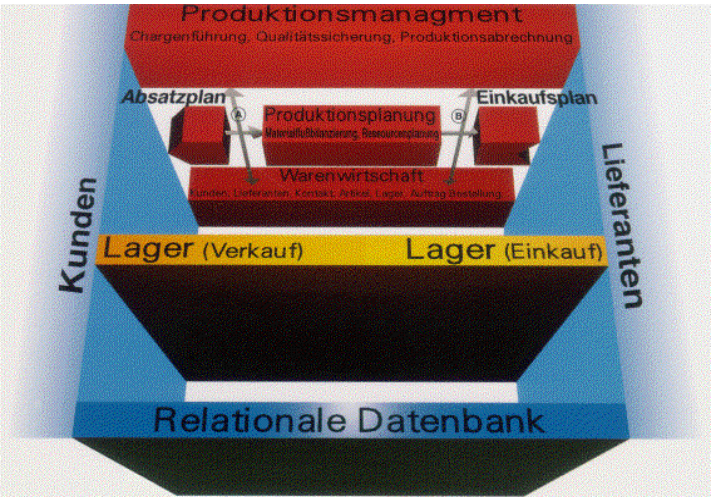
Java-Programmiersystem NeRTHUS



Entwicklerdokumentation

JPS-Klassenbibliotheken und Customizing NeRTHUS PPS

NeRTHUS Informationssysteme

Inhalt

1 Einleitung

Die Anforderungen an moderne Software sind hoch: sie soll sich möglichst flexibel an die Kundenanforderungen anpassen, soll schnell in hoher Qualität und kostengünstig entwickelt sein, sich in die unterschiedlichsten Strukturen integrieren und vieles andere mehr.

Der technische Fortschritt ist in diesem Bereich der Produktion nicht ohne Wirkung geblieben und drückt sich sehr stark in der Produktivität aus. Was früher Mannjahre an Entwicklungsarbeit benötigt hat ist heute mit modernen Technologien und Werkzeugen in wenigen Wochen erreichbar. Rapid Prototyping, Individualsoftware und Customizing wären ohne diesen Fortschritt nicht vorstellbar oder aber unvorstellbar teuer. Gerade in kleineren Unternehmen spielen diese Dinge eine existenzielle Rolle, wollen diese Unternehmen mit ihren Ressourcen am Markt bestehen.

In dieser Dokumentation werden zwei Dinge grundsätzlich beschrieben:

- Java-Programmiersystem (JPS) NeRTHUS mit den Klassenbibliotheken foundation, dbkit und appkit
- Anpassungsprogrammierung (Customizing) NeRTHUS PPS

Das NeRTHUS PPS ist die erste umfangreiche Anwendung des JPS, dem später noch andere folgen werden. In der Entwicklung des PPS-Systems ist auch ursächlich die Entwicklung des JPS begründet. Mit dem Aus von OPENSTEP durch das Aufgehen von NeXT in Apple wurde ein objektorientierte Entwicklungssystem und vor allem eine effektive Portierungsstrategie gesucht. Ersteres war schnell entschieden, ohne hier nun weitere Ausführungen über das Pro und Kontra zu Java zu machen. Der zweite Punkt stellte sich doch etwas komplizierter dar als anfangs erwartet.

Java-Dialogelemente (awt und swing) benötigen Controller-Klassen, die das entsprechende Listener-Interface implementiert haben. Möchte man z.B. ein Fenster mit einem einzigen Controller-Objekt verwalten, so muss man alle benötigten Listener-Methoden implementieren und in diesen unterschiedlichste Dialogelemente unterscheiden, um die entsprechende Funktionalität zu programmieren. Bei OPENSTEP hingegen konnte man bei jedem Dialogobjekt das Controllerobjekt mit einer frei definierten aufzurufenden Methode (action method) angeben (Interfacebuilder, nib-file). Die OPENSTEP-Philosophie ist unabhängig von den Portierungsfragen m.E. sehr viel besser, man ist nicht auf eine definierte Notation angewiesen und damit sehr viel flexibler. Insbesondere dadurch, dass man damit erst zur Laufzeit (dynamic binding) die Verbindung von Programm und Dialogoberfläche interpretieren kann. Damit ist schon die Zielstellung für diese Entwicklung erwähnt. In der Endversion soll mit XML das Layout der Dialogoberfläche und die Interaktion zwischen Dialogelemente und Programm beschrieben werden. Das entspricht dann in etwa dem nib-File des OPENSTEP-Interfacebuilder. Gleiches soll zwischen Datenbankmodell und Objektmodell wirken – eine XML-Datei, die diese Schnittstelle beschreibt und zur Laufzeit entsprechend interpretiert wird. Damit wird auch einer anderen Anforderung Rechnung getragen: die Programme sollen leicht und effektiv anpassbar sein, sowohl bei der GUI wie auch gegenüber unterschiedlicher Datenbankmodelle für die Systemintegration. Dem muss mit Beginn der Entwicklung Rechnung getragen und dazu ein Programmiersystem geschaffen werden.

Die Entwicklung des JPS steht aber noch am Anfang, der Umfang ist noch sehr zweckmäßig gering. Im Ergebnis sind aber bereits Klassenbibliotheken entstanden, die für Java-Programmierer an sich sehr interessant sind. Wer z.B. ein JTable verwenden will, wird NETable bzw. NETableBrowser sehr zu schätzen wissen. NEDbManager und NEObjectFactory, die eine sehr konfortable und sichere Schnittstelle zwischen Datenbankmodell und Objektmodell bilden, sind ein komfortabler Ersatz zu JDBC.

2 JPS-Klassenbibliotheken

Alle Klassen sind mittels JavaDoc als HTML-Dateien beschrieben und auf der Seite:

www.nerthus.de

verfügbar.

Der Umfang ist zweckmäßig auf den ersten Anwendungsfall, dem NeRTHUS PPS, zugeschnitten und deshalb noch überschaubar. Deshalb wird auch auf hochtrabende Bezeichnungen wie NeRTHUS Object Framework (NOF) verzichtet, weil es ähnlichen Vorbildern darin nicht gerecht wird. Aber gerade dbkit und appkit haben erheblich zur schnellen und qualitätsgerechten Entwicklung beigetragen, diese erst ermöglicht.

In diesem Abschnitt werden die Klassenbibliotheken einführend mit Anwendungsbeispielen beschrieben, sodaß sie für jeden Java-Programmierer effektiv einsetzbar sind. Für Programmierer des NeRTHUS PPS sind diese sogar ein muss, weil das gesamte System darauf aufbauend basiert.

Sämtliche Erweiterungen, Ideen und Anregungen sind willkommen! Der Bereich XML und andere Erweiterungen werden weiter durch NeRTHUS getragen. Je mehr Programmierer das System anwenden und ihre zu verallgemeinernden Erweiterungen dem JPS zur Verfügung stellen, je mehr Anwender werden davon in der Zukunft profitieren – so wie Sie gerade jetzt vielleicht.

2.1 Foundation Kit – com.nerthus.foundation

Wie in der Objektorientierung allgemein üblich, werden mit Beginn einer umfangreichen Entwicklung eigene Basis-Klassen implementiert. Das hat zumindest den Vorteil, dass man bei fortgeschrittener Entwicklung noch nachträglich zu verallgemeinernde Funktionen systemweit wirksam implementieren kann. Hier und da kann man sicherlich auch eine Java-Basis-Klasse von SUN weiterentwickeln. Ursprünglich war geplant, zu vielen Java-Klassen hier ein Pendant zu schaffen, z.B. String und NEString usw.. Dieser Ansatz wurde aber schnell fallen gelassen, weil der Umfang und die Funktionalität der in Java-Klassen implementierten Funktionalität sehr ausgereift ist und eine reine Umbenennung aus Portierungsgründen nicht zweckmäßig erschien.

Deshalb sind die Foundation-Klassen hier eher von untergeordneter Bedeutung und werden mehr zu Vollständigkeit erwähnt.

NEObject stellt die Root-Klasse für das JPS dar. Hier sind so einige nützliche Dinge, wie die Rückgabe des Klassennamens ohne Package u.a. implementiert. Aber eigentlich dient sie mehr dem erstgenannten Zweck, der Möglichkeit, später allgemeine Funktionen systemweit zur Verfügung zu stellen.

Beim **NECalendarDate** sieht es schon etwas anders aus. Dort wurde die Funktionalität der Java-Klassen Date und Calendar komfortabel erweitert. Eine wesentliche Ursache hierfür liegt darin, dass die Amerikaner dem Zeitbegriff Kalenderwoche (KW) keine Bedeutung zumessen und diese Funktion sehr selten implementiert ist. Für ein PPS-System wird die Kalenderwoche benötigt und somit die Methode weekOfYear() geschaffen. Für das Dbkit wurde auch ein Datentyp benötigt, der DateTime-, Timestamp- und Date-Datentypen der Datenbanken besser als java.sql.Date repräsentiert.

Die Wirkungsweise von NECalendarDate wird am folgenden Beispiel deutlich:

```
// NECalendarDate mit dem 01.01.2001 00:00:00
NECalendarDate nec = new NECalendarDate(2001, 01, 01, 0, 0, 0);
NECalendarDate nec_kw_ende;
for(a = 0; a < 53; a++) {
      if(a > 0) {
            nec.adding(0, 0, 7, 0, 0, 0);
      }
      System.out.println(
            "KW:\t" + nec.weekOfYear() +
            "\tDatum:\t" + nec.dayOfMonth() + "." +
            nec.monthOfYear() + "." +
            nec.yearOfCommonEra()
      );
      nec_kw_ende = nec.dateByAdding(0 , 0, 6, 23, 59, 59);
      System.out.println(
            "Ende KW:\t" + nec_kw_ende.getTimestampAsString()
      );
}
```

2.2 Database Kit – com.nerthus.dbkit

Die meisten Applikationen wirken mit Datenbanken zusammen, d.h. alle wesentlichen Daten werden in Datenbanken verwaltet. Damit ergibt sich immer wieder die Aufgabe, eine spezifische Schnittstelle der Datenzugriffe zu implementieren. In objektorientierten Programmsystemen (siehe verschiedene Design-Pattern) gibt es unterschiedliche Ansätze, einen Datenbanksystem in der Applikation zu modellieren, d.h. entsprechende Repräsentanten dafür zu schaffen. Häufig auftauchende Begriffe sind DBManager, DBSession, Business-Object (BO), Entity-Manager, Relation/Assocation usw.. Auch viele Prinzipien sind formuliert und haben sich sinnvoll bewährt, so z.B. die Trennung von Business-Logik und Business-Objekte, die Referenzierung über spezielle Assozation-Objekte. Das Dbkit stellt einen Anfang dar, dass alles für Java als Framework zu erschaffen, so wie es von EOF bis Forté bekannt ist. SUN stellt mit dem JDBC die Basisfunktionalität zur Verfügung. Damit aber komplexe Anwendungen zu schaffen ist nicht effektiv, noch besonders wartungsfreundlich. Eine sehr nutzbringende Technologie ist Dynamic SQL, d.h. SQL-Statements, die gegenüber dem DBMS automatisch nach den Erfordernissen generiert werden. Z.B. im NeRTHUS PPS sind ca. 100 Business-Klassen definiert. Für diese Klassen alle SELECT-, UPDATE- und INSERT-Statements zu programmieren, würde ein extremen Aufwand bedeuten. Somit ist es sinnvoll die Vorteile der Objektorientierung zu nutzen und komplexere Klassen zu implementieren, die die benötigten Funktionalitäten beinhalten.

Business-Objekte entsprechen einzelnen Datensätzen einer Tabelle/Entity. Im JPS werden die Business-Klassen von **NEEnterpriseObject** abgeleitet und gewährleisten so das Zusammenwirken mit der **NEObjectFactory**. Die NEObjectFactory entspricht wiederum genau einer Entity/DB-Tabelle und wirkt mit dem **NEDbManager** zusammen, der eine Datenbank repräsentiert und eine einzigste Datenbank-Session verwaltet.

Diesen Zusammenhang soll folgende Grafik darstellen:

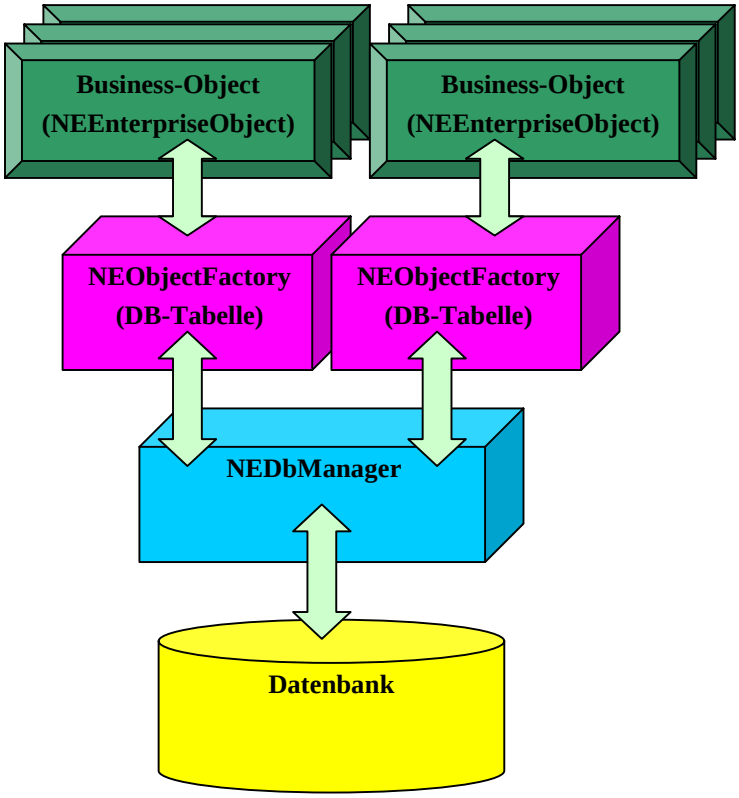


Abb. 1 Klassen des JPS-Database-Kit (dbkit)

Es ist relativ einfach zu den Klassendefinitionen der Business-Objekte zu kommen. Wenn man über ein Werkzeug verfügt, wie z.B. Sybase PowerDesigner, dann werden einen die Java-Klassen zu Datenbanktabellen generiert. Das funktioniert während dem Entwurf des Datenbankmodells oder aber durch das Reengineering vorhandener Datenbanken. Für das JPS müssen nur kleinere Veränderungen durchgeführt werden, wie z.B. die Ableitung von NEEnterpriseObject und das Ändern der Datentypen von java.sql.Date zu NECalendarDate. Ggf. kann das Werkzeug derart konfiguriert werden, dass es die Klassen gleich so generiert. Am Beispiel der Organisationseinheiten des PPS soll eine Klassendefinition dargestellt werden. Die DB-Tabelle hat folgende Definition:

```
create table OE (
      oe_id int not null,
      oeschl varchar(10) not null,
      bezeichnung varchar(100) not null,
      oetyp int
)
```

Wie im Package com.nerthus.proisys.db ersichtlich ist, sieht die Definition des zugehörigen Business-Objektes so aus:

```
//////////////////////////////////////////
// This file was generated by PowerDesigner //
//////////////////////////////////////////

public class OE extends com.nerthus.dbkit.NEEnterpriseObject
{
      public int oe_id;
      public String oeschl;
      public String bezeichnung;
      public int oetyp;
}
```

Möchte man die Klasse um weitere Attribute erweitern, so muss man darauf achten, dass diese private sind!

!! Alle public attributes sind auch Datenbankattribute, d.h. sie werden bei der Generierung der SQL-Statements durch die NEObjectFactory berücksichtigt!

Hat man das für alle benötigten Entitäten gemacht, dann ist die wesentlichste Arbeit damit erledigt. Damit kann nun beschrieben werden, wie man mittels NEDbManager und NEObjectFactory zu seinen Business-Objekten kommt. Zuerst braucht man eine DBSession, die durch den NEDbManager verwaltet wird. Die Technologie basiert auf ODBC bzw. ist derzeit nur für ODBC getestet. Es ist also administrativ nötig, das die entsprechenden ODBC-Treiber betriebssystemseitig installiert und konfiguriert sind. Das ist für die einzelnen Betriebssysteme sehr unterschiedlich und soll hier nicht beschrieben werden. Auf der Seite von Java (JDBC) gibt es folgende Entsprechungen für ausgewählte Datenbanktypen:

```
Sybase jConnect:  com.sybase.jdbc.SybDriver
JDBC/ODBC:        sun.jdbc.odbc.JdbcOdbcDriver
Oracle:           oracle.jdbc.driver.OracleDriver
```

Der **NEDbManager** wird beispielhaft folgendermaßen konfiguriert und mit der Datenbank verbunden:

```
NEDbManager dbmgr = new NEDbManager();
// 1 = DE (deutsch), 2 = US
// Datumsformat DateTime!! dd.mm.yyyy ...
dbmgr.setLanguage(1);
dbmgr.loadDriver("sun.jdbc.odbc.JdbcOdbcDriver“)
dbmgr.setUrl("jdbc:odbc");
dbmgr.setUser("sa");
dbmgr.setPassword("denkste");
dbmgr.setDatabase("staudb");
```

```
if (!dbmgr.connect()) {
      if (dbmgr.hasError()) {
        System.err.println(dbmgr.getErrorStr());
      }
} else {
      System.out.println(" ==> db is connected");
}
```

Aus welcher Quelle die Connection-Daten, die als String übergeben werden, gewonnen werden, ist egal. Oftmals kommt der Nutzername und das Passwort direkt von einem Login-Panel oder aber aus Property-Dateien (Dictionaries).

Bei Bedarf kann eine beliebige Anzahl von NEDbManagern eingerichtet werden. Die NEObjectFactory liefert die Business-Objekte ihrer Entity grundsätzlich als Array (Vector), selbst wenn nur 1 oder 0 Datensätze selektiert werden konnten. Die **NEObjectFactory** wird folgendermaßen eingerichtet und genutzt:

```
int a;
int anzahl;
Vector oedsList;
com.nerthus.proisys.db.OE theOEDS;
String oeQualifier;

NEObjectFactory  theOF =
new NEObjectFactory("com.nerthus.proisys.db.OE ", dbmgr);
// theOF.useForEnterpriseClass("com.nerthus.proisys.db.OE");

// Generation (SELECT)
oeQualifier = "oetyp = 2 order by oeschl";
oedsList = theOF.getNEObjects(oeQualifier);
anzahl = oedsList.size();
for(a = 0; a < anzahl; a++) {
      theOEDS = (com.nerthus.proisys.db.OE) oedsList.elementAt(a);
      ...
}

// Löschen (DELETE)
f = theOF.deleteNEObjects(oeQualifier);
if(f <= 0) {
      return fehler;
}
// Einfügen (INSERT )
f = theOF.insertNEObject(theOEDS);
if(f <= 0) {
      return fehler;
}
```

```
// Speichern (UPDATE)
f = theOF.updateNEObject(theOEDS, oeQualifier);
if(f <= 0) {
      return fehler;
}
```

Insert und Update kann auch zweckentsprechend geschachtelt werden, sodaß man sich bei der Programmierung nicht darum kümmern muss, ob der Datensatz in der DB bereits vorhanden ist oder nicht. Für die weiterentwickelte Version des JPS ist es geplant, das die Schlüssel eines Datensatzes bekannt sind und die Transaktionalität programmierbar ist (ObjectFactory.beginnTransaction() ...). Derzeit findet alles recht einfach mittels einem String, der als Qualifier bezeichnet wird, statt. Dieser String entspricht der WHERE-Klausel eines entsprechenden SQL-Statements. Ähnlich wird bei einem ausschließlichen ORDER BY verfahren:

```
myQualifier = "articleno";
objectList = theOF.getNEObjectsOrderBy(myQualifier);
```

Ein wichtiger Hinweis zum performanten Verhalten darf hier nicht fehlen. Die NEObjectFactory kann instanziierte Objekte wiederverwenden. Das ist besonders dann sinnvoll, wenn mehrere unterschiedliche Datensätze des gleichen Typs nur kurzzeitig selektiert und generiert werden müssen. Da in Java die Instanziierung von Objekten besonders viel Rechenzeit benötigt, wirkt sich das Verfahren sehr positiv auf das Leistungsverhalten einer Anwendung aus.

```
useForEnterpriseClass(String theClass, boolean caching)
NEObjectFactory(String theClass, NEDbManager dbmgr,
                  boolean caching)
```

2.3 Application Kit – com.nerthus.appkit

AWT- und Swing-Klassen von Java sind Dialogelemente, die einerseits Daten für den Nutzerdialog in einer bestimmten Form darstellen und andererseits auf Nutzeraktionen mit dem Senden von Events reagieren. Jedes Element hat seine eigenen Events, wozu dann im Controllerobjekt das entsprechende Listener-Interface implementiert sein muss. In den Java-Examples wird meistens nur ein einzelner Typ einer Swing-Klasse dargestellt. In einer realen Anwendung hat man aber sehr viele unterschiedliche Typen von Swing-Klassen auf einem Fenster, sodaß viele Listener-Interfaces zu implementieren sind und in den Listener-Methoden einzelne Dialogelemente unterschieden werden müssen.

Das Appkit wurde entwickelt, um das drastisch zu vereinfachen und der Dynamik von Daten einer realen Anwendung Rechnung zu tragen. Zu den wesentlichen Java-Swing-Klassen gibt es ein NeRTHUS-Pendant, das von der abstrakten Klasse **NEDialogElement** abgeleitet ist. Deshalb wird auch im weiteren Verlauf von NEDialogElementen die Rede sein.

NEDialogElemente sind Container zu ihren verwalteten Swing-Klassen und haben in sich schon das entsprechende Listener-Interface und ggf. das Datenmodell (JTable, JTree) implementiert. Container bedeutet, dass es keine Ableitungen, sondern die Orginalelemente gekapselt sind. D.h. es gibt keinerlei Einschränkungen, weil mit getComponent() jederzeit mit der Swing-Klasse gearbeitet werden kann, was i.d.R aber nicht notwendig sein wird.

Im Constructor gibt man das Controllerobjekt und die dort aufzurufende Methode (action method) an oder aber ein Delegate-Objekt, dass dann das Interface **NEDialogDelegate**:

```
handleEvent(NEDialogElement ctr, java.util.EventObject ev)
```

implementiert hat.

Z.B. die Instanziierung eines **NEButton**:

```
// NEButton(String title, Object target, String actionmethod)
NEButton myListButton = new NEButton("Liste", myCtrl, "createList");
myListButton.setBounds(120, 280, 80, 25);
...
```

Damit wird erreicht, dass beim Drücken des Buttons die Methode „createList“ des Controllerobjektes „myCtrl“ aufgerufen wird.

Beim JTextField sind weitere Vereinfachungen implementiert, sodaß z.B. Formate und Datentypen mit in das NEDialogElement einbezogen sind und so **NETextField**, **NENumberTextField** und **NEDateTextField** existiert.

```
myDate = new NEDateTextField("dd.MM.yyyy", null);
// Default NumberFormat Double mit 2 Nachkommastellen
myMaschVbkt = new NENumberTextField("", this, "changeVMasch");
mySchicht = new NENumberTextField("", this, "changeSchicht");
// Integer ohne Nachkommastellen in der Darstellung
mySchicht.setIntegerFormat(false);
```

Das vereinfacht schon die Programmierung und ist ein Schema, das Übersichtlichkeit und Einfachheit gewährleistet. Zielstellung ist eine dynamische Generierung aus XML-Dateien, wo dann Controllerobjekt und Action-Methode neben den Eigenschaften zu einem NEDialogElement angegeben werden können.

Nicht auf jedes Ereignis eines Java-Dialogelementes muss mit einem Programm reagiert werden. Das JTextField mit implementierten DocumentListener registriert nämlich schon das Verändern einzelner Zeichen usw. (CHANGE_UPDATE, INSERT_UPDATE, REMOVE_UPDATE). Bei den NEDialogElementen ist das Problem derart gelöst, dass für jedes mögliche Ereignis des verwalteten Swing-Elementes eine sogenannte **Actionmask** vereinbart ist und alle Listener-Methoden implementiert sind. Die Actionmask ist bit-orientiert (0, 1, 2, 4, 8) und es wird über eine bitweise UND-Verknüpfung festgestellt, ob auf das entsprechende Ereignis reagiert werden soll. Die Defaulteinstellung berücksichtigt erstmal das sinnvollste Ereignis, alle anderen sind maskiert (entsprechende Bit ist 0). Dadurch wird erreicht, das z.B. das NETextField nur auf den Abschluss der Eingabe (Return) mit dem Aufruf der Action-Methode reagiert.

Wer schon mal mit JTable gearbeitet und dazu in die Java-Newsgroups geschaut hat, der weiß das ist nicht trivial. Das JTable ist sicherlich das komplexeste Dialogelement von Swing. Die Java-Examples stellen nur sehr vereinfacht die Möglichkeiten dar und gehen hierbei von einem statischen Datenmodell aus. In einer realen Anwendung ist eine Tabelle jedoch selten statisch. Z.B. das Anzeigen von Datensätzen in Listen kann eine unterschiedliche Anzahl bei unterschiedlichen Suchbedingungen ergeben. Der **NETableBrowser** ist für solche Dinge ausgelegt und kann zur Laufzeit des Programms beliebig rekonfiguriert werden. **NETable** ist intern noch komplexer (TableModel) und kann zu jeder Zelle der Tabelle ein sogenannten Renderer (Darstellung) und Editor (Eingabe) verwalten. Der Verwendung von NETable bzw. NETableBrowser wurde in den JPS-Examples viel Aufmerksamkeit gewidmet. Bei der Programmierung des PPS-Systems wurde von diesen NEDialogElementen reichlich Gebrauch gemacht und die Quellen zu den Bundles stehen als weitere Anwendungsbeispiele zur Verfügung. Hier wird deshalb nur kurz ein Beispiel dargestellt. Es ist aus dem PPS-System zur Darstellung eines Lagersortiments (OEL-Dialog). Der NETableBrowser zeigt alle Lagermaterialien eines Lagers an und das sind unterschiedlich viele Datensätze je ausgewähltem Lager.

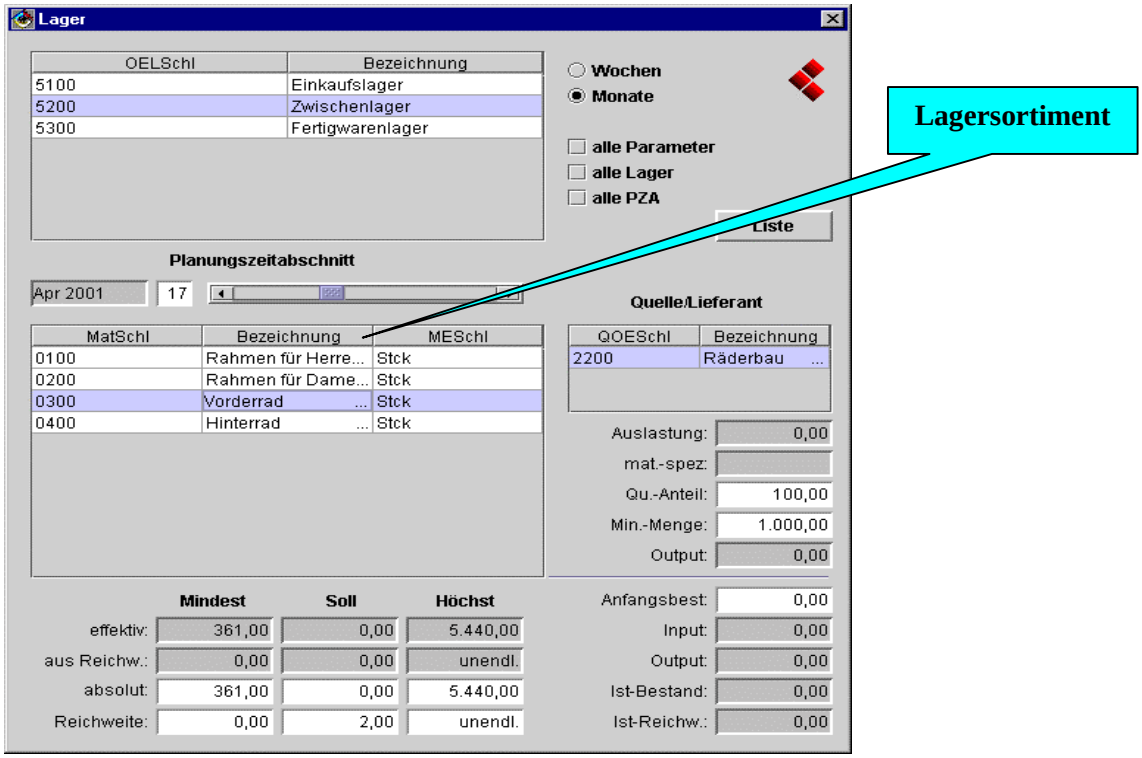


Abb. 2 OEL-Dialog als Beispiel zum NETableBrowser

Im Controllerobjekt zu diesem Dialog (com.nerthus.proisys.bundles.TIPDialog) ist folgendes implementiert:

1. Instanziierung (createOELPanel()):

```
LMBrowser = new NETableBrowser(this, "browserClick", null);
LMBrowser.setBounds(15, 230, 350, 200);
```

2. Laden des LMBrowser (loadLBrowser()):

```
// Anzahl Lagermaterialien der aktuellen OEL ermitteln
max = myOEL.lecount();
if(!LMBrowser.isDisplayed()) {
      // noch nicht angezeigt => Initialisierung der Col-Header
      LMBrowser.setTable(max, 3, true, false);
      LMBrowser.setColumnTitle("MatSchl", 0);
      LMBrowser.setColumnTitle("Bezeichnung", 1);
      LMBrowser.setColumnTitle("MESchl", 2);
} else {
      // Updaten auf eine neue Anzahl von Zeilen
      LMBrowser.updateTableModelRows(max);
}
LagerElement le;
// LMBrowser entsprechend laden
for(row = 0; row < max; row++) {
      le = myOEL.lagerelement(row);
      LMBrowser.setValueAt(le.articleno(), row, 0);
      LMBrowser.setValueAt(le.articlename(), row, 1);
      LMBrowser.setValueAt(le.unit(), row, 2);
}
// LMBrowser (re-) konfiguriert anzeigen
LMBrowser.display();
```

Aus der Instanziierung ist ersichtlich, dass die Methode browserClick() des Controllerobjektes (this = TIPDialog) aufgerufen wird. Die Defaulteinstellung des NETableBrowser ist die Einzelzeilen-Selektierung (SINGLE_ROW_SELECTION), sodaß dabei ein einzelnes Lagermaterial ausgewählt wird und dessen Lagerparameter im Fenster dargestellt werden:

```
// Nummer der selektierten Zeile
index = LMBrowser.selectedRow();
if (index != currentLM) {
      currentLM = index;
      // Lagerparameter des ausgewählten Lagermaterials anzeigen
      this.changeLM();
      // Lieferanten-Info (Quelle) zu Lagermaterial anzeigen
      currentQ = 0;
      this.loadLBrowser(QBrowser);
      // erste Quelle selektieren
      QBrowser.selectRows(currentQ, currentQ);
      this.changeQ();
}
```

In diesem Beispiel werden nur sogenannte Column-Header gebraucht. Im Bundle Auslastung (com.nerthus.proisys.bundles.Auslastung) ist ein gutes Beispiel für die zusätzliche Verwendung von Row-Header (initBrowser()):

```
myABrowser.setTable(anzahlOEP, maxWeek, true, true);
for(a = 0; a != anzahlOEP; a++) {
      myABrowser.setRowTitle(myUModell.getOESchl(a), a);
}
for(a = 0; a != maxWeek; a++) {
      myABrowser.setColumnTitle(myKalender.pzabez(a), a);
}
```

Die Zeilen werden mit dem Schlüssel der entsprechenden Organisationseinheit und die Spalten mit dem Planungszeitabschnitt (PZA – Kalenderwochen) beschriftet.

Es gäbe sicherlich noch das ein oder andere zum Appkit zu beschreiben, z.B. das NEScrollBar und NESlider auch mit Double-Zahlen erbeiten können. Die Grundprinzipien sind erläutert und die Verwendung wird für Java-Programmierer am besten aus den JPS-Examples bzw. dem PPS-System ersichtlich.

3 Anpassungsprogrammierung NeRTHUS PPS

Customizing ist ein neues Schlagwort, dass die veränderten Anforderungen an Software beschreibt. Auch das PPS-System soll flexibel an unterschiedliche Kundenanforderungen anpassbar sein und die Probleme der Kunden von verschiedenen Unternehmen lösen.

Mit dem NeRTHUS PPS (bzw. Stauseesystem TIP GmbH) ist eine erste Ausprägung zu einem Kernsystem programmiert, wo langjährige Erfahrungen zu einem allgemeingültigen Fertigungsmodell für Fließfertigungen zusammengefaßt sind. Es wird bei einem Schokoladenhersteller erfolgreich eingesetzt. Jeder Systemintegrator kann mit diesem Kernsystem und Fertigungsmodell sein eigenes PPS entwickeln oder aber diese Funktionalität seinen Produkten hinzufügen.

Dazu sind alle Vorteile der Objektorientierung nutzbar, wo durch Ableitung, Überladen und Erweiterung Funktionalität angepasst bzw. hinzugefügt werden kann. Weiterhin bietet Java einiges an Technologie, vor allem Schnittstellen wie CORBA, SAP, OCX-Bridge, Sockets um nur einige zu nennen.

In dieser Dokumentation wird die Standardschnittstelle zum Kernsystem beschrieben und erläutert, wie Bundles Werkzeuge, Module, Dialoge, Middleware u.a. das Kernsystem zu einer Anwendung aufbauen bzw. Schnittstellen zur Systemintegration bilden.

Auch in der ersten Ausprägung als PPS-System wurden alle Funktionalitäten als Bundles dem Kernsystem hinzugefügt und dienen hier als Beispiel. Das betrifft sämtliche Dialoge und Werkzeuge, Datenschnittstelle zur Datenbank und zu Dateien, Listenerstellung und -ausgabe, Excel-Anbindung mit OCX-Bridge u.v.a.m. und werden den Entwicklungspartnern mit Quellcode zur Verfügung gestellt.

Durch NeRTHUS werden weitere Bundles auf Kundenwunsch oder aber aufgrund einer zu verwirklichenden Idee entwickelt. Die Partnerunternehmen werden in ihren Projekten eigene Bundles in Rahmen ihrer Aufgabenstellungen entwickeln und zu verallgemeinernde Module dem Produkt beistellen. Damit wird in der Zukunft das Produkt mit seiner Leistungsfähigkeit und seinen Einsatzmöglichkeiten rasch wachsen.

Ab 01.01.2001 wird das Kernsystem ab Version 5.2 zentral durch die NeRTHUS und die Bundles kundenbezogen durch die Systemintegratoren verwaltet. Vereinbarte und dokumentierte Funktionalität wird in diesen entsprechenden Zuordnungen gewährleistet!

Wird mit einem Bundle Kernfunktionalität überladen bzw. erweitert, so ist dieses gesondert zu dokumentieren und zu übergeben!

Alle vorhandenen Methoden des Kernsystems sind ausreichend, um alle Daten/Objekte des Unternehmensmodells zu erreichen. Verbesserungsvorschläge bezüglich des "komfortableren" Zugriffs sind willkommen, ein Upgrade des Kernsystems erfolgt dann unmittelbar.

3.1 Entwicklungssystem und Systemvoraussetzungen

Java ist sehr hungrig auf Speicher und das sollte man bei der Einrichtung des Entwicklungssystems berücksichtigen und über mindestens 128 MB (besser 256 MB) Hauptspeicher verfügen. Das System wurde für die Java 2 Plattform entwickelt, also ab Version 1.2, derzeit ist ein 1.3 aktuell.

Zur Anpassung des PPS-Systems ist das JDK erforderlich, das ggf. mit einem Java-Entwicklungssystem geliefert wird. Mit einer Entwicklerlizenz zum NeRTHUS PPS wird IBM VisualAge für Java, Professional Edition, geliefert. Damit wurden sehr gute Erfahrungen gemacht. Andere Java-Entwicklungswerkzeuge sind aber genauso nutzbar. Die Verwendung von UML-CASE-Tools (z.B. Innovator) und dem Sybase PowerDesigner ist im Ermessen des jeweiligen Nutzers, wird aber empfohlen.

Die komprimierte Datei zum PPS-Entwicklungssystem enthält entpackt die Verzeichnisse (Java-Packages):

- com.nerthus.appkit	Application Kit
- com.nerthus.appkit.table	Application Kit NETable
- com.nerthus.dbkit	Database Kit
- com.nerthus.foundation	Foundation Kit
- com.nerthus.proisys.bundles	Standard-Bundles NeRTHUS PPS
- com.nerthus.proisys.db	Business-Klassen NeRTHUS PPS
- com.nerthus.proisys.pps	Application-Klassen NeRTHUS PPS
- com.nerthus.proisys.pps.kernel	Klassen des Kernsystems (Unternehmensmodell)
- com.nerthus.proisys.pps.staudb	Business-Klassen NeRTHUS PPS (Stauseesystem-DB)

Zur Einrichtung des Entwicklungssystems werden die Verzeichnisse mit den .java-Dateien, .class-Dateien und den Properties-Dateien in das Java-Entwicklungssystem importiert. Bei VisualAge macht man das über das Menu Files/Import, Directory und der Auswahl des Verzeichnisses der entpackten Dateien mit Selektion .java, .class und resource.

Die Datei nerthus.jar wird in das Classpath-Verzeichnis kopiert.

3.2 Kernsystem und Unternehmensmodell

Kernsystem und Unternehmensmodell sind insoweit eigentlich identisch, da alle Objekte, die das Modell einer Fertigung darstellen, gemeinsam mit den Objekten NEApplication und NEMainController das Kernsystem bilden. D.h. das Kernsystem/Unternehmensmodell beinhaltet alle Berechnungsvorschriften und bilanziert den Materialfluß pro Zeiteinheit und ermittelt daraus die Kapazitätsauslastung.

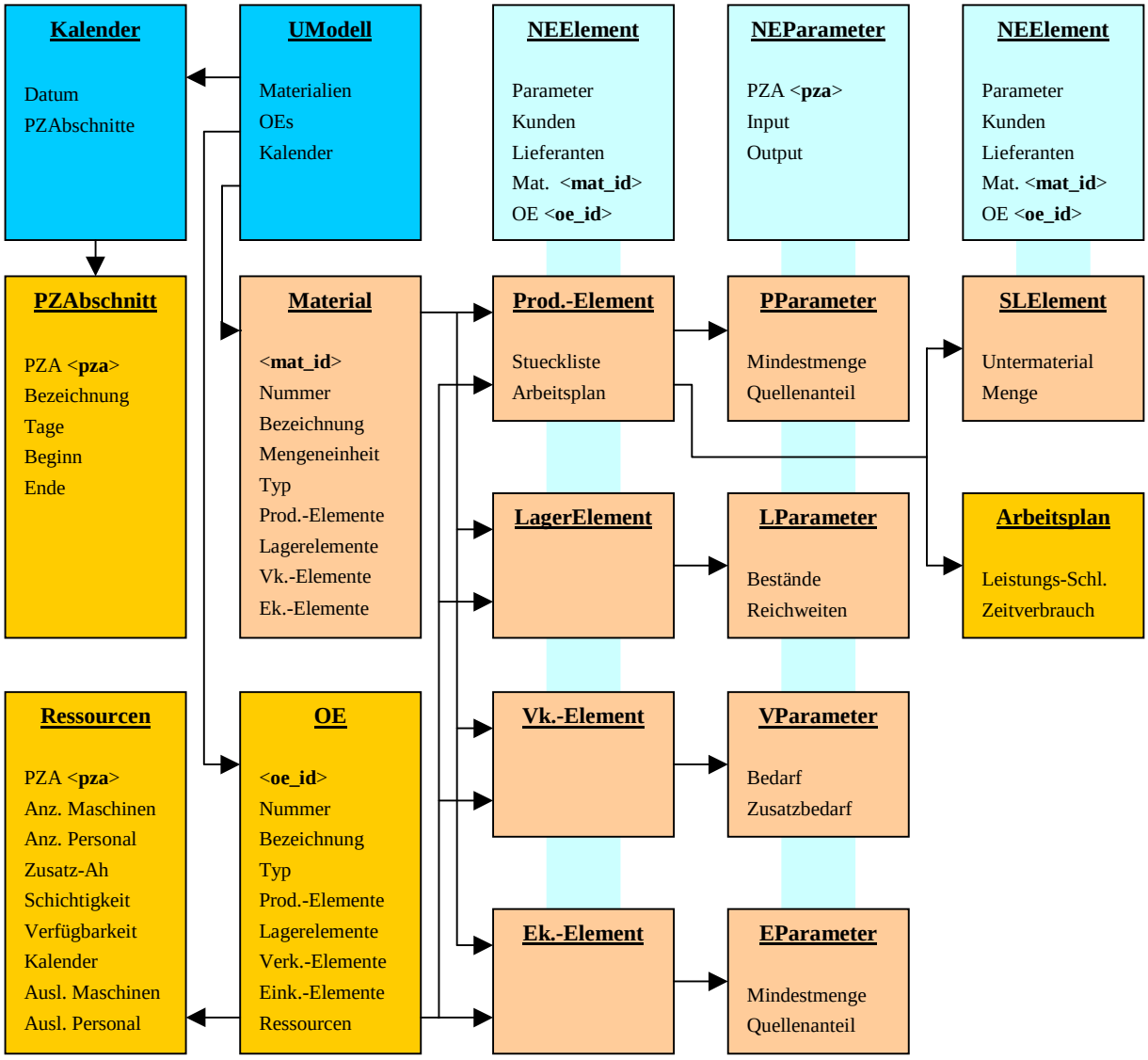


Abb. 3 Kernsystem/Unternehmensmodell

Wurde das Modell einmal komplett berechnet, so reagiert es interaktiv, d.h. auf jede Parameteränderung wird z.B. unmittelbar mit der Ausbilanzierung des Materialflusses und/oder der Kapazität reagiert. Berechnungsvorschriften (Businesslogic) und Parameter beschreiben den Sollzustand des Modells, den das Kernsystem automatisch einhält. Das ist eine sehr sinnvolle und leistungsfähige Eigenschaft für den Nutzerdialog, wo i.d.R. nur einzelne Parameter in relativ kurzer Zeit geändert werden. Natürlich kann man diese Eigenschaft auch suspendieren, wenn z.B. ein Update der Parameter aus einer Datenbank erfolgt. Dann erfolgt die Bilanzierung im Anschluss einer solchen Transaktion.

Das Objekt UModell ist das zentrale Objekt des Kernsystems und referenziert direkt bzw. indirekt über Material und Organisationseinheit alle Elemente des Modells. Im Wesentlichen verfügt ein Fertigungsmodell über 3 Dimensionen:

- Aufbauorganisation: Struktureinheiten wie Fertigungsbereiche, Abteilungen, Fertigungsnester, Kostenstellen etc. <**oe_id**>
- Ablauforganisation: Material, Materialwege, Materialfluß <**mat_id>** bzw. <**article_id**>
- Zeit: Planungszeitabschnitte PZA (Schicht, Tag, Woche, Monat, Quartal, Jahr) <**pza**>

die komplex zusammenwirken.

Die Berechnung des Modells läßt sich grob in zwei, jeweils aufeinander folgende Schritte gliedern:

1. Materialflussberechnung
2. Kapazitäts-/Ressourcenberechnung

Das ist eigentlich recht verständlich, weil ein (Wertschöpfungs-) Fertigungsprozeß immer mit (Energie-) Material- und Zeitverbrauch verbunden ist, die für einen Betrachtungsbereich und Betrachtungszeitraum ermittelt werden. Bei einem Planungsprozeß kann das wechselseitig restriktiv wirken.

Das Objekt UModell referenziert direkt das Objekt Kalender (Zeitangebot aus Kalenderdaten), ein Array von Objekten OE (Vector, Organisationseinheiten, Aufbauorganisation) und ein Array Material (Vector, Ablauforganisation).
Im NeRTHUS PPS ist es für die Zukunft vorgesehen (prinzipiell möglich) mehrere Objekte der Klasse UModell als Planvarianten zu verwalten, d.h. das die Applikation NeRTHUS PPS mehrere Kernsysteme parallel hält.

3.2.1 Betriebskalender

Der Begriff Betriebskalender ist bei mehreren Anwendern als unternehmensweite, sich aus dem Jahreskalender ergebene Betriebszeiten geläufig. Ein Monat hat z.B. 18 – 23 Arbeitstage, die je nach Tarifvertrag und Schichtsystem unterschiedliche Arbeitsstunden bedeuten, es gibt gesetzliche Feiertage und Betriebsferien. Das Objekt Kalender wirkt modellweit, d.h. die dort modellierten Zeitangebote aus Kalenderdaten wirken auf alle Organisationseinheiten (nächster Abschnitt Aufbauorganisation). Das Planungssystem beruht auf Planungszeitabschnitte (PZA), die als Einzeltupel das Zeitangebot aus dem Kalender modellieren und den Schlüssel <**pza**> haben. Alle zeitabhängigen Parameter, Daten und Berechnungsergebnisse beziehen sich auf einen PZA, d.h. Teilschlüssel ist <**pza**>.
Es gibt vereinzelt Modellierungen für Kunden, wo es erforderlich war, feinere Unterteilungen zu machen. Dort werden aus Tageszeitabschnitten (tza) Schichten, Arbeitstage mit Schichtwechsel etc. modelliert, die für einzelne Anlagen oder Abteilungen gelten. Für derartige Anforderungen ist die TIP GmbH als Systemintegrator ein erfahrener Ansprechpartner.

3.2.2 Modellelemente der Aufbauorganisation

Die Aufbauorganisation des Unternehmensmodell wird anhand von Organisationseinheiten (OE) strukturiert. Eine OE kann eine Maschine, ein Fertigungsbereich, eine Anlage, ein Fertigungsnest, eine Teilfertigung, eine Abteilung usw. umfassen und ist die kleinste planbare Einheit bezüglich der Kapazität und des Materialflusses. Es werden vier Typen von Organisationseinheiten unterschieden:

- Organisationseinheit Einkauf (OEE)
- Organisationseinheit Lager (OEL)
- Organisationseinheit Produktion (OEP)
- Organisationseinheit Verkauf (OEV)

Das PPS-System plant ganzheitlich vom Verkauf bis zum Einkauf.

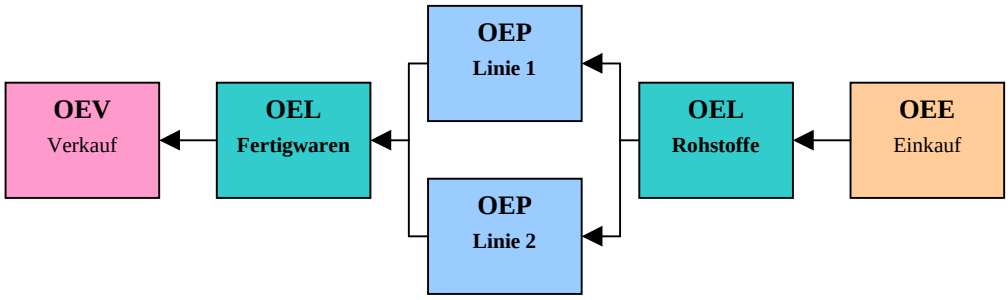


Abb. 4 Einfache Aufbauorganisation mit Organisationseinheiten

Die Objekte der Klasse OrganisationsEinheit (com.nerthus.proisys.pps.kernel) bilden im Modell die Aufbauorganisation ab. Alle Daten und Informationen, die mit OEs im Zusammenhang stehen, verfügen über den Teilschlüssel <**oe_id**>.

Je nach OE-Typ sind Elemente der Ablauforganisation (NEElement <**mat_id**>) einer OE zugeordnet:

- Verkaufselemente als Verkaufssortiment einer OEV,
- Einkaufselemente als Einkaufssortiment einer OEE,
- Lagerelemente als Lagersortiment einer OEL und OEP,
- Produktionselemente mit Stückliste und Arbeitsplan als Produktionssortiment mit spezifischen Material- und Zeitverbrauch einer OEP.

Die OEP verwaltet zusätzlich ihr zeitabhängiges (<**pza**>) Ressourcenangebot zur Berechnung der Kapazitätsauslastung.

3.2.3 Modellelemente der Ablauforganisation

Mit der Ablauforganisation wird der Materialfluss in der Fertigung beschrieben. Ein Material kann in einer Fertigung produziert, eingekauft, verkauft, gelagert und verbraucht (Stückliste/Rezeptur) werden.

Im Kernsystem wird der Materialfluss mit verketteten Objekten der Klasse NEElement abgebildet, wobei es folgende spezielle Typen gibt:

- Verkaufselement
- Lagerelement
- Produktionselement
- Einkaufselement
- Stücklistenelement

Alle NEElemente eines Materials verfügen gemeinsam über den Schlüssel <**mat_id**> bzw. <**article_id**>, werden zentral durch ein Objekt der Klasse Material und jeweils zugehörig durch ein Objekt der Klasse Organisationseinheit im Unternehmensmodell referenziert (<**oe_id**>). Die zeitabhängigen Parameter werden durch Objekte der Klasse NEParameter mit dem zusätzlichen Schlüssel <**pza**> modelliert (PParameter, LParameter, EParameter, VParameter).

Die NEElemente eines Materials stehen untereinander gerichtet im Kunden-/Lieferanten-Verhältnis (Array Supplier/Customer).

Unterschiedliche Materialien können nur ausschließlich durch Stücklistenelemente zueinander in Beziehung stehen (Mengenverhältnis Obermaterial/Untermaterial, <**omat_id**><**umat_id**>).

Ein Produkt einer Fertigung besteht in der Regel aus folgenden Elementen:

- ein Verkaufselement
- ein Lagerelement
- ein oder mehrere Produktionselemente,

ein Halbfabrikat, das nicht verkauft wird, aus:

- ein oder mehrere Stücklistenelemente (als Untermaterial)
- ein Lagerelement
- ein oder mehrere Produktionselemente
- (bei Zukaufmöglichkeit alternativ) ein Einkaufselement,

ein Rohstoff, Verpackungsmaterial, Einkaufsware aus:

- ein oder mehrere Stücklistenelemente (als Untermaterial)
- ein Lagerelement
- ein Einkaufselement.

Eine Handelsware, ein Material also, das eingekauft, gelagert und verkauft wird, kann im Unternehmensmodell dargestellt werden, spielt aber bezüglich eines Planungssystems keine Rolle.

3.3 Bundles

Bundles vervollständigen und Erweitern das Kernsystem um Werkzeuge zum grafischen Nutzerdialog, für spezielle Aufgaben, wie z.B. Auslastungsabgleich, zur Systemintegration u.v.a.m.. Kernsystem und Bundles ergeben gemeinsam eine spezifisch den Kundenanforderungen zugeschnittene PPS-Anwendung.

Damit ein Modul als Bundle mit dem Kernsystem interagiert, sind folgende Dinge notwendig:

- das Modul verfügt über ein zentrales Controllerobjekt (BundleController), dessen Klasse von com.nerthus.proisys.pps.kernel.NEController abgeleitet ist
- die Klasse des BundleControllers befindet sich in dem Bundle-Package (Default: com.nerthus.proisys.bundles)
- es ist eine Ressourcen-Datei *BundleControllerclassname.bundle* vorhanden
- das Bundle ist in der zentralen Ressourcen-Datei „proisys.conf“ unter bundles zum Laden eingetragen (bundles = TIPDialog,Auslastung,QADialog,*BundleControllerclassname*)

Den strukturellen Zusammenhang von Kernsystem und Bundles stellt die folgende Grafik dar:

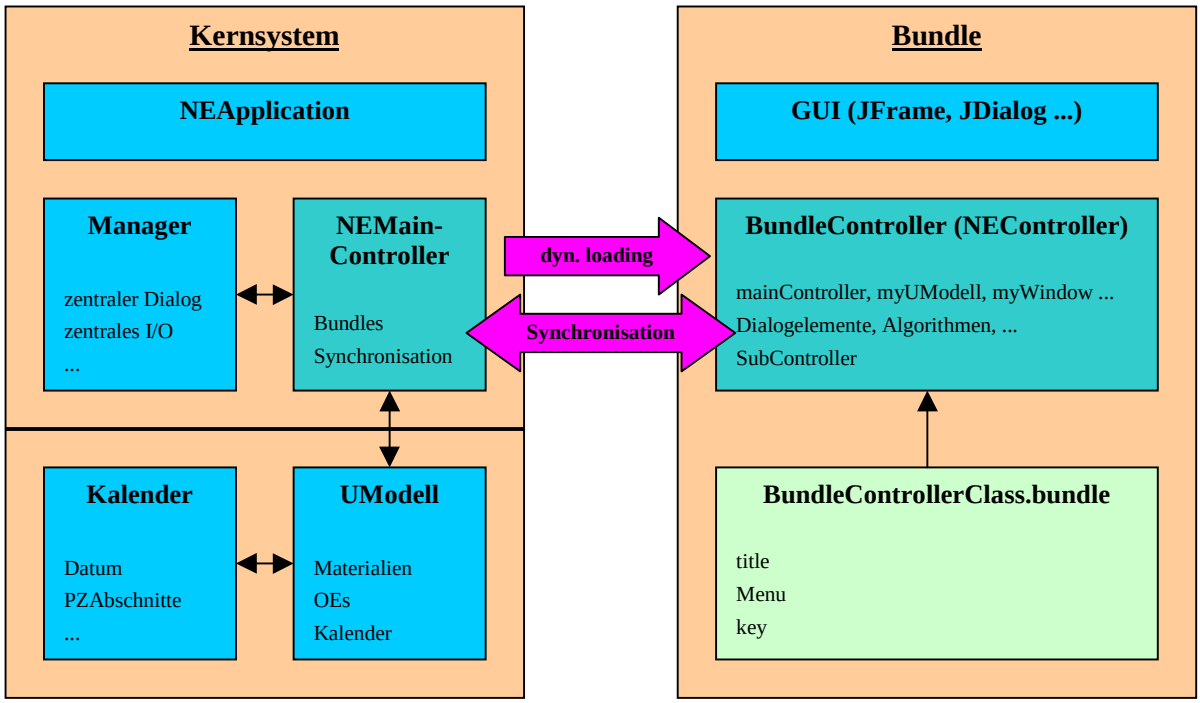


Abb. 5 Kernsystem und Bundles – NEMainController und NEController

Der MainController ist im Kernsystem das zentrale Verwaltungsobjekt für Bundles und NEController ist die abstrakte Klasse, womit das Zusammenwirken des Bundles mit dem MainController funktional sichergestellt wird.

MainController (Synchronisation) und UModell (Modelldaten) bilden die Schnittstelle der Bundles zum Kernsystem.

Die Bestandteile, ihre Wirkungsweise und Anwendung wird in den nächsten Abschnitten beschrieben.

3.3.1 BundleController (NEController)

Der BundleController ist das zentrale Verwaltungsobjekt des Bundles und seine Klasse ist eine Ableitung von NEController. Damit ist die Funktion als Bundle sichergestellt!

Der Name dieser Klasse wird in die Konfigurationsdatei proisys.conf beim Schlüssel „bundles“ eingetragen und die Klasse in dem Package, das unter „bundlePath“ angegeben ist, kopiert bzw. dort erzeugt.

```
...
bundlePath = com.nerthus.proisys.bundles
bundles = TIPDialog,Auslastung,QADialog,xxxxxxxxx
...
```

Die Werte der Schlüssel „bundles“ und „bundlePath“ werden zum Programmstart durch den MainController interpretiert, damit die Bundles registriert und dem Kernsystem zugeladen.

Die Klassendefinition ist hier am Beispiel des Auslastungsabgleichs dargestellt:

```
public class Auslastung extends com.nerthus.proisys.pps.NEController
implements NEDialogDelegate {

      private QAController myQAContr = null;
      private JDialog myAPanel = null;
      ...
```

Die Klassendefinition des NEController:

```
public class NEController extends com.nerthus.foundation.NEObject {
      protected static Properties strings = new Properties();
      protected Object myWindow;
      protected NEMainController mainController;
      protected Kalender myKalender; /* Betriebskalender */
      protected UModell myUModell;  /* Unternehmensmodell */

      protected int maxWeek;       /* max. Wochen des UModell */
      protected int maxMonth;      /* max. Monate des UModell */
      protected int maxPZA;        /* max. PZA des UModell */
      /* aktuellen Einstellungen zu maxWeek, maxMonth, maxPZA */
      protected int currentMaxWeek;
      protected int currentMaxMonth;
      protected int currentMaxPZA;
      /* mit Menupunkt ? */
      protected boolean MENU;
}
```

Aus der Klassendefinition geht hervor, das jeder BundleController folgende Eigenschaften von NEController erbt:

- strings Properties aus Ressourcen-Datei *.bundle
- myWindow Hauptfenster des Bundles (optional)
- mainController Referenz auf den MainController des Kernsystems
- myKalender Referenz auf den aktuellen Kalender des Unternehmensmodells
- myUModell Referenz auf das aktuelle Unternehmensmodell

Die Instanzevariablen maxWeek, maxMonth und maxPZA geben die maximalen Grenzen der Planungshorizonte des Unternehmensmodells aus dem Kalender wieder. Die Variablen currentMaxWeek, currentMaxMonth und currentMaxPZA sind die aktuellen Einstellungen des Unternehmensmodells, die kleiner/gleich (<=) denen des Kalenders sind.

Folgende Methoden erbt der BundleController, wobei die **fettgedruckten** i.d.R. zu überladen sind:

```
public int currentMaxMonth()
public int currentMaxPZA()
public int currentMaxWeek()
public void initWithModell(UModell modell)
public Kalender kalender()
public String key()
public int maxMonth()
public int maxPZA()
public int maxWeek()
public boolean menu()
public UModell modell()
public void openWindow()
public void setCurrentMaxMonth(int value)
public void setCurrentMaxWeek(int value)
public void setKalender(Kalender kalender)
public void setMainController(NEMainController mc)
public void setMaxMonth(int value)
public void setMaxWeek(int value)
public void setModell(UModell modell)
public String title()
public void update(boolean pzachanged)
public void updateModell(UModell modell)
```

Der BundleController wird durch den MainController instanziiert und initialisiert. Die Methoden update() und updateModell() sind Bestandteil des Synchronisationsmechanismus. Der MainController ruft die Methode updateModell() eines jeden Bundle auf, wenn das Unternehmensmodell sich geändert hat, dito update() bei jeder Datenänderung (z.B. bei einer Modellberechnung), wobei mit „pzachanged“ angegeben wird, ob sich Grenzen des Planungshorizontes geändert haben.

Danach sind diese beiden Methoden bei der Implementierung eines BundleControllers zu überladen, wobei der erste Befehl der Aufruf der Supermethode sein muss. Damit wird die Aktualisierung des

Bundles programmiert. Nach dem Beispiel des Bundles für den Auslastungsabgleich könnte es so aussehen:

```
public void update(boolean pzachanged) {
      super.update(pzachanged);
      if(myAPanel == null) { return; }
      if(!myAPanel.isVisible()) { return; }
      if(pzachanged) {
            if(WorM) {
                  myAQPZASlider.setMinMaxValue((maxWeek+2), maxPZA);
                  myAZPZASlider.setMinMaxValue((maxWeek+1),
                  (maxPZA-1));
            } else {
                  myAQPZASlider.setMinMaxValue(2, maxWeek);
                  myAZPZASlider.setMinMaxValue(1, (maxWeek-1));
            }
            currentAPZA = -1;
            this.loadBrowser();
            return;
      }
      this.loadBrowser();
      this.loadQMat();
      this.loadZMat();
      this.loadZParameter();
      this.loadQParameter();
      return;
}
```

Damit wird der Inhalt des Auslastungsfensters aktualisiert und wenn die Grenzen des Planungshorizontes sich verändert haben, dann wird ein Urzustand hergestellt.

Wird mit updateModell() ein neues Unternehmensmodell angezeigt, so ist für unser Beispiel folgendes implementiert:

```
public void updateModell(UModell modell) {
      super.updateModell(modell);

      if(myAPanel != null) {
            myAPanel.setVisible(false);
            myAPanel = null;
      }
}
```

D.h. es wird ganz simpel das Fenster geschlossen und zerstört, wohl wissend, dass bei der Erstellung des Fensters mit dem Laden der Inhalte sich auf das neue Unternehmensmodell bezogen wird.

Werden Modellberechnungen durch Aktionen im eigenen Bundle ausgelöst, so ist das PPS-System zur zentralen Aktualisierung (die Verständigung aller Bundles) zu informieren. Das geschieht durch den Aufruf der Methode:

mainController.update(false);

wodurch auch die eigene Aktualisierung angeregt wird. Z.B. setzen der Mindestmenge im Auslastungsdialog:

```
// NEDialogElement (NENumberTextField) myAQMM
x = myAQMM.doubleValue();
if(x >= 0) {
      // aktuelles Material ... ProduktionsElement
      currentAMD.pe.setMinmeng(x, currentAPZA);
      mainController.update(false);
} else {
      myAQMM.setDoubleValue(currentAMD.pe.minmeng(currentAPZA));
}
```

Die Option wird immer (false) sein, es sei denn, das Bundle ändert die Grenzen des Planungshorizontes, wie das Kontrollfeld der Anwendung.

3.3.2 Ressourcendatei eines Bundles (Properties)

Jedes Bundle braucht seine eigene Ressourcendatei, die bei der Instanziierung als Properties „strings“ geladen wird. In der Datei stehen die Informationen:

```
menu = 1                        // Menupunkt 1 = Ja, 0 = Nein
key = Auslastung                // Bundle-Schlüssel
title = Auslastungsabgleich     // Titel – z.B. für Menupunkt
```

Bisher gibt es zwei Arten von Bundles, diejenigen, die über einen Menupunkt aufgerufen werden und diejenigen, die mit oder ohne Dialogoberfläche auf der Grundlage von Ereignissen aktiv werden.

Die Ressourcendatei kann um eigene benötigte Eigenschaften erweitert werden. Die hier dokumentierten sind die, die durch den Mechanismus erforderlich sind. In den Ressourcendateien der derzeitigen Standard-Bundles ist regelmäßig noch folgender Eintrag zu finden:

```
qacontroller = QA
```

Damit wird der Schlüssel des Bundles für den Quellenanteildialog übergeben. Das ermöglicht diesen BundleController über den MainController zu referenzieren (Inter-Bundle-Dialog):

```
myQAContr = mainController.controllerWithKey(
                  strings.getProperty("qacontroller"));
```

3.3.3 Grafische Nutzeroberfläche (GUI)

In der nächsten Zukunft wird mit dem Appkit ein XML-Parser angeboten, der eine definierte XML-Datei interpretiert, die Dialogoberfläche mit den Dialogelementen erzeugt und alles mit den Controllern referenziert. Dazu wird noch ein Layoutwerkzeug gesucht, das den Entwurf der Dialogoberfläche unterstützt und XML als Ergebnis liefert.

Momentan findet bei NeRTHUS das Layout mit VisualAge statt, das Ergebnisapplet wird als Text exportiert und zur Erzeugung einer Methode createPanel() verwendet. Das ist derzeit definitiv noch etwas umständlich und uneffektiv. Es ist aber machbar und liefert ein Ergebnis, das mit der zukünftigen Lösung kompatibel ist. Die Methode createPanel() wird in der openWindow()-Methode des BundleControllers aufgerufen, wenn das Fenster myWindow gleich null ist.

```
if(myAPanel == null) {
     this.createPanel();
     ...
```

Als einfaches Beispiel dient hier die Erzeugung des Verkaufsfensters aus dem Standard-Bundle TIPDialog:

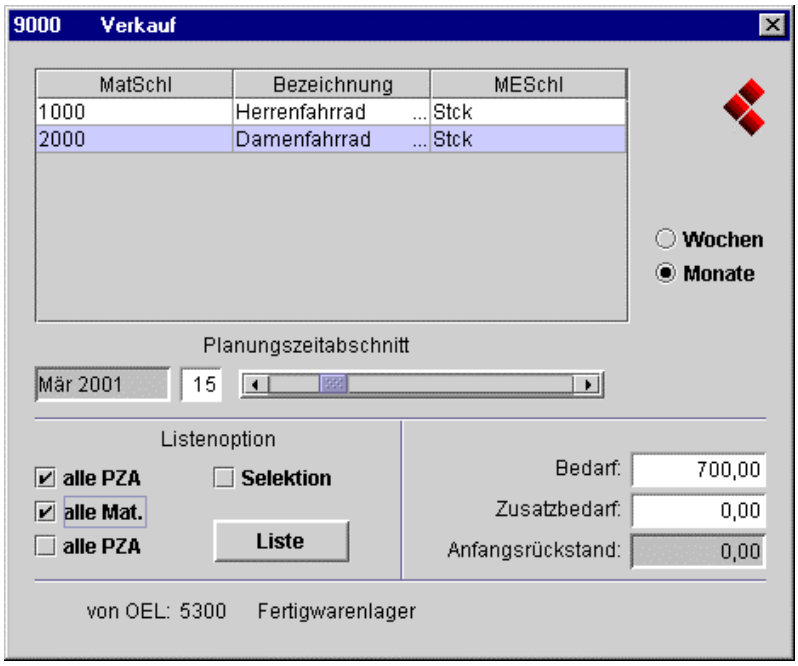


Abb. 6 Dialog Verkauf

Die Create-Methode mit der Instanziierung und Positionierung der Labels, Separatoren und Dialogelemente für dieses Fenster hat folgenden Aufbau:

```
private void createOEVPanel() {

      javax.swing.JLabel myVLabelAR = new javax.swing.JLabel();
      myVLabelAR.setName("myVLabelAR");
      myVLabelAR.setText("Anfangsrückstand:");
      ... weitere Label

      myVPZABez = new NETextField("", null);
      myVPZABez.setName("myVPZABez");
      myVPZABez.setBounds(15, 190, 80, 21);
      myVPZABez.setEditable(false);

      myVrPZA = new NENumberTextField("", this, "changeVrPZA");
      myVrPZA.setName("myVrPZA");
      myVrPZA.setBounds(100, 190, 25, 21);
      myVrPZA.setIntegerFormat(false);

      myVPZASlider = new NEScrollBar(this, "changeVPZASlider");
      myVPZASlider.setName("myVPZASlider");
      myVPZASlider.setBounds(135, 192, 215, 17);
      myVPZASlider.setOrientation(javax.swing.JScrollBar.HORIZONTAL);
      ...
      myVBedarf = new NENumberTextField("", this, "changeVParameter");
      myVBedarf.setName("myVBedarf");
      myVBedarf.setBounds(365, 240, 80, 21);

      myVAR = new NENumberTextField("", this, "changeVParameter");
      myVAR.setName("myVAR");
      myVAR.setBounds(365, 288, 80, 21);

      VBrowser = new NETableBrowser(this, "browserClick", null);
      VBrowser.setName("VBrowser");
      VBrowser.setBounds(15, 15, 350, 150);
      ... Separatoren ...

      javax.swing.JPanel theDialogContentPane = new javax.swing.JPanel();
      theDialogContentPane.setName("JDialogContentPane");
      theDialogContentPane.setLayout(null);
      theDialogContentPane.add(LIconLabel, LIconLabel.getName());
      theDialogContentPane.add(VBrowser.getComponent(), VBrowser.getName());
      theDialogContentPane.add(myVBedarf.getComponent(), myVBedarf.getName());
      ...

      myOEVPanel = new javax.swing.JDialog();
      myOEVPanel.setName("myOEVPanel");
      myOEVPanel.setResizable(false);
      myOEVPanel.setBounds(525, 40, 460, 355);
      myOEVPanel.setTitle("Verkauf");
      myOEVPanel.setContentPane(theDialogContentPane);

      return;
}
```

3.4 Kommunikation

3.4.1 Kommunikation mit dem Unternehmensmodell

Über die Funktionen der Superklasse des BundleControllers (NEController) wird sichergestellt, dass immer das aktuelle Unternehmensmodell den Bundles bekannt ist. Das aktuelle Unternehmensmodell wird durch die Instanz "myUModell" der Klasse UModell repräsentiert.

Über das Unternehmensmodell sind alle Elemente der Aufbau- und Ablauforganisation erreichbar.
Der Zusammenhang, d.h. wie kommt man von UModell über die OE bzw. das Material an die Produktionselemente, Lagerelemente, Einkaufselemente, Verkaufselemente, Stücklisten, Arbeitspläne, Ressourcen usw., wird aus der Abb. 3 des Abschnitts 3 deutlich.

Materialien (Objekte der Klasse Material) und Organsisationseinheiten (Objekte der Klasse OrganisationsEinheit) können auf zwei Arten aus dem Unternehmensmodell gewonnen werden:

- über den Datenbankschlüssel <oe_id> bzw. <mat_id>: getXXXByDBId()
- über den Index im UModell <oeidx> bzw. <matidx>: getXXXByIndex()

Das Objekt für die OEE (Einkauf) bzw. das Objekt OEV (Verkauf) erhält man über direkte Methoden getOEEinkauf() bzw. getOEVerkauf().

```
myOEV = myUModell.getOEVerkauf();
```

NEElemente verfügen alle über eine Referenz zu ihrer OE und zu ihrem Material:

```
myOE          = myLagerElement.getOE();
myMaterial    = myEinkaufsElement.getMat();
```

In OE und in Material existieren Arrays aller zugehörigen NEElemente. Das Lagersortiment einer OEL ist in dem Vector Lagerelemente (Array von Objekten der Klasse LagerElement) gespeichert und mit der Methode lagerelement(int idx) erhält man das Objekt. Dieses Prinzip wurde durchgängig eingehalten und gilt in adequater Notation für alle NEElemente, die über eine OE oder über ein Material referenziert werden sollen:

```
currentVE = myOEV.verkaufselement(currentVM);
currentLE = currentVE.getMat().lagerlagerelement();
```

Die Doppelung „lagerlager“ ist kein Schreibfehler. Es gibt Lagerelemente sowohl in einer OEL, wie auch in einer OEP, wo Verbrauchsmaterialien kurzzeitig gelagert bzw. vorgehalten werden. Mit lagerlagerelement() ist die Referenz auf das Lagerelement der OEL gewünscht.

Die Kommunikationsmoglichkeiten (Instanzen und Methoden) sind ausführlich in der Klassendokumentation des Package com.nerthus.proisys.pps.kernel (JavaDoc) beschrieben. Als Beispiele bietet sich der Quellcode vom TIPDialog-Bundle an.

3.4.2 Kommunikation unter Bundles (am Bsp. Auslastung/QADialog)

Der Dialog mit dem Quellenanteil ist ein typisches Beispiel, das hier beschrieben wird.

Der Quellenanteil ist ein Dialogparameter des Unternehmensmodell, womit im PPS-System das prozentuale Verhältnis der Materialquellen angegeben wird. Die Summe alle Quellenanteile muss 100% betragen.

Somit ist überall, wo ein Dialog mit sogenannten. Quellenparametern möglich ist, ein spezielles Werkzeug erforderlich, um die Restriktionen des Parameters Quellenanteil (nur bei mehrerer Quellen - sonst immer 100%) sicherzustellen.
Nun ist es bestimmt auch möglich, überall ein eigenes Werkzeug zu implementieren, aber eben nicht sinnvoll - systemweite Einheitlichkeit, Austauschbarkeit usw..

Zur Laufzeit des Bundles Auslastungsdialog benötigt man also den geladenen QADialog-Controller.
Den erhält man am günstigsten zum Zeitpunkt des ersten Aufrufes:

```
if(myQAContr == null) {
     myQAContr = (QAController) mainController.controllerWithKey(
                                      strings.getProperty("qacontroller")
                                 );
}
```

Um während der Programmierung die Methoden des Bundles verfügbar zu haben (Compilierung), wird auf den Mechanismus des implementierten Interface zurückgegriffen:

```
public interface QAController {
     void openQAPanel(double qavalue, int rPZA, NEElement sender);
}
```

Damit wird dann die programmierte Verwendung innerhalb des aufrufenden Bundles möglich:

```
if(myUModell.sourcecount(midx) > 1) { // mehr als eine Quelle?
     x = myAQQA.doubleValue();
     // Aufruf QA-Dialog
     myQAContr.openQAPanel((x/100), currentAPZA, currentAMD.pe);
     mainController.update(false);
}
```

3.5 Planungshorizonte

Innerhalb des NEController (nicht nur dort) sind folgende Instanzevariablen festgelegt:

```
int maxWeek;
int maxMonth;
int maxPZA;

int currentMaxWeek;
int currentMaxMonth;
int currentMaxPZA;
```

Einerseits hat es sich als sinnvoll erwiesen, zwei Planungsebenen (Monate/Wochen, Wochen/Tage ...) innerhalb eines Unternehmensmodells und dem PPS-System zu haben, um Grob- und Feinplanung in einem Zug machen zu können. Anderseits sind die Planungshorizonte der Planungsebenen (12 Monate und 100 Wochen...) flexibel einstellbar.

Ein Unternehmensmodell wird mit festen Planungshorizont für die einzelnen Planungsebenen aufgebaut.
Dieser Planungshorizont ist in kalender.conf folgendermaßen anzugeben:

```
Month = 12
Week = 12
```

für z.B. 12/12, wobei "Wochen" für die 1. Planungsebene und "Monate" für die 2. Planungsebene stehen.
Diese absoluten Grenzen des Unternehmensmodell werden durch die Variablen:

```
int maxWeek;
int maxMonth;
int maxPZA;
```

repräsentiert. Daraus ergibt sich, dass Tupel 0 ... 11 mit maxWeek = 12 zur 1. Planungsebene und Tupel 12 ... 23 mit maxPZA = 24 zur 2. Planungsebene gehört. (maxPZA = maxWeek + maxMonth)

Bei der Arbeit mit dem Unternehmensmodell kann man den maximal möglichen Planungshorizont jedoch einschränken, z.B. wenn man nur 6 Wochen/6 Monate planen möchte und somit Algorithmus (s.g. Randbehandlung) und Dialog auf 6 Wochen/6 Monate im Kontrollfeld begrenzt.

Diese Berechnungs- und Dialoggrenzen werden durch die Variablen:

```
int currentMaxWeek;
int currentMaxMonth;
int currentMaxPZA;
```

repräsentiert. Daraus ergibt sich, dass Tupel 0 ... 6 mit currentMaxWeek = 6 zur 1. Planungsebene und Tupel 12 ... 17 mit currentMaxPZA = 18 zur 2. Planungsebene gehört.

Es gilt also:

0 <= currentMaxWeek <= maxWeek 1. Planungsebene

maxWeek <= currentMaxPZA <= maxPZA 2. Planungsebene

wobei currentMaxPZA = maxWeek + currentMaxMonth und maxPZA = maxWeek + maxMonth ist.

Die Verwaltung und der Abgleich diesbezüglich wird durch NEMainController und NEController absolut sichergestellt.

Z.B. bei NEController

```
public void updateModell(UModell modell) {
     myUModell = modell;
     myKalender = myUModell.kalender();
     maxWeek = myKalender.maxweek();
     maxMonth = myKalender.maxmonth();
     maxPZA = maxWeek + maxMonth;
     currentMaxWeek = myUModell.weekPZA();
     currentMaxMonth = myUModell.monthPZA();
     currentMaxPZA = myUModell.maxPZA();
     this.update(NO);
}

public void update(boolean pzachanged) {
     if(pzachanged) {
          currentMaxWeek = myUModell.weekPZA();
          currentMaxMonth = myUModell.monthPZA();
          currentMaxPZA = myUModell.maxPZA();
     }
     return;
}
```

Beim Schreiben des eigenen BundleControllers sollte deshalb immer ein Aufruf der Supermethode in den überladenen Methoden zuerst stehen. Dann kann man die entsprechenden eigenen notwendigen Routinen programmieren.

Soll man die Berechnungs- und Dialoggrenzen im eigenen Bundle ändern können, so kann man natürlich mit einem Aufruf von **mainController.update(true)** die systemweite Aktualisierung einleiten.
Das Gleiche gilt natürlich auch, wenn man mit einem Bundle ein neues Modell lädt -
mit **mainController.updateModell(neuesModell);.**

3.6 Interaktiver und nicht interaktiver Dialog

Die Interaktivität des PPS-Systems wird durch die NEElemente (alle Klassen in Ableitung von NEElement) gewährleistet.

Wollen Sie in einem Bundle mehrfach Parameter ändern, so würde bei einem Dialog mittels Elementen jedesmal die Modellberechnung durchgeführt werden. Z.B. bei einem Parameterimport von 1000 Dialogparametern würde das Modell 1000 mal berechnet werden.

Um das zu umgehen kommuniziert man nicht über die Elemente in der Form:

Element.setXXX(value, <pza>); (interaktiver Dialog)

sondern prinzipiell über die Parameter (alle Klassen in Ableitung von NEParameter) in der Form:

Element.parameter(<pza>).setXXX(value); (nicht interaktiver Dialog).

Die Modellberechnung muß dann allerdings initiiert werden mittels:

myUModell.computeMod();

Als Beispiel für den nicht interaktiven Dialog dient das Bundle ParaManipulation. Dort werden modellweit Parameter gesetzt, wobei die Berechnung erst erfolgt, wenn alle Änderungen abgeschlossen sind.

```
...
for(b = beginnMonth; b < endeMonth; b++) {
      lp = (LParameter) le.parameter(b);
      lp.setReichw(pzl);
      lp.setMreichw(mrwl);
      if(hrwl >= 0) {
            lp.setHreichw(hrwl);
      } else {
            lp.setHreichw(-1);
      }
      ...
}
myUModell.computeMod();
mainController.update(false);
```

4 Abschließende Bemerkung

Eine Entwicklerdokumentation kann immer nur mehr oder weniger vollständig sein. Ein grundlegendes Übel liegt auch immer darin, dass der ursprüngliche, primäre Entwickler die Dokumentation schreibt und eigentlich überhaupt keine Vorstellung hat, was der Anwender seiner Kreation an Informationen braucht. Er schätzt Sachen als trivial ein, die dann den Anwendern Kopfschmerzen bereiten oder betrachtet etwas als sehr wichtig, das die Anwender ohnehin schon ahnten und wußten.

Ein paar Stunden beratender Beistand bewirken manchmal mehr als jede Dokumentation und das sei als Unterstützung ausdrücklich angeboten. Alle Informationen dazu stehen unter

www.nerthus.de

wo demnächst auch ein Entwicklerforum eingerichtet wird.

Trotzdem soll sich eine solche Dokumentation auch weiterentwickeln und verbessern, d.h. den Bedürfnissen der Anwender immer besser gerecht werden. Deshalb sind auch zum Thema Dokumentation alle Hinweise und Anregungen willkommen, auch wenn damit ein beratender Beistand vollkommen überflüssig werden sollte.

Potsdam, 03. Januar 2001

Ronald Bohlig
NeRTHUS Informationssysteme