterte Informationen anzuzeigen, die in Zukunft automatische Refactorings von Code ermöglichen.

| **Meta-Information** | **Bedeutung** |
|---|---|
| KEY | Property als Schlüssel markiert . Jede Entität muss über ein Property mit KEY verfügen. Der KEY muss aber nicht nur in der Entität, sondern auch nochmals im Tabellen-Mapping definiert werden. |

| | |
|---|---|
| RANGE (*start, stop, scale*) | Die Option RANGE steht nur für BigDecimal und int zur Verfügung. Sie schränkt den möglichen Wert von Datenfeldern ein: start <= Wert <= stop<br><br>Werte werden auch in der Benutzeroberfläche bei DelegateForms überprüft, lösen allerdings KEINE Exceptions aus, falls im Programmcode ein ungültiger Wert zugewiesen wird.<br><br>Scale gibt die Anzahl der Nachkommastellen des BigDecimal an. Wird einem Datenfeld ein BigDecimal zugewiesen, so wird die Scale angewendet. Mehr dazu bei den Empfehlungen zum Umgang mit BigDecimal. |
| LENGTH(*min, max*) | Nur bei string Datenfeldern gültig. Schränkt die Länge von Zeichenketten ein. min <= Länge <= max<br><br>Werte werden auch in der Benutzeroberfläche in Formularen (DelegateForms) überprüft, lösen allerdings KEINE Exceptions aus, falls im Programmcode ein ungültiger Wert zugewiesen wird. |
| OPPOSITE | Bei Listen-Elementen ist die Rückwärtsreferenz (back. Ref.) zur Vater-Entität mit OPPOSITE zu markieren. Wird das übersehen, so können bei der Serialisierung von Graphen Endlosrekursionen auftreten (Stack-Overflow Exception in Java). |
| CONTAINMENT | Listen die mit Containment markiert wurden, enthalten Elemente, die im Besitz der Vater Entität stehen. D.h. wird das Vater Objekt gelöscht, sind auch alle Elemente in Containment Listen zu löschen. Gegenwärtig hat diese Option noch keine Auswirkungen auf den generierten Code. |
| DEPRECATED | Markiert ein Property als “veraltet”. Referenzen auf dieses Property werden dann vom MPS mit einer Warnungen versehen. |
| NOT_PERSIST_DIRTY_IRRELEVANT | Property soll bei der Dirty Berechnung nicht berücksichtigt werden. D.h. werden Properties, die mit dieser Option versehen wurden verändert, so wird dennoch der Dirty Zustand der Entität nicht verändert.<br><br>Die Option muss berücksichtigt werden und ist für Properties wichtig, die ohnedies nicht auf der Datenbank persistiert werden. Diese Properties lassen sich dann einfach verändern, ohne dass über den Dirty Mechanismus die Entität später auf die Datenbank zurückgeschrieben wird. |

## *Besondere Unterstützung von Referenzen*

Referenzen sind bei der Modellierung von Datenstrukturen von besonderer Bedeutung, da sich dadurch ganze Objektgraphen aufbauen lassen. Objektgraphen sind logische Netzwerke von Objekten, die traversiert werden können. Die Referenz besteht bei MoWare immer aus zwei Komponenten. Dem referenzierten Objekt selbst, und dem Schlüssel des referenzierten Objektes. Obwohl bei der Definition der Datenstrukturen nur das referenzierte Objekt angegeben wird, steht auch dessen Schlüssel direkt - also unabhängig vom Objekt - zur Verfügung.

- Im Fall der Rechnung (Abbildung 19: Entität) kann auf den zugehörigen Rechungskontrollakt über das Property rekoAkt zugegriffen werden. Der Typ dieses Property ist die Entität RekoAkt.
- Neben dem Property selbst stellt MoWare automatisch auch ein Property rekoAkt#Key zur Verfügung, dessen Typ dem Schlüssel im RekoAkt entspricht (Property das im RekoAkt mit KEY markiert wurde). Wird dem Property rekoAkt ein neuer Akt zugewiesen, so wird der rekoAkt#Key automatisch angepasst.

Der direkte Zugriff auf den Schlüssel ist insbesondere beim Laden von Entitäten wichtig. Wird eine Rechnung aus der Datenbank geladen, so kann zunächst ausschließlich auf rekoAkt#Key zugegriffen werden. Die entsprechende RekoAkt Entität wird nicht automatisch "mitgeladen". Möchte der Entwickler das Objekt selbst verfügbar haben, so muss er erst über rekoAkt#Key das Objekt laden und dann anschließend rechnung.rekoAkt zuweisen. Wird ohne Ladevorgang unmittelbar auf rechnung.rekoAkt zugegriffen, so wird eine IllegalAccessError  "not initialized" ausgelöst.

Beim Speichervorgang ist eine Abfolge zu berücksichtigen. Zuerst müssen die Schlüssel von Objekten durch den Speichervorgang festgelegt werden. Dann erst können weiter Objekte gespeichert werden, die diese Objekte referenzieren. Im Fall von RekoAkt und Rechnung enthält die Rechnung eine Referenz auf den Akt. Der Anwendungsentwickler muss daher erst den Akt speichern, dann die Rechnung, die einen rekoAkt#Key enthält. ***(Bis dato known-Bug. Das zusätzliche Key Objekte wird nur bei der Zuweisung angepasst, beim Speichervorgang selbst nicht. MoWare3 .)***

## *Die Meta-Information für Datenfelder/Properties*

Seit MoWare OFX (Sommer 2013) verfügt jedes Property auch über Meta-Information. Generelle Anforderung ist es, bei den verschiedenen Datentypen Informationen zum Typ und weitere Einschränkungen selbst einstellen und abfragen zu können. Dazu zählen unter anderem bei Zeichenketten die Länge, bei int und BigDezimal ein Wertebereich und bei Stati eine Einschränkung möglicher Ausprägungen. Bei Referenzen kann eine Auswahl möglicher Kandidaten als Meta-Information abgelegt werden.

Die Meta-Information ist von den Werten selbst verschieden. D.h. wenn ein Wert neu gesetzt wird, soll dadurch nicht die Meta-Information verändert werden. Damit ist die übergeordnete Entität/ValueObject/ViewObject der richtige Ort, um die Meta-Information abzulegen. Wie beim <<propertyName>>#Key bei Referenzen kann mit <<propertyName>>#Meta auf die Meta-Informationen zugegriffen werden. Die Referenz <<propertyName>>#Meta wird durch MoWare automatisch zur Verfügung gestellt.

```
test (2) EXECUTE  "Retrive #Meta Information from String Field" {
  StringKeyObject sko = new StringKeyObject();

  assert sko.text#Meta.getMax() == 20;
  sko.text#Meta.setMinMax(1, 19);
  assert sko.text#Meta.getMin() == 1;
  assert sko.text#Meta.getMax() == 19;


}
```

Abbildung 23: Zugriff auf Meta-Informationen

Primär werden Meta-Informationen ausschließlich in Benutzeroberflächen geprüft. In Abbildung 23: Zugriff auf Meta-Informationen wurde die Länge einer Zeichenkette zwischen 1 und 19 festgelegt. Dadurch wird nun die Länge der Zeichenkette in den Editoren der Oberfläche überprüft. Der Entwickler kann im Quellcode dennoch Zuweisungen vornehmen, die der Einschränkung widersprechen. Der Entwickler ist im Quellcode nicht an die Einstellungen der Meta-Information gebunden.

Für jeden Datentyp von MoWare kann unterschiedliche Meta-Information eingestellt werden:

- Bei Status Typen können bspw. unterschiedliche Status-Ausprägungen gesperrt werden, so dass diese nicht mehr in der Oberfläche zur Auswahl stehen.
- Bei Int und BigDezimal lässt sich der Wertebereich dynamisch über die #Meta-Information einstellen. Über die RANGE() Option kann der Entwickler bereits bei der Objekt-Deklarationen eine Voreinstellung angeben.
- Bei String kann die Länge vorgegeben werden - es steht die LENGTH() Option zur Verfügung.
- Bei Referenzen können über die #Meta-Information sogenannte Scopes eingestellt werden. Scopes geben zu einer Referenz potentielle Auswahlmöglichkeiten an. So kann bspw. zu einer Artikel-Referenz ein Scope von 5 verschiedenen Artikeln angegeben werden. In der Oberfläche kann ein Anwender dann zwischen den 5 verschiedenen Artikel wählen (sie werden in einem Drop-Down Feld angezeigt).

Neben Einschränkungen können über #Meta-Informationen alle Auswahlfelder in der Oberfläche dynamisch gesteuert werden. Dafür sind folgende Methoden vorgesehen:

setEnabled() - Wert kann editiert werden - ja/nein
setOptional() - Wert muss nicht eingegeben werden / Wert kann dann auch null annehmen

Die Properties (int / BigDecimal / String / Status / Reference) verfügen derzeit über #Meta-Informationen. Virtual-Properties liefern beim Abfragen der #Meta-Information derzeit null. Für LocalDate / DateTime sind #Meta-Informationen verfügbar, allerdings ist min / max auf null gesetzt. setEnabled() / setOptional() ist allerdings verfügbar.

Hat der Anwendungsentwickler bei der Definition von Formularen bereits die Einstellung setEnabled() oder setOptional() vorgenommen (auch setMin(), setMax()), so legt er damit die Default-Einstellung des Formulars fest. Diese Einstellung wird aber durch die #Meta-Informationen überschrieben - sofern sie gesetzt wurde. Fehlen bspw. Range oder Length Angaben in den Datenfeldern der Entitäten, werden diese in der Oberfläche auch nicht kontrolliert.

Bisher wurde noch nicht genauer untersucht, ob Meta-Informationen einen besonders hohen Ressourcen Bedarf nach sich ziehen. Jedenfalls wurde die aktuelle Implementierung nicht optimiert. Zugriffe im Quellcode mit <<propertyName>>#Meta sollten gegenwärtig sparsam eingesetzt werden, falls am Konzept in Zukunft Anpassungen vorgenommen werden. Die Verwendung der Optionen von Range und Length in Objekt-Deklarationen wird empfohlen.

## *Anwendungsempfehlungen zu BigDecimal*

Der Moware-Stack verwendet für die Darstellung von Werten in Datenstrukturen grundsätzlich java.math.BigDezimal. Diese Klasse speichert Festkommawerte mit endlicher Genauigkeit, d.h. Werte werden mit unendlicher Genauigkeit gespeichert. Periodische Zahlen müssen gerundet werden. Der Datentyp int sollte damit nur mehr für Schlüssel bei Entitäten (Option KEY) oder für Zähler verwendet werden.

BigDecimal unterstützt alle bekannten Operation und Funktionen. In der Hilfsklasse MU stehen zahlreiche Operationen für BigDecimal zur Verfügung. Unter anderem Summenfunktionen oder Vergleichsfunktionen mit Angabe einer Genauigkeit. Die Klasse UserEnvironmentInformation enthält als statisches Feld einen Formatter für BigDecimal. Dieser sollte für die Ausgabe von Werten verwendet werden. Als "Default" Format wird "#,##0.00" verwendet, was zwei Nachkommastellen mit Tausender-Trennzeichen erzwingt.

Beim Arbeiten mit BigDecimal ist die richtige Präzision - Anzahl der Nachkommastellen - zu berücksichtigen. Diese sollte durch die bereits erwähnte RANGE Option spezifiziert werden. Üblich sind 0, 2 oder 4 Nachkommastellen. Die Zahl der Nachkommastellen (spezifiziert in der #Meta-Information) wird durch den Moware-Stack erst bei einer Zuweisung gesetzt. Enthält ein Objekt bspw. eine BigDecimal Variable “wert”, so wird erst bei der Zuweisung

<<object>>.wert = <<Berechnung hier>>

die Rundung angewendet. Die Berechnung selbst erfolgt in unendlicher Präzision. Damit führen folgende zwei Beispiele zum exakt selben Ergebnis:

Rechnung.gesamtWert = sum ( PosWerte * 0.96 )  führt zum exakt selben Ergebnis wie
Rechnung.gesamtWert = sum ( PosWerte ) * 0.96

Ist ein Positionswert als BigDecimal mit der Genauigkeit von 4 Stellen modelliert, so kann die Multiplikation mit 0.96 zu einem Wert mit 6 Stellen führen. Dieser wird während der Berechnung aber nicht gerundet. Erst bei der Zuweisung auf gesamtWert, wird - am Ende aller Berechnungen - der Wert auf die Zahl der Stellen des "gesamtWertes" gerundet.

Üblicherweise ist es passend, bei firmeninternen Berechnungen die unendliche Genauigkeit anzuwenden und dann erst das Endergebnis zu runden. Müssen Berechnungen von anderen gerpüft werden, ist erst festzustellen, wie sie die Berechnungen durchführen, denn evtl. runden sie Teilergebnisse. Falls notwendig, können Teilergebnisse in Berechnungen mit der Operation <<BigDezimalVariable>>.scale(n, RoundMode.HALF_UP) auf n Stellen gerundet werden.

Der Anwendungsentwickler muss Divisionen durch 0 selbst prüfen. Eine automatische Prüfung wird nicht durchgeführt. Auch Divisionen, die in periodischen Zahlen resultieren, müssen berücksichtigt werden. Daher ist bei Divisionen grundsätzlich immer divide(BigDecimal divisor, int scale, int roundingMode), mit Rounding-Mode RoundMode.HALF_UP, zu verwenden.

Die #Meta-Information beeinflusst grundsätzlich nie die Datenfelder (Properties) sondern dient nur der Validierung in Benutzeroberflächen. Eine ausnahme bilden BigDecimal Properties. Bei Zuweisung von BigDecimal Properties in Entitäen/ValueObjects/Views wird ein .scale() angewendet, falls in der #Meta-Information die Zahl der Nachkommastellen eingestellt wurde.

## *Anwendungsempfehlungen zu Status-Feldern*

Wie in Abbildung 3: Lieferaviso zu erkennen, wird eine der Ausprägungen (der Erste in der Tabelle) als Default angesehen. Wird ein neues Objekt erzeugt, so wird der Status mit der Default-Ausprägung initialisiert. Grundsätzlich sollte der Status nie auf null gesetzt werden. Ausnahmen sollten lediglich Formulare bilden. Wurde über die Meta-Information setOptional(true) gesetzt, so kann der Wert des Status null annehmen. Gibt der Endanwender in der Oberfläche keinen Status an, so wird er dann null gesetzt.

Problematisch: Wird ein Statusfeld aus der Datenbank geladen, wird gegenwärtig nicht geprüft, ob der Statuswert im Modell definiert wurde. Ferner wird ein null im Datenbankfeld bei einem Status beim Lesen über den StringConverter auf “” gesetzt - damit ist der Status dann also gerade nicht null. Fraglich ist im Moment, ob dieses Verhalten des Status in dieser Form gewünscht ist oder ob es in Zukunft angepasst werden muss.

Status können über #Meta-Information im Code in Zeichenketten (DB-Value, Short- und Long-Description) konvertiert werden. Zeichenketten können auch in Status konvertiert werden (StatusFromString Konzept in der Code-Completion). Im Quellcode sollten derartige Konvertierungen verhindert werden, da sie Fehleranfällig und mit Wartungsaufwand verbunden sind.

## *Properties mit besonderer Bedeutung*

Für Bibliotheken im Java-Enterprise Umfeld ist das Auditoring von Datensätzen und -objekten meist besonders wichtig. Festgehalten wird dabei, welche User zu welchem Zeitpunkt ein Datenobjekt verändert haben. MoWare unterstützt Auditoring in grundlegenden Formen. Gegenwärtig kann MoWare die UserId und Zeitstempel bei insert/updates auf der Datenbank automatisch ausfüllen. Dazu steht eine Intention “Make Entity auditable” bei Entitäten zur Verfügung, die vier Properties automatisch anlegt; diese müssen dann zusätzlich gemappt werden (siehe Datenbankmapping im nächsten Abschnitt). "Auditable" Properties müssen wie folgt benannt sein:

| **Property Name** | **Funktion** |
|---|---|
| zzCreatedAt | Zeitstempel (LocalDate oder DateTime), übernommen aus der Klasse UserEnvironmentInformation als Long (milliseconds); bei |

| | |
|---|---|
| | einem Insert neu gesetzt. |
| zzCreatedAtUid | UserId, übernommen aus der Klasse UserEnvironmentInformation, bei einem Insert neu gesetzt. |
| zzModifiedAt | Zeitstempel (LocalDate oder DateTime), übernommen aus der Klasse UserEnvironmentInformation als Long (milliseconds), bei Update neu gesetzt. |
| zzModifiedAtUid | UserId, übernommen aus der Klasse UserEnvironmentInformation, bei Update und Insert neu gesetzt. |

Bis dato muss die UserId im startup() der Applikation gesetzt werden. Der Anwendungsentwickler kann über die StandAloneApplicationFactory mit der Methode findInstanceByName die Instanz "userEnvironmentInformation" abfragen. Mit setUserId() ist die gewünschte ID zu setzen. Beispiele zu diesem Vorgehen finden sich in den Applikationen. In einer nächsten Version von MoWare ist die Auditoring Funktionalität zu ergänzen.

## *Technische Information (Deprecated)*

*Wichtige Einschränkung: Referenzen können gegenwärtig NICHT zur Identitätsdefinition hinzugefügt werden (resultiert ansonsten in einem Fehler beim Laden; hashCode() der Entität kann nicht berechnet werden; wenn Referenz in Entität nicht geladen -> Exception, "not initialized").* ***NICHT MEHR GÜLTIG: Referenzen können bei der Identitätsdefinition verwendet werden, es wird dann automatisch der Schlüssel der Referenz (<<name>>#KEY) und nicht die Referenz selbst verglichen.***

*Frage: Weitere Festlegung der Identität mit zusätzlichen Attributen überhaupt noch sinnvoll? Nein, im Gegenteil, das kann zu Fehlern führen, wenn die Enität noch nicht gespeichert wurde. Daher unbedingt <use standard hash> einstellen. Dann vergleich auch bei Entitäten mit 0 Schlüssel durch Session möglich "==", d.h. Entitäten innerhalb einer Session müssen nicht mit equals() verglichen werden.*

*Frage: Können Operatoren für ValueObjects überlagert werden - der besseren Lesbarkeit halber? Ist in MPS jedenfalls möglich! (Generell "Syntactical Sugar", im Moment nicht prio)*

*Vorsicht: Entitäten, Commands und Prozesse dürfen nicht gleich benannt werden. Aus den Konzepten entstehen meist einfache Java-Klassen. Im selben Namespace dürfen diese selbstverständlich nicht gleich heißen.*

## 4. Mapping und Datenbankabfragen

Die Persistierung von Datenstrukturen ist einer der wesentlichen Aufgaben bei der Erstellung von Geschäftsanwendungen. In diesem Kapitel werden alle damit in Verbindung stehenden Konzepte vorgestellt und diskutiert. Im Vergleich zu anderen Konzepten gehört das Datenbank-Mapping, die Abfragesprache und die Session Logik sicherlich zu den komplexesten Features von MoWare. Als Objektmapper wird die Sprache org.modellwerkstatt.manmap verwendet, der mit seinen Ideen an dem prominenten Java-Mapper hibernate angelehnt ist.

### *Einführung ModelRepository und Session*

**Checkin / Checkout**

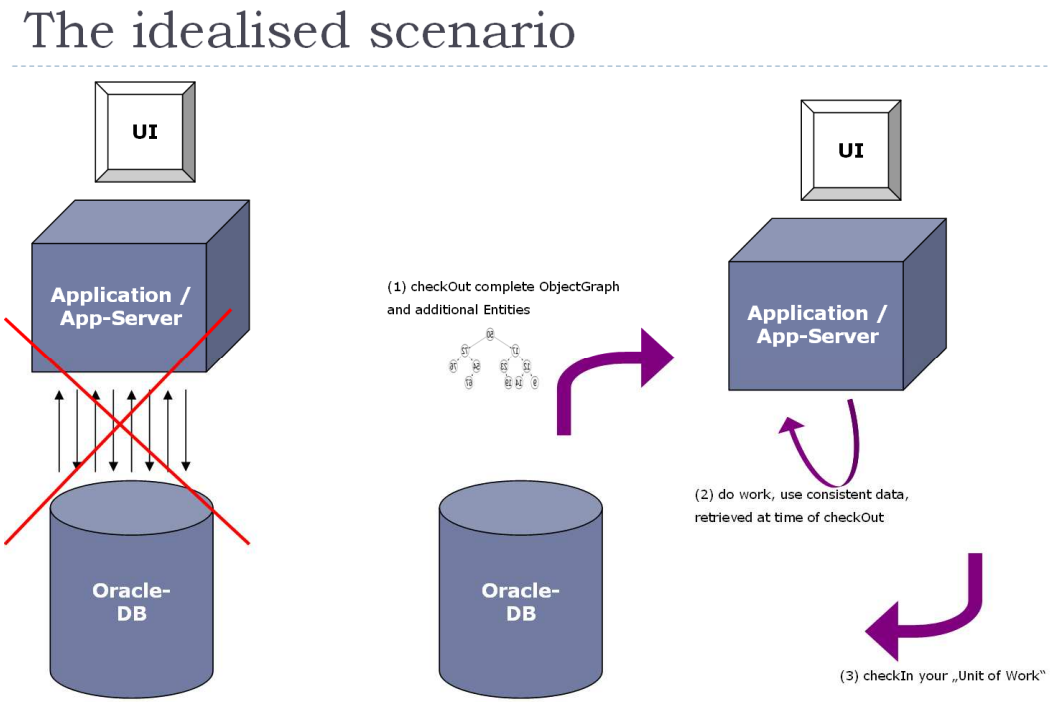



Abbildung 24: Checkin

Grundsätzlich sind MoWare Applikationen in mehrere "Unit of Work" unterteilt. Eine "Unit of Work" soll in diesem Zusammenhang ein mehr oder weniger großer Arbeitsschritt mit Benutzerinteraktion darstellen. Stellvertretend könnte die Erstellung und Korrektur eines Rechnungskontrollaktes oder das Erstellen eines Vertrages als Beispiel für eine große "Unit of Work" genannt werden.

Wie in der obigen Abbildung dargestellt, wird am Beginn eines Arbeitsvorganges ein Objektgraph von der Datenbank “ausgecheckt” (checkout). Im Fall des Rechnungskontrollaktes wird ein RekoAkt mit den enthaltenen Rechnungen und deren Positionen in einer Repository-Methode geladen und zu einem Objektgraphen zusammengestellt. Anschließend wird dieser Objektgraph in der Geschäftsanwendung bearbeitet. Entitäten können ergänzt oder verändert werden. Nachdem der Anwender die Arbeit abgeschlossen hat, wird der komplette Objektgraph durch eine Repository-Methode wieder "eingecheckt" (checkin). Damit werden allerdings nur jene Entitäten gespeichert, die als Dirty markiert wurden. Alle anderen Entitäten werden auf den Datenbanktabellen nicht verändert (weder deren Versionsnummer TCN, noch createdAt, modifiedAt Informationen).

Im Idealfall sollte der Entwickler ständiges Lesen von - und Schreiben auf die Datenbank vermeiden, da bei mobilen Zielsystemen die Verbindungen unterbrochen werden könnte. Ferner wird bei ständigem Lesen und Schreiben mit zeitlich inkonsistenten Daten gearbeitet. So kann auch beim Optimistic-Locking ständiges Lesen und Schreiben zu häufigen Versionsfehlern führen. Um inkonsistente Lese-/Schreibzyklen zu verhindern, wurde das Konzept der Session eingeführt.

**Session**

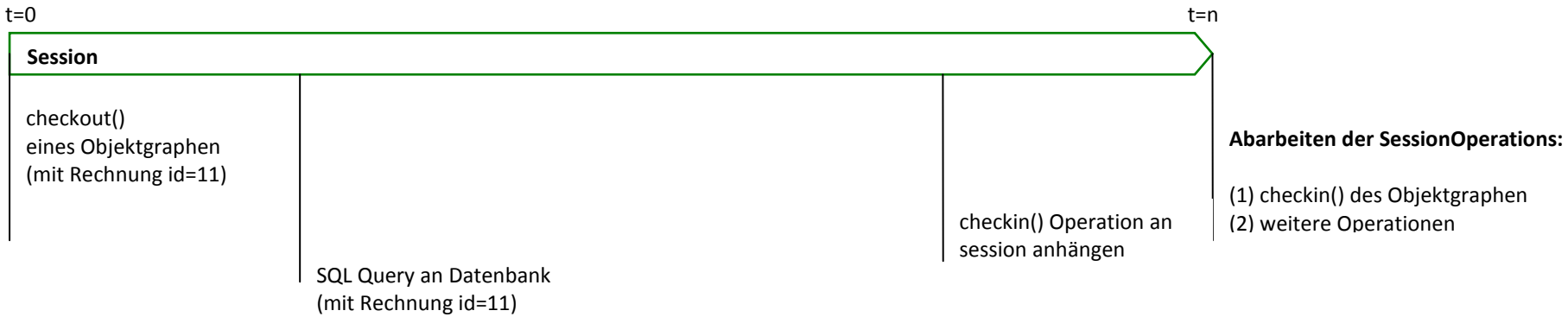



Abbildung 25: Session

Eine Session kann über mehrere Kommandos hinweg verwendet werden. Sie markiert den Anfang und das Ende einer "Unit of Work". Zu Beginn muss der Entwickler meist einen Objektgraphen laden und in einer Repo-Methode den "checkout" durchführen. Durch diesen Vorgang übernimmt der Anwender einen bestimmten Datensatz zu

Bearbeitung. Er kann ihn dann verändern. Nach abgeschlossener Arbeit gibt er den Datensatz, oder besser die Datensätze, durch einen finalen "checkin" Vorgang wieder zurück, so dass er für die Bearbeitung durch andere wieder zur Verfügung steht.

Im Quellcode wird der "checkin" nicht einfach aufgerufen, sondern er wird bei der Session zum Aufruf registriert. Wird dann die Session beendet, werden alle Operationen in einer Datenbanktransaktion abgearbeitet. In obigen Fall wird vor dem Beenden der Session der "checkin" registriert. Man beachte: Auch ein doppelter "checkin" würde keine Probleme verursachen. Bei der ersten Operation werden alle Dirty Entitäten gespeichert, bei der zweiten Operation liegt nichts mehr im Zustand Dirty vor.

Die Session dient auch als First-Level-Cache. In dem Beispiel von Abbildung 25: Session wird innerhalb der Session eine SQL Abfrage durchgeführt. Datenbankabfragen nach einem "checkout" sind nicht unüblich. Es gilt allerdings zu berücksichtigen, dass die Session bei Datenbankabfragen mitunter den Zustand von Entitäten zum Zeitpunkt des "checkouts" zurückliefert. Wurde bspw. zu Beginn der Session unter anderem eine Rechnung mit id 11 geladen und die selbe Rechnung (id 11) ist auch in der Ergebnismenge der SQL Abfrage, so wird nicht ein neues Objekt Rechnung mit id 11 instanziert, sondern die bereits erzeugte Instanz während des "checkouts" zurückgegeben. Wurden bereits Änderungen an der Instanz durchgeführt, erweckt es den Anschein, als ob diese auf der Datenbank bereits gespeichert wurden. Dies ist allerdings nicht der Fall! Die Session wirkt wie ein First Level-Cache. Sind Entitäten in der Ergebnismenge des SQL-Statements, die noch nicht in der Session geladen wurde, spiegeln diese tatsächlich den Zustand der Datenbank zu dem Abfragezeitpunkt wieder (dadurch mitunter Zeitinkonsistenz).

Datenbankabfragen können im Modus ReadOnly oder Checkout erfolgen. Entitäten, die ReadOnly geladen wurden, können nicht verändert werden. Sie können in derselben Session auch nicht nochmals ausgecheckt werden. Entitäten, die mit Checkout geladen wurden, können verändert und wieder gespeichert werden. Treten Fehler während der Ausführung auf, ist meist die komplette Session zu verwerfen.

**ModelRepository und Repo-Methoden**

ModelRepositories bilden die Schnittstelle zwischen Geschäftslogik in Kommandos und der Datenbank. Sie kapseln damit alle Datenbankabfrage in einem einheitlichen Konzept. Alle SQL selects, inserts und updates müssen in einem ModelRepository durchgeführt werden. ModelRepositories sind für alle Datenbankinteraktionen verantwortlich: checkin, delete, checkout und natürlich einfache Read-Only Abfragen (find).

Wie bereits angesprochen beginnt der übliche Ablauf einer "Unit of Work" mit einem "checkout" der relevanten Datenstrukturen. Dann werden diese verändert und schließlich mit einer Session-Operation wieder "eingecheckt". Im einfachsten Fall können Checkout und Checkin eines Objektgraphen als zwei Methoden in einem Model-Repository abgebildet werden. Die Methoden in einem ModelRepository sollen allerdings nicht nur einfache Listen von Entitäten liefern. Sie erzeugen ganze Objektbäume (z.B. Rechnungsakt mit Rechnungen und Rechnungspositionen in einem Objektgraphen) und liefern die Wurzel der Objektgraphen zurück. Alle notwendigen Referenzen und Kind-Entitäten in Listen sollen im Repository geladen werden, so dass es bei späteren Zugriffen auf Referenzen nicht zu IllegalAccessExceptions kommt (siehe dazu Abschnitt Referenzen in Entitäten im vorigen Kapitel).

## *Die PersistenceDescription*

Die PersistenceDescription besteht aus mehreren Mapping-Vorschriften, die die Abbildung von Entitäten auf die Datenbanktabellen festlegen. Dabei kann eine Entität mit unterschiedlichen Vorschriften abgebildet werden, bspw. auf verschiedene Tabellen oder auch mit eingeschränkter Felderzahl zur Optimierung der Performance und aus Sicherheitsüberlegungen.

```
map Rechnung as MapRechnung
  on table "X_RECHNUNG" options OPTIMISTIC_LOCK {
  int id -> "id" KEY, AUTOID("SX_RECHNUNG")
  int nrBeleg -> "NUM_BELEG" <opt>
  LocalDate datRech -> "belegDat" <opt>
  LocalDate datEingang -> "eingangsDat" <opt>
  LocalDate datFaellig -> "faelligDat" <opt>
  string nrRechnungL -> "extBelegNr" <opt>
  string refRechnungL -> "ref_extbelegNr" <opt>
  embedded bestellung {
    int nummer -> "num_bestellung" <opt>
    LocalDate datum -> "dat_bestellung" <opt>
  }

  embedded lieferung {
    string nummer -> "num_lieferung" <opt>
    LocalDate datum -> "dat_lieferung" <opt>
  }

  ref VorgangsOrt lieferant: int id -> "lieferant"
  embedded glnLieferant {
    string code -> "glnLieferant" <opt>
  }

  ref VorgangsOrt kaufer: int id -> "kaufer"
  ref VorgangsOrt lieferStelle: int id -> "lieferStelle"
  status steuerCode -> "steuerCode" <opt>
  string uid -> "cod_uid" <opt>
  status typBeleg -> "typBeleg" <opt>
  status typRechnung -> "typRechnung" <opt>
  status typErfassung -> "typErfassung" <opt>
  embedded summeBrutto {
    BigDecimal betrag -> "btr_brutto" <opt>
    status einheit -> "eh_brutto" <opt>
  }

  embedded summeWarenWert {
    BigDecimal betrag -> "btr_warenWert" <opt>
    status einheit -> "eh_warenWert" <opt>
  }

  embedded summeZuAb {
    BigDecimal betrag -> "btr_zuab" <opt>
    status einheit -> "eh_zuab" <opt>
  }

  ref RekoAkt rekoAkt: int id -> "refRekoAkt"
  status status -> "status" <opt>
}
```

Abbildung 26: Mapping-Vorschrift

Oben dargestellt ist die Mapping-Vorschrift für eine Rechnungsentität, bezeichnet als MapRechnung. Als Option wurde OPTIMISTIC_LOCK angegeben, d.h. auf der Datenbank Tabelle "X_RECHNUNG" wird zusätzlich ein TCN Feld für Versionsnummern angelegt und vom Mapper verwaltet. Damit können Entitäten nur dann gespeichert werden, wenn die Versionsnummer checkin/checkout übereinstimmt. Sollten andere Anwender in der Zwischenzeit einen Speichervorgang durchführen, ist ein erneuter checkin nicht mehr möglich. Es ist ratsam, grundsätzlich immer auf Optimistic Locking zurückzugreifen.

Die id Property wird auf das Feld id abgebildet und in der Tabelle als eindeutiger Schlüssel verstanden, wobei bei einem SQL Insert Statement die Nummer erst aus der Sequenz SX_RECHNUNG gezogen wird (Option KEY, AUTOID). Weitere Zuordnungen zwischen LocalDate und int auf Tabellenspalten sind selbsterklärend. Diverse ValueObjects in der Entität Rechnung (z.B. bei Referenz bestellung, lieferung) wurden mit einem embedded Mapping abgebildet. Die Felder der ValueObjects werden dabei direkt in die Tabelle der Entität übernommen. Referenzen wurden mit Ref-Mappings integriert. Dabei wurde der Schlüssel der referenzierten Entität auf eine Tabellenspalte abgebildet.

Primitive Datentypen werden direkt auf ein Feld (FieldMapping), Referenzen abhängig vom Schlüssel der referenzierten Entität (meist nur int, daher FieldMapping) und ValueObjects auf mehrere Felder (EmbeddedMapping) abgebildet. IncludeMapping kann ein bereits bestehendes Mapping in eine weitere Mapping-Vorschrift für eine Entität übernehmen. Ferner stehen Optionen wie KEY, AUTOID, NOTNULL, SIZE, INDEX etc. zur Verfügung, die in generierte SQL Tabellenbeschreibungen übernommen werden.

Werden Properties in Entitäten (int, string, etc.) auf die Datenbank geschrieben, so konvertiert MoWare diese zuerst mit TypeHandlern. TypeHandler werden auch zur Konvertierung beim Lesen von Daten verwendet. Sie wenden gegenwärtig folgende Konventionen an.

| **DatenTyp** | **Lesen** | **Schreiben** |
|---|---|---|
| integer | bei null auf Datenbank, Konvertierung auf 0 | bei null als Java Int-Objekt, Konvertierung auf 0 |
| integer (als Schlüssel bei Referenzen) | Keine Referenz bei<br><br>value == 0 \|\| value == -1 \|\| value == null | Keine Referenz wird als 0 geschrieben |
| BigDezimal | null wird auf 0.0 konvertiert | null wird auf 0.0 konvertiert. Dadurch können später auf Spalten ohne Probleme Aggregationsfunktionen ausgeführt werden.<br><br>Möchte man auf die Datenbank einen Tri-State Wert wie null schreiben, so muss man neben einem BigDecimal noch einen Status verwenden. |
| DateTime | null wird **nicht konvertiert**, Java Objekt ebenfalls null | Java Objekt null wird nicht konvertiert, null auf DB |
| LocalDate | null wird **nicht konvertiert**, Java Objekt ebenfalls null.<br><br>Da Datum ebenfalls als Timestamp auf der Datenbank, können beliebige TimeStamp-Felder gelesen werden (automatische | Java Objekt null wird nicht konvertiert, null auf DB<br><br>Da auch Datum als TimeStamp gespeichert, wird Zeit auf 00:00:000 gesetzt. |

| | | |
|---|---|---|
| | Konvertierung zu Date) | |
| String | null wird in Java Objekt “Leerstring” konvertiert (“”) | Wunsch: null Java Objekt wird in “Leerstring” konvertiert (“”) und auf DB geschrieben<br><br>Bei Oracle konvertiert die Datenbank den Inhalt einer Varchar Zelle auf null, wenn die Zeichenkette leer ist ("") |

Um Joins zu erlauben, können auch Listen von Entitäten/ValueObjects in Mappings deklariert werden. Listen sind nicht anfällig bei Veränderung des hashCode() und lassen sich auch einfach sortieren. Listen sind gegenwärtig ausschließlich mit Entitäten/ValueObjects als Elemente zu verwenden, d.h. grundsätzlich ist es nicht vorgesehen, Listen mit Primitiven (etwa BigDezimal/Int oder string etc.) im Code zu verwenden.

*Lessons Learned:*

*Auch Oracle-Felder wie ROW-NUM lassen sich über Mappings erfassen. Dadruch können dann auch Abfragen darauf zugreifen, z.B. zur Einschränkung von Ergebnis-Sets (nur 1000 Zeilen; Vorsicht - ROW-NUM berücksichtigt Sortierung nicht).*

*Weiters zu berücksichtigen: Bei SQL-Insert Operation nehmen nicht abgebildete Felder (bei nicht vollständiges Mapping) in der Tabelle dann den Wert null an.*

## *Datenbankabfragen mit ManMap*

Datenbankabfragen werden nicht in SQL, sondern mit eigens dafür vorgesehenen Konzepten geschrieben. Damit vereinfachen sich Abfragen wesentlich, da der Anwendungsentwickler nur mehr mit Objekten und nicht mehr mit einzelnen Feldern bei der Datenbankinteraktion arbeitet. ManMap unterstützt folgende Funktionalitäten:

| Funktionalität | Beschreibung |
|---|---|
| <<MappingVorschrift>>.get(<<key>>) | Lädt eine Entität mit dem Schlüssel <<key>> über gegebene Mappingvorschrift von der Datenbank. Exakt eine Entität muss in der entsprechenden Datenbank Tabelle gefunden werden, ansonsten wird eine IncorrectResultSetColumnCountException() geworfen. |
| <<MV>>.where(<<condition>>) | Lädt eine Liste von Entitäten von einer Datenbanktabelle, für welche die angegebene Bedingung gilt. Für das RowMapping wird die angegebene MappingVorschrift verwendet. Liste kann auch 0 Entitäten enthalten. |
| <<MV>>.where(<<condition>>) | Lädt eine Liste von Entitäten, wobei diese zusätzlich auf der Datenbank sortiert werden (aufsteigend, absteigend). Es können mehrere |

| | |
|---|---|
| .sortBy(<<field>>, ASC/DESC) | sortBy() angefügt werden. Ein where() ist zwingend, kann aber 1==1 als Bedingung enthalten. |
| <<MV>>.reload(<<Entität>>) | Lädt die gegebene Entität (wurde zuvor bereits von der Datenbank geladen) nochmals nach. Dabei wird das gegebene Objekt nicht zerstört. Objekte, die diese Entität referenzieren, behalten eine gültige Referenz.<br><br>DEPRECATED: WIRD NICHT MEHR UNTERSTUETZT. |
| save with <<MV> (auto) <<obj>> | Speichert eine Entität mit gegebener MappingVorschrift auf die Datenbank. Gewählt werden kann zwischen insert, update und auto. Auto führt ein SQL insert aus, wenn bei<br>- Integer-Schlüsseln der Wert <= 0<br>- String-Schlüsseln = null oder “”<br>- Value-Object Schlüsseln isNull() == true ist<br>ansonsten wird ein SQL update ausgeführt.<br><br>Save wird nur ausgeführt, falls die zu speichernde Entität als Dirty markiert wurde.<br><br>Save muss immer in Transaktionen ausgeführt werden, d.h. save Aufrufe müssen immer an Session mit addOperation() gehängt werden, anschließend mit startTransactionAndFlush() Transaktion ausführen. (siehe Abschnitt zur Session) |
| delete with <<MV>> <<obj>> | Delete löscht eine Entität mit gegebener MappingVorschrift. |

Bei Where-Abfragen konvertiert ManMap LocalDate Datentypen auf Timestamp, bei denen entsprechend auf den Tag (Beginn / Ende) abgegrenzt wird. Auch größer / kleiner Operatoren werden korrekt auf den Tag abgegrenzt. Zusätzlich lassen sich bei Abfragen auch DateTime Felder auf das Datum eingrenzen. Dazu muss bei der entsprechenden DateTime MappingReference in der Bedingung im Inspektorfenster das “*force to* “ auf “LocalDate” gestellt werden.

In Where-Abfragen steht zusätzlich ein In-Operator, ein Like-Operator und ein Optional-Operator zur Verfügung. Der In-Operater bildet das Äquivalent zum SQL IN Operator. Er kann gegenwärtig mit int und string Objekten angewendet werden. Der Optional-Operator prüft erst, ob der Operand null (oder bei int 0) entspricht. Falls dies der Fall ist, wird die Bedingung übergangen. Der Like-Operator entspricht dem SQL LIKE Operator. Er muss mit CRTL-Space “like” gewählt werden.

Where-Abfragen lassen sich mit joins versehen. So kann bspw. eine **Referenz** der Entität A auf Entität B direkt mit einer Abfrage über Mapping-Vorschrift für Entität A und Join der Referenz mit Mapping-Vorschrift B geladen werden. Ist ein Schlüssel auf B in A eingetragen, die referenzierte Enität B existiert allerdings nicht, so kommt es zu keinem Fehler bei der Abfrage (im Gegensatz zu einem get(), das eine Exception wirft). Es kann später allerdings zu einem IllegalAccessError kommen, wird auf die Referenz zugegriffen (Grund: Schlüssel hat einen Wert, Entität aber nicht geladen). *Um Fehler sofort zu erkennen, sollten Referenzen mit get() nachgeladen werden, außer es ist eine Bedingung auf die Referenz notwendig.*

Zusammenfassend ist in Abbildung 27: Rechnung checkout und Abbildung 28: Rechnung checkin ein komplettes Beispiel mit get() und where() Abfragen dargestellt. Der checkin verwendet die save Operation.

```
//
CHECKOUT Rechnung checkoutRechnung(int id) {
  Rechnung rechnung = MapRechnung <join> get(id) ReadOnly;

  // Details zu einer Rechnung auschecken. Vorgangsorte, Positionen,....
  rechnung.lieferant = MapVorgangsOrt <join> get(rechnung.lieferant#Key) ReadOnly;
  rechnung.lieferant.partei = MapPartei <join> get(rechnung.lieferant.partei#Key) ReadOnly;
  rechnung.kaufer = MapVorgangsOrt <join> get(rechnung.kaufer#Key) ReadOnly;
  rechnung.kaufer.partei = MapPartei <join> get(rechnung.kaufer.partei#Key) ReadOnly;
  if (rechnung.lieferStelle#Key != 0) {
    rechnung.lieferStelle = MapVorgangsOrt <join> get(rechnung.lieferStelle#Key) ReadOnly;
  }
  rechnung.rechPos = MapRechPos <join> where({~rechpos,  => rechpos.beleg.id == rechnung.id  }) Checkout;
  rechnung.rechPos.forEach({~it =>
    it.beleg = rechnung;
    it.artikel = MapArtikelWWS <join> get(it.artikel#Key) ReadOnly;
  });
  rechnung.steuerPos = MapSteuer <join> where({~steuer,  => steuer.beleg.id == rechnung.id  }) Checkout;
  rechnung.steuerPos.forEach({~it => it.beleg = rechnung; });
  rechnung.zuAbPos = MapZuAb <join> where({~zuab,  => zuab.beleg.id == rechnung.id  }) Checkout;
  rechnung.zuAbPos.forEach({~it => it.beleg = rechnung; });
  rechnung.rechKorrInfos = MapRechKorrInfo <join> where({~rechkorrinfo,  => rechkorrinfo.beleg.id == rechnung.id  })
      Checkout;
  rechnung.rechKorrInfos.forEach({~it => it.beleg = rechnung; });
  rechnung;
}
```

Abbildung 27: Rechnung checkout

```
//
CHECKIN void checkinRechnung(Rechnung rechnung) {
  // Wenn die interne BelegNr noch nicht gesetzt ist wird die Nummer hier gezogen
  save with MapRechnung, auto (rechnung);
  rechnung.rechPos.forEach({~it => save with MapRechPos, auto (it); });
  rechnung.steuerPos.forEach({~it => save with MapSteuer, auto (it); });
  rechnung.zuAbPos.forEach({~it => save with MapZuAb, auto (it); });
  rechnung.rechKorrInfos.forEach({~it => save with MapRechKorrInfo, auto (it); });
}
```

Abbildung 28: Rechnung checkin

Hinweis: Am Ende jeder Abfrage ist entweder Checkout oder ReadOnly vermerkt.

## *Ergänzungen zu Referenzen*

In diesem Abschnitt wird die Referenzierung zwischen Entitäten nochmals aufgegriffen. Referenzen werden nicht automatisch von ManMap aufgelöst. Es findet auch kein sogenanntes Lazy-Loading statt. Angenommen eine Entität A enthält eine Referenz “refB” auf ein Objekt B, so steht in der Entität A sowohl der Schüssel von B, als auch die Entität B selbst zur Verfügung. Der Schlüssel zu B kann mit refB#Key, die Entität mit refB abgefragt werden.

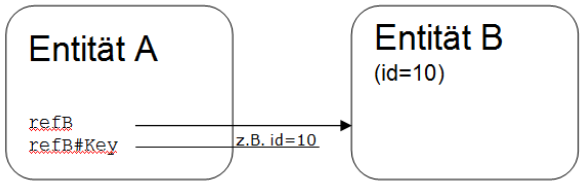



Abbildung 29: Referenzen

Wird Entität A mit Hilfe von einem Mapping geladen, wird nur das Klassenfeld “Schlüssel von B” geladen - sprich das Property refB#Key. Die Referenz refB muss bei Bedarf entweder manuell mit einer <<MappingVorschrif für B>>.get() abgefragt und refB zugewiesen, oder auch mit einem join unter Angabe der Mapping-Vorschrift für B direkt in der Abfrage von A geladen werden.

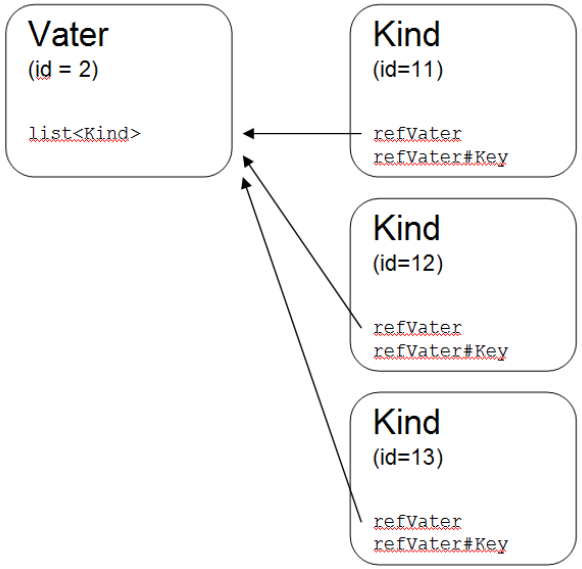


Abbildung 30 Liste von Kind-Entitäten

Listen werden üblicherweise über eine sogenannte Rückwärts-Referenz (eine Referenz auf das Vater-Objekt, OPPOSITE) in SQL Schemata abgebildet. Bei dieser Abbildung wird auch von einem CONTAINMENT der Listen-Entitäten (Kind-Entitäten) in der Vater-Entität gesprochen. In der Mapping-Vorschrift der Vater-Entität muss die Liste grundsätzlich nicht modelliert werden. Eine Liste von Kind-Entitäten lässt sich der Vater-Entität anhängen, indem eine zusätzliche Abfrage mit der Kind Mapping-Vorschrift über die Rückwärtsreferenz ausgeführt wird, bspw. in der Form KindMapping.where(refVater#Key == 2). Sollen allerdings Kind-Entitäten über ein SQL Join mit der Vater-Entität geladen werden, muss die Liste auch im Mapping modelliert werden. *Alternativ könnte man bei ManMap in Zukunft direkt bei der Join-Abfrage die zu verwendende Kind Mapping-Vorschrift angeben. Nicht CONTAINMENT Listen müssen über Indirektion (eine zusätzliche Tabelle und eine Verbindungsentität) aufgebaut werden.*

Bei Join-Abfragen liegen im Moment **Einschränkungen vor:** Queries mit mehreren Join/RefSelects können zu Problemen führen, insbesondere dann, wenn mit einer Session gearbeitet wird (was dem Normalfall entspricht). Diese Einschränkungen können mit einer ManMap Ver 2.0 behoben werden, bei dem where-Queries dann zu optimieren sind.

Selbstverständlich müssen auch Referenzen in Entitäten nicht zwingend gemappt werden. Referenzen/Fremdschlüssel sind vielleicht nicht in der Datenbanktabelle vermerkt, sondern müssen aus weiteren Tabellen berechnet und beim Laden vom Objektgraphen gesetzt werden. Dadurch können sich Entitäten dann wesentlich von Datenbanktabellen unterscheiden - was richtig ist. Entitäten sind nicht mit Tabellen gleichzusetzen. Eine Entität ist ein für die Domäne passendes Datenobjekt.

Zur Verwendung von zusammengesetzten Schlüsseln: ValueObjects lassen sich auch als eindeutiger Schlüssel einer Entität verwenden. So ist z.B. ein Valueobject Konto mit (KontoNr, KostenstelleNr) als Schlüssel einer Entität denkbar. Allerdings sollte dann zusätzlich eine isNull() Methode im ValueObjekt implementiert werden, die prüft, ob der Schlüssel als nicht gesetzt gilt (z.B. wenn alle Integer-Felder = 0 und String-Felder = “”; das entspricht gerade der MoWare default Initialisierung). Dieser “nicht gesetzt” Schlüssel wird dann bei Referenzen herangezogen, die als "nicht gesetzt" gelesen werden sollen. Als Vergleich dazu: Bei einfachen Integer-Schlüsseln wird 0 als nicht gesetzt verstanden. Folglich kann mit MoWare <<obj>>.refA#Key.isNull geprüft werden, ob der Schlüssel nicht gesetzt wurde. (<<obj>>.refA#Key liefert NICHT null oder 0 , sondern eben ein Leer-Valueobjekt; <<obj>>.refA selbst liefert null - siehe dazu Test 47 im ManMapTestCases)

## *Die Session*

Eines der wichtiges Konzepte in Zusammenhang mit Datenbankabfragen stellt die Session dar. Sie wurde in ihren Grundzügen bereits in der Einführung dieses Kapitels angesprochen. Eine Session bildet einen abgeschlossenen Kontext, eine "Unit of Work", in der mehrere kleine Arbeitsschritte zusammengefasst und schlussendlich bestätigt werden.

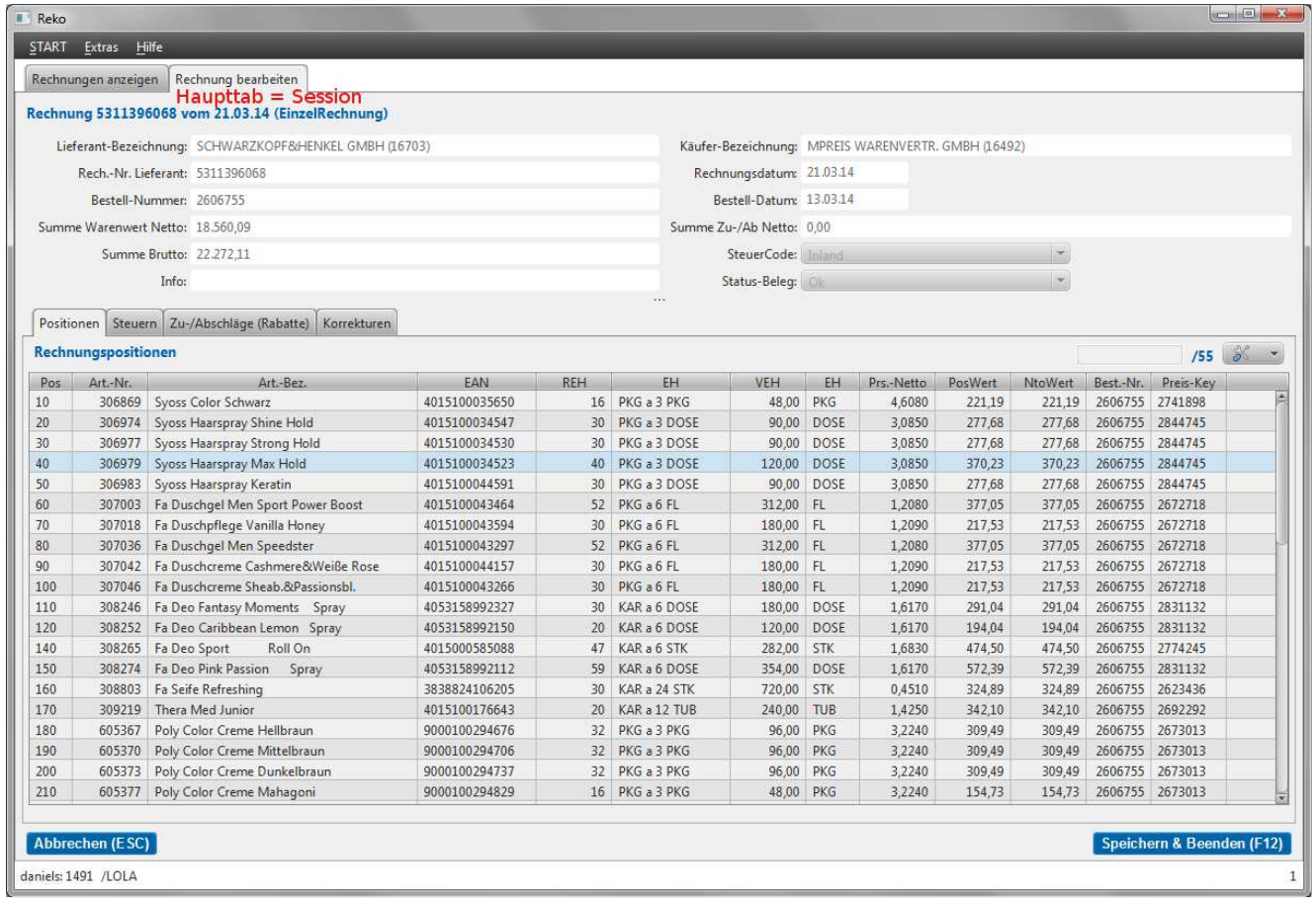


Abbildung 31: Session

Im Screenshot wird wieder auf das bewährte Beispiel der Rechnungskontrolle zurückgegriffen. Im aktiven Tab wird das Kommando "Rechnung bearbeiten" ausgeführt. Eine Rechnung mit diversen Positionen wurde geladen und dann der Anzeige übergeben. Der Anwender kann die Rechnung nun mit verschiedenen weiteren Kommandos verändern, bspw. Positionsdaten anpassen. All diese Änderungen werden aber nicht sofort auf die Datenbank zurückgeschrieben, sondern nur an der Rechnung im Speicher vollzogen. Erst nach Abschluss aller Arbeiten kann der Anwender das Kommando "Rechnung bearbeiten" beenden (mit der Schaltfläche "Speichern"), womit die Session beendet wird und die Rechnung mit allen Änderungen auf die Datenbank geschrieben wird (checkin). Wird das Kommando abgebrochen, so wird die Rechnung verworfen und

keine Änderungen in der Datenbank gespeichert. Das Bearbeiten einer Rechnung ist in diese Fall als "Unit of Work" zu sehen. Die Session, die im Hintergrund ausgeführt wird, entspricht gerade einem Tab im Hauptfenster. Grundsätzlich ist jedem Tab im Hauptfenster eine eigene Session zugeordnet.

Eine Session verfügt über zwei wesentliche Eigenschaften:

1. Sie stellt sicher, dass nur eine Instanz einer Entität mit gegebenem Schlüssel pro Session instanziert wird, d.h. eine Rechnung mit der id = 10 kann zwar öfter von der Datenbank abgefragt werden, als Ergebnis erhält der Entwickler allerdings immer dieselbe Rechnungsinstanz, nämlich die der Session. So ist es zwar möglich beliebig viele neue Rechnungsinstanzen in einer Session zu erzeugen, von der Datenbank wird zu einem gegebenen Schlüssel nur eine Instanz erzeugt.

   ***Die Session stellt sicher, dass nur eine Instanz eines Objektes in der Session verfügbar ist.*** Dazu überprüft sie folgendes:

   - Eine Entität, die ausgecheckt wurde, kann nicht nochmals ausgecheckt werden. Das selbe gilt natürlich auch für Listen.
   - Eine Entität, die ausgecheckt wurde, kann auch nicht mit find “readOnly” gesucht werden.
   - Eine Entität, die mit ReadOnly geladen wurde, kann in derselben Session nicht mehr “ausgecheckt” werden.
   - Eine Entität, die mit ReadOnly geladen wurde, kann nochmals in einer Session abgefragt werden - es wird allerdings die selbe Instanz zurückgegeben.

   Wird eine neue Entität gespeichert, wird sie in der Session als “checked out” (als read/write) hinzugefügt.

   Alle Ergebnisse aller Abfragen werden durch die Session zwischengespeichert. Wurde eine Entität bereits geladen und es erfolgt anschließend eine get() Abfrage, so wird nicht mehr mit der Datenbank interagiert. Über den eindeutigen Datenbankschlüssel kann die Entität identifiziert und unmittelbar von der Session selbst zurückgegeben werden. Wird mit Where() abgefragt, muss ein Ergebnis-Set auf der Datenbank berechnet werden. Es werden dann allerdings bereits geladene Entitäten aus dem Cache von der Session zurückgegeben. Entitäten in der Session werden während dem Lebenszyklus nicht verworfen, wodurch auch Speicher nicht freigegeben werden kann. Dazu muss die Session explizit angewiesen oder beendet werden.

2. Speicher- oder Löschvorgänge werden nicht unmittelbar ausgeführt, sondern können lediglich einer Session angehängt werden. Alle checkinXXX() bzw. deleteXXX() Methoden von ModelRepositories werden automatisch als Session-Operations der Session angehängt. Wird die Session abgebrochen, wird keine der Operationen ausgeführt. Beendet der Anwender die Session, werden alle gesammelten Session-Operations in einer gemeinsamen Datenbanktransaktion abgearbeitet. MoWare verwendet dieses Vorgehen, um “lange Transaktionen” auf der Datenbank zu verhindern. Zudem werden dadurch entweder alle Anwenderänderungen oder eben keine übertragen.

   Die Session wird grundsätzlich durch den Anwendungsentwickler nicht berücksichtigt, da MoWare diese einfach im Hintergrund mitführt und alle Interaktionen automatisch koordiniert (auch das Anhängen von Session-Operations). Dennoch kann mit dem Schlüsselwort session auf die aktuelle Session zugegriffen werden. Folgende Methoden sind verfügbar: Mit addOperation() können der aktuellen Session manuell beliebig viele Operationen angehängt werden. Jede Operation kann mit einer beschreibenden Zeichenkette versehen werden. Auf diese Weise kann sie der Entwickler später nochmals aufgreifen und gegebenenfalls entfernen (Methode removeOperation). Wichtig ist auch die Methode findDirtyEntities(), mit der alle in der Session als Dirty markierte Entitäten geprüft werden können. Nach einem einfachen checkout sollte sich keine der Entitäten im Zustand Dirty befinden (siehe dazu auch Property Option NOT_PERSIST_DIRTY_IRRELEVANT). Weitere Metoden

sind durch die Code-Completion ersichtlich.

Oft wird gefordert, dass beim Fehlschlagen einer Transaktion ein Fehler-Flag in einer (weiteren) Entität gesetzt wird. Mit der Methode setTransactionExceptionOperation() kann einer geplanten Transaktion eine Fehleroperation hinzugefügt werden. Mit Hilfe einer Transaktions-Savepoint-Logik wird diese Operation im Fehlerfall ausgeführt und auch gleich ein Commit gesendet. Die Session sollte anschließend geschlossen werden.

**Wichtig: Entitäten/ValueObjects oder ViewObjects dürfen keinesfalls zwischen Sessions ausgetauscht werden. Falls ein Austausch erfolgen muss, ist nur der Schlüssel (ID) zu übergeben. Die entsprechende Session muss das Objekt selbst erneut laden (durch checkout oder ReadOnly)!!!**

## *Die ReadOnly Session*

Mit Herbst 2013 wurden ReadOnly Sessions eingeführt. Ganze Sessions können intern ReadOnly gesetzt werden. Dadurch werden dann auch beim Checkout in einem ModelRepository Entitäten nur ReadOnly geladen. ReadOnly Sessions können vor allem dann verwendet werden, wenn bestimmte Benutzer keine Bearbeitungsrechte gewährt werden soll.

- session.isReadOnly() gibt bei einer Session an, ob sie gegenwärtig im ReadOnly Mode ausgeführt wird.
- In einem Formular (DelegateForm) werden automatisch alle Editoren mit setEnabled(false) in einen ReadOnly Mode gesetzt (Default-Wert). Kann mit #Meta-Information wieder überschrieben werden.
- Bei ReadOnly Sessions führt das Abarbeiten von Session-Operations (startTransactionAndFlush()) zu einer Exception. Das Abarbeiten der Operationen ist nicht möglich, sodass keinesfalls Änderungen in die Datenbank zurückgeschrieben werden.
- *In Bezug auf Kommandos: Veränderungen von Graphen etc. sind immer in der FINAL_OK_CONCLUSION von Kommandos durchzuführen. Buttons, die mit "done" das aktuelle Kommando abschließen, sind im ReadOnly Modus nicht verfügbar, d.h. Page Conclusions die "done" enthalten, können nur ausgeführt werden, wenn session.isReadOnly() false ist. Daher sollte alle Logik, die einen Graphen verändert, nach Möglichkeit in der FINAL_OK_CONCLUSION stehen. Damit ist dann sichergestellt, dass in ReadOnly Sessions keine Daten manipuliert werden können. Nur ein Kommando - Abbruch ist durch den Anwender möglich.*
- *In Bezug auf Prozesse: In der Prozess-Definition können Commands mit ReadOnly Sessions gestartet werden (ReadOnly Command im Inspektor-Fenster). Man unterscheidet nun zwischen, Enabled/Disabled und ReadOnly/ReadWrite im Inspektor.*
- *Zukünftiges Feature in MoWare: Pessimistic Locking könnte Commands mit ReadOnly Session starten, falls wesentliche Entitäten in zweitem Haupt-Tab oder von anderen Anwendern bereits ausgecheckt wurden.*

## *Aggregation vs. Komposition*

Aus UML stammen die beiden weit verbreiteten Begriffe Aggregation und Komposition, die an dieser Stelle genauer erläutert werden sollen. Sie bringen grundlegende Eigenschaften mit sich, dessen sich der Entwickler beim Arbeiten mit MoWare bewusst sein sollte.

Aggregation und Komposition sind zwei spezifische Formen von Assoziationen. Assoziationen wurden bisher als Referenzen bezeichnet. Sie verbinden Datenobjekte (Enitäten/ViewObjects) untereinander. Assoziationen können vier verschiedene Formen annehmen:

1. Unidirektional: Ein Objekt A referenziert ein weiteres Objekt B, aber B referenziert nicht A. Damit kann nur vom Objekt A auf das Objekt B navigiert werden, aber nicht umgekehrt. Diese Situation treffen wir an, wenn Positionen Artikel referenzieren.

2. Bidirektional: Zwei Objekte referenzieren sich gegenseitig. Diese Situation treffen wir an, wenn eine Belegposition den Belegkopf referenziert und der Belegkopf über eine Liste Belegpositionen enthält. In MoWare muss der Entwickler dafür Sorge tragen, dass die Referenzen in beiden Objekten gesetzt werden, d.h. eine Position in einer Liste einfügen und in der Position die Rückwärtsreferenz zum Kopf setzen. Das Management dieser Assoziation ist daher etwas aufwändiger und verkompliziert den Code. Daher wird gegenwärtig empfohlen, keine Rückwärtsreferenz in Positionen zu modellieren (bzw. nur als Integer Property). Sie muss im Quellcode dann auch nicht gepflegt werden.

   *Ein Vorgriff auf Kommandos: Sollte von der Position aus dennoch zum Kopf navigiert werden müssen, so kann man das Kopf-Objekt als Parameter dem Kommando zur Verfügung stellen (der* Selektionskontroller *in Oberflächen unterstützt diese Vorgehensweise). Wird dieses Pattern angewendet, muss nur die Liste im Kopf verwaltet werden, was wiederum die MPS Collection Sprache optimal unterstützt. (Alternativ ist es in Zukunft vorstellbar, bei Containment eine Operation .parent in MoWare zu unterstützen).*

3. Aggregation: Eine Aggregation entspricht einer "hat eine" Beziehung und ist damit spezifischer als eine Assoziation. Es handelt sich um eine Assoziation die ein Teil-Ganzes Beziehung repräsentiert, insbesondere bei Sammlungen (z.B. Listen). So ist bspw. die Beziehung zwischen einem Institut in einer Universität und deren Professoren eine Aggregation. Ein Institut hat mehrere Professoren. Allerdings - und das ist besonders wichtig - sind die Lebenzyklen der Objekte voneinander unabhängig. D.h. wird das Institut aufgelöst, dann gibt es die Professoren weiterhin. Wird das Aggregat zerstört, bleiben die Elemente erhalten.

4. Reflexiv: Von Reflexiven Assoziationen wird gesprochen, wenn sich ein Datenobjekt selbst referenziert, d.h. eine Instanz eines Objekte referenziert eine Instanz vom selben Objekttyp.

Kompositionen sind nun eine besondere Variante von Assoziationen, die noch spezifischer sind als die Aggregationen. Der Lebenszyklus zwischen Elementen und der Komposition sind voneinander abhängig. Wird die Komposition zerstört, so werden auch dessen Elemente zerstört. In MoWare wird diese Beziehung mit der Option CONTAINMENT ausgezeichnet.

Kompositionen sind mit MoWare sehr einfach handhabbar. Der einfachste Fall ist wohl das Verhältnis zwischen einem BelegKopf und dessen Positionen. Wird der BelegKopf gelöscht, sind auch die Positionen zu löschen. Wird der BelegKopf ausgecheckt, werden auch dessen Positionen ausgecheckt. Bearbeitet ein Anwender den Kopf, so können auch die Positionen für die Bearbeitung durch andere gesperrt werden. Bei einem checkin werden erst die Positionen in der Liste gespeichert, dann der Kopf. Wird eine Position aus der Liste entfernt, so wird der Löschvorgang gleichzeitig als Session-Operation zusätzlich registriert. Die selbe Position kann nicht mehr der Liste angehängt werden. Der Benutzer kann bei Bedarf lediglich eine neue Position mit selbem Inhalt erzeugen. Diese wird dann in die Liste aufgenommen und beim checkin ebenfalls gespeichert - mit neuem Schlüssel.

Aggregationen werfen hingegen verschiedenste Probleme auf. Nehmen wir als Beispiel die Verbindung zwischen Rechnungskontrollakt und Rechnungen. Ein Akt kann mehrere Rechnungen enthalten. Wird der Akt zerstört, so dürfen die Rechnungen allerdings nicht gelöscht werden. Anwender müssten ansonsten bereits erfasste Rechnungen nochmals erfassen. Wird der Akt zerstört, so sollten vielmehr die enthaltenen Rechnungen für andere Akte zur Verfügung stehen. Modelliert wird eine derartige Verbindung bisher immer analog dem Containment Fall. Der RekoAkt führt eine Liste von Rechnungen, die Rechnung weist eine Back-Referenz zum RekoAkt auf.

Verschiedene Anwendungsfälle sind unter anderem:

- Rechnung wird aus RekoAkt entfernt, dann wieder hinzugefügt: Angenommen die Rechnungen werden beim checkout des Aktes geladen, dann wird eine Rechnung aus dem Akt entfernt (aus der Liste). Die Back-Referenz in der Rechnung muss dann auf 0 gesetzt und gespeichert werden, was durch Anhängen einer Session-Operation geschehen könnte. Die Rechnung wird also nicht gelöscht, sondern nur die Referenz auf den Akt 0 gesetzt und anschließend auf der Datenbank gespeichert.

  Möchte der Anwender diese Rechnung wieder anhängen, so liefert die Datebankabfrage diese Rechnung nicht als "freie" Rechnung, da sie noch nicht auf der Datenbank gespeichert, sondern eben nur zum Speichern markiert wurde.

  Könnte der Anwender die Rechnung der Liste wieder hinzufügen, so dürfte die Session-Operation zum Speichern an der Session bleiben, allerdings darf nicht die Rekoakt Referenz in der Operation selbst 0 gesetzt werden. Ansonsten könnte nach Speichern des Aktes die Rechnung durch die Operation erst wieder aus dem Akt entfernt werden (zumindest auf der Datenbank).

- Rechnung wird hinzugefügt, dann verändert und dann wieder entfernt: Wird nicht nur die Back-Referenz auf den RekoAkt in der Rechnung verändert, sondern auch Werte/Positionen in der Rechnung, so könnte das zu intransparentem Verhalten führen. Angenommen der Anwender hängt eine Rechnung dem Akt an und kann auch in der selben Session angehängte Rechnungen anpassen. Er kann dann die Rechnung verändern und wieder aus dem Akt entfernen, womit sie als Session-Operation angehängt wird. Beim Beenden der Session, wird dann über die Session-Operations die Rechnungsentität gespeichert - offensichtlich mit den Änderungen daran. Es ist unklar, ob der Anwender die Änderungen an der Rechnung, die er wieder entfernt hat, gespeichert haben möchte.

- Wird die Liste von Rechnungen, die potentiell an einen RekoAkt angehängt werden kann, ausgecheckt oder ReadOnly geladen? Damit verbunden ist die Frage, ob die Rechnungen zur Bearbeitung für andere gesperrt oder freigegeben ist.

- Sollte evtl. ein eigenes Verbindungsobjekt RekoAkt / Rechnung herangezogen werden? Es könnten dann nur Speichervorgängen mit dem Verbindungsobjekt durchgeführt werden ???

Die Diskussion Aggregation vs. Composition soll auf die damit verbundenen Probleme im Zusammenhang mit der aktuellen MoWare Session hinweisen. Containment Assoziationen werden mit den aktuellen Sessions passend unterstützt. Verbesserungspotentiale bergen vor allem Aggregationen, die im Moment mit einem ReferenceSupport unterstützt werden. ReferenceSupport ist eine Java Utility-Klasse, die im Rahmen dieses Manuals nicht erläutert wird. Der ReferenceSupport ist etwas Komplex in der Handhabung. In Zukunft soll er durch ein einfacheres Konzept ersetzt werden.

## *Manuelle SQL  Abfragen*

Mit der Version Sommer 2014 verfügt Manmap auch über Möglichkeiten zum direkten Arbeiten mit SQL. Grundsätzlich sollten direkte Abfragen allerdings nicht verwendet werden. Zum einen sind sie nicht in das Session-Konzept integriert. Anwendungsentwickler müssen selbständig die Session manuell nachführen. Schnell könnten Entitäten mit gleichem Schlüssel in mehreren Instanzen vorliegen. Für Dritte aber auch bei langfristiger Wartung sind dann Abläufe schwer nachzuvollziehen. Zum anderen könnten sich aber Mapping-Vorschriften mit Tabellenbezeichnungen ändern. Direktes SQL wird dann natürlich nicht automatisch angepasst, generiertes SQL durch Mapping Abfragen schon.

```
//
READONLY IntKeyObject findIntKeyObjectById(int id) {
  // Check if num of params fit string count '?'

  list<IntKeyObject> objects2 = MapSELECT "SELECT * FROM TESTOBJECT_TABLE WHERE KEY=?" (id) {~row =>
    IntKeyObject i = new IntKeyObject();
    // first col starts with 0
    i.id = row.getAsInteger(0);
    // second one is 1
    i.text = row.getAsString(1);
    return i;
  } // check with session ;

  // this object is not mangaed by a session
  // what is particualarly dangerous
  return objects2.first;
}


//
CHECKIN Integer checkinPlainSQLObj(IntKeyObject edited) {
  // check for dirty yourself ...
  MapUPDATE "UPDATE TESTOBJECT_TABLE SET TXT_BEMERKUNG=?, KEY=? WHERE KEY=?" (edited.text, edited.id, edited.id)
      // check with session ;

}
```

Abbildung 32: SQL direkt

In der Abbildung sind zwei direkte SQL Anweisungen dargestellt. Unterschieden werden muss zwischen SELECT und UPDATE Anweisungen. Die MapUPDATE Anweisung erlaubt die Übergabe einer SQL Zeichenkette und der notwendigen Parameter. Der Entwickler kann aber auch eine Variable vom Typ string referenzieren, sollte das SQL Statement selbst dynamisch generiert werden. Vor der Ausführung wandelt MapUPDATE die angegebenen Parameter mit ManMap Typehandler um (siehe Tabelle Typehandler im vorigen Abschnitt). Die Parameteranzahl muss der Zahl der ?-Zeichen in der SQL Zeichenkette entsprechen. Nacheinander, in der angegebenen Reihenfolge, werden die Parameter eingefügt. Bei dynamischem SQL mit unterschiedlicher Zahl von Parametern müsste der Entwickler mehrere MapUPDATE verwenden. Der Rückgabewert ist immer integer - das ist die Anzahl der veränderten Datensätze. Damit ist MapUPDATE auf für SQL insert und delete Anweisungen die richtige Wahl.

Die MapSELECT Anweisung ist unwesentlich komplizierter. Übergeben wird die SQL Zeichenkette (oder eine string Variable) und die Parameter. Analog dem MapUPDATE werden die Parameter konvertiert. Zusätzlich ist ein Closure anzugeben, das eine Zeile der Datenbank in ein beliebiges Objekt konvertiert. Dem Closure wird der Parameter "row" übergeben. Er verfügt über verschiedene Methoden, mit denen Daten aus der aktuellen Zeile direkt in einen Typ konvertiert werden können - wie bei den Parametern mittels Typehandler. Das erzeugte Objekt ist mit der return Anweisung im Closure zurückzugeben. Das Closure wird für jede Zeile einmal aufgerufen. MapSELECT liefert dann eine list der abgepackten Objekte zurück. Es müssen bei einem MapSELECT immer Objekte vom selben Typ gebildet werden. Innerhalb eines MapSELECT - genauer im Closure - kann man keine weiteren Abfragen an die Datenbank stellen.

Es soll an dieser Stelle auch nochmals darauf hingewiesen werden, dass der Entwickler die Session bei direkten SQL Abfragen selbst verwalten muss. So kann er z.B. vor einem Update prüfen, ob ein Objekt Dirty markiert wurde. Schwieriger gestaltet sich bei einem MapSELECT die Integration von Entitäten in die Session. Das Vermeiden doppelter Instanzen oder ReadOnly und Checkout Unterscheidung obliegen dem Anwendungsentwickler.

## 5. Kommando und Prozesse

Die Logik in Applikationen setzt sich aus Prozessen und Kommandos zusammen. Während Prozesse Datenstrukturen prüfen und Kommandos freigeben oder sperren, verändern die Kommando auch Entäten, ValueObjects und ViewObjects. Gerade die Verkettung mehrerer Kommandos und die Handhabung der Session im Hintergrund verlangt Detailwissen.

Kommandos bilden abgeschlossene Benutzerinteraktionen ab. Ein Kommando wird entweder erfolgreich ausgeführt oder vollständig abgebrochen. Kommandos folgen einer Undo/Redo Logik. Verändert ein Kommando eine Datenstruktur (den Objektgraphen), so kann die Veränderung rückgängig gemacht werden. Der Anwendungsentwickler muss also jede Veränderung der Datenstruktur zwingend in einem Kommando abbilden. Das Kommando selbst kann aus der Oberfläche gestartet werden, bspw. über einen Menüeintrag, einen Button oder auch über die “Enter-Taste” (entspricht auch einem Doppelklick) auf einer Tabellenzeile. Auch aus dem Programmcode lassen sich Kommandos direkt ohne Oberfläche ausführen, vor allem zu Testzwecken.

Im Gegensatz zu einfachen Methoden umfassen Kommandos neben Logik auch Interaktionen mit der Benutzeroberfläche. D.h. Kommandos können Pages/Formulare in der Oberfläche anzeigen, die mit Schaltflächen (sogenannten Conclusions) versehen sind. Üblich sind bspw. “Ok” oder auch “Schließen”. Das Kommando verwaltet die Seite und kann die Benutzerinteraktion auch auswerten. Wie bereits bei der Einführung angesprochen, kann ein Kommando keine, eine oder mehrere Seiten umfassen. Ein Kommando mit keiner Seite benötigt offensichtlich keine Benutzerinteraktion (z.B. Rechnung auf freigegeben setzen). Bei einem Kommando mit mehreren Seiten spricht man von einem Wizzard. Anwender können mit einem Wizzard durch mehrere Interaktionsaufgaben geführt werden.

## *Grundsätzlicher Kommandoablauf*

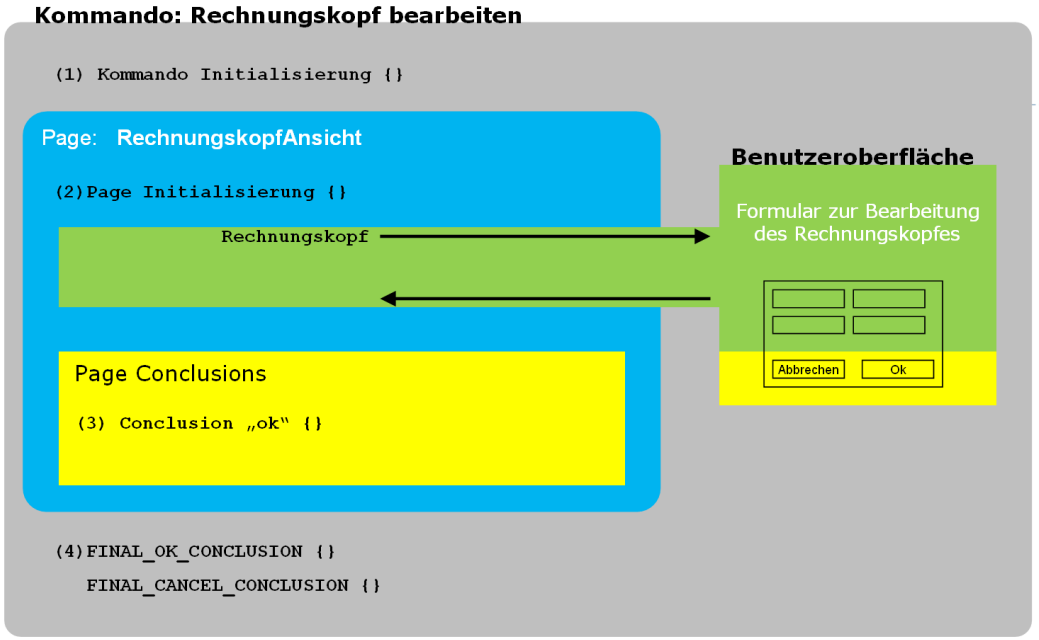



Abbildung 33: Kommando konzeptionell

In der obigen Abbildung ist prototypisch eine Kommando mit einer Seite dargestellt. Das Kommando "Rechnungskopf bearbeiten" soll bspw. dem Endanwender eine Bearbeitung der Entität Rechnung erlauben. Dazu wird eine aktuelle Instanz der Rechnungsentität dem Kommando als Parameter übergeben. In 4 verschiedenen Abschnitten kann das Kommando Logik implementieren, d.h. an 4 Stellen lässt sich Java-Code hinterlegen.

1. Kommando Initialisierung: Dieser Abschnitt wird als erstes ausgeführt, nachdem das Kommando gestartet wurde. In einem Kommando deklarierte lokale Variablen jeglichen Typs lassen sich an dieser Stelle initialisieren. Vorbereitungen und Überprüfungen lassen sich durchführen. Der Abschnitt wird jedenfalls nur einmal, nämlich beim Start des Kommandos, ausgeführt. Der Entwickler kann an dieser Stelle das Kommando auch mit dem sogenannten cancel Statement und einer Meldung abbrechen.

2. Page Initialisierung: Nach der Kommando Initialisierung wird automatisch die erste Seite geladen. Dabei wir als erstes die Page Initialisierung durchgeführt. Auch in diesem Abschnitt kann auf lokale Variablen des Kommandos zugegriffen werden. Die Page Initialisierung bietet sich vor allem auch zum Laden von Daten aus der Datenbank an, da während dieses Abschnitts ein Busy-Wheel, eine Sanduhr, auf der Oberfläche angezeigt wird. Jedenfalls muss die Page Initialisierung ein

Datenobjekt zurückgeben, das dann in der Benutzeroberfläche angezeigt werden soll. Auch eine Listen von Objekten kann zurückgegeben werden.

Zu berücksichtigen ist, dass dieser Abschnitt mehrmals ausgeführt werden könnte, bpsw. bei einem "Reload" der Seite oder auch bei Rückkehr zu dieser Seite in einem Wizzard (Kommando mit mehreren Seiten). Auch an dieser Stelle kann das Kommando mit cancel abgebrochen werden. Es lassen sich auch mit dem Statement flag Meldungen an die Benutzeroberfläche weitergeben.

Die Definition der Benutzeroberfläche selbst ist nicht Gegenstand des Kommandos. Formulare für Desktop-Anwendungen werden mit Konzepten der Sprache Forms3 beschrieben. In Zukunft könnten Formulare mit anderen Anordnungen für Mobiles/Tablets beschrieben werden. Das Kommando kann ohne Anpassungen übernommen werden. Es werden lediglich die Formulare getauscht.

3. Eine Seite wird mit sogenannten Conclusions beendet. Conclusions repräsentieren in der Oberfläche Schaltflächen am unteren Ende der Formulare. Eine Cancel-Conclusion, die zum Abbruch des gesamten Kommandos führt, steht immer automatisch zur Verfügung und muss vom Entwickler nicht modelliert werden. Der Name jeder weiteren Conclusion entspricht zugleich dem Schaltflächentext. Ferner kann zu jeder Conclusion auch wieder Logik in Java formuliert werden.

   Auch in Conclusions stehen cancel und flag als Statement zur Verfügung. Mit cancel lässt sich das gesamte Kommando abbrechen, mit flag kann eine Meldung im aktuellen Formular auf der Oberfläche ausgegeben werden. Auf diese Weise lassen sich diverse Probleme bei Überprüfungen an den Anwender melden. Schließlich ist ein done Statement vorhanden, um eine beabsichtigte, erfolgreiche Beendigung des Kommandos anzuzeigen. Das Statement page XXX ist vorgesehen, um zwischen verschiedenen Seiten bei Wizzards zu wechseln.

4. Der letzte Abschnitt bilden die Command-Conclusions: Im Gegensatz zu den Conclusions bei Seiten gibt es auf der Ebene des Kommandos nur zwei Conclusions, nämlich FINAL_OK und FINAL_CANCEL. Erstere wird ausgeführt, wenn das Kommando erfolgreich abgeschlossen werden kann, letztere dann, wenn das Kommando durch die Geschäftslogik - sprich ein cancel Statement im Code - oder durch den Benutzer (Cancel Schaltfläche) beendet wird. In den FINAL Conclusions selbst stehen flag, cancel oder done nicht mehr zur Verfügung.

   Gegenwärtig sollen Veränderungen an den Datenstrukturen ausschließlich in der FINAL_OK / FINAL_CANCEL durchgeführt werden. In Zukunft ist als wesentliche Ergänzung von MoWare ein automatisches Undo auf Ebene der Datenstrukturen vorgesehen. Wird ein Kommando abgebrochen, so werden Parameter des Kommandos automatisch wieder in den Zustand vor Aufruf des Kommandos zurückversetzt.

   Als dritte Conclusion, die in der Abbildung allerdings nicht dargestellt wurde, ist auch noch die FINAL_EXCEPTION_CONCLUSION zu nennen. Diese Conclusion wird ausgeführt, wenn ein technischer Fehler - eine Java Exception - auftritt, während das Kommando ausgeführt wird. Auch wenn in der FINAL_OK oder FINAL_CANCEL Exceptions auftreten, wird diese Conclusion ausgeführt. Üblicherweise muss in der FINAL_EXCEPTION_CONCLUSION nicht reagiert werden, da das voreingestellte Verhalten in den meisten Fällen angepasst ist.

Kommandos weisen Parameter auf. Die Datenstruktur(en), die verändert werden sollen, werden dem Kommando von außen (meist über die Benutzeroberfläche) übergeben. So wird beispielsweise ein Kommando aus der Oberfläche gestartet, dem dann die aktuell selektierte Rechnung (mit dessen Positionen) übergeben wird. Kommandos haben

keinen Rückgabewert. An einem Bespiel lässt sich die Funktionsweise von Kommandos nochmals plastischer verdeutlichen. Im Folgenden wird das Kommando Rechnungskopf bearbeiten aus der Rechnungskontroll-Applikation herangezogen.

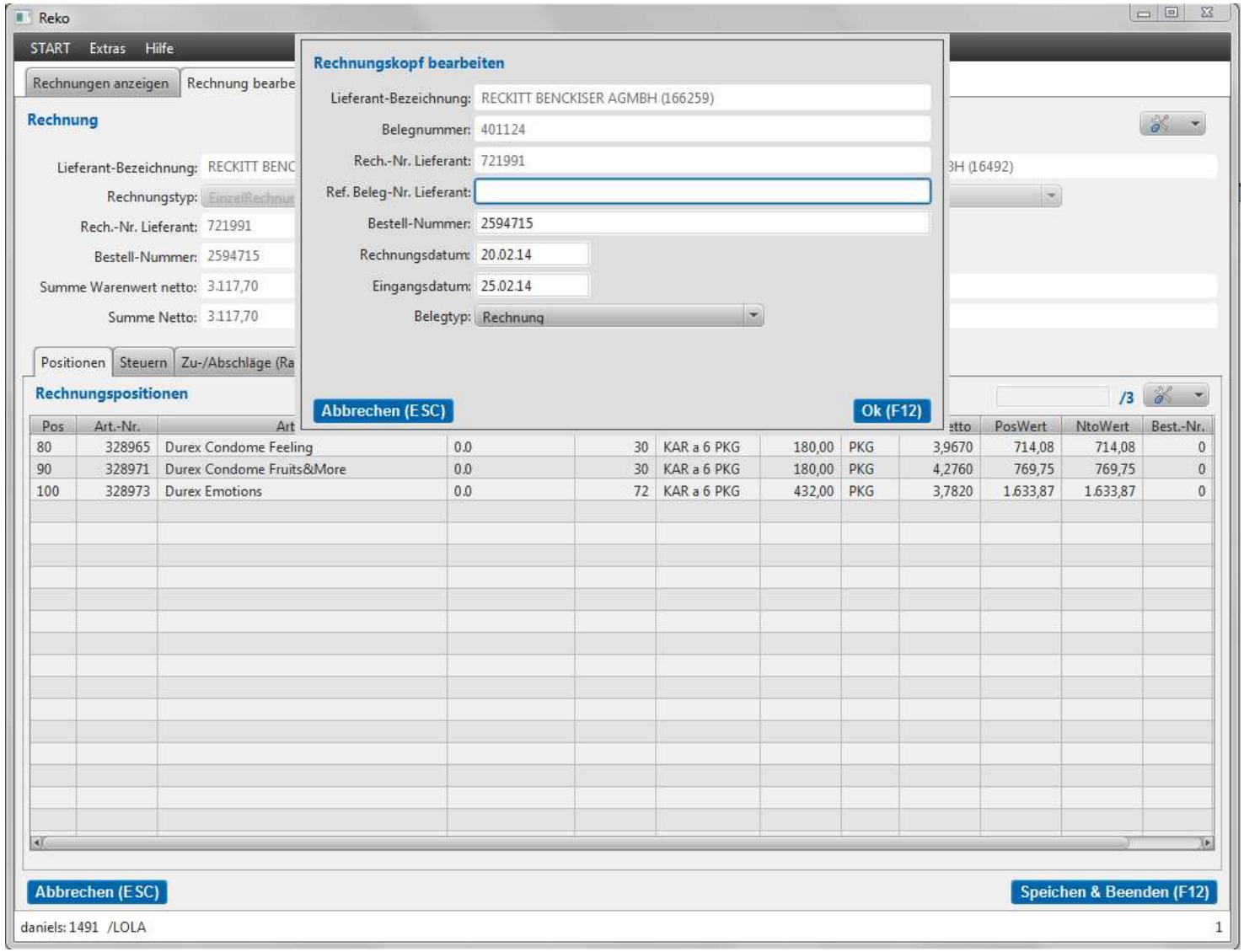



Abbildung 34: Oberfläche Rechnungskopf bearbeiten

```
command 'Rechnungskopf bearbeiten' in process RechnungsProcess with Rechnung rechnung

command parameter:
<< ... >>

local variables:
<< ... >>


<docu>


command settings:
command type: GRAPH_EDIT
enabled if: <cond>
question on external abort: <msg>
title addon: <msg>
command icon: HafinaDefaults.ICON_EDIT

command init:
<func>


command pages:
 page 'Standard'
  form bound to list< Rechnung > as  form

  page init:
    pageLoadFunc()->Object {
      rechnung;
    }

  then, calc page title: <title>

  set scopes for page:
    <no scopeConceptFunc>

  page conclusions:
    conclusion 'Ok'  (enabled if: <cond>)
      request 'save data' from page form: save hotkey: SAVE
      func()->void {
        rechnung.updateSumme();
        done (run FINAL_OK_CONCLUSION)
      }


FINAL_OK_CONCLUSION:
  <func>

  check process, then: DO_NOT_COMMIT_SESSION
  notification: <msg>
  selection(s)/update(s) on parent: <no finalSelection>


FINAL_CANCEL_CONCLUSION: // do revert first
  <func>


EXCEPTION_CONCLUSION: exception // do revert first
  <func>
```

Abbildung 35: Kommando

Das Kommando "Rechnungskopf bearbeiten" ist denkbar einfach gehalten:

- Das Kommando wurde dem Prozess "RechnungsProcess" zugeordnet. Damit steht als Variable "rechnung" im Kommando automatisch die Rechnung, das Prozessdokument, zur Verfügung. Zusätzliche Kommando Parameter wurden in diesem Beispiel nicht deklariert, eben sowenig, wie zusätzliche lokale Variablen innerhalb des Kommandos.

- Bei den Einstellungen zum Kommando wurde GRAPH_EDIT als command type gewählt. D.h. das Kommando übernimmt die Session des Aufrufers, es wird kein checkout und keine checkin Operation in diesem Kommando durchgeführt (dazu in späteren Abschnitten mehr). Ansonsten wurde nur ein Icon zu dem Kommando spezifiziert.

- Das Kommando besteht aus einer Seite "Standard". Diese Seite muss dann in der Applikation mit einem Formular verbunden werden, das eine Liste von Rechnungen (oder eine einzelne Rechnung) anzeigen kann. In der Page Initialisierung wird die Rechnung (Variable rechnung) an die Oberfläche übergeben.

- Neben der Abbrechen Conclusion ist eine Ok Conclusion verfügbar. Wird diese Schaltfläche vom Anwender betätigt, so muss erst das Formular angewiesen werden, alle Änderungen in der Oberfläche auf das Objekt Rechnung (Variable rechnung) zurückzuspeichern. Dann werden durch die Methode updateSumme() diverse Summen der Rechnung neu berechnet. Mit dem done Statement wird das Kommando erfolgreich beendet.

- Keine FINAL Conclusion wurde implementiert. Es könnten aber jeweils Funktionen angegeben werden.

## *Page und Kommando Conclusions*

**ad Page Conclusions:**

Bei diesem Beispiel wurde erstmals bei einer Page Conclusion das Formular der Seite gesteuert. Es stehen zwei Anweisungen zur Verfügung:

| **Conclusion Typ** | **Beschreibung** |
|---|---|
| NO_SAVE | Eine Conclusion von diesem Typ fordert kein “Save” beim Formular an. Die Änderungen an der Oberfläche werden nicht auf die Datenobjekte zurückgespeichert. Dadurch wird auch die Validierung nicht ausgelöst. Üblicherweise bei Schaltflächen eingesetzt, die ähnlich der "Abbrechen" Funktionalität den Inhalt des Formulars verwerfen. |
| SAVE | Eine Conclusion von diesem Typ fordert ein “Save” beim Formular an und löst damit auch die Validierung aus (muss bspw. bei einem Save-Button verwendet werden, sodass der Inhalt des Formulars auf Datenobjekt übertragen wird). |

Häufig wird in der Page Conclusions die flag Anweisung verwendet. Eine flag Anweisung bricht das aufrufende Kommando nicht ab, sondern zeigt nur die angegebene Zeichenkette im aktuellen Formular an. Dabei wird nicht nochmals die pageInit() abgearbeitet, sondern es wird auf erneute Eingaben des Benutzers gewartet. Die Anweisung kann aber auch in Page Initialisierung verwendet werden. Allerdings wird die Initialisierung an dieser Stelle dann abgebrochen und die Meldung im Formular angezeigt. Dieses Verhalten ist nicht immer gewünscht.

**ad Command Conclusions:**

| | |
|---|---|
| FINAL_OK_CONCLUSION | Wird ein Kommando erfolgreich beendet (mit einem done Statement), so wird die FINAL_OK_CONCLUSION ausgeführt. Bei der FINAL_OK_CONCLUSION lässt sich Geschäftslogik in Form von Java Statements codieren. Bei erfolgreichem Beenden werden diese Statements abgearbeitet, dann wird der Prozess geprüft. Eventuell sind Zustandsänderungen im Prozessdokument durchzuführen. Anschließend könnte die Session "commited" werden. D.h. alle anstehenden Session Operations sollen dann in einer Datenbanktransaktion abgearbeitet werden. Gerade dieser Vorgang hängt aber wesentlich mit dem Kommando-Typ zusammen (siehe nächster Abschnitt). Als letzter Schritt können noch Objekte mit dem Vater-Kommando (häufig einem SEARCH_VIEW) ausgetauscht werden<br><br>Ruft der Entwickler Repo-Methoden von der FINAL_OK_CONCLUSION aus auf, so werden diese nicht sofort abgearbeitet, sondern als Session-Operation der Session angehängt. Bei der call Anweisung wird das durch "add to session operation" angedeutet.<br><br>`FINAL_OK_CONCLUSION:`<br>`  func()->void {`<br>`    call(add to session operation)ParteienRepo.checkinVorgangsortRechnung(defaults, idDefaults == 0) ;`<br>`  }`<br><br>`  check process, then: COMMIT_SESSION`<br>`  notification: <msg>`<br>`  selection(s)/update(s) on parent: defaults`<br><br>Abbildung 36: checkin |
| FINAL_CANCEL_CONCLUSION | Wird ein Command mit cancel abgebrochen, so wird die FINAL CANCEL_CONCLUSION ausgeführt. Der Entwickler kann bei der cancel Anweisung zusätzlich eine Zeichenkette mit dem Grund des Abbruchs angeben. Ist die Länge der Zeichenkette >0, so wird der Grund in einem Meldungsfenster angezeigt. |
| EXCEPTION_CONCLUSION | Tritt eine Ausnahme auf, wird immer die EXCEPTION_CONCLUSION ausgeführt. Dort kann der Typ der Ausnahme ermittelt und entsprechend reagiert werden. Die Ausnahme wird dann “nach Außen” zum Aufrufer weitergegeben (User Interface). Dort wird die Meldung als “Technischer Fehler: ” + Exception-Text |

| | |
|---|---|
| | angezeigt und ein Stacktrace auf der Konsole ausgegeben. |

**ad Aufruf von Logik in Services:**

Grundsätzlich sollte notwendiger Applikationscode direkt in Kommandos codiert werden. Command-Init, Page-Init, Page-Conclusion und Command-Conclusion bieten dafür ausreichend Möglichkeiten. Ziel ist es, Ordnung und Überblick im Code zu schaffen. Code, der thematisch mit einem Kommando verbunden ist, findet sich dann ausschließlich im Kommando selbst wieder.

Falls notwendig kann der Entwickler Service Methoden aus Kommandos mit der call Anweisung aufrufen. Dem Service wird dabei automatisch die aktuelle Session des aufrufenden Kommandos übergeben. Werden aus diesem Service wiederum Methoden eines ModelRepositories (oder Service) aufgerufen, erhält auch dieses automatisch die aktuelle Session des Aufrufers. Sowohl ModelRepositories als auch Services können mit dem aufrufenden Kommando interagieren.

Das Kommando kann auch von Services temporär unter- oder vollständig abgebrochen werden. Um eine Meldung anzuzeigen ist das flag Statement in Services zu verwenden (flag “Zeichenkette”), für einen vollständigen Abbruch des Kommandos aus einem Service ist das cancel Statement (cancel ”Zeichenkette”) zu verwenden.

*Kommandos können bis dato keine weiteren Kommandos starten. Falls gefordert, wären diese jedenfalls asynchron zu starten, d.h. in einem weiteren Fenster. Es ist unklar, ob Kommandos weitere Kommandos starten können - falls dies nur über Fake-Pages erfolgt. Was passiert dann bei Abbruch? Es müsste dann die komplette Kommando-Hierarchie abgebrochen werden.)*

**ad Exception-Handling in Services:**

Bei Services oder Kommandos können verschiedenste Exceptions auftreten. Wird bspw. in einem Service ein File geöffnet, sind IOExceptions zu erwarten. Die Service-Methode (Methode A) muss daher mit throws IOException in der Methoden-Deklaration angeben, dass diese auftreten kann. Service Methoden (Methode B), die wiederum Methode A aufruft, muss ebenfalls diese Exception ankündigen (throws IO Exception). Das aufrufende Kommando fängt grundsätzlich alle Exceptions ab, womit auch das Kommando selbst mit einem technischen Fehler abgebrochen wird.

Ein gebräuchliches Muster ist das Umwandeln von Exceptions zu einem cancel Statement. Angenommen in einem Service tritt eine NumberFormatException auf, so kann diese direkt lokal mit einem try/catch abgefangen werden. Im catch wird dann das aktuelle Command mit dem cancel abgebrochen. Dem Anwender kann eine programmspezifische Nachricht in der Form “Beim Konvertieren ihrer Rechnung …. “ angezeigt werden.

Gerade bei IOException (File nicht gefunden) muss geprüft werden, welches Verhalten gewünscht wird. Meist macht es Sinn, eine IOException (als technischer Fehler) in eine cancel Anweisung mit passenderem Fehlertext zu übersetzen.

## *Unterschiedlich Kommando Typen*

Ein wesentliches Kriterium stellt bei Kommandos der Kommandotyp dar, da er dessen grundsätzliche Eigenschaften bestimmt. Zwei Typen wurden bereits verwendet: Im Einführungsbeispiel ein SEARCH_VIEW Kommando und in diesem Kapitel ein GRAPH_EDIT. Gerade diese beiden Kommandos sind typische Vertreter von Kommandos, die man als Long-Running (SEARCH_VIEW) oder Short-Running (GRAPH_EDIT) charakterisieren könnte.

Ein Search-View ist in dem Sinne Long-Running, als dass es nicht erst beendet werden muss, um weitere Kommandos zu starten. Er wird in einem Haupt-Tab ausgeführt. Weitere Kommandos lassen sich parallel in weiteren Haupt-Tabs starten. Anwender können zwischen den Kommandos in Haupt-Tabs wechseln. Üblicherweise wird ein Search View auch nicht sofort vom Anwender wieder beendet. Vielmehr wird von einem Objekt in der Ergebnisliste in ein weiteres Kommando verzweigt, um dieses Objekt evtl. im Detail zu laden und zu editieren. Dieses häufig anzutreffende Pattern wird in einem späteren Abschnitt noch im Detail gezeigt. Wie in früheren Abschnitten festgehalten, verfügen Kommandos in Haupt-Tabs über eigene, unabhängige Sessions.

Ein Graph-Edit ist ein typischer Vertreter für ein Short-Running Kommand. Der Entwickler kann in einem Graph-Edit keinen vollständigen checkout/edit/checkin Zyklus abbilden. Ein Graph-Edit übernimmt die Session des Kommandos, von dem aus er gestartet wurde. Es besteht dadurch eine Abhängigkeit zu einem Vater-Kommando. Außerdem werden Formulare eines Graph-Edit immer in einem modalen Fenster angezeigt. Anwender müssen das Fenster, genauer gesagt das Kommando, mit einer Conclusion erst schließen, bevor sie weiter mit der Applikation interagieren können. Graph-Edit Kommandos weisen dadurch einen kurzen Lebenszyklus auf.

Unterschieden werden kann bei den Typen grundsätzlich ob

(a) eine neue Session gestartet wird oder nicht, und
(b) ob das Kommando in einem Haupt-Tab (Long-Running) ausgeführt wird, oder in einem modalen Subfenster (Short-Running).

Die Typen im einzelnen:

| **command type** | **Beschreibung** |
|---|---|
| SEARCH_VIEW<br>(new session) | Startet neue Session, wird in einem Haupt-Tab ausgeführt.<br><br>Kommando, um nach Entitäten auf der Datenbank zu suchen, z.B. Rechnungen; Wird immer mit einem Einschränkungsformular (z.B. Datum, Lieferant etc.) als Seite 1 verwendet, auf Seite 2 wird dann das Ergebnis der Suche angezeigt. Entitäten im SEARCH_VIEW werden Read-Only angezeigt, d.h. die Session darf in der FINAL_OK_CONCLUSION nicht "commited" werden.<br><br>Wird auf der Page 2 eine Ergebnisliste angezeigt, so sollte man die Liste über Repositories im Page-Init der Page 2 laden. Beim Wechseln von Seiten wird automatisch die Session gelöscht - um genau zu sein - alle KeyStores (Cache) der Session. Damit wird erzwungen, dass bei einem Seitenwechsel alle Entitäten tatsächlich von der Datenbank erneut geladen werden. |

| | |
|---|---|
| | Da bis dato kein automatisches "Reload" von Listen unterstützt wird, muss in der Seite 2 eine Page Conclusion “Aktualisieren” vorgesehen werden, die die Seite 2 nochmals lädt. Es wird immer der aktuelle Zustand von der Datenbank geladen.<br><br>Aus Ergebnislisten können keine Objekte entfernt werden, zumindest nicht, ohne dass der Endanwender ein "Reload" anweist. Der Anwendungsentwickler kann allerdings neue Objekte der Ergebnisliste anhängen - ohne "Reload". Angehängte Objekte entsprechen dann mitunter nicht dem Suchkriterium. |
| GRAPH_OWNER<br>(new session) | *Startet neue Session, wird in einem Haupt-Tab ausgeführt.*<br><br>Ein GRAPH-OWNER ist das richtige Kommando, um einen Objektgraph im Detail anzuzeigen und zu verändern. Verwendet wird es häufig nach einer Suche, um die “markierte” Entität in der Ergebnisliste mit zusätzlichen Informationen (z.B den Positionen) bearbeiten zu können. Das Kommando bildet einen kompletten Zyklus mit dem (1) checkout der notwendigen Daten, (2) dem verändern der Daten und (3) dem checkin der veränderten Daten ab.<br><br>Das GRAPH_OWNER Kommando öffnet immer einen neuen Haupt-Tab und instanziert eine neue Session. Dabei wird dem Kommando üblicherweise der Schlüssel derjenigen Entität übergeben, die sich an der Wurzel des Objektgraphen befindet. Mit diesem Schlüssel lässt sich in einem ersten Schritt dann der zu bearbeitende Objektgraph laden. Häufig ist die Wurzel gleichzeitig das Prozessdokument.<br>Das Prozessdokument ist beim Aufruf eines GRAPH_OWNER Kommandos null und muss dann meist mit <name> = call(Repo.checkoutFunction(id)) sofort gesetzt werden.<br><br>Als Page-Conclusion ist ein "Speichern und Beenden" vorzusehen, welche das GRAPH_OWNER Kommando beendet. In der FINAL_OK_CONCLUSION selbst wird dann der checkin als Session Operation angehängt und COMMIT_SESSION eingestellt, sodass alle Operationen der Session abgearbeitet werden.<br><br>Bricht der Benutzer das Kommando mit der Schaltfläche "Abbrechen" ab, erfolgt erst eine Rückfrage in der Form "Änderungen gehen verloren, wirklich abbrechen?". Der Anwender muss diese nochmals mit "OK" bestätigen.<br><br>Ein GRAPH-OWNER Kommando verfügt über eine eigene Session und dadurch über einen eigenen Zwischenspeicher. Der Zwischenspeicher behält alle geladenen Entitäten. Zugriffe auf Read-Only Entitäten mit get() Operationen führen zu keinem weiteren Datenbankzugriff und sind entsprechend schnell. Pro Haupt-Tab werden Entitäten zwischengespeichert, was evtl. auch Endanwendern erklärt werden sollte. |
| GRAPH_EDIT | *Startet keine neue Session, wird in einem modalen Fenster ausgeführt.*<br><br>Einfaches Kommando, das ein Objekt des Objektgraphen verändert. Es werden keine Operationen zur Session hinzugefügt, lediglich der Objektgraph manipuliert. Die Session wird vom Vater-Kommando übernommen, daher ist kein COMMIT_SESSION in diesem Kommando möglich. |

| | |
|---|---|
| | Eingesetzt wird dieses Kommando vor allem für einfache Editorfenster (Kopfpdaten editieren / markierte Position editieren).<br><br>Grundsätzlich sollten GRAPH_EDIT Kommandos keine weiteren Objekte nachladen. So ist sichergestellt, dass GRAPH_EDIT in Hinblick auf Performance schnell abgearbeitet wird. |
| GRAPH_ADD | *Startet keine neue Session, wird in einem modalen Fenster ausgeführt.*<br><br>Das GRAPH_ADD verhaltet sich exakt wie das GRAPH_EDIT Kommando, außer dass kein bestehendes Objekt verändert, sondern ein neues an den Objektgraphen angehängt wird. Dazu wird erst das Objekt erzeugt, der Anwender kann es in einem modalen Fenster editieren, dann wird es schließlich in der FINAL_OK_CONCLUSION dem Objektgraphen angehängt. |
| OPERATION_ADD | *Startet keine neue Session, wird in einem modalen Fenster ausgeführt.*<br><br>Manipuliert Elemente am geladenen Objektgraphen und hängt neue Operationen an die Session an. Im Vergleich zu GRAPH Kommandos wird nicht nur der Objektgraph, sondern auch der Zustand der Session verändert. OPERATION_ADD Kommandos sind grundsätzlich eher zu vermeiden. Sie sind meist etwas undurchsichtig.<br><br>Beispiel eines OPERATION_ADD-Kommand: Endgültiges Löschen einer Position aus einem Objektgraphen. Dazu: (1) Positions-Entität aus dem Objektgraphen entfernen und (2) Operation in der FINAL_OK_CONCLUSION an die Session anhängen, um die Entitäts-Position auch von der Datenbanktabelle zu löschen.<br><br>Die Anwendung von OPERATION_ADD Kommandos wird in zukünftigen MoWare Versionen einfacher gestaltet. Insbesondere besseres Session-Handling birgt große Potentiale. |
| NEW_GRAPH<br>(new session) | *Startet neue Session, wird in einem Haupt-Tab ausgeführt.*<br><br>NEW_GRAPH Kommandos werden verwendet, um einen neuen Objektgraphen zu erzeugen. Dazu wird häufig ein Wizzard ähnlicher Aufbau verwendet.<br><br>Gegenwärtig entspricht NEW_GRAPH exakt dem GRAPH_OWNER. |
| SUB_GRAPH_OWNER<br>(new session) | *Startet neue Session, wird in einem modalen Fenster ausgeführt.*<br><br>Diese Kommando bildet eine Ausnahme. Es verfügt über eine eigene Session, das modale Fenster verleiht im allerdings den Charakter eines Short-Running Kommandos. Es muss erst beendet werden, um mit der Applikation weiter arbeiten zu können. |

| | |
|---|---|
| | SUB_GRAPH_OWNER Kommandos werden nur dann verwendet, wenn bereits ein GRAPH_OWNER/NEW_GRAPH mit eigener Session ausgeführt wird und man aber zusätzlich eine neue Session kurzzeitig benötigt.<br><br>Häufig wird ein SUB_GRAPH_OWNER verwendet, wenn besondere Details zu dem aktuell bearbeiteten Objektgraphen von der Datenbank geladen und berechnet werden, diese dann aber wieder zu verwerfen sind. Keinesfalls dürfen Entitäten aus einem SUB_GRAPH_OWNER mit dem GRAPH_OWNER/NEW_GRAPH Kommando ausgetauscht werden. Es dürfen grundsätzlich keine Objekte zwischen Sessions getauscht werden.<br><br>Bei einem Abbruch erfolgt keine Abfrage "Wirklich abbrechen", sondern es gehen sofort alle Daten der Session verloren. Anwendungsentwickler sollten dieses Kommando grundsätzlich vermeiden. Es ist nur als Lösung zu sehen, wenn "alle Stricke reißen". |

## *Prozess und Kommandos*

Jedes Kommando ist genau einem Prozess zuzuordnen. Der Prozess bildet damit eine große logische Komponente in einer Applikation, die zahlreiche Kommandos unter einem gemeinsamen Dach vereint. Eng verbunden mit dem Begriff Prozess sind Zustände, Bedingungen und Rollen. Ein Prozess befindet sich in verschiedenen Zuständen, die durch Bedingungen gewechselt werden können. Je nach Zustand des Prozesses stehen verschiedene Kommandos zur Verfügung - andere sind nicht freigegeben. Rollen bilden Rechtekonzepte ab, die ebenfalls die Freigabe von Kommandos beeinflussen.

Der Prozess ist immer an ein Prozessdokument gebunden, das den Zustand des Prozesses widerspiegelt. Das Prozess-Dokument ist meist eine Entität, so dass diese einfach auf der Datenbank persistiert werden kann. Der Prozesszustand wird im Prozessdokument als Property vom Typ Status modelliert, so dass der Prozesszustand dem Wert des Statusfeldes in der Entität entspricht.

Im Vergleich zu einem Workflow-System (z.B. BPMN) gibt es also keine eigene Prozessinstanz, die durch die Applikation verwaltet werden müsste. Die in MoWare implementierte Form der Prozesse mit einem Prozessdokument ist transparent, bietet allerdings nicht die Möglichkeiten eines klassischen Business Prozess Modelling Frameworks (vgl. dazu DroolsFlow oder Activiti). Es wäre aber denkbar, ein derartiges Workflow-System zu späterem Zeitpunkt in MoWare zu integrieren.

Prozesse lassen sich in MoWare gegenwärtig nicht in Verbindung bringen, verknüpfen oder vereinen.

# Prozess + Bedingungen

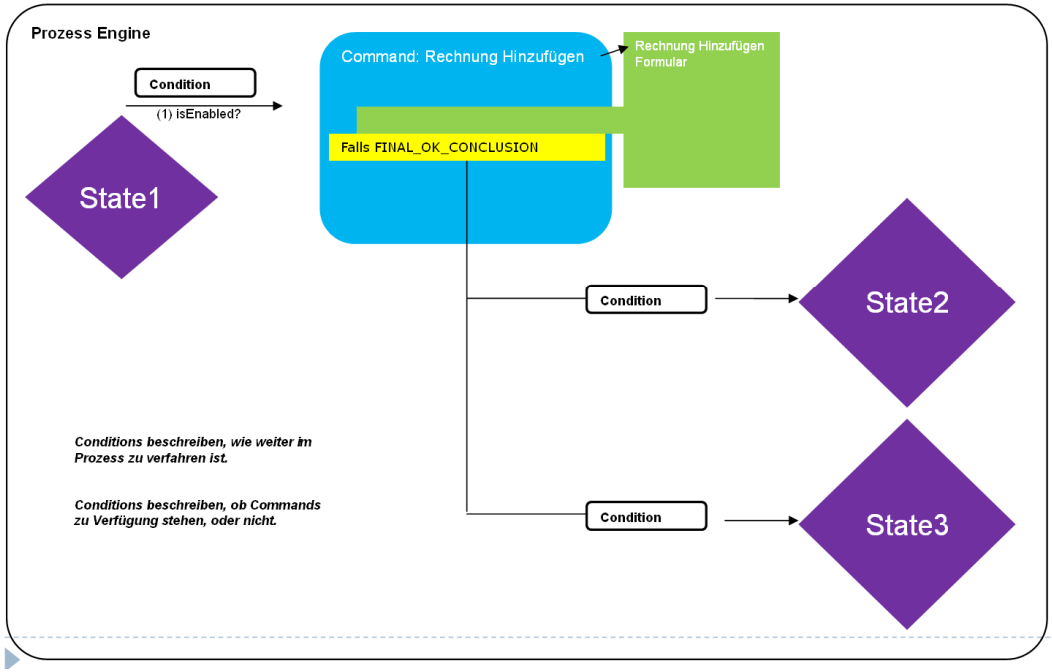



Abbildung 37: Prozess konzeptionell

Ein Zustand in einem Prozess dient zwei Zielen. Einerseits bestimmt der Zustand, welche Kommandos ausgeführt werden können - und welche nicht, andererseits sind in einem Zustand Bedingungen vermerkt, die, falls erfüllt, zu einem Zustandswechsel führen. Kommandos werden nicht selbstständig von der Prozessengine gestartet, sie müssen von “außen” durch den Endanwender angefragt werden (“on trigger”). Zustandswechsel können aber automatisch ("auto") oder jeweils nach dem erfolgreichen Beenden eines Kommandos geprüft und ausgeführt werden. Sie benötigen folglich keinen zusätzlichen Trigger. Ist nach erfolgreichem Beenden eines Kommandos keine der im Zustand angegebenen Bedingungen erfüllt, verbleibt der Prozess im aktuellen Zustand.

MoWare startet Kommandos immer über den Prozess, d.h. bei der Ausführung legt der Prozess eine Instanz des Kommandos an. Wird das Kommando mit der FINAL_CANCEL_CONCLUSION oder mit der FINAL_EXCEPTION_CONLUSION beendet, wird weder der Prozesszustand noch irgendwelche Bedingungen geprüft. Es wird verfahren, als ob das Kommando nicht aufgerufen wurde. Wird das Kommando hingegen mit FINAL_OK_CONCLUSION beendet, so übernimmt der Prozess die Kontrolle und evaluiert Zustände, Bedingungen und Trigger. Wenn notwendig, wird ein Zustandsübergang durchgeführt.

```
process 'Rechnungskontrolle'  using RekoAkt akt process-status-field is status

  <docu>


  creators and state-independent commands:  // and open a session?
    on trigger [<cndt> / <rls>] No Userinterface -> < >
    on trigger [<cndt> / <rls>] Rekoakt erstellen -> < >
    on trigger [<cndt> / <rls>] Rekoakte anzeigen -> < >
    on trigger [<cndt> / <rls>] No Userinterface -> < >
    on trigger [<cndt> / <rls>] Rekoakt bearbeiten -> < >
    on trigger [<cndt> / <rls>] Aufgabenliste drucken -> < >
    on trigger [<cndt> / <rls>] Rech.-Korrektur anschauen -> < >
    on trigger [<cndt> / <rls>] Rech.-Korrekturen drucken -> < >
    on trigger [<cndt> / <rls>] Gutschrift anfordern -> < >
    on trigger [<cndt> / <rls>] EDI Rechnungen anzeigen -> < >
    on trigger [<cndt> / <rls>] Rekoakt aus EDI erstellen -> < >
    on trigger [<cndt> / <rls>] Proforma hinzufügen -> < >
    on trigger [<cndt> / <rls>] Proforma entfernen -> < >


  states:
    state Angelegt: [<cndt>]
      help/docu text:
        Testdoku
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt> / <rls>] Proforma hinzufügen -> < >
      on trigger [<cndt> / <rls>] Proforma entfernen -> < >
      on trigger [<cndt> / <rls>] Rechnungsposition hinzufügen -> < >
      on trigger [<cndt> / <rls>] WEB korrigieren -> < >
      on trigger [<cndt> / <rls>] Rech.-Preis korrigieren -> < >
      on trigger [<cndt> / <rls>] Rech.-Menge korrigieren -> < >
      auto [Alle Rechnungen Ok && Alle Rekopositionen Ok && Alle Aufgaben abgeschlossen] -> Ok
      auto [Offene Aufgaben WEB] -> WEBKorrektur
      on exit: <exp>


    state WEBKorrektur: [<cndt>]
      help/docu text:
        Es wurden WEB-Kürzungen angefordert, die noch nicht abgeschlossen sind
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt> / <rls>] Aufgabe abschliessen -> < >
      auto [Alle Aufgaben abgeschlossen && !(Alle Rechnungen Ok && Alle Rekopositionen Ok)] -> Angelegt
      auto [Alle Rechnungen Ok && Alle Rekopositionen Ok && Alle Aufgaben abgeschlossen] -> Ok
      on exit: <exp>


    state Ok: [<cndt>]
      help/docu text:
        Wenn der Rekoakt im Status Korrekt ist, kann er freigegeben werden. Es werden dann alle Rechnungen
        Freigegeben und alle offenen Aufgaben als Erledigt markiert.
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt> / <rls>] Rechnungen freigeben -> < >
      on trigger [<cndt> / <rls>] Rech.-Preis korrigieren -> < >
      on trigger [<cndt> / <rls>] Rech.-Menge korrigieren -> < >
      auto [Alle Rechnungen freigegeben] -> Freigegeben
      on exit: <exp>
```

Abbildung 38: Prozess Teil 1

```
    state WEBKorrektur: [<cndt>]
      help/docu text:
        Es wurden WEB-Kürzungen angefordert, die noch nicht abgeschlossen sind
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt> / <rls>] Aufgabe abschliessen -> < >
      auto [Alle Aufgaben abgeschlossen && !(Alle Rechnungen Ok && Alle Rekopositionen Ok)] -> Angelegt
      auto [Alle Rechnungen Ok && Alle Rekopositionen Ok && Alle Aufgaben abgeschlossen] -> Ok
      on exit: <exp>


    state Ok: [<cndt>]
      help/docu text:
        Wenn der Rekoakt im Status Korrekt ist, kann er freigegeben werden. Es werden dann alle Rechnungen
        Freigegeben und alle offenen Aufgaben als Erledigt markiert.
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt> / <rls>] Rechnungen freigeben -> < >
      on trigger [<cndt> / <rls>] Rech.-Preis korrigieren -> < >
      on trigger [<cndt> / <rls>] Rech.-Menge korrigieren -> < >
      auto [Alle Rechnungen freigegeben] -> Angelegt
      on exit: <exp>



  conditions:
    condition 'Alle Rechnungen Ok'
      "Alle Rechnungen sind im Status Erfasst"
      akt.rechnungen.all({~it => it.status == RechnungsStatus.Erfasst; })
      <no helptext>

    condition 'Alle Rekopositionen Ok'
      "Alle Rekopositionen sind innerhalb der vorgeschriebenen Toleranz"
      akt.posStatus == RekoAktPosStatus.Ok
      <no helptext>

    condition 'Alle Rechnungen freigegeben'
      "Alle Rechnungen des Aktes wurde freigegeben"
      akt.rechnungen.all({~it => it.status == RechnungsStatus.Freigegeben; })
      <no helptext>

    condition 'Alle Aufgaben abgeschlossen'
      "Es muessen alle Aufgaben zum Akt abgeschlossen sein"
      akt.aufgaben.all({~it => it.status == Status.Erledigt; })
      <no helptext>

    condition 'Offene Aufgaben WEB'
      <no message>
      akt.aufgaben.any({~it => it.empfaenger == Empfaenger.Warenwirtschaft && it.status != Status.Erledigt; })
      <no helptext>
```

Abbildung 39: Prozess Teil 2

Wie in den Abbildungen zu sehen, ist beim Namen des Prozesses auch das Prozessdokument vom Typ RekoAkt als Variable "rekoAkt" spezifiziert. Die Variable ist in der gesamten Prozessdefinition verfügbar. Das Property status im RekoAkt soll als Prozessstatus fungieren. Der Prozess selbst ist in verschiedene Abschnitte gegliedert:

- Er beginnt mit dem Abschnitt "creators and state independent views". Dort sind all jene Kommandos aufgelistet, die unabhängig vom Status im RekoAkt (oder falls noch kein Akt vorliegt) gestartet werden können. Kommandos für die Suche oder zum Erzeugen ganzer Akte sind typische Beispiele. Kommandos in diesem Abschnitt müssen eine neue Session starten und - ganz wichtig - falls noch kein Prozessdokument vorliegt dieses erzeugen und der Prozessvariable im Kommando zuweisen. Ansonsten ist kein Prozessdokument nach Beendigung des Kommandos vorhanden und der Prozess wird auch keine weiteren Prüfungen/Zustandswechsel durchführen. Grundsätzlich soll man möglichst wenig Kommandos in diesem Abschnitt vermerken, da sie nicht durch den Prozess "geschützt" werden.

  Kommandos in diesem Abschnitt werden in der Benutzeroberfläche nie "disabled" (grau hinterlegt, d.h. nicht wählbar) angezeigt. Der Endanwender kann sie unabhängig vom Prozessstatus immer ausführen.

- Es folgt für jede mögliche Statusausprägung ein eigener Abschnitt, d.h. die Zahl der Statusabschnitte muss der Zahl der Ausprägungen des Status im Prozessdokument entsprechen. Für jeden Abschnitt gilt folgendes:

  Die Kommandos des Abschnitts können ausgeführt werden, wenn sich das Prozessdokument im entsprechenden Status befindet. Der Entwickler kann in jedem "on trigger" nach dem Kommandonamen einen Zielstatus angegeben. Wird das Kommando erfolgreich beendet, so wird der Zustand gewechselt. D.h. vom aktuellen Zustand wird erst die "on exit" Anweisung (falls vorhanden), dann die "on entry" Anweisung des Zielzustands/Abschnitts ausgeführt. Ist bei einem "on trigger" kein Zielzustand angegeben, prüft der Prozess alle "auto" Konzepte des aktuellen Abschnitts. Die erste Bedingung die greift (der Reihenfolge von oben nach unten) wird dann für den Zustandswechsel herangezogen. Aber nicht nur "auto" Konzepte können Bedingungen enthalten, sondern auch die "on trigger" können Bedingungen aufnehmen. Nur wenn diese auf true evaluieren, kann der Endanwender das Kommando starten. Ansonsten ist das Kommando "disabled" in der Oberfläche.

  Jeder Zustandsabschnitt enthält auch eine Bedingung. Diese Bedingung muss zu jedem Zeitpunkt gültig sein, zu dem sich der Prozess in diesem Zustand befindet. Dadurch können Validierungen implementiert werden. Ist eine Bedingung bei der Validierung nicht erfüllt, so wird der Message-Text in einem Dialogfenster ausgegeben.

  "auto" Trigger werden immer nach dem erfolgreichen Beenden eine Kommandos geprüft. Evaluiert die Bedingung des "auto" auf true, wird genau ein Zustandswechsel durchgeführt. Im Zielzustand werden keine weiteren "auto" evaluiert.

- Der letzte Abschnitt umfasst die Bedingungen mit den Condition Konzepten. Eine Bedingung hat
  - einen Namen (Spaces erlaubt)
  - einen Text, der angezeigt wird, falls die Bedingung geprüft wird und auf false evaluiert (Prüfung bspw. bei Prozess Validierung)
  - die Bedingung selbst, als Java-Expression formuliert
  - einen zusätzlichen Hilfetext

  Bedingungen können per Namen im Code verwendet werden, ohne dass sie bereits mit Java-Expression formuliert wurden. So kann der Entwickler bereits von Beginn an wesentliche Bedingungen in ihrer Bedeutung festhalten. Alle für den Prozess wichtigen Bedingungen und Zustandsübergänge sollen im Prozesskonzept selbst erfasst werden. Dritte können dann alleine aufgrund der Prozessdefinition einen ausreichenden Überblick über die Software und deren Funktionalitäten erlangen. Der Prozess ist eine wichtige Dokumentation.

## *Prozess und Rollen*

Rechte in einer Anwendung werden durch Rollen festgelegt, d.h. Anwender dürfen bestimmte Aktionen nur dann ausführen, wenn sie über eine bestimmte Rolle verfügen. Rollen können damit neben Prozessen Kommandos freigeben oder sperren. Rollen können aber nicht nur den Zugriff auf Kommandos, sondern auch den Zugriff auf Objekte einschränken (z.B. Einschränkung von Lieferanten für unterschiedliche Anwender). Nur bestimmte Instanzen eines Objekttyps können dann von einem Benutzer bearbeitet werden.

Zusätzlich wird zwischen ReadOnly und Read/Write Sessions unterschieden. Der Entwickler kann über Rollen die Kommandos mit ReadOnly oder Read/Write Sessions starten (nur bei GRAPH_OWNER/NEW_GRAPH/SUB_GRAPH_OWNER Kommandos relevant). In einer ReadOnly Session werden zwingend alle Entitäten im Zustand ReadOnly geladen, auch wenn in Repo-Methoden bei Abfragen checkout angegeben wurde. Dadurch ist es für den Endanwender nicht möglich, geladene Objekte zu verändern. Außerdem werden Schaltflächen, die ein Kommando mit FINAL_OK_CONCLUSION beenden, "disabled" geschalten (nicht wählbar). Auch Editoren in Formularen (z.B. Textfelder) sind im disabled Zustand, so dass der Endanwender Daten nicht verändern kann.

Im nachfolgenden Bild ist der RekoProzess in einem Tab geöffnet. Der Cursor befindet sich auf einem "on trigger" , der das Kommando "Rechnungskopf bearbeiten" startet. Das Inspektor-Fenster zeigt die Zugriffsrechte. Es gibt keine Einschränkungen für ReadOnly, d.h. jeder kann das Kommando mit einer ReadOnly Session starten, nur Anwender mit der Rolle Einkäufer können das Kommando mit einer ReadWrite Session ausführen.  Um die Rolle zu prüfen, wird dem role() Konzept der Lieferant der Rechnung übergeben. MoWare bietet das role () Konzept nicht nur im Inspektor-Fenster an, es ist überall im Quellcode verfügbar.

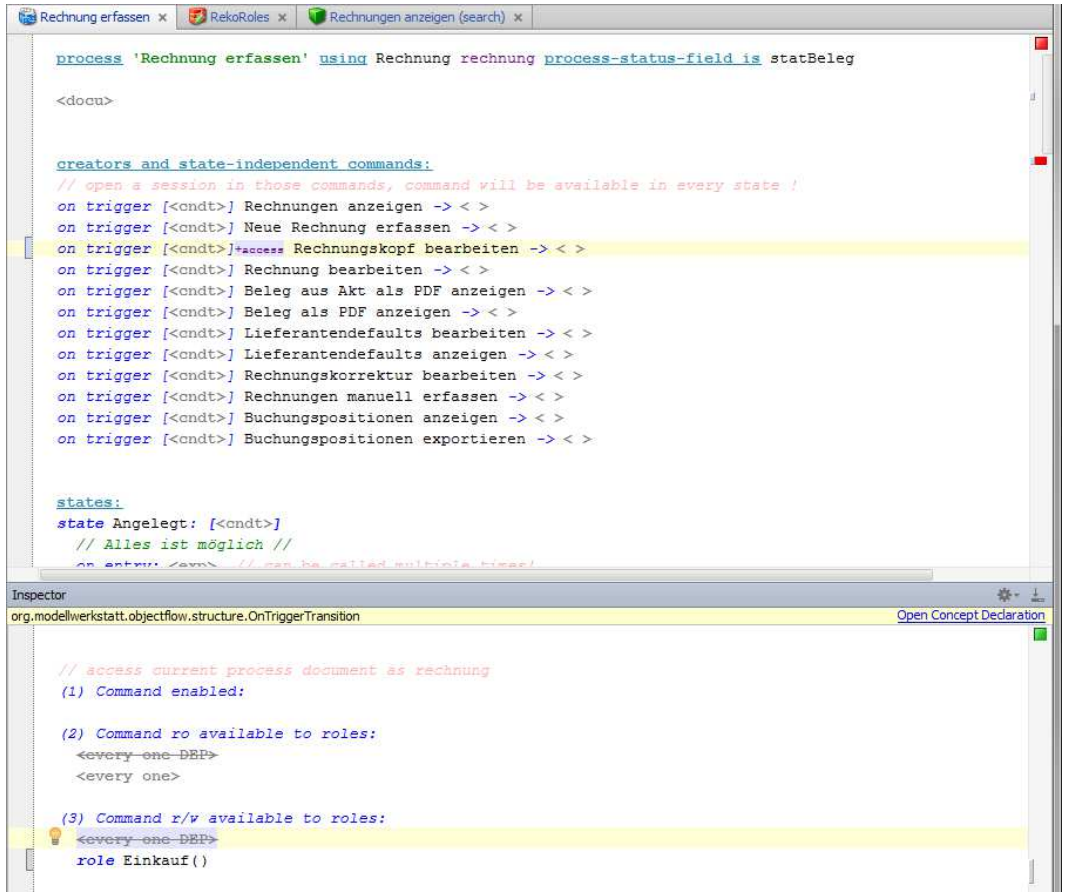



Abbildung 40: Inspector-Fenster

```
Roles and scopes repository RekoRoles


static roles:
static role 'ADMIN'
  overwrites/is also static role Reko Pruefer
  <no doc>
  is(userEnvironment)->boolean {
    list<AppManager> appmanager = call MitarbeiterRepo.findAppManagerByAdUser("REKO", userEnvironment.getUsername()) ;
    return appmanager.any({~it => it.OPTIONS.contains("ADMIN"); });
  }


static role 'Reko Pruefer'
  overwrites/is also static role Einkauf, RekoSachbearbeiter
  <no doc>
  is(userEnvironment)->boolean {
    list<AppManager> appmanager = call MitarbeiterRepo.findAppManagerByAdUser("REKO", userEnvironment.getUsername()) ;
    return appmanager.any({~it => it.OPTIONS.contains("PRUEFER"); });
  }


static role 'Einkauf'
  overwrites/is also static role <is also>
  <no doc>
  is(userEnvironment)->boolean {
    list<AppManager> appmanager = call MitarbeiterRepo.findAppManagerByAdUser("REKO", userEnvironment.getUsername()) ;
    return appmanager.any({~it => it.OPTIONS.contains("EINKAUF"); });
  }


static role 'RekoSachbearbeiter'
  overwrites/is also static role <is also>
  <no doc>
  is(userEnvironment)->boolean {
    list<AppManager> appmanager = call MitarbeiterRepo.findAppManagerByAdUser("REKO", userEnvironment.getUsername()) ;
    return appmanager.any({~it => it.OPTIONS.contains("READONLY"); });
  }



dynamic roles:
<roles>

scopes:
<scopes>
```

Abbildung 41: Rollen

Es werden grundsätzlich drei verschiedene Berechtigungsverfahren unterschieden: statische Rollen, dynamische Rollen und Scopes.

- ad statische Rollen: Statische Rollen werden einmalig beim Programmstart berechnet. Sie hängen meist ausschließlich vom userEnvironment ab und werden zum freischalten/sperren von Kommandos verwendet. In der obigen Abbildung wurden vier verschiedene Rollen definiert. Mit "Overwrites/is also role" lassen sich Hierarchien abbilden. In diesem Beispiel erhält ein Anwender, der die Rolle RekoPruefer erhält, automatisch auch die Rollen Einkauf und RekoSachbearbeiter. Dadurch kann der Anwender auch Kommandos starten, welche die Rollen Einkauf oder RekoSachbearbeiter benötigen. Schließlich ist bei allen Rollen die is() Funktion implementiert. Als Parameter sind bei der Funktion userEnvironment genannt.

- ad dynamische Rollen: Dynamische Rollen hängen nicht nur vom userEnvironment ab, sie hängen zusätzlich von einem weiteren Objekt (meist eine Entität) ab. Damit lassen sich Rollen bilden, die bspw. abhängig von einem Vorgangsort vergeben werden. In der aktuellen Implementierung wird die Berechnung der Rollen nicht gecacht. Es darf bei Repozugriffen daher nur mit get() gearbeitet werden, so dass die Session das Caching übernimmt.

- ad Scopes: Scopes berechnen eine Liste von Objekten, auf die bestimmte Benutzer oder bestimmte Rollen Zugriff haben. Sie können auf das userEnvironment und auf die Rollendefinitionen zugreifen. Zusätzlich können sie noch von weiteren Parametern abhängen ("depends on"). Der Entwickler verwendet scopes häufig bei Such Kommandos. Endanwender könnne in Suchkommandos dann nur jene Daten suchen, zu denen sie auch berechtigt sind. Wie bei Rollen mit role() können Scopes mit scope() verwendet werden.

  Wird das scope an eine Datenbankabfrage übergeben, um diese Einzuschränken (ManMap in Operator), so kann zusätzlich mit optional() gearbeitet werden. Bei null wird das optional() dann gerade nicht evaluiert, d.h. nicht zur weiteren Einschränkung von SQL Selects herangezogen. Das entspricht wiederum genau der scope() Logik von "keine Einschränkung".

  In der nachfolgenden Abbildung wurde ein scope über Rollen eingeschränkt. Die Berechnung selbst ist in diesem Beispiel nicht sinnvoll.

```
scope 'Vorgangsorte'
  restricts VorgangsOrt
  depends on <param>
  <no doc>
  scope(userEnvironment)->list<?> {
    if (role ADMIN()) {
      call ParteienRepo.findVorgangsOrte() ;

    } else if (role Einkauf()) {
      call ParteienRepo.findVorgangsOrte() ;

    }

    return new arraylist<VorgangsOrt>;
  }
```

Abbildung 42: Scope

## *Aufruf von Kommandos*

Der Anwendungsentwickler kann Kommandos nicht wie Repository- oder Service-Methoden mit dem Konzept call starten. Speziell für Kommandos bietet MoWare daher das Konzept run() und on trigger run() an. Soll ein Kommando in der Oberfläche über eine Schaltfläche oder über das Menü gestartet werden, so ist on trigger run zu verwenden.

```
on trigger "Rechnungen anzeigen" run RechnungsProcess.Rechnungen anzeigen(null, ModeFilter.NotDefined)
  view for page: ListView = RechnungListFc
  view for page: Suchen = DokumentAuswahlSearchFC
hk: UNDEFINED
```

Abbildung 43: CommandTrigger

Das Label der Schaltfläche oder des Menüeintrags, der das Kommando startet, entspricht grundsätzlich immer dem Kommandoname. Mit der Intention "Add Custom Trigger Text" kann der Anwendungsentwickler auch ein spezifisches Label angeben, was in diesem Beispiel als Zeichenkette in grün erkennbar ist. Das "Rechnungen anzeigen" Kommando übernimmt zwei Parameter, zum einen eine Rechnung als Prozessdokument (hier null gesetzt), zum anderen einen Status. Wesentlich an dieser Stelle: Die Pages ListView und Suchen wurden mit zwei Formularen verbunden. Ein HotKey wurde beim on trigger run keiner angegeben. Im nächsten Kapitel werden wir die Formulare und vor allem deren Definition genau erörtern.

Kommandos können aber auch direkt in Java Code - wie Funktionen - aufgerufen werden, was das Schreiben von Tests ermöglicht. Dem Konzept “run” müssen dann für jede Page sogenannte fakeUi Seiten - also keine echten Forms3 Seiten, sondern Substitute - übergeben werden. Diese fakeUi entsprechen simulierten Anwendereingaben. Sie lösen auch gleich die angegebene Conclusion aus. Damit lassen sich komplette Interaktion simulieren.

```
test (0) EXECUTE  "Rechnung anhand von EDI Rechung erzeugen. " {
  session = StandAloneApplicationFactory.getNewManMapSession();
  list<int> ediIds = new arraylist<int>{643419, 643391, 643383, 643375};
  EdiRechnung ediRechKopf = call EdiRepo.findEdiRechnung(ediIds.first):session ;


  session = StandAloneApplicationFactory.getNewManMapSession();
  // aufruf mit übergabe der session
  run EDI-Import.Rechnung aus Edi erstellen(ediRechKopf)
    fake ui input at page: Standard
      fakeUi(boundObjects)->void {
        Rechnung rechnung = boundObjects.first;
        System.out.println(rechnung);
      }
      and conclude with Speichern & Beenden

  with session

}
```

Abbildung 44: run Konzept

In der Abbildung wurde ein Test implementiert. Zwei Sessions werden verwendet, um den Kopf einer Edi-Rechnung zu laden und anschließend eine Rechnung zu erzeugen. Die Page "Standard" des Kommandos "Rechnung aus Edi erstellen" wurde als fakeUi implementiert. Offensichtlich kopiert das Kommando eine Liste von Rechnungen auf die Oberfläche (vermutlich nur Länge 1), die im fakeUi als boundobjects Parameter zur Verfügung steht. Als Conclusion wird anschließend "Speichern & Beenden" gewählt. Dem run wird eine Session (session Variable) übergeben.

Auch Prozesse und deren Bedingungen können direkt im Java Code verwendet werden. Dazu dient das Konzept

in <<Process>> is <<Condition or Role>>(<<process document>>)

Für RolesAndPermissions sind die beiden Konzepte role_is() und role_scope() vorgesehen, die direkt im Java Code angewendet werden können.

## *Konventionen*

Das automatische Überprüfen von Bedingungen mit anschließendem Zustandswechsel sollte immer am Ende der on trigger/auto Liste angegeben werden. Trigger / Auto sollten zur besseren Übersichtlichkeit nicht gemischt werden. Trigger zu Beginn der Zustandsdeklaration, Auto am Ende.

Der Status eines Prozessdokuments sollte nicht direkt verändert werden, der Prozess ist dafür zuständig. Gerade auch Prozessdokumente eines weiteren Prozesses sollten nicht manipuliert werden. Der weitere Prozess sollte selbstständig über Bedingungen der Situation entsprechend das Dokument anpassen. Dazu kann der Prozess mit

recheck <<Process>> (<< Document >>)

nochmals evaluiert, d.h., Status angepasst, werden.

Am Beispiel Abschließen eines Prozessdokumentes und eröffnen eines neuen: Häufig sind Buchungen einem Prozessdokument anzuhängen, bis eine bestimmte Bedingung dieses Dokument schließt. Weitere Buchungen sind dann einem neu erzeugten Dokument anzuhängen. Dies kann mit Hilfe von Prozessen folgendermaßen gelöst werden:

- Aktuelle Buchung über Kommando dem Prozessdokument anhängen
- Prozess prüft anschließend Abschlussbedingung (auto) und verändert evtl. Zustand des Prozessdokumentes (auf abgeschlossen)
- Zustand des Prozessdokumentes nochmals prüfen und gegebenenfalls
  - Prozessdokument speichern
  - und neues Prozessdokument erzeugen

## *Problems and Solutions*

| | |
|---|---|
| IllegalStateException | Die Prozess Engine kann aus mehreren Gründen eine IllegalStateException werfen:<br><br>(1) Eine Validierungs-Bedingung zu einem Zustand ist nicht erfüllt, obwohl das Prozess Dokument diesen Zustand des Prozesses anzeigt<br><br>(2) Ein Prozess wurde beauftragt ein Kommando zu starten, welches er nicht kennt.<br><br>(3) Ein Prozess wurde beauftragt ein Kommando zu starten, dessen enabled Bedingungen (+ Prozess-Zustand) zu diesem Zeitpunkt nicht positiv evaluiert wurden. |

## *Ergänzungen zu MoWare 3, RC30;*

Mit dem Releasecandidate 30 wurden die ersten Features für Variantenmanagement implementiert. In der Konfiguration kann aus 6 verschiedene Varianten gewählt werden. Mit ifvariant können Codeblöcke variantenspezifisch ausgeführt werden.

SUGAR nimmt als Variante eine besondere Position ein. Treten in der Variante SUGAR java Exceptions auf, so wird bei Oberflächen kein Stacktrace ausgegeben, sondern die Exception mit zugehörigem Trace wird geloggt. An der Oberfläche wird lediglich "Das Kommando konnte am System nicht ausgeführt werden" ausgegeben. Der Anwendungsentwickler kann sich auftretende Exceptions und deren Trace per Mail zukommen lassen, indem er einen Maillogger in der Konfiguration einstellt.

# 6. Benutzeroberflächen

Neben der Geschäftslogik und der Datenbankschnittstelle sind Oberflächen zentralstes Element von Geschäftsanwendungen. Gerade die Erstellung von Oberflächen ist üblicherweise sehr zeitaufwendig, wenn man nicht nur an die Erstellung der grafischen Elemente, sondern auch an deren Verwaltung in der Anwendung denkt. Mit der Sprache Forms3 unterstützt MoWare sehr einfach die Erstellung von ERP-ähnlichen Desktopanwendungen. ERP-ähnlich soll in diesem Zusammenhang bedeuten: Daten werden in Tabellen und Formularen angezeigt, die Anordnung von Tabellen und Formularen ist über Anwendungen hinweg ähnlich, spezifischen Widgets zur Darstellung werden vermieden. Insgesamt wird ein hohes Maß an Standardisierung angestrebt, auch wenn in Einzelfällen andere Darstellungen in der Oberfläche für Endanwender vielleicht günstiger wären.

Konzepte der Benutzeroberfläche sind strikt von der Geschäftslogik getrennt, sodass sie einfach ausgetauscht werden können, ohne die restlichen Teile der Applikation anpassen zu müssen. Eine Migration auf Mobiles oder Tablets ist dadurch möglich.

Im folgenden werden die wichtigsten Konzepte von Forms3 im Detail erläutert. Dann wird gezeigt, wie eine Desktopanwendung/Desktop Plugin erzeugt werden kann. Zuerst muss allerdings anhand eines Beispiels - die Anzeige eines Rechnungskontrollaktes - in das sehr grundlegende Konzept von Master- /Detailansichten eingeführt werden.

In der folgenden Abbildung ist ein Akt mit zwei Rechnungen dargestellt. In der Tabelle unterhalb der Rechnungen werden Rechnungspositionen angezeigt und zwar ausschließlich diejenigen, die der aktuell selektierten Rechnung zugeordnet sind. Wird die zweite Rechnung selektiert (bspw. durch Mausklick), dann ändern sich automatisch die Positionsdaten. Nur Rechnungspositionen der zweiten Rechnung werden angezeigt. Die Tabelle der Rechnungspositionen zeigt Details des Masters - der Rechnung - an.

Im Hintergrund der Ansichten arbeitet immer der Selektionskontroller. Er verwaltet die unterschiedlichen Ansichten und kann Master/Detail Anordnungen unbegrenzter Tiefe und über Reiter hinweg steuern. Zu jedem Zeitpunkt kann er das selektierte Element eines bestimmten Objekttyps angeben. In der dargestellten Ansicht liefert bspw. getSelected(RekoAkt) den aktuellen Akt - es gibt ohnedies nur den angezeigten, getSelected(Rechnungsposition) die Position 20 "Blistex Lip Conditioner". Natürlich können auch mehrere Objekte vom selben Typ in einer Tabelle selektiert werden (bspw. Strg Taste und Mausklick). getSelected(<<Typ>>) liefert dann null, da kein einzelnes Objekt selektiert wurde. Verwendet werden muss stattdessen getSelectedList(<<Typ>>).

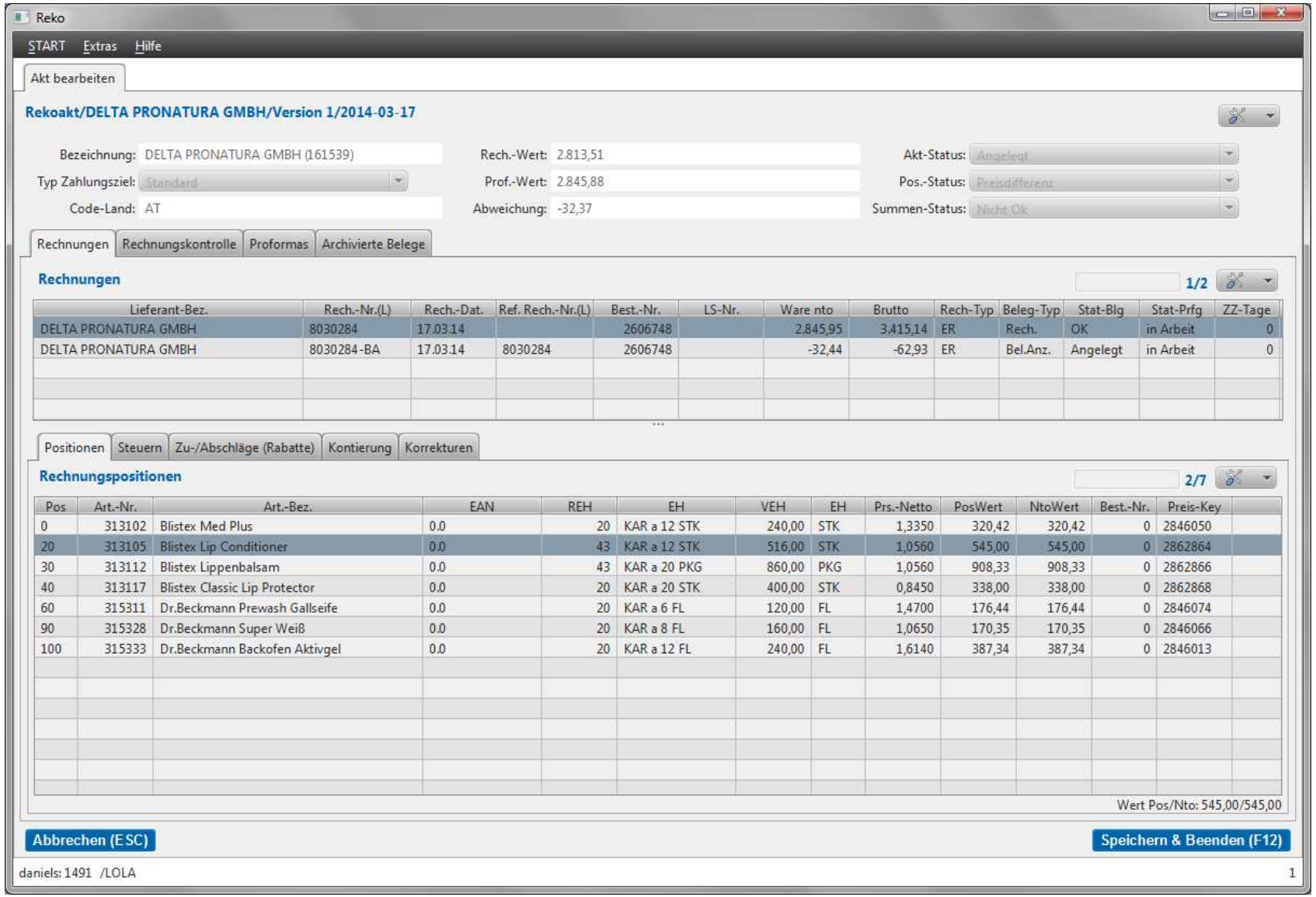


Abbildung 45: Oberfläche Rechnung

## *Forms der Benutzeroberfläche*

Die gesamte Benutzeroberfläche wird durch Forms beschrieben, die ineinander geschachtelt werden. Forms3 unterstützt in der aktuellen Version dazu die folgenden Konzepte:

TableForm - kann eine Liste von Objekten des selben Typs in einer Tabelle anzeigen
TabContainer - kann mehrere Reiter mit weiteren Formularen aufnehmen, d.h primär für Layoutzwecke geeignet
DelegateForm - ein Eingabeformular, das Editoren für verschiedene Properties von Objekten bereitstellt
FormContainer - kann weitere Formulare anordnen, d.h. ebenfalls für Layoutzwecke vorgesehen.

Wie zu erkennen ist, wird bei den Konzepten wird zwischen Container und Forms unterschieden. Container dienen dem Layouting, währen die Forms der eigentlichen Anzeige von Daten dienen. Sie können die Daten auch editieren.

### *TableForm*

Das TableForm Konzept dient der Modellierung von Tabellen. Wie alle Konzepte der Oberflächen ist auch die Tabelle recht einfach zu handhaben. Neben einem Namen ist der Typ anzugeben, der in einer Tabelle visualisiert wird. Im nachfolgenden Fall ist das eine Liste von Rechnungen. Der Typ kann bei TableForm Konzepten grundsätzlich immer nur eine Liste von Objekten annehmen, nie ein einzelnes Objekt.

Der Anwendungsentwickler kann die einzelnen Spalten mit Hilfe der Variable boundObject konfigurieren. BoundObject steht stellvertretend für ein Objekt der Liste, um auf die Properties des Objekts zuzugreifen. Mit setProperty() werden die verschiedenen Spalten der Tabelle konfiguriert. SetWidth() legt die Breite fest. (Vorsicht: JavaFX verhält sich undeterministisch, wird 81 Pixel als Breite gewählt - kein Witz). Die Tabellenüberschrift wird aus der Short-Description berechnet, die in den Objekten (Entität, ValueObject, ViewObject) angegeben wird. Ist ein einfacher Pfad angegeben, wird nur die Short-Description vom Property selbst verwendet (z.B. im Fall von boundObject.id). Umfasst der Pfad mehr als ein Property, werden die Short-Descriptions entlang des Pfades mit "-" als Trennzeichen zusammengesetzt. Gerade diese Eigenschaft ist zu berücksichtigen, wenn in Entitäten bspw. ValueObjects oder Referenzen verwendet und beschriftet werden. Das Anzeigeformat wird ebenfalls direkt vom Zielproperty der Spalte abgeleitet. Wie der Name bereits ausdrückt, kann mit <overwrite label> eine Zeichenkette als Tabellenüberschrift, mit <overwrite format> ein anderes Format spezifiziert werden. Aus der Perspektive der Standardisierung ist dies allerdings zu vermeiden.


Abbildung 46: TableForm

Für diese Tabelle wurden keine Anweisungen zum Laden von Daten hinterlegt. Möglich wäre dies durch Einstellungen beim "loaded with selected". Dort kann der Entwickler einen Objekttyp und ein Property angeben, von dem Daten in die Tabelle geladen werden. RekoAkt.rechnungen wäre eine passende Konfiguration. Vom aktuell selektierten Rechnungskontrollakt würde auf das Property "rechnungen" zugegriffen. Objekte dieser Liste werden in die Tabelle geladen. Bei TableForm muss das angegebene Property also immer vom Typ list<> sein. Eine andere Möglichkeit, Daten in Formulare zu laden, stellt das Include Konzept dar. Das Konzept wird im Verlauf dieses Kapitels noch gezeigt.

```
TableForm TfXRechnungen (list<Rechnung> with boundObject loaded with selected: <class>.<property>)
  label: <no heading>
  select first: false
  <advanced selections>

  MenuButton " "  (image: <no imageString> )
    on trigger run RekoProcess.Akt bearbeiten(null, getSelected(Rechnung.class))
      view for page: Standard = RekoAktFc
    hk: UNDEFINED


     ---------- Seperator ----------

    on trigger run RechnungsProcess.Rechnung bearbeiten(getSelected(Rechnung.class))
      view for page: Standard = RechnungFc
    hk: UNDEFINED

    on trigger run RechnungsProcess.Neue Rechnung erfassen(null)
      view for page: PageRechnungLieferant = RechBeleg1Fc
      view for page: PageRechnungskopf = RechBeleg2Fc
      view for page: GutschriftenAuswahl = RechnungListFc
      view for page: PageSumme = SteuerFc
      view for page: Rechnung = RechnungFc
    hk: UNDEFINED


     ---------- Seperator ----------

    on trigger run RechnungsProcess.Belastungsanzeige drucken(getSelected(Rechnung.class), "view")
      <views>
    hk: UNDEFINED

    on trigger run RechnungsProcess.Beleg drucken(getSelected(Rechnung.class), "view")
      <views>
    hk: UNDEFINED



  setProperty(boundObject.id) setWidth(50) <overwrite label> <overwrite format> ;
  setProperty(boundObject.nrBeleg) setWidth(60) <overwrite label> <overwrite format> ;
  setProperty(boundObject.datRechnung) setWidth(75) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferant.bezId) setWidth(250) <overwrite label> <overwrite format> ;
  setProperty(boundObject.nrRechnungL) setWidth(100) <overwrite label> <overwrite format> ;
  setProperty(boundObject.refRechnungL) setWidth(100) <overwrite label> <overwrite format> ;
```

Abbildung 47: TableForm mit Trigger

Das in dieser Abbildung gezeigte TableForm entspricht genau dem ursprünglichen TfXRechnungen, allerdings wurden Trigger der Tabelle hinzugefügt. Diese Trigger starten weitere Kommandos und werden in der Oberfläche als Schaltflächen neben dem Suchfeld der Tabelle platziert. Die bei einer Tabelle modellierten Trigger werden zusätzlich als Kontextmenü in der Tabelle angezeigt.  Parametrisiert werden die Kommandos der Trigger mit getSelected() Abfragen und entsprechenden Formularen für die Pages. Ferner wird bei zwei Kommandos die Zeichenkette "view" als zusätzliche Einstellung übergeben. Ein Status als Konstante zu wählen wäre passende.  Das Seperatorkonzept erzeugt einen Trennstrich im Menü.

```
TableForm TfRechPos (list<RechPos> with boundObject loaded with selected: <class>.<property>)
  label: "Rechnungspositionen"    (+ advanced selection)
  select first: false
  setTableSummaryLine(selectedObjects)->string: "Wert Pos/Nto: " +
      UserEnvironmentInformation.decimalFormat.format(MU.sum(selectedObjects.select({~it => it.wertPos.betrag; }))) +
      "/" + UserEnvironmentInformation.decimalFormat.format(
      MU.sum(selectedObjects.select({~it => it.wertNetto.betrag; })))

  MenuButton " "  (image: <no imageString> )
    on trigger run RechnungsProcess.Rechnungsposition bearbeiten(
        getSelected(Rechnung.class), getSelected(RechPos.class), getSelectedList(RekoAkt.class).size > 0)
      view for page: Standard = RechPosEditFC
    hk: ENTER

    on trigger run RechnungsProcess.Rechnungsposition hinzufügen(
        getSelected(Rechnung.class), getSelectedList(RekoAkt.class).size > 0)
      view for page: Standard = RechPosEditFC
    hk: ADD

    on trigger run RechnungsProcess.Rechnungspositionen löschen(
        getSelected(Rechnung.class), getSelectedList(RechPos.class), (getSelectedList(RekoAkt.class).size > 0))
      <views>
    hk: UNDEFINED



  setProperty(boundObject.posNr) setWidth(40) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artikel.id) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artBezRechnung) setWidth(250) <overwrite label> <overwrite format> ;
  setProperty(boundObject.ean.code) setWidth(110) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.betrag) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.txtEh) setWidth(110) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.betragVeh) setWidth(70) <overwrite label> <overwrite format> ;
```

Abbildung 48: Tabelle Rechnungspositionen

Als letztes Beispiel einer TabelForm wird hier die Tabelle für Rechnungspositionen gezeigt. Der Tabelle wurde mit einer Überschrift versehen ("label"). Ferner wurde auch eine TableSummaryLine eingestellt. Diese Zeile wird am unteren Ende der Tabelle in der Benutzeroberfläche angezeigt. Sie bietet genügend Platz um den Betrag und den Nettoertrag der selektierten Positionen (Formatierung mit richtigem Format durch DecimalFormatter) anzuzeigen. Bei der Modellierung der Trigger griff der Entwickler auf

einen Trick zurück. Offensichtlich wird die Tabellendeklaration in unterschiedlichen Ansichten verwendet: Bei Kommandos, die auch über einen RekoAkt verfügen und bei Kommandos ohne Akt. Mit getSelectedList() wird nun abgefragt, ob ein Akt verfügbar ist. Das Ergebnis ist true oder false. (Wie wir später noch sehen, werden Trigger mit getSelected() disabled, wenn kein selektiertes, einzelnes Objekt verfügbar ist. Mit getSelectedList() kann diese Funktion umgangen werden, da sie keine Auswirkung auf enabled/disabled Zustand von Trigger in der Oberfläche nach sich zieht.)

Alle Tabellen unterstützen mit CRTL-C eine Kopierfunktion. Dabei werden die Daten der selektierten Zeilen in der Tabellen mit zugehörigen Spaltenüberschriften als CSV in die Zwischenablage kopiert. Die Tabellen verfügen auch über ein Suchfeld. Gibt der Anwender eine oder mehrere Zeichen ein, wird die Tabelle nach diesen Zeichen durchsucht. Alle Datenobjekte in der Tabelle werden dabei auf eine Zeichenkette konvertiert. Angezeigt werden in der Tabelle dann nur noch jene Zeilen, die die gesuchte Zeichenkette enthalten.

### *DelegateForm*

Das DelegateForm ist das Komplexeste aller Formulare. Es dient primär als Editorformular zur Bearbeitung von Entitäten, ValueObjects oder ViewsObjects. In einem DelegateForm sind eine Zahl von Delegates deklariert, meist in mehreren Spalten. Delegates sind die kleinsten Editorkomponenten der Forms3 Sprache. Sie können einzelne Properties von Objekten bearbeiten. Für jeden Property-Typ gibt es ein eigenes Delegate. In der nachfolgenden Abbildung ist ein einfaches DelegateForm dargestellt.

```
DelegateForm RechKopfWizzDF (Rechnung as boundObject loaded with selected: Rechnung.<property>)
  col(|): ["1*"]
  label: <no lable>
  StringDelegate refLieferant setProperty(boundObject.lieferant.bezeichnung), setEnabled(false) ;
  IntegerDelegate belegNr setProperty(boundObject.nrBeleg), setEnabled(false) ;
  StringDelegate extBelegNr setProperty(boundObject.nrRechnungL), setEnabled(false) ;
  StringDelegate bezExtBelegNr setProperty(boundObject.refRechnungL) ;
  IntegerDelegate bestellNr setProperty(boundObject.bestellung.nummer) ;
  LocalDateDelegate rechnungsDatum setProperty(boundObject.datRechnung) ;
  LocalDateDelegate eingangsDatum setProperty(boundObject.datValuta) ;
  StatusDelegate typ setProperty(boundObject.typBeleg) ;

  <no onLoad>
  <no onStore>
  <no onValidate>
```

Abbildung 49. DelegateForm

Das DelegateForm dient der Bearbeitung eines Rechnungskopfes. Neben dem Namen wird, wie beim TableForm, der Objekt-Typ angegeben, der auf das Formular geladen werden kann - in diesem Fall ein Objekt vom Typ Rechnung, das im DelegateForm selbst als Variable boundObject zur Verfügung steht. Durch "loaded with selected" wird der Selektionskontroller angewiesen, das Formular mit der aktuell selektierten Rechnung zu laden.

Das Formular enthält zahlreiche Delegates unterschiedlichen Typs. Der Entwickler hat eine Spalte mit maximaler Breite definiert (col "1*"). Die Delegates werden daher einfach untereinander angeordnet. Jedes Deleagate wurde mit setProperty an ein Property des boundObject gebunden. Mit setEnabled(false) wurden drei Delegates für die Bearbeitung gesperrt. Als Label wird automatisch die Long-Desc des Properties herangezogen, das Format wird ebenfalls von dort übernommen. Mit setLabel() und setFormat() könnten diese Eigenschaften überschreiben werden. Die Beschriftungen folgen der selben Logik analog den Spaltenüberschriften der Tabellen.

Spalten können im DelegateForm Konzept durch die Col-Eigenschaft eingestellt werden. Die Notation für diese Eigenschaft basiert gegenwärtig auf Zeichenketten (In zukünftige MoWare Versionen ist diese Notation sicherlich anzupassen). Der Entwickler muss eine oder mehrere Spalten durch deren Größe modellieren. Grundsätzliche gelten dabei folgende Regeln

- "1*" wird sehr häufig verwendet und kennzeichnet ein Gewicht (durch den Stern) mit dem Wert von eins. Wird keine weitere Spalte angegeben, so nimmt diese Spalte den größtmöglichen Platz ein.
- "2* ist wiederum ein Gewicht. Es muss in Kombination mit anderen Gewichten verwendet werden. Eine Kombination ["2*", "4*"] bedeutet: Das Formular nimmt den ganzen verfügbaren Platz ein. Die erste Spalte ist 2/6 und die zweite Spalte ist 4/6 von diesem Platz breit.
- Drei gleichbreite Spalten können sehr einfache mit ["1*, "1*", "1*"] erzeugt werden.
- Pixelangaben statt Gewichte werden nicht mehr unterstützt. Früher konnte der Entwickler mit ["150", "150"] ein Formular mit zwei Spalten zu je 150 Pixel festlegen.

Etwas komplexer ist das nachfolgende DelegateForm. Es wird mit der aktuell selektierten Entität vom Typ Steuer geladen und zeigt drei Dezimalwerte an. Dabei wurde bei zwei DecimalDelegate ein OnUpadte() mit einer Berechnungsvorschrift formuliert. Verändert der Anwender einen Wert im Delegate, wird ständig die OnUpdate Funktion ausgeführt. Im folgenden Fall wird bei Veränderung im dPZ der Wert im dBS neu berechnet.

Bei jedem Delegate kann der Anwendungsentwickler ein Update-Closure mit der Intention “add an update closure” installieren. Immer, wenn beim Delegate eine neue Eingabe (Tastendruck) erfolgt und der aktuelle Wert des Delegates gültig ist, wird dieses Update-Closure ausgeführt. Der Wert jedes Delegate kann mit dessen Methoden getValue() und setValue() verändert werden (z.B. für automatischen Brutto/Netto Rechner). Ferner steht die Methode setValidationErrorText() zur Verfügung, um einen Fehler anzuzeigen.

```
DelegateForm SteuerDf (Steuer as boundObject loaded with selected: Steuer.<property>)
  col(|): ["1*"]
  label: <no lable>
  DecimalDelegate dPZ setProperty(boundObject.wert.przSteuer) onUpdate({ =>
        dBS.setValue((dPZ.getValue() * dBN.getValue()).divide(100.0d, 2, BigDecimal.ROUND_HALF_UP)); }) ;
  DecimalDelegate dBN setProperty(boundObject.wert.
        betragNetto) onUpdate({ => dBS.setValue((dPZ.getValue() * dBN.getValue()).
        divide(100.0d, 2, BigDecimal.ROUND_HALF_UP)); }) ;
  DecimalDelegate dBS setProperty(boundObject.wert.betragSteuer), setEnabled(false) ;

  onLoad(boundObject)->void {
    dBN.requestFocus();
  }
  <no onStore>
  <no onValidate>
```

Abbildung 50: komplexes DelegateForm

Zusätzlich wurde bei dem Beispiel eine OnLoad() Funktion definiert. Sie wird aufgerufen, bevor das boundObject (Parameter) an das Formular selbst weitergegeben wird. In der OnLoad() kann auch das voreingestellte Fokusverhalten angepasst werden. Hier erhält das zweite Delegate (dBN) den Eingabefokus.

Ad Eingabefokus: bei DelegateForms erhält immer das erste Delegate von oben den Fokus. Wurde es mit setEnabled(false) in ReadOnly gesetzt, so wird der Fokus an das nächste Delegate weitergegeben. Bei Validierung des Formulars wir der Fokus an das erste Delegate mit Validierungsfehler übergeben (mehr zur Validierung folgt).

In dem komplexeren DelegateForm ist am Pfad im setProperty() zu erkennen, dass die Dezimalwerte selbst nicht Felder in der Entität Steuer sind, sondern im Feld wert. Bei dem Feld handelt es sich um ein Rechnungswert ValueObject. Ohne auf das Objekt selbst genauer einzugehen, ist von der Definition eines ValueObjects her klar, dass dessen Inhalt nicht verändert werden darf. Das DelegateFrom ersetzt ganze ValueObjects daher automatisch, wenn einer der Werte im ValueObject verändert wird. Im vorliegenden Fall wird das gesamte Rechnungswert Objekt ersetzt, wenn im Formular einer der drei Dezimalwerte verändert wird.

Neben der OnLoad() Funktion kann ein DelegateForm auch eine OnValidate() enthalten. Das OnValidate() wird immer dann aufgerufen, wenn eine Page-Conclusion des zugrundeliegenden Kommandos ausgeführt werden soll. Ist in dieser Page-Conclusion "request save" auf true gestellt, wird erst von allen aktiven Formularen die OnValidate() abgearbeitet. Führt keine zu einem Validierungshinweis, kommt die Page-Conclusion selbst zum Zug, ansonsten wird der Hinweis angezeigt (direkt im/neben Delegate). Im nachfolgenden Beispiel eines DelegateForms wird das LocalDateDelegate geprüft, falls dessen Wert null entspricht, wird ein Fehlertext angezeigt und das Formular bleibt auf der Benutzeroberfläche sichtbar (keine Conclusion wird ausgeführt).

```
DelegateForm DokumentAuswahlSearchDF (DokumentFilter as boundObject loaded with selected: DokumentFilter.<property>)
  col(|): ["1*"]
  label: <no lable>
  ReferenceDelegate refLieferant setSuggestionFieldFormat("langBez"), setProperty(boundObject.lieferant),
      setOptional(true) ;
  LocalDateDelegate inpBis setProperty(boundObject.datBis), setOptional(true) ;
  LocalDateDelegate inpVon setProperty(boundObject.datVon), setOptional(true) ;
  IntegerDelegate inpAnzTage setProperty(boundObject.anzTage) ;
  StatusDelegate dMode setProperty(boundObject.mode) ;

  <no onLoad>
  <no onStore>
  onValidate()->void {
    if (inpBis.getValue() == null) {
      inpBis.setValidationErrorText("gültiges Datum eingeben");
    }
  }
```

Abbildung 51: Validierungs-Funktion

Liegen keine Validierungsfehler vor, so werden alle Daten von dem Delegate in das boundObject zurückgespeichert. Allerdings werden nicht alle Properties, sondern nur jene, bei denen sich der Wert geändert hat, zugewiesen.

Folgende Delegates zählen zu den Wichtigsten, die in Formularen verwendet werden können.

| Delegate | Beschreibung |
|---|---|
| IntegerDelegate | Anzeigen/Verändern von ganzzahligen Werten. Dezimalwerte (mit Komma) führen automatisch zu einem Validierungsfehler, so wie auch Texteingaben. |
| DecimalDelegate | Anzeigen/Verändern von Dezimalzahlen. Texteingaben führen automatisch zu Validierungsfehler. Der Wertebereich kann mit setMinimum() und setMaximum() beim Delegate eingestellt werden. Wurde nichts spezifiziert, so wird auf die RANGE Eigenschaft von BigDecimal-Properties zurückgegriffen, die zur Laufzeit auch als Meta-Information vorliegt. |
| StringDelegate | Anzeigen/Verändern von Zeichenketten. Mit setMinium() und setMaximum() kann eine Textlänge zur Validierung angegeben werden. Gerade die Textlängen müssen üblicherweise geprüft werden, da zu lange Zeichenketten in Datenbankspalten zu Fehler führen können; Reagiert auch auf die LENGTH() Eigenschaft, die in Properties vom Typ string angegeben werden kann. |
| DateTimeDateDelegate | Anzeigen/Verändern von Datum (nicht von Zeit) bei DateTime Properties. |
| LocalDateDelegate | Anzeigen/Verändern von Datum (Zeit nicht vorhanden) bei LocalDate Properties. |

| | |
|---|---|
| StatusDelegate | Anzeigen/Verändern eines Status-Attributes |
| ReferenceDelegate | Anzeigen/Verändern von Referenzen mit Hilfe eines Autocomplete-Textfeldes (z.B. Auswahl eines Lieferanten, Auswahl eines Artikels).<br><br>Neben einem setProperty() muss der Entwickler bei ReferenceDelegates immer auch ein setSuggestionFieldFormat("<<PropertyName>>, <<PropertyName>>") angeben. Dabei kann er eine oder mehrere Properties nennen (als Zeichenkette bis data, mit Komma getrennt), die zur Berechnung des Anzeigetextes der Autocompletion verwendet werden. Im Beispiel oben wurde der Entität Lieferant eine virtual Property "langBez" hinzugefügt, die einen kurzen Beschreibungstext berechnet. Selbstverständlich können auch normale Properties angegeben werden.<br><br>Die Angabe der Properties durch Zeichenketten ist problematisch. Ändern sich Namen der Properties, treten erst zur Laufzeit Fehler auf. In zukünftigen MoWare Versionen ist die Notation anzupassen. |

Jedes der Delegates unterstützt zahlreiche Funktionen zur Parametrisierung: So bspw.

- setLabel(), um den Beschriftungstext zu überschreiben,
- setFormat(), um gegebenenfalls ein Format zu überschreiben,
- setEditabel(), um ein Delegate nur zur Anzeige von Daten zu verwenden
- setOptional(), um gültige Eingaben bei einem Delegate nicht zu erzwingen (insbesondere bei ReferenceDelegate, StatusDelegate und DateDelegates). Wird kein gültiger Wert beim Delegate eingegeben, so setzt das Delegate den Wert auf 0 (int) oder auf null (bei Objekten). Gerade bei Formularen zur Spezifikation von Suchkriterien, deren Angabe nicht verpflichtend ist, wird das setOptional() gerne verwendet.

### *FormContainer*

Der FormContainer wird, wie bereits angesprochen, für Layoutaufgaben verwendet. Er erzeugt dem sogenannten Grid-Layout ähnliche Anordnungen. Dazu kann der Anwendungsentwickler eine Zahl von Zeilen und Spalten definieren. Alle untergeordneten Forms werden dann der Reihe nach in diesem Grid eingefügt, beginnend von rechts oben nach links unten. Die Layout Notation in Form von Zeichenketten für Spalten und Zeilen wurde beim DelegateForm bereits angeschnitten. An dieser Stelle nochmals eine Zusammenfassung:

- Mit einem Stern versehen Zeichenketten geben Gewichte an. "1*" kennzeichnet ein Gewicht mit dem Wert von eins. "2* ist wiederum ein Gewicht. Es muss in Kombination mit anderen Gewichten verwendet werden. Eine Kombination ["2*", "4*"] bedeutet, dass das Formular nimmt den ganzen verfügbaren Platz ein. Die erste Spalte ist 2/6 und die zweite Spalte ist 4/6 von diesem Platz breit. Gewichte füllen immer den maximal möglichen Platz aus.
- Drei gleichbreite Spalten können sehr einfache mit ["1*, "1*", "1*"] erzeugt werden.

- "-1" weist Layoutmanager an, möglichst wenig Platz zu verwenden. -1 muss bei DelegateForms verwendet werden, um diese kompakt darzustellen.

In der aktuellen Version von JavaFX können verschiedene Spalten/Zeilen Anordnungen noch zu Problemen führen. Mit Java8 sind diese Probleme behoben. Die Spalten/Zeilen-Notation wird dann auch im MoWare angepasst.

Im nachfolgenden Bild ist ein FormContainer dargestellt, der nur eine Spalte verwendet. In zwei Zeilen wird ein DelegateForm und ein TableForm verwendet. Das DelegateForm wurde mit "-1" minimiert, die Tabelle mit "1*" maximiert. Der FormContainer wird von außen mit einer Rechnung geladen. Das DelegateForm erwartet ebenfalls eine Rechnung und lädt sofort die Selektierte. Die TableForm benötigt eine Liste von RechPos und wird über den Selektionskontroller mit der rechPos von der aktuell selektierten Rechnung geladen.

```
FormContainer RechnungFc (Rechnung as boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["-1", "1*"]
  label: "Rechnung"
<no command trigger>

DelegateForm XRechnungDF2 (Rechnung as boundObject loaded with selected: Rechnung.<property>)
  col(|): ["1*", "1*"]
  label: <no lable>
  StringDelegate d11 setProperty(boundObject.lieferant.bezId), setEnabled(false) ;
  StringDelegate d12 setProperty(boundObject.kaufer.bezId), setEnabled(false) ;
  StringDelegate d41 setProperty(boundObject.nrRechnungL), setEnabled(false) ;
  LocalDateDelegate d42 setProperty(boundObject.datRechnung), setEnabled(false) ;
  IntegerDelegate d51 setProperty(boundObject.bestellung.nummer), setEnabled(false) ;
  LocalDateDelegate d52 setProperty(boundObject.bestellung.datum), setEnabled(false) ;
  DecimalDelegate d61 setProperty(boundObject.summeWarenWert.betrag), setEnabled(false) ;
  DecimalDelegate d62 setProperty(boundObject.summeZuAb.betrag), setEnabled(false) ;
  DecimalDelegate d72 setProperty(boundObject.summeBrutto.betrag), setEnabled(false) ;
  StatusDelegate dSteuerCode setProperty(boundObject.steuerCode), setEnabled(false) ;
  StringDelegate dMeldeInfo setProperty(boundObject.meldeInfo.langText), setEnabled(false) ;
  StatusDelegate dStatus setProperty(boundObject.statBeleg), setEnabled(false) ;

  <no onLoad>
  <no onStore>
  <no onValidate>

TableForm TfRechPositionen (list<RechPos> with boundObject loaded with selected: Rechnung.rechPos)
  label: "Rechnungspositionen"
  select first: false
  <table summary line>

  <no command trigger>

  setProperty(boundObject.posNr) setWidth(40) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artikel.id) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artBezRechnung) setWidth(250) <overwrite label> <overwrite format> ;
  setProperty(boundObject.ean.code) setWidth(110) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.betrag) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.txtEh) setWidth(110) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.betragVeh) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.menge.txtVeh) setWidth(50) <overwrite label> <overwrite format> ;
  setProperty(boundObject.prsNettoVeh) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.wertPos.betrag) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.wertNetto.betrag) setWidth(70) <overwrite label> <overwrite format> ;
  setProperty(boundObject.bestellung.nummer) setWidth(60) <overwrite label> <overwrite format> ;
  setProperty(boundObject.preisKey) setWidth(70) <overwrite label> <overwrite format> ;
```

Abbildung 52: Rechnungsansicht

### *TabContainer*

Der TabContainer ist wohl das einfachste Konzept von Forms3. Der Anwendungsentwickler kann nichts einstellen, da das Konzept keine verstellbaren Eigenschaften hat. Lediglich die Zahl der Tabs (Reiter) und deren Inhalt kann modelliert werden. Im Beispiel sind 4 Tabs, die mit Hilfe des Selektionskontrollers mit Daten befüllt werden. Unterschiedliche Properties werden über die selektierte Rechnung geladen.

```
TabContainer TcRechnung type: tab
  tab name: "Positionen"
    Include TfRechPos (container expects: list<RechPos>) (load with: Rechnung.rechPos ) ;
  tab name: "Steuern"
    Include TfSteuerPos (container expects: list<Steuer>) (load with: Rechnung.steuerPos ) ;
  tab name: "Zu-/Abschläge (Rabatte)"
    Include TfZuAbPos (container expects: list<ZuAb>) (load with: Rechnung.zuAbPos ) ;
  tab name: "Korrekturen"
    Include RechKorrPosListTf (container expects: list<RechKorrInfo>) (load with: Rechnung.rechKorrInfos ) ;
```

Abbildung 53: TabContainer

### *Include - Modularisierung von Anzeigehierarchien*

Forms lassen sich mit dem Include-Konzept in mehreren FormContainer oder TabContainer verwenden. Auf diese Weise muss der Anwendungsentwickler Forms, die an mehreren Stellen bzw. in unterschiedlichen Ansichten verwendet werden, nicht doppelt modellieren. Außerdem ist durch Verwendung von Include sichergestellt, dass Ansichten einheitlich gestaltet sind. Zum Laden mit dem Include-Konzept wird in diesem Kapitel weiter unten nochmals bei Erläuterungen zum Selektionskontroller eingegangen.

Die klassen MessageBox ist im MoWare gegenwärtig nicht implementiert. In Zukunft müssen aber Meldungskonzepte angedacht werden. Meldungen für Benutzer können mit dem notification() Statement jetzt schon ausgegeben werden. Es wird dann jeweils die letzte, aufgetretene Nachricht in der Status-Zeile angezeigt. Zukünftig werden Nachrichten prominenter in der Oberfächte dargestellt. JavaFx8 bietet verschiedene Möglichkeiten.

## *Der Selektionskontroller*

Der Selektionskontroller ist zentrales Element von Forms3 Oberflächen. Er kann sowohl mehrfach geschachtelte Master-Detail Ansichten automatisch verwalten, als auch umfangreiche Anzeigehierarchien. Seine Funktionalität lässt sich anhand von zwei Punkten darlegen:

- Mit Hilfe des Selektionskontroller kann auf jeder Stufe einer Anzeigehierarchie das selektierte Objekt (auch Objekte) eines bestimmten Typs ermittelt werden. So kann man bspw. sehr tief in einer Anzeigehierarchie ein weiter oberhalb selektiertes Objekt abfragen. Möchte man bspw. in einer Tabellenansicht (Rechnungspositionen) einen CommandTrigger setzen und benötigt dazu die Rechnung, kann die "oberhalb" selektierte Rechnung mit getSelected(Rechnung) ermittelt werden.

  *Bei komplexen Ansichten wird in einer Anzeige-Hierarchie häufig der selbe Typ mehrmals angezeigt (z.B. Buchungen in zwei verschiedenen Tabs.) getSelected(Buchung.class) ist daher nicht mehr eindeutig! Der* Selektionskontroller *wurde nun umgestellt, so dass er nur noch selektierte Elemente aus der übergeordneten Hierarchiestufen zu dem Abfragecontext liefert. Genauer: Er liefert selektierte Elemente aus der ersten übergeordneten Ansicht mit betreffendem Typ.*

- Mit Hilfe des Selektionskontroller werden bei den häufig verwendeten Master-Detail-Ansichten die Forms (Details) automatisch geladen. D.h. je nach selektiertem Master Objekt, wird eine Liste von Detail-Objekten durch den Kontroller in die Detail-View geladen. Durch diese Funktionalität können in Forms3 Ansichten relativ zu Selektionen geladen werden. Bspw. lassen sich Rechnungspositionen relativ zur selektierten Rechnung mit "Rechnung.rechnungspositionen" laden. Erstere gibt den Selektions-Objekt-Typ an, letzteres die Liste, welche die Positionen hält.

Erstellte TableForms, DelegateForms, FormContainer oder Includes müssen immer an Entitäten oder ViewObjekte gebunden werden (TableForms müssen immer an Listen gebunden werden). Neben dem Typ (der boundObject Variable) kann immer noch ein "loaded with" spezifiziert werden. Dieses "loaded with" gibt an, wie der Inhalt der Liste zu laden ist. Es sind grundsätzlich zwei verschiedene Ladeverfahren zu unterscheiden:

| | |
|---|---|
| Statisches Laden | Beim statischen Laden wird direkt eine Klasse A im <class> Feld des load-with angegeben. Ferner kann - muss aber nicht - zusätzlich ein <property> gewählt werden.<br><br>1. Wird kein <property> angegeben: Wenn bei Selektionskontroller eine Auswahl vom Type A vorliegt, wird das Form mit dieser Selektion geladen.<br><br>2. Wird ein <property> angegeben: Wenn beim Selektionskontroller eine Auswahl vom Type A vorliegt, wird auf das Property der Selektion zugegriffen und das Ergebnis ins Form geladen.<br><br>Steht keine Selektion vom Typ A oder mehrere Selektionen vom Typ A (z.B. Elemente einer Liste) zur Verfügung, wird das Form mit einer leeren Liste/ keinem Objekt geladen (entspricht Inhalt des Forms gelöscht). |
| Dynamisches Laden | Dynamisches Laden wird bei der Verwendung des Include-Konzeptes verwendet. Dabei bestimmt nicht das eingebundene Form, welche Selektion für das Laden relevant ist, sondern das Include selbst verfügt über ein load-with Konzept.<br><br>Analog den Erläuterungen zum "load-with" beim statischen Laden verhält sich auch das load-with des Includes. |

Wie bereits angesprochen kann die Situation entstehen, dass ein Objekttyp (z.B. Rechnungspositionen) in einer Anzeigehierarchie in mehreren Tabellen angezeigt wird. Denkbar wären zwei Tabellen, eine mit offenen Positionen, eine mit abgeschlossenen Positionen. Wird in einer Tabelle ein Objekt selektiert, so wird die Selektion aber in der kompletten Anzeigehierarchie verbreitet. In Tabellen, die ebenfalls Objekte vom selektierten Typ enthalten, das selektierte Objekt (die Instanz) aber nicht, wird die Selektion gelöscht. Es wird dann ein * bei dem n/m Indikator angezeigt. Das zeigt vorerst an, dass zwar eine Selektion mit passendem Typ vorliegt, in dieser Tabelle aber nicht angezeigt wird. In diesem Fall werden dann Kind-Formulare - wie zu erwarten - nicht geladen, falls das Binding über Selektion geschieht.

Bei getSelected() Abfragen in einem Trigger, prüft der Selektionskontroller erst lokal ab, d.h. die Prüfung erfolgt in dem FormContainer oder TableForm, in dem der Trigger selbst modelliert wurde. Bei einem Trigger in einem TableForm wird erst geprüft, ob die Tabelle selbst über eine Selektion vom angefragten Typ verfügt. Falls nicht, wird die Anfrage an das Vaterformular weitergegeben. Auf diese Weise populieren sich Anfragen die Anzeigehierarchie "hoch", bis zum Wurzelformular. Trigger, die sich auf Inhalte der Tabelle beziehen, sollen daher in der Tabelle selbst parametrisiert werden.

Es liegt zu jedem Zeitpunkt nur eine Selektion eines bestimmten Objekttyps vor. Es ist daher zulässig, beim obersten Selektionskontroller mit getSelected() eine eindeutige Abfrage zu stellen, die sich auf die Anzeigehierarchie bezieht. Damit könnten auf oberster Hierarchiestufe Trigger installiert werden, die sich auf Kindformulare "weiter unten" beziehen.

Es wird sehr schwierig, die Anzeigehierarchie nachzuvollziehen, wenn an unterschiedlichen Stellen in der Hierarchie Formulare verwendet werden, die den selben Objekttyp visualisieren. Das soll daher möglichst verhindert werden.

## *HotKeys*

Bei Hotkeys muss grundsätzlich zwischen HotKeys in Page-Conclusions und Hotkeys für CommandTrigger (Forms3) unterschieden werden. Während Erstere nur für Conclusions in Kommandos definiert werden können, sind Letztere in den Formularbeschreibungen vermerkt. Für Conclusions sind gegenwärtig folgende HotKeys zulässig.

| **Bezeichnung in OFX (zur Auswahl stehend)** | **Belegung** |
|---|---|
| SAVE | “Ende” / END Taste |
| ESCAPE | ”Escape” Taste |
| BACK | “F11” Taste |
| NEXT | “F12” Taste |
| INSERT | “INS” Taste |

Für Hotkeys, die Command Trigger auslösen, sind alle Tasten zulässig. In Zukunft wird dies noch genauer zu spezifizieren sein (Richtlinien). Grundsätzlich ist folgendes zu berücksichtigen:

- CommandTrigger in TableForms werden nur dann ausgelöst, wenn der Tastaturfokus sich auf der Tabelle selbst befindet. Ferner wird ein Doppelklick auf einer Tabellenzeile als ENTER Taste gewertet.
- CommandTrigger in FormContainer werden nur ausgelöst, wenn diese am Top-Level FormContainer (höchsten in der Anzeigehierachie) angebracht wurden. CommandTrigger in untergeordneten FormContainer sind nicht vorgesehen, bzw. sollen nicht verwendet werden. Es gilt: CommandTrigger nur in einer Tabelle oder im Top-Level FormContainer anbringen!

## *Das Application Konzept*

Neben den Formularen muss für eine Desktopapplikation auch ein “Application” Konzept angelegt werden. Das Konzept legt die zu verwendende Konfiguration fest, vor allem aber die Menüdefinition. Das Konzept dient auch als “Einstiegspunkt” einer Anwendung. Die Startup-Funktion wird unmittelbar nach dem Laden der Konfiguration ausgeführt. Auf Wunsch kann username (string) und passwort (int) abgefragt werden. Das Passwort wird zwar als Zeichenkette abgefragt, dann aber als hashCode Wert dem Anwendungsentwickler in der startup() Funktion übergeben. Wie in der nachfolgenden Abbildung ersichtlich, wird im startup() die Benutzer-Authentifizierung vorgenommen.

```
Application Reko

configuration: RekoConfigLola

version information: 1

show login screen: false

startup(username, password)->boolean {
  string adUser = System.getProperty("user.name");
  UserEnvironmentInformation envInf = (UserEnvironmentInformation)
      StandAloneApplicationFactory.findInstanceByName("userEnviormentInformation");
  System.out.println(adUser);
  Mitarbeiter user = call MitarbeiterRepo.findMitarbeiteByAdUser(adUser):null .first;

  if (user == null) { return false; }
  envInf.setUserId(user.KEY_MITARBEITER);
  envInf.setUsername(adUser);
  true;
}

'start' menu definition:
  on trigger "Rechnungen anzeigen" run RechnungsProcess.Rechnungen anzeigen(null, ModeFilter.NotDefined)
    view for page: ListView = RechnungListFc
    view for page: Suchen = DokumentAuswahlSearchFC
  hk: UNDEFINED

  on trigger run ProformaProcess.Proformas anzeigen(null)
    view for page: Suchen = DokumentAuswahlSearchFC
    view for page: ListView = ProfListFc
  hk: UNDEFINED

  on trigger run EDI-Import.EDI Rechnungen erfassen(null, "")
    view for page: Suchen = DokumentAuswahlSearchFC
    view for page: ListView = EdiRechListFc
  hk: UNDEFINED

  on trigger run RechnungsProcess.Buchungspositionen anzeigen(null)
    view for page: ListView = BuchPosListFC
  hk: UNDEFINED
```

Abbildung 54: Applikationskonzept

In einer Applikation können Kommandos mit Hilfe von CommandTrigger gestartet werden. Die CommandTrigger werden im Menü angezeigt. Auch Seperatoren und Sub-Menüs sind verfügbar. So ist bspw. laut Abbildung im Start-Menu ein CommandTrigger, der das Kommando “Rechnungen anzeigen” startet. Für die Pages des Kommandos wurden Formulare angegeben. Jeder Page in einem Kommando muss ein passendes Formular (d.h. mit selbem Inhaltstyp) übergeben werden.

In der Abbildung ist außerdem ein prototypischer Start einer UI-Applikation dargesetellt. Dabei werden Zugänge geprüft und userId / username in der UserEnvironmentInformation gesetzt (bspw. notwendig für Auditable bei Manmap - wird in Zukunft verbessert).

## *Problems and Solutions*

| | |
|---|---|
| java.lang.IllegalStateException: No selection object available for type at .. | Kann auftreten, wenn in einer Klassenhierarchie auf das Parent Objekt abgefragt wird, obwohl das abgeleitete Objekt geladen wurde. |
| Selektionen löschen sich, nach dem Schließen von Eingabefeldern<br><br>oder<br><br>Selektionen löschen sich beim Initialisieren von hierarchischen Anzeigen | Mehrere Tabellen, die den selben Elementtyp laden, haben select first auf true. Wenn das erste Element in diesen Tabellen sich unterscheidet, löschen sich die Selektionen gegenseitig. Man sollte nur bei einer Tabelle select first verwenden. |
| ***Was passiert, wenn ein gesuchtes Scope nicht vom Command zur Verfügung gestellt wird?*** | |
| 81 Pixel als Spaltenbreite in Tabellen | Sollte bei JavaFx 7 nicht eingestellt werden. Führt zu vielen Merkwürdigkeiten. |

## 7. Beispiele zu typischen Abläufen

Bisher wurden vor allem verschiedene Konzepte von MoWare isoliert erklärt. In diesem Abschnitt werden nun typische Abläufe vorgestellt, die in den meisten Geschäftsanwendungen zu finden sind. Wichtig ist, dass sich Anwendungsentwickler möglichst restriktiv an die vorgestellten Muster halten, da die meiste Funktionalitäten von MoWare speziell vor dessen Hintergrund entwickelt wurden. Gerade auch in Zukunft werden diese Muster die ersten sein, deren Funktionieren auf neuen Laufzeitumgebungen sichergestellt wird. Abweichungen können also nicht nur im aktuellen MoWare zu Unwägbarkeiten führen - das kann der Anwendungsentwickler ja noch ständig "ausprobieren" - sondern sie können vor allem in Zukunft einfache Migrationen von Anwendungen verhindern.

### *Das SEARCH_VIEW Kommando*

Im Einführungskapitel wurde bereits der prototypische SEARCH_VIEW Kommando im Detail erklärt. Es dient dem Suchen von Entitäten auf der Datenbank. Es stellt für den Endanwender eine Sammlung von Entitäten zusammen, die einem bestimmten Kriterium entsprechen. Im folgenden wird das Beispiel nicht nochmals dargestellt, allerdings soll es um eine wesentliche Funktionalität erweitert werden.

Dem Endanwender werden auf der ersten Seite des SEARCH_VIEW zwei Datum-Delegates angezeigt (siehe Abbildung 5: Suchformular). Das erste Delegate ist "disabled", das zweite Delegate ist kein Pflichtfeld. Häufig sind Suchkriterien aber optional, d.h. sie müssen nicht vom Anwender ausgefüllt werden.

```
FormContainer FCAuswahlBeleg (AuswahlBelegAnzeige as boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["1*"]
  label: "Auswahl Lieferaviso"
<no command trigger>

DelegateForm DFAuswahlBeleg (AuswahlBelegAnzeige as boundObject loaded with selected: AuswahlBelegAnzeige.<property>)
  col(|): ["-1"]
  label: <no lable>
  LocalDateDelegate f1 setProperty(boundObject.datVon), setEnabled(false) ;
  LocalDateDelegate f2 setProperty(boundObject.datBis), setOptional(true) ;

  <no onLoad>
  <no onStore>
  <no onValidate>
```

Abbildung 55: Kriterium-Formular

```
READONLY list<LieferAviso> findeLieferAviso(LocalDate von, LocalDate bis) {
  list<LieferAviso> result = null;

  result = MapLieferAviso <join> where({~lieferaviso,  =>
      lieferaviso.lieferDatum >= von && optional(lieferaviso.lieferDatum <= bis)  }) ReadOnly;

  result;
}
```

Abbildung 56: Repo-Methode

Gegenüber dem Einführungsbeispiel wird nur an zwei Stellen etwas verändert. Im Formular wird das LocalDateDelegate für datBis mit setOptional(true) versehen. Damit ist das Datumsfeld kein Pflichtfeld mehr, boundObject.datBis kann also auch den Wert null annehmen. Die Repository-Methode "findeLieferAviso" wird ebenfalls angepasst. In der Abfrage mit Hilfe von MapLieferAviso wird das optional() Konzept verwendet. Die "Und" (&&) Verknüpfung und die Bedingung "lieferaviso.lieferDatum <= bis" wird nur dann herangezogen, wenn der Parameter - in diesem Fall "bis" - nicht null annimmt. Das "bis" Lieferdatum wird nur dann in der Suche berücksichtigt, wenn es im Formular angegeben wurde. Sicher lohnt es sich, das Eingangsbeispiel nochmals durchzusehen, nachdem in den Kapiteln zuvor alle wesentlichen Konzepte erläutert wurden.

**Ergänzende Hinweise zum vorgestellten Suchkommando**

In dem Beispiel des Suchkommandos sind einige Aspekte aufgezeigt, die nicht offensichtlich sind. Suchkommandos müssen sich an die vorgestellten Konventionen halten. Insbesondere ist auf folgendes zu achten:

| | |
|---|---|
| Kriterium auf der ersten Seite | Das Suchkommando berücksichtigt auf der ersten Seite ein Kriterium, mit dem die Suche eingeschränkt werden kann. Es obliegt der Verantwortung des Anwendungsentwickler keine große Mengen an Daten zu laden. Das Kriterium befindet sich auf der ersten Seite, die Ergebnisliste auf der Zweiten. |
| Ergebnisliste auf der zweiten Seite, direkt gebunden | Die Ergebnisliste wird direkt vom Page-Init der zweiten Seite zurückgegeben und an das Formular weitergegeben. Evtl. Updates oder Ergänzungen zu der Ergebnisliste mit weiteren Kommandos ist nur möglich, wenn die Ergebnisliste unmittelbar vom Page-Init zurückgeliefert wird. Die Ergebnisliste sollte daher nicht als Liste im Kriterium selbst modelliert werden. |
| Kriterum mit toString() auf dynamischen Page-Header | Die aktuelle Auswahl durch das Suchkriterium wurde mit der toString() Methode zusammengefasst, die als Page-Header über der Ergebnisliste angezeigt wurde. Auf diese Weise erkennt der Benutzer, wie die aktuelle Ergebnisliste eingeschränkt wurde. Entitäten der Ergebnisliste können verändert werden, sodass sie dann nicht mehr dem Suchkriterium entsprechen. Sie verbleiben aber dennoch in der Liste. Ebenso können neue Entitäten durch weitere Kommandos der Ergebnisliste angehängt werden. Auch diese könnten nicht dem Kriterium entsprechen, werden aber dennoch angezeigt. |
| Schnelle Ladefunktion ohne zusätzliche Entitäten (Performance) | Die Repository Methode um die Lieferaviso Entitäten zu laden ist sehr einfach gehalten. Es werden außer dem LieferavisoKopf keine weiteren Entitäten geladen (z.B. Positionsdaten). Dadurch ist der Ladevorgang sehr schnell und effizient. Wollte der Anwender Positionen zu einem Lieferaviso einsehen, müsste er eine Detailansicht für das Lieferaviso ausführen. Alle Entitäten wurden im Suchkommando readonly geladen. |
| Langsame vs. schnelle Operationen | Grundsätzlich sind alle Operationen, die im Arbeitsspeicher ausgeführt werden, als schnell zu charakterisieren. Darunter fallen Berechnungen, Sortierungen etc. Zugriffe auf die Datenbank oder andere Netzwerkkomponenten sind hingegen langsamer und mit Wartezeiten für den Anwender verbunden. Diese Operationen sollten immer im Page-Init ausgeführt werden und nicht im Command-Init. Denn im Command-Init wird keine Sanduhr/Busy Indicator eingeblendet, im Page-Init hingegen schon. |

## *Das GRAPH_OWNER Kommando*

Direkt an das Beispiel des SEARCH_VIEW für Lieferaviso soll das folgende Beispiel für einen GRAPH_OWNER und das nachfolgende Beispiel eines GRAPH_EDIT anschließen.
Folgender Anwendungsfall wird unterstellt:

Schritt 1: Der Endanwender verwendet die Suche, um ein Lieferaviso auszuwählen.
Schritt 2: Anschließend möchte er das Lieferaviso im Detail laden, um
Schritt 3: Daten im Kopf zu ändern.

Um das Lieferaviso im Detail zu laden, ist ein GRAPH_OWNER Kommando zu verwenden.

- Dem Kommando wird der Schlüssel (id) des Aviso übergeben,
- er führt einen checkout unter Verwendung einer Repo-Methode durch,
- lädt also den Kopf und die Positionen und
- bringt diese zur Anzeige.

Nachfolgend wird das Kommando und der Prozess, dem das Kommando angehört, dargestellt. Die Oberfläche sieht wie folgt aus:

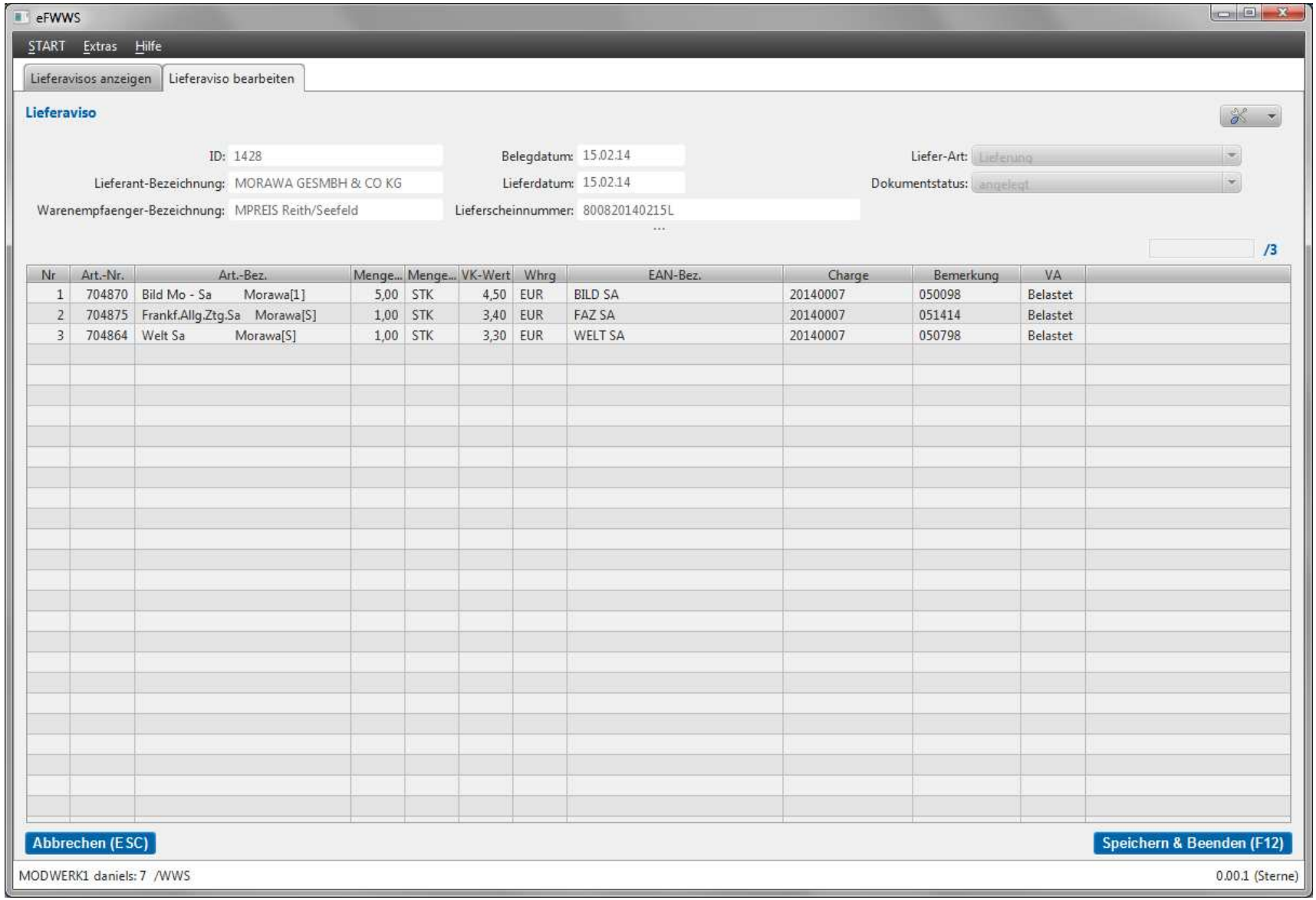


Abbildung 57: Lieferaviso Detailansicht

Der Prozess ist recht einfach gestaltet. Wie zu erkennen, ist das "Lieferaviso bearbeiten" Kommando im Abschnitt "creators and state-independent commands" genannt. Damit kann das Kommando gestartet werden, ohne dass der Prozess Prüfungen vornimmt, d.h. das Prozessdokument muss nicht vorliegen. Es kann dem GRAPH_OWNER als erster Parameter null übergeben werden. Grundsätzlich gilt: GRAPH_OWNER starten eine neue Session und dürfen daher keine Objekte anderer Sessions übernehmen.

```
process 'LieferAvisoProc' using LieferAviso doc process-status-field is dokumentStatus

<docu>


creators and state-independent commands:
// open a session in those commands, command will be available in every state !
on trigger [<cndt>] Lieferaviso bearbeiten -> < >
on trigger [<cndt>] Lieferavisos anzeigen -> < >


states:
state Angelegt: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  on trigger [<cndt>] Lieferaviso bearbeitet -> Bearbeitet
  on exit: <exp>


state Bearbeitet: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  on trigger [<cndt>] Lieferaviso zurücksetzen -> Angelegt
  on exit: <exp>


state Abgeschlossen: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>


state Fehler: [<cndt>]
  <docu>
  on entry: <exp>  // can be called multiple times!
  <transitions>
  on exit: <exp>
```

Abbildung 58: LieferAviso-Prozess

In der folgenden Abbildung ist das Kommando abgebildet. Dem Kommando übergeben wir den Schlüssel (id) als Integer-Wert, so dass das Kommando aufgrund des Schlüssels den checkout durchführen kann. Das Lieferaviso setzen wir sogleich im ersten Schritt  als Prozessdokument.

Das Prozessdokument wird auf die Oberfläche gespielt. Es steht eine Page-Conclusion "Speichern und Beenden" zur Verfügung. Eine "Abbrechen" Page-Conclusion wird automatisch von MoWare hinzugefügt. Sie bricht das Kommando ab und führt die FINAL_CANCEL_CONCLUSION aus (diese wurde aber nicht in der Abbildung dargestellt, da keine Arbeiten durchzuführen sind). Bei der FINAL_OK_CONCLUSION sind sehr wohl Arbeiten durchzuführen:

- Das Prozessdokument wird wieder eingecheckt. Falls Objekte sich im Dirty-Zustand befinden, werden sie auf die Datenbank gespeichert. Die call Anweisung legt eine neue Session Operation an. Sie wird noch nicht ausgeführt.

- Anschließend wird die Kontrolle an den Prozess zurück übergeben. Da nun ein Prozessdokument vorhanden ist, befindet sich gleichzeitig auch der Prozess in einem spezifischen Zustand - in dem des Dokumentes. Erst wird geprüft, ob das Kommando als onTrigger im aktuellen Prozesszustand genannt wird, falls ja, wird zum Zielzustand übergegangen, falls nein, wird im Abschnitt "creators and state-independent commands" nach einem Trigger für das Kommando gesucht und zu dessen Zielzustand übergegangen. Dann wird geprüft, ob eine "auto" Bedingung greift.

- Die Session wird commited, d.h. alle Session-Operationen werden in einer Datenbanktransaktion abgearbeitet.

- Schließlich wird das Lieferaviso doc and das Vaterkommando (der SEARCH_VIEW, von dem aus dieses Kommando gestartet wurde) übergeben. Ist dort die Ergebnisliste aktiv, so wird das Aviso an dessen Selektionskontroller übergeben. Dieser prüft die Wurzelobjekte des angezeigten Objektgraphen. Wird ein Objekt mit selbem Schlüssel gefunden, wird es ausgetauscht. Ansonsten wird es der Liste in der Wurzel angehängt. (Es können grundsätzlich auch mehrere Objekte an Vater-Kommandos übergeben werden. Gegenwärtig implementieren nur SEARCH_VIEW den Austauschmechanismus.)

**Ergänzende Hinweise zum vorgestellten GRAPH_OWNER Kommando**

| | |
|---|---|
| Hinweise zur aktuellen Ansicht im Page-Header | Nicht nur im SEARCH_VIEW, auch in GRAPH_OWNER wird nun eine Zusammenfassung des aktuell angezeigten Dokuments an der Oberfläche angezeigt. Dazu muss man im Kommando bei jeder Seite den Page-Header einstellen. Im dargestellten Fall wurde das verabsäumt. |
| DelegateForms für die Bearbeitung sperren | In GRAPH_OWNER werden die Details zu einem Datenobjekt visualisiert. Alle DelegateForms werden aber in der Ansicht für die Bearbeitung gesperrt. Für die Bearbeitung ist immer zusätzlich ein GRAPH_EDIT mit eigener Ansicht zu verwenden. |

```
command 'Lieferaviso bearbeiten' in process LieferAvisoProc with LieferAviso doc  OR null (set proc doc first)!

command parameter:
int id

local variables:
<< ... >>


<docu>


command settings:
command type: GRAPH_OWNER (new session)
enabled if: <cond>
question on external abort: <msg>
title addon: <msg>
command icon: <no icon>

command init:
func()->void {
  <no statements>
}


command pages:
page 'AusgAviso'
 form bound to list< LieferAviso > as  form

 page init:
   pageLoadFunc()->Object {
     // Checkout Aviso
     doc = call EingangsBelegRepo.checkoutLieferAviso(id) ;
     doc;
   }

 then, calc page title: <title>

 set scopes for page:
   <no scopeConceptFunc>

 page conclusions:
   conclusion 'Speichern & Beenden'  (enabled if: <cond>)
     request 'save data' from page form: save hotkey: SAVE
     func()->void {
       done (run FINAL_OK_CONCLUSION)
     }


FINAL_OK_CONCLUSION:
  func()->void {
    call(add to session operation)EingangsBelegRepo.checkinLieferAviso(doc) ;
  }

  check process, then: COMMIT_SESSION
  notification: <msg>
  selection(s)/update(s) on parent: doc
```

Abbildung 59: Lieferaviso bearbeiten

Das Formular für die Detailansicht ist nachfolgend gezeigt. Das DelegateForm lädt das aktuell selektierte LieferAviso, welches als einziges Objekt von der Page-Init des Kommandos zurückgeliefert und vom Selektionskontroller sogleich automatisch selektiert wurde. Liefert ein Page-Init nur ein Objekt, wird es automatisch vom Kontroller auch selektiert. Das TableForm ist an die Positionen des selektierten LieferAviso gebunden (LieferAviso.pos). Im FormContainer sind zwei weitere Kommandos als Trigger eingebunden, die der Endanwender von diesem Formular aus starten kann.

```
FormContainer LADetail (LieferAviso as boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["-1", "1*"]
  label: "Lieferaviso"
MenuButton " "  (image: <no imageString> )
  on trigger run LieferAvisoProc.Lieferaviso bearbeitet(getSelected(LieferAviso.class))
    <views>
  hk: UNDEFINED

  on trigger run LieferAvisoProc.Lieferaviso zurücksetzen(getSelected(LieferAviso.class))
    <views>
  hk: UNDEFINED



DelegateForm LaDetailDF (LieferAviso as boundObject loaded with selected: LieferAviso.<property>)
  col(|): ["1*", "1*", "1*"]
  label: <no lable>
  IntegerDelegate f1 setProperty(boundObject.id), setEnabled(false) ;
  LocalDateDelegate f5 setProperty(boundObject.ausstellDatum), setEnabled(false) ;
  StatusDelegate f8 setProperty(boundObject.lieferArt), setEnabled(false) ;
  StringDelegate f2 setProperty(boundObject.lieferant.bezeichnung), setEnabled(false) ;
  LocalDateDelegate f6 setProperty(boundObject.lieferDatum), setEnabled(false) ;
  StatusDelegate f9 setProperty(boundObject.dokumentStatus), setEnabled(false) ;
  StringDelegate f4 setProperty(boundObject.warenEmpfaenger.bezeichnung), setEnabled(false) ;
  StringDelegate f7 setProperty(boundObject.lsNummer), setEnabled(false) ;

  <no onLoad>
  <no onStore>
  <no onValidate>

TableForm LaPos (list<LieferAvisoPos> with boundObject loaded with selected: LieferAviso.pos)
  label: <no heading>
  select first: false
  <table summary line>

  <no command trigger>

  setProperty(boundObject.id) setWidth(0) <overwrite label> <overwrite format> ;
  setProperty(boundObject.posNum) setWidth(40) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artikel.id) setWidth(60) <overwrite label> <overwrite format> ;
  setProperty(boundObject.artikel.bezeichnung) setWidth(200) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferMenge.betrag) setWidth(50) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferMenge.txtEh) setWidth(50) <overwrite label> <overwrite format> ;
  setProperty(boundObject.verkaufsWert.betrag) setWidth(50) <overwrite label> <overwrite format> ;
  setProperty(boundObject.verkaufsWert.einheit) setWidth(50) "Whrg" <overwrite format> ;
  setProperty(boundObject.eanBezeichnung) setWidth(200) <overwrite label> <overwrite format> ;
  setProperty(boundObject.charge) setWidth(120) <overwrite label> <overwrite format> ;
  setProperty(boundObject.bemerkung) setWidth(100) <overwrite label> <overwrite format> ;
  setProperty(boundObject.verrechungsArt) setWidth(60) <overwrite label> <overwrite format> ;
```

Abbildung 60: Formular für Lieferaviso

Doch wie wird das Kommando "LieferAvsio bearbeiten" gestartet? Hier lohnt sich nochmals einen Blick auf das Formular mit der Ergebnisliste des SEARCH_VIEW Kommandos zu werfen.  In der Tabelle mit der Ergebnisliste wurde ein "on trigger" eingefügt, der das Kommando startet. Parametrisiert ist es mit null und der id des aktuell selektierten Aviso. Ein HotKey "Enter" wurde angegeben, der auch einem Doppelklick mit der Maus entspricht.

```
FormContainer FCLieferAvisoListView (list<LieferAviso> with boundObject loaded with selected: <class>.<property>)
  cols(|): ["1*"] rows(-) ["1*"]
  label: <no heading>
<no command trigger>

TableForm TFAvisoKoepfe (list<LieferAviso> with boundObject loaded with selected: <class>.<property>)
  label: <no heading>
  select first: false
  <table summary line>

  MenuButton " "  (image: <no imageString> )
    on trigger run LieferAvisoProc.Lieferaviso bearbeiten(null, getSelected(LieferAviso.class).id)
      view for page: AusgAviso = LADetail
    hk: ENTER



  setProperty(boundObject.id) setWidth(60) <overwrite label> <overwrite format> ;
  setProperty(boundObject.ausstellDatum) setWidth(80) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferant.partei#Key) setWidth(80) "Lief-Nr." <overwrite format> ;
  setProperty(boundObject.lieferant.bezeichnung) setWidth(200) "Lieferant" <overwrite format> ;
  setProperty(boundObject.lsNummer) setWidth(120) <overwrite label> <overwrite format> ;
  setProperty(boundObject.lieferArt) setWidth(80) <overwrite label> <overwrite format> ;
  setProperty(boundObject.dokumentStatus) setWidth(80) <overwrite label> <overwrite format> ;
```

Abbildung 61: Ergebnisformular

## *Das GRAPH_EDIT Kommando*

Die GRAPH_EDIT Kommandos sind die Einfachsten von allen Kommandos. Sie übernehmen die Session des Vater-Kommandos und verändern lediglich Objekte im Graphen. In diesem Beispiel soll die Entität LieferAvsio bearbeitet werden können. Dazu ist immer ein GRAPH_EDIT Kommando zu verwenden, da im GRAPH_OWNER die DelegateForms immer disabled werden. Der Endandwender soll dort nicht direkt in den Delegates-Feldern bearbeiten können.

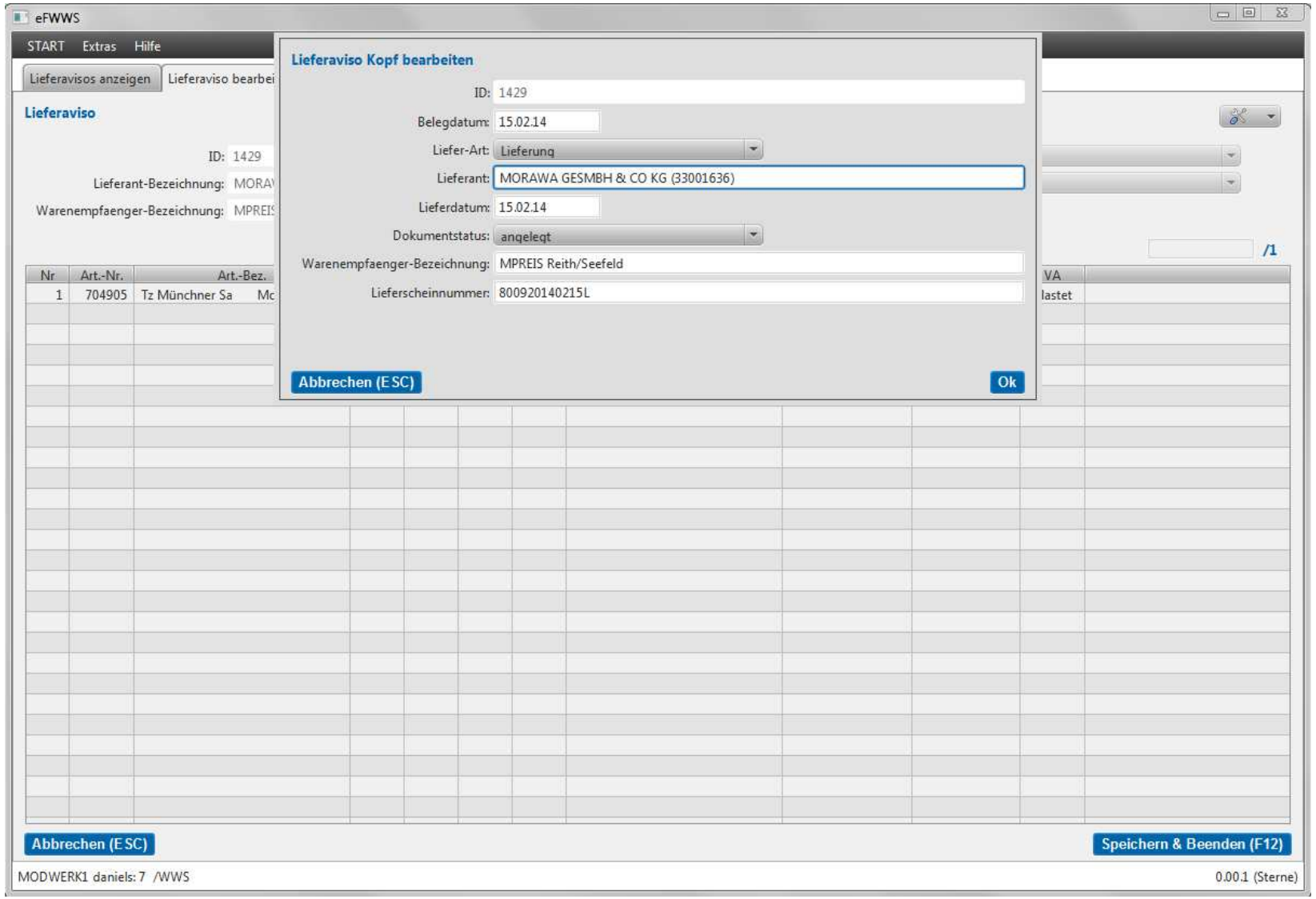


Abbildung 62: Lieferaviso bearbeiten

```
command 'Lieferaviso Kopf bearbeiten' in process LieferAvisoProc with LieferAviso doc

command parameter:
<< ... >>

local variables:
<< ... >>


<docu>


command settings:
command type: GRAPH_EDIT
enabled if: <cond>
question on external abort: <msg>
title addon: <msg>
command icon: <no icon>

command init:
<func>


command pages:
page 'Kopf Editor'
  form bound to list< LieferAviso > as  form

  page init:
    pageLoadFunc()->Object {
      doc;
    }

  then, calc page title: <title>

  set scopes for page:
    pageSetScopesFunc()->void {
      doc.lieferant#Meta.setScope(call ParteienRepo.findeFFWSLieferanten() );
    }

  page conclusions:
    conclusion 'Ok'  (enabled if: <cond>)
      request 'save data' from page form: save hotkey: NONE
      func()->void {
        done (run FINAL_OK_CONCLUSION)
      }


FINAL_OK_CONCLUSION:
  func()->void {
    <no statements>
  }

  check process, then: DO_NOT_COMMIT_SESSION
  notification: <msg>
  selection(s)/update(s) on parent: doc
```

Abbildung 63: Kopf bearbeiten

Da das Prozessdokument selbst in diesem Beispiel editiert werden soll, verfügt das Kommando nur über einen Parameter (das Prozessdokument muss ohnedies immer angegeben werden). Weder Funktionen für die Command-Init() noch für die FINAL_OK_CONCLUSION sind notwendig. Lediglich eine Page "Kopf Editor" wird eingefügt. Diese kopiert die zu editierende Entiät doc auf die Oberfläche (in der Page-Init()). Eine Page-Conclusion "Ok" beendet das Kommando. Die Daten aus der Oberfläche werden zuvor in die Entität zurückkopiert ("request save data from page form").

Neu ist die Berechnung eines Scopes in der dargestellten Seite. Die Entität LieferAviso verfügt über eine Referenz auf einen Lieferanten. Referenzen werden in der Oberfläche grundsätzlich als autocomplete Drop-Down-Felder dargestellt. Nun stellt sich nur mehr die Frage, welche Auswahlmöglichkeiten im Drop-Down darzustellen sind. Diese Auswahlmöglichkeiten für Referenzen werden durch Scopes berechnet. Die Möglichkeiten werden wiederum in der Meta-Information von Referenzen hinterlegt. Das Referenz-Delegate fordert in diesem Beispiel automatisch über die Meta-Information die Liste von möglichen Lieferanten ab.

Die Session wird selbstverständlich nicht commited. Sie muss mit "Speichern und Beenden" im GRAPH_OWNER Vater-Kommando commited werden. Gleichzeitig bedeutet das auch, dass die Änderungen am LieferAviso nicht sofort auf die Datenbank gespeichert werden.  Erst wenn alle Änderungen abgeschlossen sind, wird der GRAPH_OWNER durch den Endanwender beendet.

## *Bearbeiten von Belegen (prototypisch)*

Zusammenfassend sind hier nochmals Abläufe zum Bearbeiten von Belegen erklärt. In einer Such-Maske kann der Benutzer nach bestimmten Kriterien suchen (Page1), dann wird im das Ergebnis der Suche (verschiedene Aviso) auf einer Page2 angezeigt. Offensichtlich handelt es sich dabei um ein SEARCH_VIEW, bei dem keines der Objekte ausgecheckt wird (nur readonly geladen). Die Session kann nicht commited werden. In der Ergebnisliste kann der Benutzer eine Entität zum Bearbeiten wählen, ein GRAPH_OWNER Kommando wird dadurch gestartet, welches erst das gewählte Objekt über dessen ID auscheckt.

- GRAPH_OWNER startet eine neue Session,
- Entität wird über ID ausgecheckt,
- checkin der Entität wird als Session-Operation in der FINAL_OK_CONCLUSION hinzugefügt,
- Benutzer kann über Formular Entität bearbeiten (Page1)
- Benutzer kann Kommando abbrechen oder
- mit einer Save Page-Conclusion "Speichern & Beenden" das  Kommando beenden
- das Kommando führt eine COMMIT_SESSION in der FINAL_OK_CONCLUSION aus
- Das Objekt in der Ergebnisliste des SEARCH_VIEW wird ersetzt

Im folgenden Beispiel sind die drei Kommandos beteiligt, die oben bereits im Detail erläutert wurden. Ein SEARCH_VIEW, ein GRAPH_OWNER und ein GRAPH_EDIT. Üblicherweise wird dieses Pattern verwendet, um einen Beleg oder ein Dokument zu bearbeiten. Der Ablauf ist wie folgt:

- Über einen SEARCH_VIEW kann der Benutzer nach dem Lieferaviso suchen
- Er kann auf Page 2 des SEARCH_VIEW eine Entität/Aviso zum Bearbeiten markieren
- GRAPH_OWNER startet eine neue Session
- Entität wird über ID ausgecheckt, vollständiger Objektgraph wird geladen (Aviso + Positionen)
- checkin der Entität wird als Session-Operation in der FINAL_OK_CONCLUSION hinzugefügt,
- Benutzer kann über Formulare Daten der Entität ansehen (Page 1). Alle Felder/Delegates sind ReadOnly, so dass die Daten nicht bearbeitet werden können.
- Benutzer kann mit einem Kommando “Kopfdaten Aviso ändern” das Aviso bearbeiten. Dabei wird ein GRAPH_OWNER Kommando gestartet (keine neue Session, Entität wird direkt als Parameter übergeben oder ist - wie in diesem Fall - das Prozessdokument selbst)
- Benutzer bearbeitet und schließt den GRAPH_EDIT mit "Ok" Page-Conclusion ("request save from form" sollte nicht vergessen werden) .
- Es wird automatisch ein Reload der Ansicht des GRAPH_OWNER durchgeführt, da der Objektgraph verändert wurde.
- Benutzer kann GRAPH_OWNER Kommando abbrechen oder
- mit einer Save Page-Conclusion "Speichern & Beenden" das GRAPH_OWNER Kommando beenden
- das Kommando führt eine COMMIT_SESSION in der FINAL_OK_CONCLUSION aus
- Das gerade editierte Objekt Aviso wird in der Ergebnisliste ausgetauscht und ist daher auf dem aktuellen Stand

## 8. Drucken von Objekten

MoWare bietet auch Unterstützung zum Drucken von Entitäten, ValueObjects und ViewObjects. Es stehen sogar zwei verschiedene Tools zur Verfügung. In der Vergangenheit wurde vor allem FopLand (org.modellwerkstatt.FopLand) verwendet. FopLand ist eine sehr einfache Sprache, die stark an Apache XSL-FO angelegt ist. Es lassen sich damit einfach druckbare Dokumente erzeugen. Für die Sprache selbst liegt keine Dokumentation vor. Das Code-Completion Menü von MPS sowie die Apache FO Dokumentation geben Anleitung.

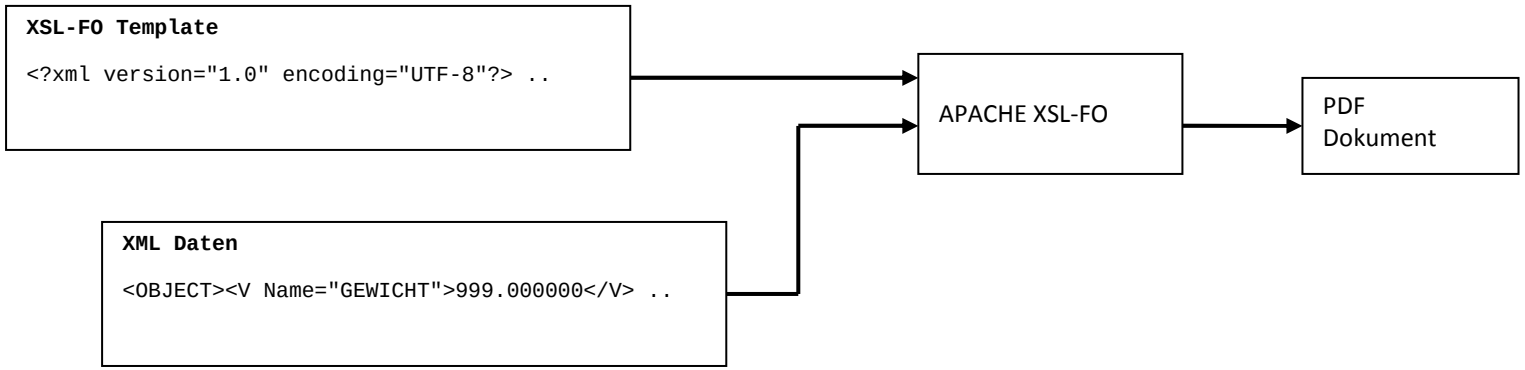



Abbildung 64: XSL-FO

Mit der FopLand-Sprache wird ein XSL-FO Template generiert. Darin enthalten sind alle Formatierungs- und Layoutvorschriften. Allerdings sind keine Daten enthalten. Daten werden aus Entitäten, ViewObjects oder ValueObjects gewonnen, indem sie zu XML serialisiert werden. Anschließend kombiniert die APACHE XSL-FO Runtime zur Laufzeit diese beiden Artifakte und erzeugt daraus eine PDF-Datei.

MoWare führt die Serialisierung von Objekten in XML durch. Dazu steht die Methode .toXml() zur Verfügung. Serialisiert wird dann nicht nur eine Instanz, sondern auch alle Kindelemente, d.h. der komplette Objektgraph wird als Zeichenkette abgebildet. Häufig wurde über die Kapitel hinweg die Rechnung als Beispiel für eine komplexere Datenstruktur verwendet. Wird bei einer Rechnungsinstanz "rechnung" dann bspw. rechnung.toXml() aufgerufen, so werden auch die Positionsdaten wie zuAbPositionen oder rechnungsPositionen serialisiert. Alle Properties, VirtualProperties (int, string, status, LocalDate, DateTime, BigDecimal) sowie auch Referenzen sind in der XML Zeichenkette enthalten.

Der Anwendungsentwickler muss darauf achten, Rückwärtsreferenzen mit OPPOSITE zu markieren. Rückwärts-Referenzen ermöglichen von einem Kindelement zum Vaterelement zu traversieren, also bspw. von einer Belegposition zum Belegkopf. Enthält der Belegkopf gleichzeitig eine Liste von Belegpositionen, ergibt sich ein zirkulärer Schluss. Während der Serialisierung kommt es zu einer Endlosrekursion und zu einem StackOverflow. OPPOSITE unterbricht die Rekursion.

Mit dem XFO Designer hat MoWare auch einen WYSIWYG Editor integriert. Mit dem Designer können XSL-FO Templates direkt editiert werden. Eine eigene Dokumentation ist für dieses Produkt verfügbar.

Die XSL-FO Laufzeitumgebung ist in MoWare integriert. Als Utility-Klasse soll der Anwendungsentwickler auf MUPrint zurückgreifen. Sie stellt zwei Methoden zur Verfügung, die XML Daten und XSL-FO Template zusammenführen und dann das resultierende PDF in einen FileOutputStream schreiben. In der nachfolgenden Abbildung ist ein einfaches Beispiel dargestellt.

```
File pdf = File.createTempFile("rechnung", ".pdf");
string xml = rechnung.toXML();

InputStream xslt = Rechnung.class.getResourceAsStream("rechnung.xslt");

MUPrint.printXFODesignerTemplate(xml, xslt, new FileOutputStream(pdf), null);
```

Abbildung 65: MUPrint Utility

Das Xslt Template wird über die Rechnung.class geladen. Die Java-VM sucht das Xslt Dokument damit an der Position im Filesystem, an der sich auch die Rechnung.class - also die Entität - befindet. Befindet sie die Java Klasse in einem Jar-Archiv, so wird auch das Sslt von dort geladen.

Der XFO Designer unterstützt Barcodes nicht direkt. MoWare kann allerdings Barcodes vor dem XSL-FO Prozessing einfügen. Dazu muss im Template an Position der Barcodes einfach

BARCODE4J(/OBJECT/NRBELEG)

vermerkt werden. /OBJECT/NRBELEG ist der Pfad zu dem Property, in dem die Barcodenummer enthalten ist: OBJECT ist demnach die Rechnung, NRBELEG der Name des Propertys in der Rechnung. Die ganze Zeichenkette wurde Uppercase konvertiert.

## 9. Batch-Jobs auf dem Server

MoWare unterstützt zwei verschiedene Formen von Programmen. Programme mit Benutzeroberflächen, kurzum, ERP ähnliche Desktop-Applikationen, und BatchJobs. BatchJobs sind zyklische Programme, die auf einem Server ausgeführt werden und über keine Oberfläche verfügen. Häufig laden sie eine Liste von Belegen aus einer Datenbank, verarbeiten diese und schreiben das Ergebnis wieder zurück. Batchjobs werden auch zum Importieren von Daten verwendet, bspw. dem Lesen vom Filesystem und Ablegen in der Datenbank.

Mit MoWare lassen sich derartige Aufgaben mit dem Konzept BatchJob unabhängig von der Laufzeitumgebung formulieren. Ziel ist es, die Jobs in Form von zyklischen Aufgaben mit Hilfe der bekannten Kommandos zu erstellen. Unerheblich soll dabei sein, in welchem Umfeld die BatchJobs ausgeführt werden (z.B. einfache JVM, Application-Server, etc.) Die Hauptelemente eines Batchjobs sind Tasks. Tasks werden parallel ausgeführt - jeweils ein Task entspricht einem Thread. Damit können Batchjobs bereits auf einfachste Weise parallelisieren. Es ist allerdings darauf zu achten, dass Sessions keinesfalls zwischen Tasks geteilt werden. Jeder Task sollte mit eigener Sessions an seiner UnitOfWork arbeiten.

Der Entwickler kann einen Task grundsätzlich zyklisch oder zu einem bestimmten Zeitpunkt ausführen lassen, d.h., jeder Task muss einzeln konfiguriert werden. Dazu wird die Cron-Expression Syntax aus Unix übernommen (http://en.wikipedia.org/wiki/Cron). Neben einer Cron-Expression kann einem Task zusätzlich ein "additional idle timeout" angegeben werden, welches der Task nach Abarbeitung (eines Zyklus!) verstreichen lässt.

Eine Cron-Expression ist recht mächtig und kann verschiedenste Zyklen oder Startzeitpunkte für Tasks ausdrücken. Sie besteht im Moware-Stack aus 6 Feldern, die folgende Bedeutung haben:

```
task “mytask1” {
*   *   *   *   *   *
┬   ┬   ┬   ┬   ┬   ┬
│   │   │   │   │   │
│   │   │   │   │   └──── day of week (0 - 7) (0 or 7 are Sunday, or use names)
│   │   │   │   └──────── month (1 - 12)
│   │   │   └────────────── day of month (1 - 31)
│   │   └──────────────────── hour (0 - 23)
│   └────────────────────────── min (0 - 59)
└──────────────────────────────── sec (0 - 59)
```

| Cron-Expression | Beschreibung |
|---|---|
| * * * * * * | Setzt man die Cron-Expression auf * * * * * *, so bedeutet das: egal welche Sekunde, welche Minute etc. der Task wird erneut gestartet, wenn er beendet wurde (nach einem Zyklus). |
| 00 * * * * * | Setzt man die Expression auf 00 * * * * * , so wird der Task hingegen nur bei Sekunde = 00 neu gestartet (Minute, Stunde usw. belanglos). |
| 00 00 03 * * * | Ein Task soll immer um 3 Uhr in der Früh gestartet werden |
| * * * * * * | Ein weiterer Fall: Der Task soll ständig ausgeführt werden, wenn allerdings nichts abzuarbeiten ist, soll er 10 Sekunden pausieren. Dazu kann die Cron-Expression auf * * * * * gesetzt werden und das "additional time-out" auf 10000ms. |

Aufgrund der cron Unterstützung prüft jeder Batchjob beim Starten erst, ob die Uhrzeit des Servers, auf dem die Java Virtual-Machine ausgeführt wird, der Uhrzeit des verbundenen SQL-Servers entspricht (Toleranz 5min.)

```
Batchjob BerichtLoaderApp


configuration for this test: BerichtLoaderAtLola
on startup: {

  statusBerichtLoader = "Berichtloader erfolgreich intitialisiert => Warten auf ersten Durchlauf";

  // Fuer Testzwecke wird anzTage hier gesetzt
  /*
  anzTage = "200";
  */
  log DEBUG : "Startup Ende mit Mode: " + actMode + ", Berichtmode: " + actModeBerichte + ", OPMode: " + actModeOP
}
on shutdown: {
  log ERROR : "Der Berichtloader wurde beendet"
}


properties and static methods of Batchjob:

  MONITORING_VALUE private string actModeOP = <default>;  // Mode offene Posten verbuchen
  MONITORING_VALUE private string actModeBerichte = <default>;  // Mode Berichte Erstellen
  MONITORING_VALUE private string actMode = <default>;  // Mode ist
  MONITORING_VALUE private string statusBerichtLoader = <default>;  // Status Berichte erstellen
  MONITORING_VALUE private string version = "2013-12-19";  // Versionsnummer als Datum
  CONFIG_VALUE private string anzTage = "200";  // Anzahl Tage zurueck
  CONFIG_VALUE private string modeOP = <default>;  // KassaBuchbons aus XML-Files erzeugen (Active|NotActive)
  CONFIG_VALUE private string modeBerichte = <default>;  // KassaBons aus XML-Files erzeugen (Active|NotActive)
  CONFIG_VALUE private string mode = <default>;  // Setzt Mode: (Produktion)
```

Abbildung 66: BatchJob Einstellungen

Wie in der nachfolgenden Abbildung ersichtlich, benötigen BatchJobs - analog den Desktop-Applikationen - eine Konfiguration. Ferner können startup() und shutdown() Funktionen angegeben werden. Zudem lassen sich in statischen Methoden Hilfsfunktionen abbilden. Jeder BatchJob kann über zahlreiche Properties verfügen. Unterschieden werden die Properties wie folgt:

| **Property** | **Erläuterung** |
|---|---|
| CONFIG_VALUE | Das Property wird als Konfiguration verstanden (nur Type string möglich). Beim Startup des BatchJobs wird dann aus dem objectflow.properties File der Wert von <<BatchJobName>>.<<PropertyName>> gelesen und in der Property abgelegt. |
| MONITORING_VALUE | Das Property dient als Monitoring Wert und wird über JMX exportiert. Die Beschreibung (// Kommentar) sollte ausgefüllt werden, da diese in JMX angezeigt wird. Werden Int oder Long als Typ verwendet, können diese mit Hilfe von Grafiken in JMX sogar visualisiert werden. Bei Objekten wird toString() aufgerufen, dadurch ist keine Visualisierung möglich.<br><br>Für Überwachungswerte wurden speziell zwei Klassen angedacht. Die ObjectflowRT Measure Klasse bietet die Möglichkeit Zykluszeiten zu messen. |

Damit kann ein Task oder auch eine einzelne Codestelle geprüft werden. Die Klasse Counter verfügt über einen ok und über ein failure Zähler. Beiden Klassen kann ein Name des Messwerts übergeben werden. Häufig übergibt der Anwendungsentwickler Instanzen dieser Klassen einem Kommando, um Messungen durchzuführen.

PRIVATE_VALUE — Private entspricht einem Wert, der verwendet werden kann, um Information zwischen Task's auszutauschen. Der Wert ist aber nicht synchronisiert, sondern einfach volatile.

```
tasks of Batchjob:

task 'Bericht Loader' {
  cron setting: second * , minute * , hour * , day of month * , month * , day of week *
  additional idle timeout: 100000 ms

  // Job wird alle halben Stunden ausgeführt
  DateTime actTime = new_DateTimeFromServer();
  IOFXSession searchSession = StandAloneApplicationFactory.getNewManMapSession();

  run Filialabrechnung.KassaBonUebersicht anzeigen(null, Integer.parseInt(anzTage)) with searchSession
    fake ui input at page: Standard
     fakeUi(boundObjects)->void {


       foreach kBU in boundObjects {
         log DEBUG : "Fil " + kBU.filialNummer + ", Lade " + kBU.geldlade + ", Datum " +
             kBU.datum.toString("dd.MM.yy") + "  => " + kBU.anzahl + " Bons"

         IOFXSession bucherSession = StandAloneApplicationFactory.getNewManMapSession();
         run Ladenabrechnung.Ladenbons verbuchen(null, kBU.filialNummer, kBU.geldlade, kBU.datum) with bucherSession
           <views>


       }

     }
     and conclude with Done

  log DEBUG : "=> Berichte erstellen abgeschlossen"
```

Abbildung 67: BatchJob Task

In der obigen Abbildung ist für Illustrationszwecke eine einfacher Task dargstellt. Er greift auf zwei Kommandos zurück, um Ladenbons zu verbuchen. Dazu wird im Task erst eine Session für das Suchkommando erzeugt, d.h. das Kommando "KassaBonUebersicht anzeigen" ist vom Typ SEARCH_VIEW. Dem run wird zugleich die dafür vorgesehende Session "searchSession" übergeben. Anstatt eines Formulars für Benutzeroberflächen wurde ein fake ui für die Page "Standard" des Kommandos angegeben. In eine foreach Schleife wird für jeden KassaBon aus der Liste der boundObjects, also dem Suchergebnis, das Verbuchen-Kommando aufgerufen (Typ GRAPH_OWNER, eigene Session

bucherSession, offensichtlich keine Page - der Code steht im Command-Init() oder in der FINAL_OK_CONCLUSION). Das Kommando wird in der FINAL_OK_CONCLUSION die Session "commiten". Dadurch muss bei der nächsten Iteration eine neue erzeugt werden.

Bricht ein "Ladenbons verbuchen" mit cancel ab - es wird also mit einer FINAL_CANCEL_CONCLUSION beendet - so ist in diesem Beispiel die fake-ui Seite davon nicht betroffen. Es wird einfach die nächste Iteration in der foreach-Schleife ausgeführt, schließlich liegt auch kein Fehlerzustand vor. Lediglich das "Ladenbons verbuchen" wurde intern abgebrochen - was völlig legitim ist. Tritt allerdings eine Exception auf und die FINAL_EXCEPTION_CONCLUSION wird aufgerufen, so wird auch das äußere Suchkommando abgebrochen (ebenfall mit der FINAL_EXCEPTION_CONCLUSION).

Sind alle KassaBon verbucht, wird das Suchkommando mit der Page-Conclusion "Done" beendet.

BatchJobs behandeln eigenständig die nachfolgenden Exceptions. Dabei wird zwar der Task mit Exception abgebrochen, allerdings wird er dann erneut gestartet.

| **Exception** | **Behandlung** |
|---|---|
| InterruptedException | Der aktuelle Task wurde unterbrochen: Task abbruch, nächster start des Tasks sofort. |
| BadSqlGrammarException | Abbruch des BatchJobs. Bei Fehler ORA-02049 "timeout: distributed transaction waiting for lock tips", 5 Minuten Pause, dann Task erneut starten. |
| TransactionException | Fehler während Transaktion, 5 Minuten Pause, dann Task erneut starten. |
| DataAccessResourceFailureException | Datenbank Verbindung verloren. 5 Minuten Pause, dann Task erneut starten. Der Connection-Pool wird hoffentlich eine neue Verbindung in der Zwischenzeit organisiert haben. |
| DeadlockLoserDataAccessException | "Generic exception thrown when the current process was a deadlock loser, and its transaction rolled back". Mehrfache Zugriffe auf gelockte Ressourcen in Oracle mit Timeout. 5 Minuten Pause, dann Task erneut starten. |
| Exception | Abbruch des BatchJobs |

Im Gegensatz zu User-Interface Applikationen ist bei BatchJobs der Connection-Pool nicht auf 1, sondern auf die Anzahl der verwendeten Tasks einzustellen (in der Configuration). Tasks führen üblicherweise viele Transaktionen in kurzer Zeit aus. Mehrere Datenbankverbindungen sind daher notwendig, so dass sich Tasks nicht gegenseitig behindern, d.h., sich gegenseitig auf Datenbankverbindungen warten lassen.

## *Ergänzung zu MoWare 3, RC30*

Mit dem Releasecandidate 30 von MoWare auf MPS3 wurde das Konzept vom BatchJob angepasst. Es können mit der neuen Version sehr einfach mehrere Instanzen eines Tasks gebildet und parallel ausgeführt werden. Wie in der nachfolgenden Abbildung zu erkennen, kann beim Task die Zahl der Instanzen konfiguriert werden. Bei der execute() Methode ist dann die Zahl der Instanzen pro Task insgesamt (numberOfInstances) und die aktuelle Instanznummer selbst (instanceNumber) als Parameter verfügbar. Anhand von der Instanznummer kann Arbeit aufgeteilt werden, bspw. durch die Bildung von Nummernkreisen, wobei jede Instanz eine Submenge davon bearbeitet.

Bei Konfigurationen muss die "max-pool size" auf die Gesamtanzahl aller Instanzen gesetzt werden. Es werden dann vom Connectionpool entsprechend viele Verbindungen zur Datenbank zur Verfügung gestellt.

```
task 'Bank Barlosung verbuchen'
  cron setting: second * , minute * , hour * , day of month * , month * , day of week *
  additional idle timeout: 1000000 ms
  task instances to create: 1

  execute(instanceNumber, numberOfInstances)->void {


    IOFXSession searchSession = StandAloneApplicationFactory.getNewManMapSession();
    // 1. Schritt: Jede noch nicht verbuchte Bankzahlung einem Bankbericht zuordnen,
    // einen Vorgang mit 2 Buchungen erstellen
    // versuchen eine Zuordnung zu finden und wenn es noch keine gibt, eine neue erstellen
    run Bankabrechnung.Offene Bankeingaenge anzeigen(null, TypZuordnung.BarlosungEinzahlung) with searchSession
      fake ui input at page: Standard
       fakeUi(boundObjects)->void {
         foreach kassaBonUebersicht in boundObjects {
           IOFXSession session = StandAloneApplicationFactory.getNewManMapSession();
```

Abbildung 68: Task

## 10. Tools

MoWare stellt neben den domänenspezifischen Sprachen und Laufzeitumgebungen auch verschiedene Tools in der MPS Entwicklungsumgebung zur Verfügung. Dabei handelt es sich um Scripts oder Plugins. Je nachdem können Entwickler entweder über das Menü Tools->Scripts oder direkt über Tools darauf zugreifen.

### *Importieren von Testdaten*

Sehr einfach können Daten aus einer SQL-Tabelle oder aus Excel in Modelle importiert werden. Dazu einfach im gewünschten Modell ein Konzept "TestData" anlegen und unter Tools->"Parse Data from CSV" wählen.

In der nächsten Abbildung ist der Import-Dialog abgebildet. In der ersten Zeile des Textes ist die Ziel-Entität für die Daten genannt, anschließend die Überschrift (meist Property-Bezeichnungen, in diesem Beispiel aber Tabellenspaltenüberschriften) und anschließend die Daten selbst. Werden Tabellenspaltenüberschriften verwendet, so wird über mögliche Mappings das entsprechende Property aufgelöst. Der Entwickler kann auf die Testdaten über deren Namen zugreifen. Sie bilden eine Liste der Entitäten.

Die zu importierende Tabelle muss aus mindestens 3 Spalten bestehen. Eine oder zwei Spalten werden vom Importer ignoriert.

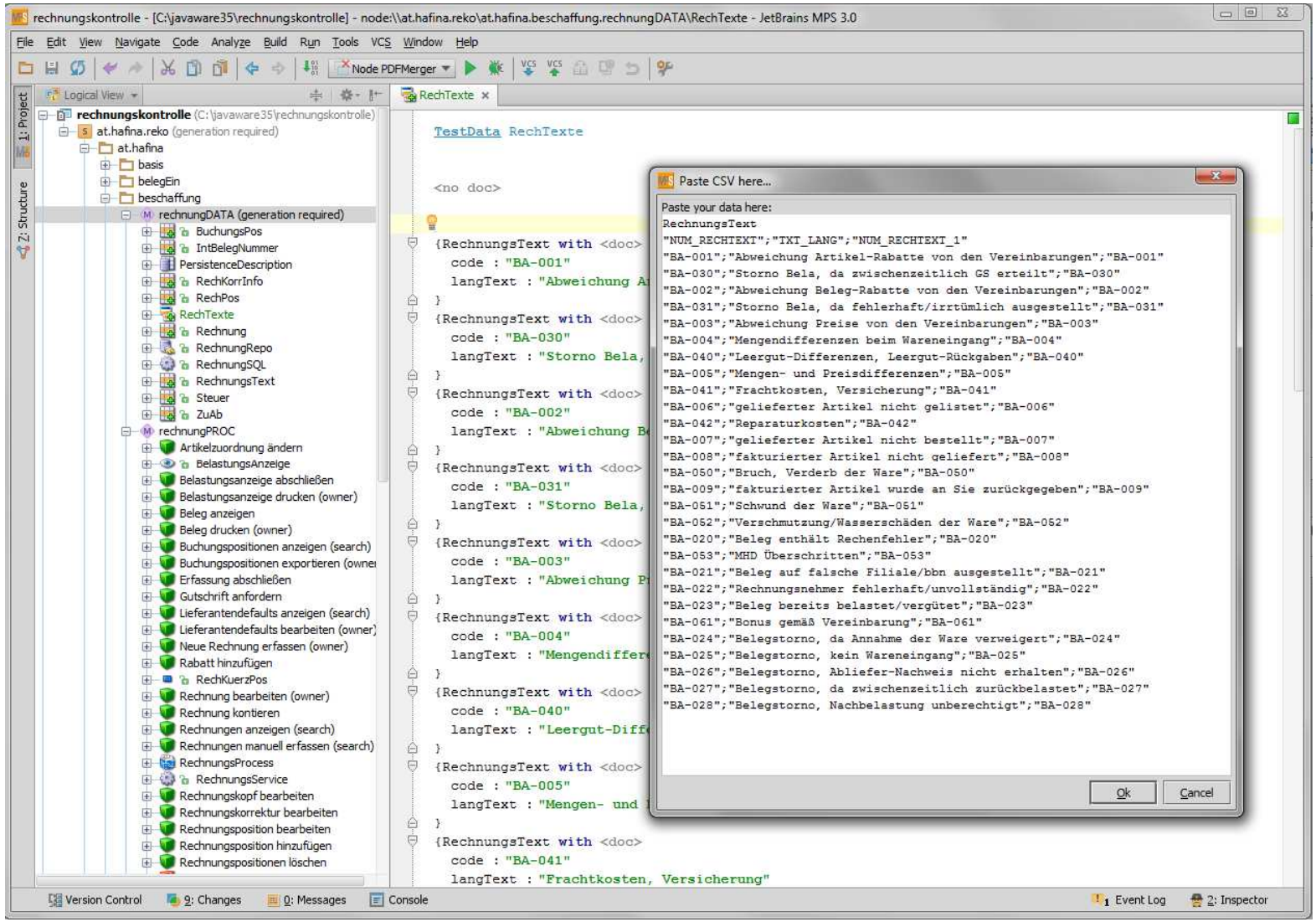


Abbildung 69: Datenimport

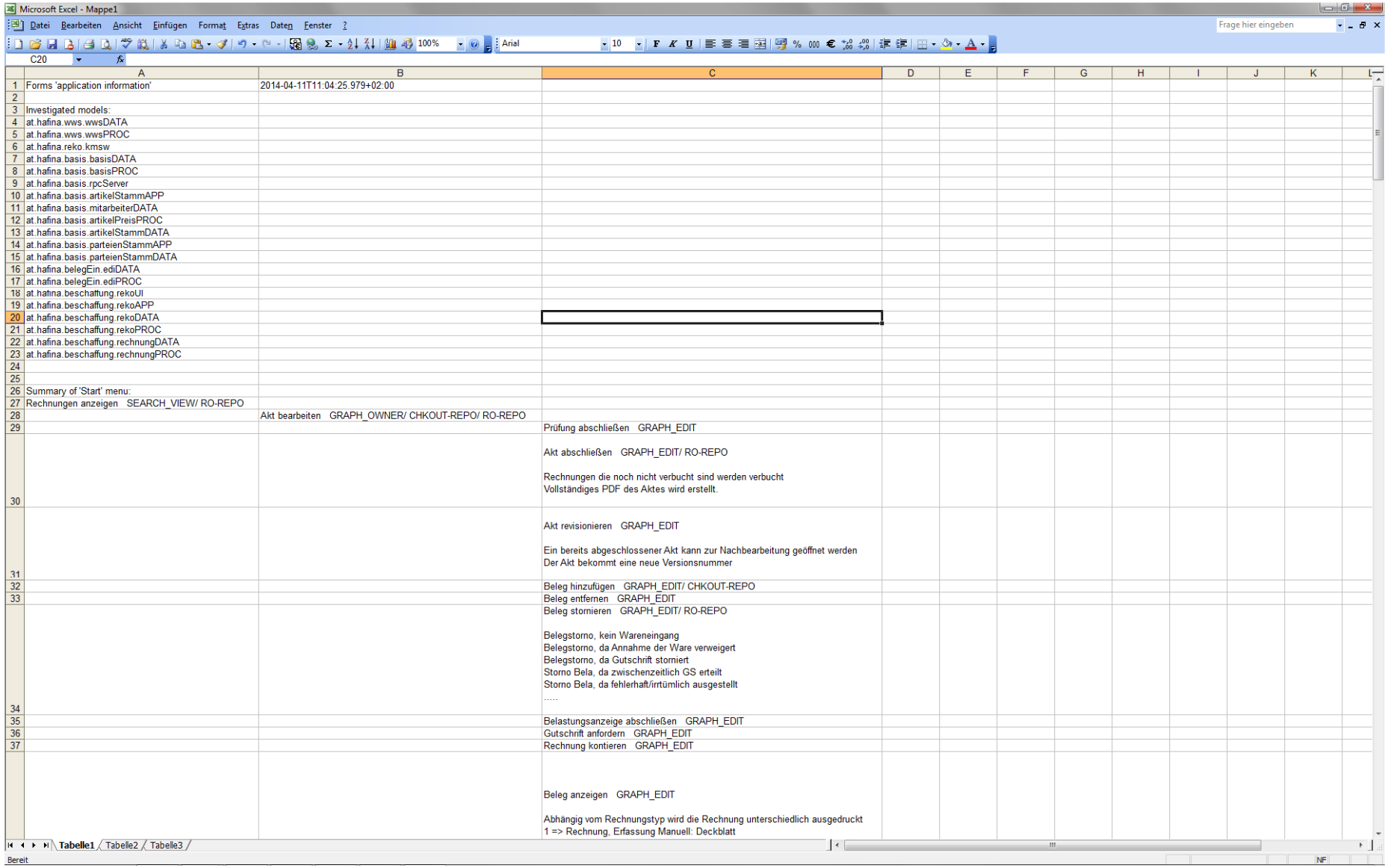
## *Erzeugen von Dokumentation*

Software lebt auch von der Dokumentation. MoWare unterstützt aktiv Dokumentation von Software im Quellcode, die über einfache Kommentare hinausgeht. Bereits angesprochen wurde, dass in Objekten bei Properties Kommentare hinterlegt werden können, die in der Oberfläche später bei entsprechenden Felder als Tooltips angezeigt werden. Auch in Kommandos kann Dokumentation abgebildet werden. Im nachfolgenden Bild wurde eine kurze Beschreibung nach dem Abschnitt "local variables" eingefügt.

```
command 'Beleg anzeigen' in process RechnungsProcess with Rechnung rechnung  OR null (set proc doc first)!

command parameter:
string mode

local variables:
string user


// Abhängig vom Rechnungstyp wird die Rechnung unterschiedlich ausgedruckt //
   1 => Rechnung, Erfassung Manuell: Deckblatt
   2 => Rechnung, Erfassung EDI: Gesamte Rechnung
   3 => Kürzung: Brief
   4 => Gutschrift, Erfassung Manuell: Deckblatt
   4 => Gutschrift, Erfassung EDI: Gesamte Rechnung


command settings:
command type: GRAPH_EDIT
enabled if: <cond>
question on external abort: <msg>
title addon: <msg>
command icon: HafinaDefaults.ICON_SETTINGS

command init:
func()->void {
  File pdf = call RechnungsService.createPDF(rechnung) ;
  if (mode != null && mode == "print") {
    Desktop.getDesktop().print(pdf);
  } else {
    Desktop.getDesktop().browse(pdf.toURI());
    /*
    Desktop.getDesktop().open(pdf);
    */

  }
  pdf.deleteOnExit();
}
```

Abbildung 70: Dokumentation

Gerade die Anordnung von Kommandos in der Software und die kurzen Kommentare zu den Kommandos kann MoWare als sogenannter Hierarchical Application Tree zusammenfassen. In dem Baum erkennt man, welche Kommandos von welchen Kommandos aus gestartet werden können. An dem folgenden Beispiel, lässt sich die Idee einfach zeigen.




| | A | B | C |
|---|---|---|---|
| 1 | Forms 'application information' | 2014-04-11T11:04:25.979+02:00 | |
| 2 | | | |
| 3 | Investigated models: | | |
| 4 | at.hafina.wws.wwsDATA | | |
| 5 | at.hafina.wws.wwsPROC | | |
| 6 | at.hafina.reko.kmsw | | |
| 7 | at.hafina.basis.basisDATA | | |
| 8 | at.hafina.basis.basisPROC | | |
| 9 | at.hafina.basis.rpcServer | | |
| 10 | at.hafina.basis.artikelStammAPP | | |
| 11 | at.hafina.basis.mitarbeiterDATA | | |
| 12 | at.hafina.basis.artikelPreisPROC | | |
| 13 | at.hafina.basis.artikelStammDATA | | |
| 14 | at.hafina.basis.parteienStammAPP | | |
| 15 | at.hafina.basis.parteienStammDATA | | |
| 16 | at.hafina.belegEin.ediDATA | | |
| 17 | at.hafina.belegEin.ediPROC | | |
| 18 | at.hafina.beschaffung.rekoUI | | |
| 19 | at.hafina.beschaffung.rekoAPP | | |
| 20 | at.hafina.beschaffung.rekoDATA | | |
| 21 | at.hafina.beschaffung.rekoPROC | | |
| 22 | at.hafina.beschaffung.rechnungDATA | | |
| 23 | at.hafina.beschaffung.rechnungPROC | | |
| 24 | | | |
| 25 | | | |
| 26 | Summary of 'Start' menu: | | |
| 27 | Rechnungen anzeigen SEARCH_VIEW/ RO-REPO | | |
| 28 | | Akt bearbeiten GRAPH_OWNER/ CHKOUT-REPO/ RO-REPO | |
| 29 | | | Prüfung abschließen GRAPH_EDIT |
| 30 | | | Akt abschließen GRAPH_EDIT/ RO-REPO<br><br>Rechnungen die noch nicht verbucht sind werden verbucht<br>Vollständiges PDF des Aktes wird erstellt. |
| 31 | | | Akt revisionieren GRAPH_EDIT<br><br>Ein bereits abgeschlossener Akt kann zur Nachbearbeitung geöffnet werden<br>Der Akt bekommt eine neue Versionsnummer |
| 32 | | | Beleg hinzufügen GRAPH_EDIT/ CHKOUT-REPO |
| 33 | | | Beleg entfernen GRAPH_EDIT |
| 34 | | | Beleg stornieren GRAPH_EDIT/ RO-REPO<br><br>Belegstorno, kein Wareneingang<br>Belegstorno, da Annahme der Ware verweigert<br>Belegstorno, da Gutschrift storniert<br>Storno Bela, da zwischenzeitlich GS erteilt<br>Storno Bela, da fehlerhaft/irrtümlich ausgestellt<br>..... |
| 35 | | | Belastungsanzeige abschließen GRAPH_EDIT |
| 36 | | | Gutschrift anfordern GRAPH_EDIT |
| 37 | | | Rechnung kontieren GRAPH_EDIT |
| | | | Beleg anzeigen GRAPH_EDIT<br><br>Abhängig vom Rechnungstyp wird die Rechnung unterschiedlich ausgedruckt<br>1 => Rechnung, Erfassung Manuell: Deckblatt |

Abbildung 71: HAT Übersicht

Unter Tools->"Forms application information" steht ein Werkzeug zur Verfügung, um das obige Bild zu erzeugen. Unter "Summary of 'Start' menu" ist ersichtlich, dass

- das Kommando "Rechnungen anzeigen" wählbar ist
- nach ausführen des Kommandos in der Benutzeroberfläche das Kommando "Akt bearbeiten" wählbar ist
- in der Oberfläche dieses Kommandos dann "Prüfung abschließen" wählbar ist

Nicht bei allen Kommandos wurde im obigen Fall eine Dokumentation/Kommentar hinterlegt. Bei "Akt abschließen" in diesem Beispiel schon.

Dokumentationsaspekte werden in zukünftigen Versionen von MoWare noch ausgebaut. Insbesondere eine Anlehnung an BPML birgt große Potentiale.