SAP®

ABAP®-Programmierung

EDITION

Edition SAP®

Bernd Matzke
ABAP/4
3., erweiterte Auflage, 1999, ISBN 3-8273-1553-0

Thomas Teufel, Jürgen Röhricht, Peter Willems
SAP Prozessbibliothek:
SAP Prozesse mit Knowledge Maps analysieren und verstehen
1. Auflage, 2000, ISBN 3-8273-1603-0
SAP Prozesse: Vertrieb und Customer Service
1. Auflage, 2000, ISBN 3-8273-1602-2
SAP Prozesse: Planung, Beschaffung und Produktion
1. Auflage, 2000, ISBN 3-8273-1601-4
SAP Prozesse: Finanzwesen und Controlling
1. Auflage, 2000, ISBN 3-8273-1600-6

Gerhard Oberniedermaier
Vertriebslogistik mit SAP R/3
1. Auflage, 2000, ISBN 3-8273-1610-3

Gerhard Oberniedermaier, Marcus Geiß
SAP R/3 Systeme effizient testen
2., erweiterte Auflage, 2000, ISBN 3-8273-1561-1

Andreas Berthold, Ulrich Mende
SAP Business Workflow
2., aktualisierte Auflage, 2000, ISBN 3-8273-1687-1

Michael Ullrich
SAP R/3 Der schnelle Einstieg
1. Auflage, 2000, ISBN 3-8273-1646-4

Michael Umlauff
SAP R/3 Übungsbuch
1. Auflage, 2001, ISBN 3-8273-1788-6

Michael Umlauff, Walter Dirnhofer
ABAP Übungsbuch
1. Auflage, 2001, ISBN 3-8273-1789-4

Erich Dräger
Projektmanagement mit SAP R/3
2. Auflage, 2001, ISBN 3-8273-1707-X

Gerhard Oberniedermaier
Daten- und Dokumentenmanagement mit SAP R/3
1. Auflage, 2001, ISBN 3-8273-1761-4

EDITION SAP®

PEARSON Education

ADDISON-WESLEY

Rainer Riekert

ABAP®-Programmierung

Fortgeschrittene Programmiertechniken für ABAP

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

**Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.**

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

04 03 02 01

ISBN 3-8273-1754-1

Einbandgestaltung:	Barbara Thoben, Köln
Lektorat:	Christian Schneider, cschneider@pearson.de
Korrektorat:	Jutta Hoppe, München
Herstellung:	Elisabeth Egger, eegger@pearson.de
Satz:	mediaService, Siegen. Gesetzt aus der Palatino mit FrameMaker
Druck:	Media-Print, Paderborn

Printed in Germany

Für Robert Senger

Inhaltsverzeichnis

Vorwort

Bei meiner Arbeit in den letzten Jahren sah ich mich immer wieder mit Problemen konfrontiert, zu denen es keine oder nur unzureichende Dokumentation oder Literatur gab. Aufgrund der Schwierigkeiten, hilfreiche Informationen zu den Themen zu finden erwies sich die Realisierung der Aufgaben oft als sehr schwierig. Eine logische Konsequenz daraus war die Überlegung, das gewonnene Wissen für Kollegen festzuhalten, um das Wissen im Team zur Verfügung zu stellen. Der Aufwand, eine solche Dokumentation zu erstellen ist jedoch sehr hoch und das Tagesgeschäft lässt leider oftmals viel zu wenig Raum für diese, vermeindlich unproduktiven, Tätigkeiten. Und so kommt es, dass viele Programmierer immer wieder für die gleichen Probleme Lösungen erarbeiten müssen.

Dieses Buch setzt Kenntnisse in der Programmiersprache ABAP voraus. Es handelt sich hierbei also nicht in erster Linie um ein Lehrbuch für die Programmiersprache ABAP. Hier sei auf entsprechende Bücher wie z. B. das sehr gute Buch ABAP/4 von Bernd Matzke verwiesen.

Das Ziel dieses Buches ist ein anderes. Es soll die Programmiersprache ABAP in der Anwendung zeigen. Die Kapitel versuchen jeweils einen Aspekt der Programmierung in ABAP detailliert und praxisnah zu betrachten. Dies umfasst zum Teil konkrete Programmierprobleme wie z.B. die Programmierung mit den neuen Dialog-Controls in Kapitel 3. Andere Kapitel behandeln nicht so konkrete Themen, sondern zielen vielmehr darauf ab, über das reine Handwerk der Implementierung von Programmen hinaus zu gehen und Fertigkeiten zu vermitteln, die die Qualität Ihrer Programme steigern sollen. Als Beispiel hierfür sei Kapitel 2 erwähnt.

Die Kapitel dieses Buches bilden in sich abgeschlossene Einheiten, die in beliebiger Reihenfolge gelesen werden können. Ausgenommen hiervon sind die Kapitel 3 und 4, denen das gleiche Beispielprogramm zugrunde liegt.

Das erste Kapitel fällt im Vergleich zu den anderen Kapiteln etwas aus dem Rahmen. Es ist gedacht, demjenigen, der gerade Kenntnisse in ABAP erworben hat, aber noch keine eigenen Programme geschrieben hat, eine Hilfe an die Hand zu geben, wie man den zunächst leeren Bildschirm mit Leben füllen kann. Aber auch dem erfahreneren Programmierer kann das Kapitel sicherlich noch die eine oder andere Anregung zur Gestaltung seiner Programme geben.

Die Quelltexte der meisten Beispielprogramme dieses Buches finden Sie auf der beiliegenden CD. Da jedoch in manchen Kapiteln das zuvor erstellte Beispielprogramm weiterentwickelt wird, um neue oder andere Möglichkeiten aufzuzeigen, können nicht alle Programmversionen lückenlos zur Verfügung gestellt werden.

Sämtliche Programme wurden in einem R/3-System Release 4.6C entwickelt und getestet.

Da das Format der Transportdateien sich immer wieder ändert und in den meisten Fällen sehr schwierig ist, fremde Programme in das System einzuspielen, wurde darauf verzichtet, direkt importierbare Dateien für die Beispiele zur Verfügung zu stellen. Die Objekte der Entwicklungsumgebung, die nicht als Textdateien zur Verfügung gestellt werden können, müssen dabei vom Leser manuell erstellt werden. Anhand der Ausführungen in den jeweiligen Kapiteln sollte dies jedoch kein Problem darstellen.

Zum Schluss möchte ich noch Herrn Christian Schneider vom Verlag Addison-Wesley für seine Geduld bei der Erstellung dieses Buches danken. Ihnen, dem Leser, wünsche ich viel Spaß beim Lesen verbunden mit der Hoffnung, dass Ihnen viel mühsames Ausprobieren bei den dargestellten Themen erspart bleibt.

Rainer Riekert
Hannover im Mai 2001

1 Grundlagen zur ABAP-Programmierung

1.1 Allgemeines

Dieses Kapitel ist in erster Linie an die Leser gerichtet, die zwar die Syntax der Sprache ABAP kennen, aber erst wenige Programme geschrieben haben.

Die Ausführungen dieses Kapitels sollen nicht als Regeln verstanden werden, die vom Programmierer eingehalten werden müssen. Es handelt sich vielmehr um Anregungen, die sich aus der täglichen Arbeit mit R/3 und ABAP entwickelt haben und dem Neuling helfen sollen, sich schneller im System zurechtzufinden.

Sicher sind nicht alle hier dargestellten Anregungen für jeden Leser sinnvoll oder praktikabel. Vielmehr sollen die Ausführungen dem Leser helfen, seinen Programmierstil zu entwikkeln oder weiterzuentwickeln. Besonderes Augenmerk verdienen in diesem Punkt hausinterne Konventionen, die in fast jedem Betrieb, der R/3 einsetzt, in der einen oder anderen Form vorhanden sind.

1.2 ABAP-Reports

Beim Anlegen eines neuen Programms in ABAP muss im Dynpro »Programmattribute« (siehe Abbildung 1.1) der Typ des Programms festgelegt werden. Programme vom Typ »Report« (in älteren Releaseständen Typ »1«) werden auch ausführbare Programme genannt.

Reports können weiter unterteilt werden in logische Datenbanken, deren Zweck darin besteht, Daten aus der Datenbank zu extrahieren und anderen Programmen zur Verfügung zu stellen und Online-Reports (auch SQL-Reports genannt). Reports, die eine logische Datenbank implementieren, sind alleine nicht ablauffähig und bedürfen immer eines rufenden Programms, um sinnvolle Ergebnisse zu liefern.

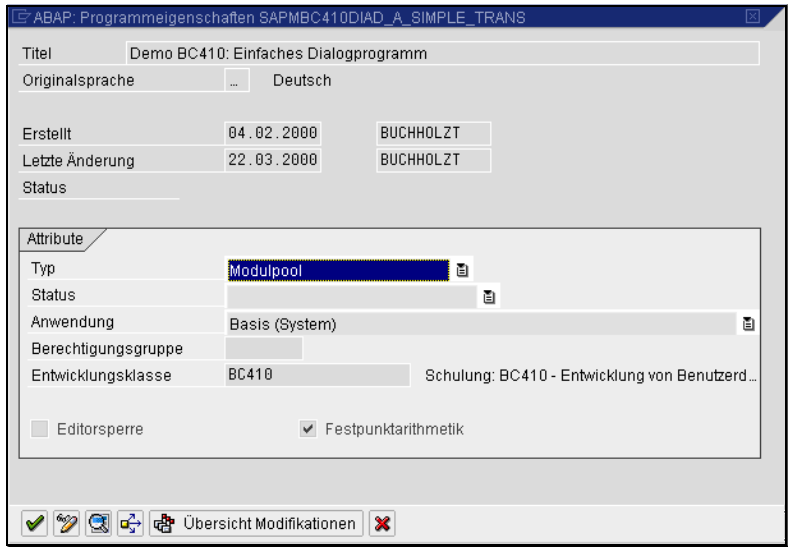

Abbildung 1.1
Programmeigenschaften (© SAP AG)

Es gibt drei Möglichkeiten, einen Report in R/3 zu starten. Bei der Definition einer Dialogtransaktion kann entweder ein Start-Dynpro eines Modulpools oder ein Online-Report definiert werden (siehe Abbildung 1.2). Im letzten Fall wird beim Start der Transaktion automatisch der Report geladen und zur Ausführung gebracht. Wenn eine solche Zuordnung nicht existiert, kann ein Online-Report über die Transaktion `SA38` direkt gestartet werden. Der wesentliche Unterschied dieser beiden Methoden liegt im Berechtigungswesen in R/3. Benutzer, die für die Transaktion `SA38` berechtigt sind, können alle Reports starten, die dies nicht durch einen expliziten `AUTHORITY CHECK` verhindern. Bei Dialogtransaktionen kann die Berechtigungsprüfung an jede einzelne Transaktion gebunden werden, was – bei Transaktionen, die Reports aufrufen – die explizite Berechtigungsprüfung am Anfang des Reports nicht ersetzen kann, sondern lediglich auf höherer Ebene ergänzt.

In der Transaktion `SA38` können Reports auch über ihren eindeutigen Namen gestartet werden. Dies ist insbesondere für Programme sinnvoll, denen kein Transaktionscode zugewiesen wurde. Aus ABAP-Programmen heraus können Reports über das `SUBMIT`-Kommando der Sprache ABAP gestartet werden.

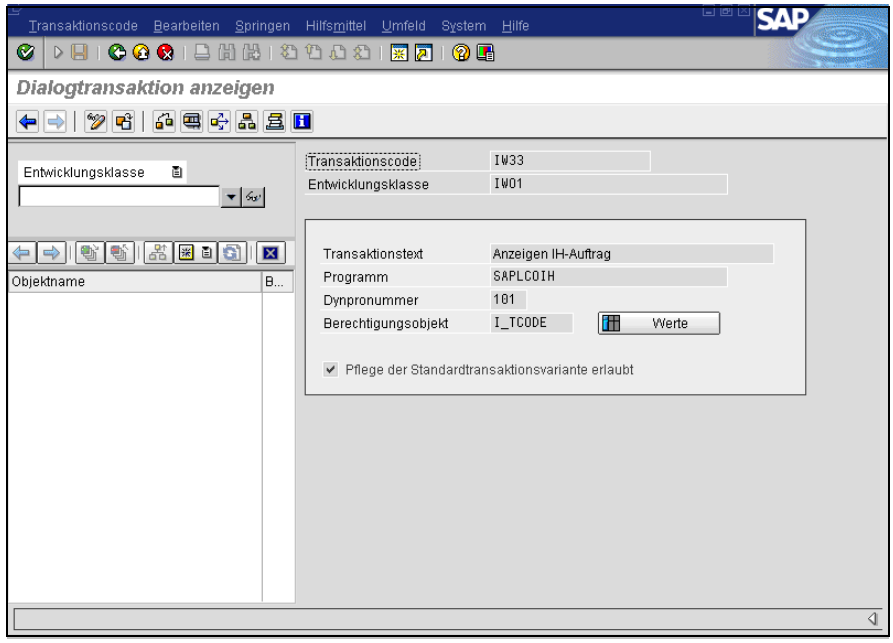

Abbildung 1.2
Eigenschaften eines Transaktionscode (© SAP AG)

1.2.1 Aufbau von Reports

Ein Report in ABAP beginnt immer mit der `REPORT`-Klausel. In dieser können neben dem Programmnamen weitere Angaben bzgl. der Listenbreite, der Höhe einer Seite und der Kopfzeilen im Online-Report bzw. dem Namen der im Report realisierten logischen Datenbank angegeben werden. Außerdem wird in der `REPORT`-Anweisung die standardmäßig verwendete Nachrichtenklasse festgelegt.

Nach der `REPORT`-Anweisung folgt der eigentliche Programmcode, der folgenden Aufbau hat:

```
REPORT reportname.
[Datendeklarationen}
[Definition des Selektionsbildschirmes]
[Ereignisblöcke]
[Unterprogramme]
```

Die dargestellte Reihenfolge für die einzelnen Blöcke ist nicht festgelegt, bietet sich aber an, um lesbaren Programmcode zu erzeugen. Generell gilt, dass Datendeklarationen, die nicht in Unterprogrammen erfolgen, globale Gültigkeit haben. Programmcode, der nicht in Unterprogrammen steht, entweder explizit oder implizit zu einem Ereignisblock oder Modul gehört, wird niemals ausge-

führt. Obwohl Unterprogramme generell an jeder Stelle im Programm stehen können, hat sich die Platzierung der Unterprogramme am Ende des Reports bewährt. Aus dem gleichen Grund ist die Definition des Selektionsbildschirms vor den Ereignisblöcken üblich.

Reports können sowohl in einer Datei[1] als auch in mehreren Dateien angelegt werden. Soll die Implementierung eines Reports auf mehrere Dateien verteilt werden, kann dies mit der INCLUDE-Anweisung in ABAP geschehen. Die Syntax der INCLUDE-Anweisung ist denkbar einfach.

```
INCLUDE <filename>.
```

Um ein neues Include zu erzeugen, wird die Include-Anweisung wie dargestellt in den Programmcode eingefügt und die neue Datei erzeugt, indem ein Doppelclick auf den Dateinamen durchgeführt wird (falls die Datei noch nicht existiert). Obwohl Includes als eigene Dateien mit eigenem Programmtyp realisiert werden, können sie ausschließlich im Kontext eines übergeordneten Programms verwendet werden. Hierbei ist die Anzahl der Programme, die ein Include einbinden, nicht beschränkt, Includes können also als Code-Pool für mehrfach verwendeten Programmcode verwendet werden.

INCLUDE-Anweisungen können an jeder beliebigen Stelle im ABAP-Programm stehen. Der Text der Include-Datei wird behandelt als stünde er an der entsprechenden Stelle in dem aufrufenden Programm, das heißt aber auch, dass sich der Code an der betreffenden Stelle nahtlos in den Code des Programmes einfügen muss. Wird beispielsweise ein Include mit der Implementierung eines Ereignisblocks innerhalb einer FORM-Routine eingebunden, führt dies unweigerlich zu Fehlern.

1.2.2 Datendeklarationen

In Anschluss an die REPORT-Klausel folgen die globalen Datendeklarationen, die wiederum unterschieden werden können in

- Tabellen
- Typen
- Konstanten
- Strukturen/interne Tabellen
- einfache Daten

1. Der in diesem Zusammenhang verwendete Begriff Datei ist nicht mit einer Datei auf der Ebene des jeweiligen Datei-Systems gleichzusetzen. Vielmehr werden ABAP-Programme und alle dazugehörigen Objekte der Programmierung in der R/3-Datenbank abgelegt. Der Begriff Datei ist hier also als Modularisierungseinheit zu sehen, deren Eigenschaften denen einer Datei auf Dateisystemebene sehr ähnlich sind.

Die Reihenfolge dieser Deklarationen ist beliebig. Die hier gezeigte Reihenfolge versucht die Daten mit absteigender Wichtigkeit im Programm darzustellen.

Tabellendeklarationen

Tabellendeklarationen werden in ABAP mit der TABLES-Anweisung durchgeführt. Diese ist für jede im Programm verwendete Datenbanktabelle notwendig. Weiterhin müssen Data Dictionary-Strukturen, die im Programm direkt verwendet werden sollen, auf diese Art deklariert werden. Werden Strukturen nur zur Beschreibung anderer Datenfelder mit der LIKE-Anweisung oder der INCLUDE-STRUCTURE-Anweisung benutzt, ist eine Deklaration mit der TABLES-Anweisung hingegen nicht notwendig.

Ein wichtiges Ziel der Programmierung sollte stets sein, den erzeugten Programmcode so lesbar wie möglich zu gestalten. Aus diesem Grund ist es sinnvoll, hinter die Tabellennamen als Kommentar zumindest den Beginn der Kurzbeschreibung der jeweiligen Tabelle oder Struktur zu setzten.

Besser lesbar als

```
TABLES: AUFK, CRHD, CRTX, EQUI, MARA, MAKT.
```

wäre demnach die Gestaltung des Codes nach folgendem Muster:

```
TABLES: AUFK,          " Auftragsstammdaten
        CRHD,          " Arbeitsplatz Kopf
        CRTX,          " Arbeitsplatz/ Fertigungshilf...
        EQUI,          " Equipment Stammdaten
        MARA,          " Materialstamm: Allgemeine Daten
        MAKX.          " Materialkurztexte
```

Es sollte immer bedacht werden, dass ein anderer Programmierer ein Programm lesen und verstehen muss, der nicht so tief in der jeweiligen Materie steckt, wie der ursprüngliche Autor. Auch erleichtert die hier gezeigte alphabetische Sortierung der Deklarationen ein schnelles Zurechtfinden in der Aufzählung – insbesondere dann, wenn sehr viele Tabellen verwendet werden.

Eigene Datentypen

In ABAP besteht die Möglichkeit, basierend auf den ABAP-Typen oder Data Dictionary-Elementen neue Datentypen innerhalb eines Programms zu deklarieren.

Die Deklarationen eigener Datentypen erfolgt mit der TYPES-Anweisung nach der Syntax:

```
TYPES <t> [ TYPE <type> | LIKE <dd-objekt> ].
```

Während die Form mit TYPE Bezug zu einem ABAP-Typ (egal ob elementar oder benutzerdefiniert) herstellt, referenziert die Variante mit LIKE ein Data Dictionary-Objekt.

Diese Datentypen sind ausschließlich in dem Programm verwendbar, in dem sie deklariert werden. Sollen eigene Datentypen über mehrere Programme hinweg verwendet werden, ist die Definition der Datentypen in einem so genannten TYPE-POOL (Programmtyp »T«) abzulegen. Man spricht hierbei von globalen Typdefinitionen, da diese an beliebigen Stellen im R/3-System verwendet werden können. Sollen in einem Programm globale Typdeklarationen verwendet werden, müssen diese zuerst über die ABAP-Anweisung

```
TYPE-POOLS <typepoolname>.
```

im Programm sichtbar gemacht werden. Die TYPE-POOLS-Anweisung sollte in diesem Fall im Bereich der TABLES-Anweisungen im Programm abgelegt werden. Näheres zu TYPE-POOLS kann in Abschnitt 1.4 nachgelesen werden.

Eigene Datentypen sind sinnvoll, wenn innerhalb eines Programms oder (bei globalen Typdefinitionen) über mehrere Programme hinweg mehrere Datenobjekte einen Sachverhalt beschreiben, für den keine Data Dictionary-Beschreibung existiert, oder der explizit nicht an ein Data Dictionary-Objekt gebunden werden soll. Sie sind insbesondere dann sinnvoll, wenn eine Entkopplung von den vordefinierten SAP-Datentypen gefordert ist. Durch Referenzierung von eigenen Datentypen können alle Datenobjekte, die den gleichen Typ referenzieren, bei Bedarf auf einmal geändert werden.

Das folgende Beispiel soll die Verwendung eigener Datentypen demonstrieren.

```
...
TYPES T_ANSWER(1) TYPE C.
...
CONSTANTS: BEGIN OF C_ANSWER,
             YES TYPE T_ANSWER VALUE 'J',
             NO  TYPE T_ANSWER VALUE 'N',
           END OF C_ANSWER.
...
DATA: G_ANSWER TYPE T_ANSWER.
...
IF G_ANSWER = C_ANSWER-YES.
  ...
ENDIF.
...
```

Durch die Verwendung des Datentyps T_ANSWER sowohl in der Definition der Konstanten als auch der des Datenfeldes G_ANSWER findet hier eine weitgehende Entkopplung von der Realisierung als einstelliges Charakter-Felds statt. Soll die Repräsentation auf einen anderen Datentyp verändert werden, kann dies einfach durch Änderung der Typ-Deklaration und der global gültigen Konstanten

realisiert werden. Eine Modifikation des eigentlichen Programmcodes ist nicht notwendig, bei häufiger Verwendung des Datentyps eine erhebliche Erleichterung.

Werden die TYPES- und CONSTANTS-Anweisungen in einer eigenen Typgruppe abgelegt, erfolgt hiermit eine vollkommene Entkopplung der Datentypen vom Programmcode.

> In diesem Programmbeispiel wurden für die Namensgebung der Programmobjekte bestimmte Regeln zugrunde gelegt. Näheres über die hier verwendeten Konventionen wird in Kapitel 1.4 dargestellt.

Konstanten

Bereits das kleine Beispiel im vorigen Abschnitt zeigt deutlich, wie sinnvoll die Verwendung von Konstanten in Programmen ist. Konstanten werden in ABAP genauso wie normale Datenobjekte entweder mit Bezug zu einem Typ oder mit Bezug auf ein Data Dictionary-Element definiert.

Im Programmcode sollten im Normalfall keine literalen Informationen verwendet werden. Stattdessen bietet sich die Verwendung von Konstanten an. Dadurch wird zum einen sichergestellt, dass für den gleichen Sachverhalt immer der gleiche Wert verwendet wird, zum anderen vermindert es den Aufwand, eine Einstellung im Programm zu ändern, erheblich.

Konstanten können entweder als einfache Datenobjekte oder – wie im Beispiel gezeigt – als Struktur definiert werden. Die Verwendung von Strukturen bietet sich bei alternativen Werten für einen Sachverhalt, wie es im abgebildeten Beispiel der Fall ist, an.

Im Normalfall werden Konstanten als globale Datenobjekt erzeugt, gleichwohl ist es möglich, Konstanten im lokalen Datenraum einer Unterroutine zu definieren, macht aber normalerweise keinen Sinn, da Unterprogramme in der Regel nicht so lang sind und der Rationalisierungseffekt gering wäre.

Strukturen

Den bisher beschriebenen Informationen folgen die Datendeklarationen für die globalen Datenobjekte des Programms. Diese werden mit der DATA-Anweisung der ABAP-Sprache deklariert. Für Strukturen lautet die Grundform dieser Anweisung

```
DATA: BEGIN OF <struktur>,
        <feldname> [ TYPE <typ> | LIKE <dd-objekt> ],
        ...
      END OF <struktur>.
```

Die Anzahl der Felder in einer Struktur ist beliebig. Ebenso können in einer Struktur sowohl Felder mit Typbezug als auch Felder mit Data Dictionary-Bezug deklariert werden. Die Deklaration der Struktur endet mit der Sequenz `END OF <struktur>`.

Existiert bereits eine Data Dictionary-Struktur mit dem gleichen Aufbau, wie es das Datenobjekt hat, kann das Datenobjekt auch durch die vereinfachte Schreibweise

```
DATA: <struktur> LIKE <dd-struktur>.
```

deklariert werden.

Soll eine interne Struktur den gleichen Aufbau wie eine bereits im Data Dictionary definierte Struktur oder Tabelle haben und noch weitere Felder hinzufügen, ist das folgende Konstrukt passend.

```
DATA: BEGIN OF <struktur>,
        <feld> [ TYPE <typ>  | LIKE <dd-objekt> ].
        INCLUDE-STRUCTURE <dd-struktur>.
DATA: END OF <struktur>.
```

Auf diese Weise können neue Felder beliebig mit Data Dictionary-Strukturen kombiniert werden. Es können sowohl mehrere neue Felder in der Struktur angelegt werden, als auch Strukturen aus dem Data Dictionary eingebunden werden.

Zu beachten ist lediglich, dass die Anweisung `INCLUDE-STRUCTURE` eine eigene ABAP-Anweisung ist. Die vorige `DATA`-Anweisung muss also mit einem Punkt abgeschlossen und die Folgende erneut mit dem Schlüsselwort `DATA` eingeleitet werden.

Interne Tabellen

Interne Tabellen werden genauso wie Strukturen deklariert. Auch hier stehen die drei oben beschriebenen Methoden zur Verfügung, werden jedoch um ein weiteres Schlüsselwort ergänzt.

Im ersten Fall wird die `BEGIN OF`-Anweisung um das Schlüsselwort `OCCURS`, gefolgt von der initialen Größe des zur Verfügung gestellten Datenbereichs ergänzt. In früheren Versionen von R/3 musste hier ein möglichst nahe an der erwarteten Tabellengröße liegender Wert angegeben werden. In neueren Releaseständen von R/3 spielt dies keine Rolle mehr und es kann generell der Wert 0 verwendet werden.

```
DATA: BEGIN OF <itab> OCCURS 0,
        <feldname> [ TYPE <typ> | LIKE <dd-objekt> ],
        ...
      END OF <itab>.
```

Bei der zweiten Variante muss beachtet werden, dass R/3 in diesem Fall standardmäßig keine Kopfzeile zur Tabelle definiert. Diese muss bei Bedarf explizit durch den Zusatz `WITH HEADER LINE` definiert werden. Die vollständige Anweisung zur Deklaration einer internen Tabelle mit Dictionary-Bezug lautet demnach

```
DATA: <itab> LIKE <dd-struktur> OCCURS 0 [ WITH HEADER LINE ].
```

In Release 4.0 sind umfangreiche Möglichkeiten bei der Definition von internen Tabellen hinzugekommen. So können interne Tabellen jetzt wie Datenbanktabellen mit einem Schlüsselbereich versehen oder automatisch sortiert abgelegt werden. Bei diesen Definitionen handelt es sich immer um eine Erweiterung der oben dargestellten Methoden. Die bisherigen Anweisungen zur Verwendung interner Tabellen funktionieren nach wie vor und werden jetzt unter dem Tabellentyp `STANDARD TABLE` zusammengefasst. Die neuen Funktionalitäten werden in den Tabellentypen `HASHED TABLE` bzw. `SORTED TABLE` definiert.

Einfache Datenfelder

An letzter Stelle des Deklarationsbereichs eines Programms stehen schließlich die Deklarationen der einfachen Datenfelder. Diese werden, so wie die internen Tabellen und Strukturen, mit der `DATA`-Anweisung der ABAP-Sprache deklariert.

Die Syntax der Anweisung in diesem Fall lautet:

```
DATA <feld> [ TYPE <typ> | LIKE <dd-objekt> ].
```

Bei der Programmierung von Reports sollte die Verwendung von globalen Datenobjekten möglichst vermieden werden. Werden keine Unterprogramme zur Strukturierung des Programmcodes verwendet, bleibt jedoch keine andere Wahl als mit global gültigen Datenobjekten zu arbeiten. Bei Verwendung von Unterprogrammtechniken (siehe auch Kapitel 1.5) sollte darauf geachtet werden, dass Unterprogramme keinen Zugriff auf globale Datenobjekte durchführen. Nur wenn dies gewährleistet ist, können die Unterprogramme ggf. auch extern von anderen Programmen genutzt werden. Dies gilt jedoch nicht für den Zugriff auf Tabellen oder Konstanten. Da diese Datenobjekte nicht veränderbar sind, ist ein globaler Zugriff auch von Unterprogrammen unproblematisch.

1.2.3 Der Selektionsbildschirm

Bei der Programmierung von Reports muss nicht auf die Verwendung von Dynpros verzichtet werden. Standardmäßig hat ein Report ein Dynpro, das so genannte Standard-Selektions-Dynpro mit der vordefinierten Dynpro-Nummer `1000`. Bei Bedarf können beliebig viele andere Dynpros innerhalb eines Programms definiert werden. Zu beachten ist hierbei lediglich, dass die Dynpro-Nummer `1000` nicht verwendet werden darf und dass die Dynpro-Nummern

von Selektions-Dynpros sich nicht mit den Dynpro-Nummern von anderen, im Screen-Painter definierten Dynpros überschneiden darf.

Ein Selektions-Dynpro wird zwischen den Anweisungen

```
SELECTION-SCREEN BEGIN OF SCREEN <dynnr> [ AS WINDOW ].
```

und

```
SELECTION-SCREEN END OF SCREEN <dynnr>.
```

definiert. Werden diese Kommandos nicht verwendet, beziehen sich die Angaben implizit auf das Standard-Selektions-Dynpro mit der Nummer `1000`. Über den optionalen Zusatz `AS WINDOW` kann ein Selektions-Dynpro als modales Dialogfenster definiert werden. Warnungen und Meldungen, die während der Verarbeitung des Dynpros auftreten, werden in diesem Fall ebenfalls als modale Dialogfenster dargestellt.

Zwischen diesen beiden Anweisungen (falls vorhanden) wird der Aufbau des Dynpros beschrieben. Hierzu stehen die ABAP-Kommandos `PARAMETERS`, `SELECT-OPTIONS` sowie `SELECTION-SCREEN` zur Verfügung.

> Im Programmcode des Reports werden diese Dynpros mit dem ABAP-Kommando `CALL SELECTION SCREEN <Dynpro-Nummer>` aufgerufen. Dies gilt sowohl für die nichtmodalen Fenster als auch die modalen, in einem eigenen Fenster dargestellten Dynpros.
>
> Ein Selektions-Dynpro kann auch mit einer zuvor gespeicherten Variante aufgerufen werden. Hierzu wird die `CALL SELECTION SCREEN`-Anweisung folgendermaßen erweitert:
>
> ```
> CALL SELECTION SCREEN <dynnr> USING SELECTION-SET <variante>.
> ```
>
> Wenn Reports mit der `SUBMIT`-Anweisung gestartet werden, kann das zu verwendende Selektions-Dynpro als Parameter an die `SUBMIT`-Anweisung angefügt werden. Reports mit mehrere Selektions-Dynpros werden in der folgenden Form gestartet:
>
> ```
> SUBMIT <report> USING SELECTION-SCREEN <dynnr>.
> ```
>
> In den Ereignisblöcken kann das aktuelle Selektionsbild über die Systemvariable `SY-DYNNR` ausgewertet werden.

Datenfelder mit der PARAMETERS-Anweisung darstellen

Einfache Datenfelder können auf Selektionsbildschirmen mit der `PARAMETERS`-Anweisung dargestellt werden. Die Grundform dieser Anweisung lautet

```
PARAMETERS <parameter> [ TYPE <typ> | LIKE <dd-objekt> ].
```

Die Grundform dieser Deklaration ist identisch mit der Grundform der DATA-Anweisung. Gleichzeitig wird mit dieser Anweisung auch das Feld <p> als globales Datenobjekt angelegt.

Wenn ein Parameter mit Dictionary-Bezug deklariert wird, werden automatisch die Eigenschaften des entsprechenden Feldes für das Dynpro-Feld übernommen. Dies schließt auch die F1-Hilfe sowie die Eingabehilfe F4 ein. Soll auch die Werthilfeprüfung aus der Data Dictionary-Definition übernommen werden, ist der optionale Zusatz VALUE CHECK an die PARAMETERS-Anweisung anzufügen.

Über den Zusatz DEFAULT <wert> kann das Dynpro-Feld mit einem Wert vorbelegt werden. Diese Vorbelegung erfolgt vor der Ausführung aller Ereignisblöcke. Während der Verarbeitung der Ereignisblöcke kann dieser Defaultwert demnach wieder überschrieben werden.

Character-Felder mit der Länge 1 können auf dem Selektions-Dynpro auch als Ankreuzfelder bzw. Auswahlfelder dargestellt werden.

Die Definition als Ankreuzfeld erfolgt mit dem Zusatz AS CHECKBOX. Soll das Ankreuzfeld beim Starten des Dynpros vorselektiert sein, kann dies mit dem Zusatz DEFAULT 'X' erreicht werden.

Eine Gruppe von Auswahlfeldern wird auf dem Dynpro generiert, indem das Feld mit dem Zusatz RADIOBUTTON GROUP <gruppenname> deklariert wird. Alle Felder einer Auswahlfeld-Gruppe müssen hierbei den gleichen Gruppennamen verwenden. Auch hier gilt, dass das Feld, welches initial ausgewählt sein soll, mit dem Zusatz DEFAULT 'X' versehen werden muss.

Weitere Optionen sind bei der Verwendung der PARAMETERS-Anweisung die Optionen OBLIGATORY für Pflichtfelder, DECIMALS <n> für die Anzahl von Dezimalstellen bei Datenfeldern vom Typ »P«, NO-DISPLAY für Datenfelder, die auf dem Selektions-Dynpro nicht angezeigt werden sollen, sowie MEMORY ID <pid> für Felder, die aus dem SPA/GPA-Memory vorbelegt werden sollen, bzw. das Memory aktualisieren sollen.

Selektionen

Während über die PARAMETERS-Anweisung lediglich Einzelwerte erfasst werden können, besteht bei SELECT-OPTIONS die Möglichkeit komplexere Wertabgrenzungen zu erfassen. Über die ABAP-Anweisung

```
SELECT-OPTIONS <option> FOR <feld>.
```

wird eine interne Tabelle mit dem in Tabelle 1.1 dargestellten Aufbau deklariert. Diese Tabelle kann in der WHERE-Klausel der SELECT-Anweisung direkt mit dem Operator IN verwendet werden.

Feldname	Typ	Inhalt
SIGN	CHAR 1	Kennzeichnet, ob die Werte dieser Tabellenzeile zur Treffermenge dazugehören (»I«) oder nicht dazugehören (»E«).
OPTION	CHAR 2	Selektionsoperator
LOW	<feld>	Untere Feldgrenze
HIGH	<feld>	Obere Feldgrenze

Tabelle 1.1
Struktur der internen Tabellen für SELECT-OPTIONS

Der Vergleichsoperator in derartigen Selektionstabellen kann die gleichen Werte annehmen, wie Vergleichsoperatoren in ABAP. Es handelt sich hierbei um EQ, NE, GT, LE, LT, CP, NP für einfache Vergleiche sowie BT und NB für Intervalle.

Über die zusätzliche Option NO INTERVALS kann festgelegt werden, dass für die Selektion lediglich Einzelwerte zulässig sind. Auf dem Selektions-Dynpro wird dann lediglich ein Eingabefeld für den Parameter angeboten und auch im Dynpro »Selektionsoptionen« ist es nicht möglich, Intervalle zu definieren.

Im Gegensatz zu Parametern, die auf dem Selektions-Dynpro als ein Feld dargestellt werden, werden Selektionen durch zwei Felder und eine zusätzliche Schaltfläche abgebildet.

Eine solche Tabelle kann – ohne Selektionsbildschirm – auch mit dem ABAP-Kommando RANGES deklariert werden. Die RANGES-Anweisung hat den Aufbau

```
RANGES <rangetab> FOR <feld>.
```

Gestaltung des Selektionsbildschirms

Während mit den Kommandos PARAMETERS und SELECT-OPTIONS Eingabefelder auf dem Selektions-Dynpro erstellt werden können, dient das Kommando SELECTION-SCREEN mit seiner Vielzahl von möglichen Parametern ausschließlich der Gestaltung der dargestellten Elemente.

Die hier dargestellte Aufzählung von Möglichkeiten erhebt keinen Anspruch auf Vollständigkeit, soll jedoch die wichtigsten Optionen zur Gestaltung von Selektionsbildschirmen aufzeigen.

Leerzeilen auf dem Selektions-Dynpro können mit der Anweisung

```
SELECTION-SCREEN SKIP [ <n> ].
```

erzeugt werden. <n> steht hierbei für die Anzahl Leerzeilen. Wird <n> nicht angegeben, wird als Default der Wert 1 angenommen.

Um eine horizontale Trennlinie auf dem Bildschirm zu erzeugen, verwendet man die Anweisung

```
SELECTION-SCREEN ULINE [ [ / ] <pos>(<len>) ] [ MODIF ID <key> ].
```

Für die Position der Linie gelten die gleichen Parameter wie bei der WRITE-Anweisung in Listen. Mit dem optionalen Zusatz MODIF ID <key> kann auf die Linie zugegriffen werden, um dynamische Änderungen am Dynpro-Layout vorzunehmen (siehe Abschnitt 2.2.3).

Kommentare und Zeichenketten werden auf dem Dynpro mit der Anweisung

```
SELECTION-SCREEN COMMENT [ / ] <pos(len)> <text> [ FOR FIELD <feld> ]
                                                 [ MODIF ID <key> ].
```

erzeugt. Bei Kommentaren kann, wie bei horizontalen Linien, die Position innerhalb der Bildschirmzeile über die optionalen Parameter / <pos(len)> festgelegt werden. Soll der Text einen Bezug zu einem Eingabefeld auf dem Dynpro haben, kann dieser mit der Option FOR FIELD <feld> hergestellt werden.

Sollen mehrere Elemente in einer Dynpro-Zeile erscheinen, müssen diese zwischen die Anweisungen

```
SELECTION-SCREEN BEGIN OF LINE.
```

und

```
SELECTION-SCREEN END OF LINE.
```

platziert werden. Innerhalb einer Zeile kann die Position eines Elementes mit der Anweisung

```
SELECTION-SCREEN POSITION <pos>.
```

festgelegt werden.

Elemente auf einem Dynpro können zu Gruppen zusammengefasst werden. Diese Gruppen können bei Bedarf mit einem Rahmen eingefasst und dieser mit einem Titel versehen werden. Hierzu müssen die Elemente der Gruppe zwischen den Anweisungen

```
SELECTION-SCREEN BEGIN OF BLOCK <block> [ WITH FRAME [ TITLE <titel> ] ]
                                        [ NO INTERVALS ].
```

und

```
SELECTION-SCREEN END OF BLOCK <block>.
```

gesetzt werden.

So wie im Dynpro-Editor ist es auch auf Selektionsbildschirmen möglich, eine Drucktaste darzustellen und dieser einen Funktionscode zuzuweisen. Auch hierfür wird die SELECTION-SCREEN-Anweisung verwendet.

```
SELECTION-SCREEN PUSHBUTTON [ / ] <pos(len> <text>
                             USER-COMMAND <ucom> [ MODIF ID <key> ].
```

Innerhalb einer Zeile kann die Position und Größe des Buttons identisch mit den Optionen beim PARAMETERS-Kommando oder bei der Option ULINE beim SELECTION-SCREEN-Kommando definiert werden.

Wird die Drucktaste auf dem Dynpro gedrückt, wird sofort das Ereignis AT SELECTION-SCREEN (siehe Kapitel 1.2.4) ausgelöst. Welcher Button gedrückt wurde, kann aus dem Feld UCOMM der Systemstruktur SSCRFIELDS entnommen werden. Zu beachten ist, dass diese Struktur vor der Verwendung in einer TABLES-Anweisung deklariert werden muss.

Über die Option MODIF ID <key> kann die Schaltfläche dynamisch im Programm verändert werden (z.B. ausgeblendet).

Drucktasten auf dem Selektionsbild

Wie bei der Dialogprogrammierung auch, besteht in Reports die Möglichkeit, in der Drucktastenleiste am oberen Bilschirmrand eigene Schaltflächen darzustellen. SAP sieht hier bis zu fünf vom Programmierer zu belegende Drucktasten vor. Hierzu wird ebenfalls das ABAP-Kommando SELECTION-SCREEN in der folgenden Form verwendet:

```
SELECTION-SCREEN FUNCTION KEY <fkey>.
```

Der Wert <fkey> kann hierbei die Werte »1« bis »5« annehmen und identifiziert die einzelnen Schaltflächen. Im Programm müssen die Schaltflächen anschließend mit einer Beschriftung versehen werden. Hierzu stehen in der Struktur SSCRFIELDS die Felder FUNCTXT_01 bis FUNCTXT_05 zur Verfügung. Wird eine der Schaltflächen im Dynpro gedrückt, wird sofort das Ereignis AT SELECTION-SCREEN ausgelöst. Der Funktionscode des gedrückten Buttons wird im Feld SSCRFIELDS-UCOMM abgelegt und ist FC01 bis FC05 für die fünf möglichen Schaltflächen.

Das folgende, an sich vollkommen nutzlose Beispiel soll die Verwendung von Drucktasten auf dem Selektions-Dynpro verdeutlichen.

```
REPORT ZBTEST.

TABLES: SCCRFIELDS.

* Konstanten für die Funtionscodes
CONSTANTS: BEGIN OF C_FCODES,
             FC01  LIKE SSCRFIELDS-UCOMM  VALUE 'FC01',
             FC02  LIKE SSCRFIELDS-UCOMM  VALUE 'FC02',
             FC03  LIKE SSCRFIELDS-UCOMM  VALUE 'FC03',
           END OF C_FCODES.

PARAMETERS: P_UCOMM LIKE SSCRFIELDS-UCOMM.
```

```
SELECTION-SCREEN: FUNCTION KEY 1,
                  FUNCTION KEY 2,
                  FUNCTION KEY 3.

INITIALIZATION.
* Beschriftung der Drucktasten setzten
  SSCRFIELDS-FUNCTXT_01 = TEXT-001.
  SSCRFIELDS-FUNCTXT_02 = TEXT-002.
  SSCRFIELDS-FUNCTXT_03 = TEXT-003.

AT SELECTION-SCREEN.
* Auswerten der gedruckten Drucktaste
  CASE SSCRFIELDS-UCOMM.
    WHEN C_FCODES-FC01.
      P_UCOMM = C_FCODES-FC01.

    WHEN C_FCODES-FC02.
      P_UCOMM = C_FCODES-FC02.

    WHEN C_FCODES-FC03.
      P_UCOMM = C_FCODES-FC03.

  ENDCASE.

START-OF-SELECTION.
* Ausgabe des Inhalts des Feldes
  WRITE: / TEXT-004,
           P_UCOMM.
```

Im Beispiel wird auf dem Selektions-Dynpro ein Feld ausgeben, da ein leeres Selektions-Dynpro in R/3 nicht angezeigt wird. Über die Schaltflächen kann man das angezeigte Feld mit den Werten `FC01` bis `FC03` vorbelegen.

Die Beschriftung der Drucktasten wird über die Textelemente des Reports ermittelt. Nach dem Drücken der F8-Taste wird der `START-OF-SELECTION`-Block ausgeführt, der den aktuellen Wert des Eingabefeldes des Selektions-Dynpros ausgibt.

1.2.4 Ereignisblöcke - Lebenszyklus eines Reports

Die Verarbeitung von Reports erfolgt ereignisgesteuert, d.h. bei Eintreten bestimmter Ereignisse werden definierte Programmteile ausgeführt. Unterschieden werden kann hierbei zwischen Ereignissen, die während der Dialogverarbeitung des Reports und Ereignissen, die während der Listenausgabe des Reports ausgelöst werden.

Die Verarbeitung eines Reports beginnt immer mit dem Ereignis LOAD-OF-PROGRAMM (siehe Abbildung 1.3). Dieses Ereignis, es wird auch Programmkonstruktor genannt, wird ausgelöst, sobald der Report geladen wird. Es wird vor allen anderen Ereignisblöcken ausgeführt.

Diesem Ereignis folgt immer das Ereignis INITIALIZE. Es wird jedesmal beim Starten des Reports prozessiert. Im Verarbeitungsblock dieses Ereignisses können Steuertabellen gepuffert oder Variable initialisiert werden. Weiterhin können in diesem Bereich dynamische Änderungen am Selektions-Dynpro vorgenommen werden.

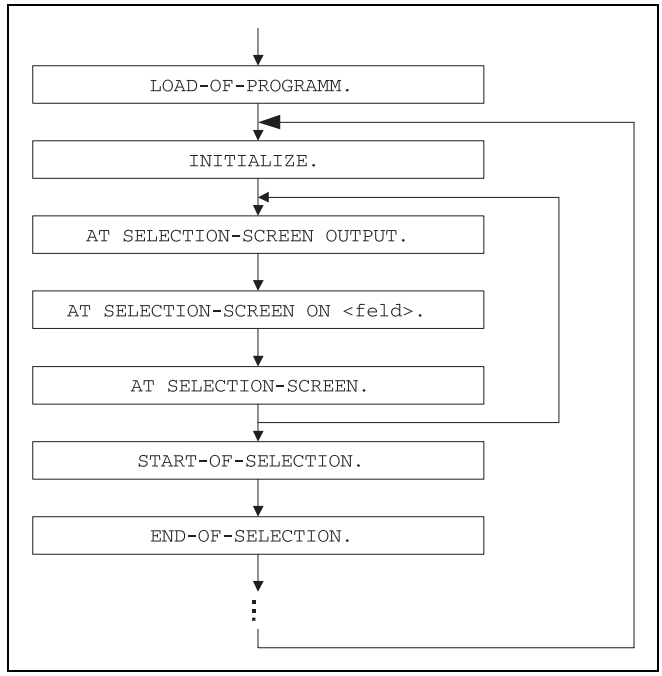

Abbildung 1.3
Ereignisfolge im Report - Dialogteil

Die gleichen Aufgaben können jedoch auch im Verarbeitungsblock des Ereignisses AT SELECTION-SCREEN OUTPUT, dem letzten Ereignisblock, der ausgeführt wird, bevor das Selektions-Dynpro prozessiert wird, durchgeführt werden. Der Unterschied zum Ereignis INITIALIZE liegt darin, dass AT SELECTION-SCREEN OUTPUT jedesmal prozessiert wird, bevor das Selektions-Dynpro ausgegeben wird. Dies ist beispielsweise nach jedem Betätigen der ↵-Taste der Fall.

Nach dem Abarbeiten des Ereignisblocks AT SELECTION-SCREEN OUTPUT wird das Dynpro auf dem Bildschirm des Benutzers angezeigt. Der Benutzer hat jetzt die Möglichkeit, Eingaben zu tätigen. Sobald ein Funktionscode ausgelöst wird, der

die Kontrolle wieder an den ABAP-Prozessor zurückgibt, wird zunächst für jedes Feld des Selektionsbildschirms das Ereignis AT SELECTION-SCREEN FIELD <feld> ausgelöst. Hier können Prüfungen auf der Feldebene vorgenommen werden. Wird in diesen Ereignisblöcken eine Fehlermeldung (E-Message) ausgegeben, ist lediglich dieses Feld des Dynpros eingabebereit. Im Anschluss an diese Ereignisse wird das Ereignis AT SELECTION-SCREEN ausgelöst. Hier werden die Eingaben des Benutzers für alle Felder plausibilisiert und bei Bedarf Fehlermeldungen ausgegeben. Fehlermeldungen in diesem Block bewirken, dass alle Felder des Selektions-Dynpros eingabebereit bleiben.

Ableitungen von Feldinhalten sind in der Regel besser im AT SELECTION-SCREEN OUTPUT-Block aufgehoben, da sie auch dann funktionieren, wenn im INITIALIZE oder auf anderem Wege Felder auf dem Dynpro vorbelegt werden. Wenn der Benutzer keinen Funktionscode mit Typ »E« (Exit-Command) ausgelöst oder die Selektion mit F8 gestartet hat, kehrt die Verarbeitung anschließend wieder in den AT SELECTION-SCREEN OUTPUT-Block zurück und prozessiert das Dynpro erneut.

Betätigt der Benutzer die F8-Taste, werden wie oben beschrieben zunächst die AT SELECTION-SCREEN-Ereignisblöcke abgearbeitet und anschließend der Verarbeitungsteil des Reports mit dem Block START-OF-SELECTION ausgeführt. Hier findet die eigentliche Listenverarbeitung eines Reports statt. Bei konsequenter Anwendung sollten hier lediglich die Daten für die Listenverarbeitung selektiert und im anschließend auszuführenden Ereignisblock END-OF-SELECTION ausgegeben werden. In der Praxis wird auf diese Teilung häufig verzichtet und die gesamte Programmlogik des Reports im START-OF-SELECTION-Ereignis realisiert.

Mit dem END-OF-SELECTION-Ereignisblock ist der dunkel verarbeitete Teil des Programms beendet. R/3 wechselt dann in den Listenprozessor und zeigt die mit WRITE und ähnlichen Routinen erzeugten Ausgaben auf dem Bildschirm des Benutzers an. Diese Anzeige erfolgt immer *nach vollständiger Abarbeitung* der hier genannten Programmteile. Die Darstellung von Zwischenergebnissen, wie es in anderen Programmiersprachen möglich ist, ist in ABAP nicht realisierbar.

Nachdem der Report abgearbeitet wurde und das Listenbild verlassen wird, wird der Report automatisch erneut vom System gestartet. Hierbei wird der LOAD-OF-PROGRAMM-Ereignisblock nicht erneut aufgerufen. Die Ausführung des Programms beginnt demnach mit dem INITIALIZE-Block. In diesem können jedoch in diesem Fall keine Dynpro-Felder belegt werden, da diese im anschließenden AT SELECTION-SCREEN OUTPUT-Ereignisblock mit den Werten des vorigen Programmlaufs versehen werden.

Im Anschluss an den Dialogteil des Reports erfolgt die Listenausgabe. Hier werden die mittels WRITE-Anweisung ausgegebenen Zeilen auf dem Bildschirm des Benutzers ausgegeben. Während der Listenausgabe und in der anschließenden Listenanzeige werden verschiedene Ereignisse ausgelöst, die es dem Programmierer ermöglichen, auf die Seitengestaltung Einfluss zu nehmen. So wird am

Anfang jeder Seite das Ereignis TOP-OF-PAGE ausgelöst. Hier kann ein zusätzlicher Seitenkopf oder Ähnliches ausgegeben werden.

Während der Listanzeige werden die Ereignisse AT LINE-SELECTION, AT USER-COMMAND sowie AT PFxxx prozessiert. Ersteres wird ausgelöst, wenn der Benutzer einen Doppelklick auf eine der Listenzeilen durchführt oder den Cursor auf die gewünschte Zeile setzt und F2 betätigt[2]. Das Ereignis AT USER-COMMAND wird ausgeführt, wenn ein im GUI-Status definierter Funktionscode ausgelöst wurde. Um diese Funktionalität zu nutzen, muss ein eigener Listenstatus definiert werden. Dies ist nicht notwendig, wenn die Funktionstasten mit dem Ereignis AT PF<xx> ausgewertet werden, wobei <xx> für die Funktionstasten von 01 bis 24 steht. SAP empfiehlt dieses Vorgehen jedoch lediglich für vorläufige Testversionen von Programmen. Sollen in einem produktiven Umfeld Funktionstasten in der Listenanzeige ausgewertet werden, wird das Vorgehen mit AT USER-COMMAND nahegelegt.

Werden diese drei Ereignisse in einem Programm ausgewertet, spricht man von interaktivem Reporting. Auch hier gilt, dass der Report, wenn die Listenanzeige verlassen wird, erneut gestartet und mit dem Ereignis INITIALIZE fortgefahren wird.

1.3 Dialogtransaktionen

1.3.1 Allgemeiner Aufbau

Im Gegensatz zu Report-Programmen ist die Verwendung von Include-Dateien bei Dialogtransaktionen nicht zu vermeiden. Neben dem TOP-Include, in dem alle globalen Deklarationen des Programms enthalten sein sollten, gibt es im Normalfall mindestens ein Include für PBO-Module und ein Include für PAI-Module. Weiterhin werden Unterprogramme in speziellen Includes abgelegt. SAP schlägt für diese Includes Namen vor, die nach Möglichkeit verwendet werden sollten. Wie erwähnt, wird für jeden Include-Typ mindestens eine Datei angelegt. Aus Gründen der Übersichtlichkeit und Strukturierung des Codes empfielt es sich jedoch, nicht zu sparsam mit Includes umzugehen. Legen Sie neue Includes für abgrenzbare Bereiche an, ggf. für jedes Dynpro eigene PBO- und PAI-Includes. Denken Sie dabei immer daran, dass später ein anderer Programmierer sich in Ihrem Code zurechtfinden muss, ohne Sie fragen zu können. Denken Sie bei der Programmgestaltung an Ihre heutigen und künftigen Kollegen.

2. Hierzu muss im aktuellen Status ein Funktionscode PICK existieren und der Funktionstaste F2 zugewiesen sein. Im standardmäßig verwendeten GUI der Listendarstellung ist dies der Fall.

1.3.2 Ablaufsteuerung und Module

In der Ablaufsteuerung von Dynpros können so genannte Module aufgerufen werden. Module sind spezielle Unterprogramme der Ablaufsteuerung. Module haben keine Schnittstelle für Parameter, können also ausschließlich auf globale Datenfelder zugreifen. Die Implementierung von Modulen sollte lediglich einfache Strukturen enthalten. Komplexe Logik ist besser in FORM-Routinen zu implementieren, die wiederum von den Modulen aufgerufen werden. Diese FORM-Routinen erhalten sämtliche benötigten Parameter in der Aufrufschnittstelle übergeben und nutzen keine globalen Datenfelder, um an anderer Stelle wiederverwendet werden zu können.

Die Namen von Modulen sollten immer die Dynpro-Nummer, in der sie verwendet werden, beinhalten. Soll auf mehreren Dynpros die gleiche Funktionalität ausgeführt werden, ist das passende Mittel, für jedes Dynpro eigene Module zu implementieren und die eigentlich auszuführende Logik in gemeinsam genutzte FORM-Routinen auszulagern.

In der Ablauflogik können Module auf drei verschiedene Arten aufgerufen werden.

Als einfache Aufrufe nach dem Muster:

```
MODULE <modulname> [ ON EXIT COMMAND ].
```

Modulaufrufe, die ohne Bezug zu einem Dynpro-Element stehen, werden auf jeden Fall während der Ablaufsteuerung aufgerufen. Bei Verwendung dieser Methode ist unbedingt auf die Ausführungen des nächsten Abschnitts bezüglich des Transports der Dynpro-Feldinhalte in die Programmvariable zu achten. Feldinhalte stehen erst nach Aufruf einer FIELD-Anweisung (siehe unten) für das jeweilige Feld im Programm zur Verfügung.

Wird während der Verarbeitung des Moduls eine Fehlermeldung (Message-Typ E) ausgegeben, ist kein Feld des Dynpros eingabebereit. Der Benutzer kann diesen Zustand nur durch ein Exit-Command (siehe unten) verlassen.

Wird der Zusatz ON EXIT COMMAND angegeben, wird das Modul als Erstes aufgerufen, wenn ein Funktionscode, der als Exit-Command (Funktionstyp E) definiert wurde, aufgerufen wird. Es muss hierbei beachtet werden, dass vor Aufruf des Moduls kein Transport von Feldinhalten in Programmvariablen stattfindet und dementsprechend in diesem Modul kein Zugriff auf die aktuellen Dynpro-Werte besteht. Weiterhin ist zu beachten, dass in der Ablaufsteuerung eines Dynpros lediglich ein Modul den Zusatz ON EXIT COMMAND haben darf.

Soll ein Modulaufruf mit Bezug zu einem Dynpro-Element realisiert werden, so wird die folgende Form verwendet:

```
FIELD <dynprofeld> MODULE <modulname> [ ON REQUEST | ON INPUT ].
```

In diesem Fall wird zunächst der Feldinhalt aus dem Dynpro-Feld in die Programmvariable transportiert und anschließend das Modul aufgerufen. Der optionale Zusatz ON REQUEST bewirkt, dass das Modul nur aufgerufen wird, wenn im Dynpro-Element eine Veränderung vorgenommen wurde, während der Zusatz ON INPUT bewirkt, dass das Modul immer aufgerufen wird, wenn der Feldinhalt des Dynpro-Elements nicht initial ist.

Wird innerhalb der Verarbeitung des Moduls eine Fehlermeldung ausgegeben (E-Message), sind alle Dynpro-Felder bis auf das in der FIELD-Anweisung angegebene Feld gegen Eingaben gesperrt. Lediglich das definierte Dynpro-Feld ist eingabebereit. Der Benutzer muss durch Veränderung der Eingabe in diesem Feld den Fehler beheben oder ein Exit-Command (siehe oben) auslösen. Andere Module werden in diesem Zustand nicht verarbeitet.

Sollen Prüfungen über mehrere Dynpro-Felder vorgenommen werden, so können diese in einer CHAIN-Anweisung gruppiert und gemeinsam ausgewertet werden. Zwischen den Anweisungen CHAIN und ENDCHAIN können mehrere FIELD-Anweisungen und ggf. mehrere MODULE-Aufrufe stehen.

```
CHAIN.
    FIELD: <feld1>,
           <feld2>,
           ....
           <feldn>.
    MODULE <modul1>.
    MODULE <modul2>.
    MODULE <modul3> [ ON CHAIN-REQUEST | ON CHAIN-INPUT ].
  ENDCHAIN.
```

Es werden nacheinander alle angegebenen Module aufgerufen. Zuvor werden die Inhalte der angegebenen Felder in die entsprechenden Programmvariablen transportiert. Der optionale Zusatz ON CHAIN-REQUEST entspricht dem Zusatz ON REQUEST bei einfachen FIELD-Anweisungen und bewirkt, dass das Modul nur aufgerufen wird, wenn sich der Inhalt von mindestens einem Feld der FIELD-Kette geändert hat. Auch ON CHAIN-INPUT entspricht dem Pendant bei einfachen FIELD-Anweisungen und bewirkt, dass das Modul aufgerufen wird, wenn der Feldinhalt mindestens eines Elements der Kette nicht initial ist.

Wird während der Verarbeitung eines der Module eine E-Message ausgeben, sind nur die in der CHAIN-Kette angegebenen Felder eingabebereit. Der Benutzer muss den Fehler entweder innerhalb dieser Felder beheben oder ein Exit-Command auslösen.

Ein Dynpro-Feld kann beliebig oft in der Ablaufsteuerung eines Dynpros verwendet werden. Dadurch können Plausibilisierungen und die dabei notwendige Feldsteuerung sehr fein programmiert werden.

Eine spezielle Anweisung in der Ablaufsteuerung bildet die LOOP-Schleife, die Verwendung findet, wenn auf dem Dynpro entweder Step-Loops oder Table-Views enthalten sind.

1.3.3 Feldinhalte in der Ablaufsteuerung

Die Feldinhalte aus dem Dynpro werden nicht unmittelbar beim Ereignis PAI in die entsprechenden Datenfelder im ABAP übernommen. Die Übertragung eines Feldinhaltes findet explizit bei einer entsprechenden FIELD-Anweisung und implizit am Ende der Ablaufsteuerung des PAI statt. Vor diesen Zeitpunkten entsprechen die Inhalte der Programmvariablen den Werten vor dem Ereignis PAI und nicht den Werten, die im Dynpro sichtbar sind.

Wenn sichergestellt werden soll, dass die Dynpro-Daten zu einem Zeitpunkt in die ABAP-Variable transportiert sind, müssen für alle Felder explizite FIELD-Anweisungen gegebenenfalls ohne Modul-Aufruf implementiert werden.

Eine weitere Möglichkeit, Daten aus dem Dynpro explizit zu übernehmen oder vor Ablauf des PBO im Dynpro zur Anzeige zu bringen, besteht in der Nutzung von zwei Funktionsbausteinen, die SAP zu diesem Zweck zur Verfügung stellt. Es handelt sich um den Funktionsbaustein DYNP_VALUES_READ zum Auslesen von Dynpro-Inhalten und DYNP_VALUES_UPDATE zum sofortigen Setzen von Dynpro-Feldinhalten. SAP hat diese beiden Funktionsbausteine – besonders DYNP_VALUE_READ – für die Verwendung in Werthilfen, d.h. PROCESS ON VALUE-REQUEST-Blöcken vorgesehen. Im normalen PAI kann auf die Verwendung dieses Funktionsbausteins leicht verzichtet werden, wenn ein expliziter Feldtransport mit der FIELDS-Anweisung durchgeführt wird.

DYNP_VALUES_READ

Mit dem Funktionsbaustein DYNP_VALUES_READ können Feldwerte aus dem Dynpro auch ohne FIELDS-Anweisung ausgelesen werden. Dieser Funktionsbaustein ist insbesondere dann nützlich, wenn eine eigene Werthilfe im Ereignisblock von PROCESS ON VALUE-REQUEST programmiert wird und diese Werthilfe abhängig von anderen Feldinhalten reagieren soll.

Die Schnittstelle des Funktionsbausteins hat untenstehenden Aufbau.

```
FUNCTION DYNP_VALUE_READ
   IMPORTING
      DYNAME
      DYNUMB
      TRANSLATE_TO_UPPER
   TABLES
      DYNPFIELDS
   EXCEPTIONS
      INVALID_ABAPWORKAREA = 01
```

```
INVALID_DYNPROFIELD  = 02
INVALID_DYNPRONAME   = 03
INVALID_Dynpro-Nummer = 04
INVALID_REQUEST      = 05
NO_FIELDDESCRIPTION  = 06
UNDEFINED_ERROR      = 07.
```

Im Parameter DYNAME wird der Name des Modulpools, in dem das betreffende Dynpro definiert wurde, übergeben. Das Feld DYNUMB enthält die Nummer des Dynpros im jeweiligen Modulpool. Mit dem Parameter TRANSLATE_TO_UPPER kann festgelegt werden, dass der Inhalt der übertragenen Felder in Großbuchstaben konvertiert wird. Schließlich wird dem Funktionsbaustein eine interne Tabelle übergeben, die beim Aufruf des Funktionsbausteins lediglich die Namen der zu übertragenden Felder enthält und nach erfolgreichem Aufruf auch die Werte der jeweiligen Felder sowie ein Flag, ob das Feld auf dem Dynpro eingabebereit ist oder nicht. Dieser Tabellenparameter hat den Aufbau der Dictionary-Struktur DYNPREAD, wie in Tabelle 1.2 dargestellt.

Feldname	Datentyp	Bedeutung
FIELDNAME	CHAR 132	Name des Dynpro-Feldes
STEPL	INT	Loop-Zeile
FIELDVALUE	CHAR 132	Inhalt des Dynpro-Feldes
FIELDINP	CHAR 1	Flag, ob Feld eingabebereit ist

Tabelle 1.2
Aufbau der Struktur DYNPREAD

Sollen mit dieser Funktion Feldinhalte aus einem Step-Loop oder einem Table-View ausgelesen werden, muss zunächst durch einen Aufruf der Funktion DYNP_GET_STEPL die gewünschte Loop-Zeile gesetzt werden, bevor mit DYNP_VALUE_READ der Feldinhalt dieser Zeile ausgelesen werden kann. Die Feldnamen des Parameters DYNPFIELDS enthalten keinen Index oder Ähnliches, um eine bestimmte Loop-Zeile zu bestimmen.

Die Aufruf-Schnittstelle dieses Funktionsbausteins ist folgendermaßen aufgebaut:

```
FUNCTION DYNP_GET_STEPL
```

DYNP_VALUES_UPDATE

Mit dem Funktionsbaustein DYNP_VALUE_UPDATE können Feldinhalte in einem Dynpro vor dem dazugehörgen PBO-Ereignis gesetzt werden. Dies macht vor allen Dingen dann Sinn, wenn mit einer Werthilfe mehrere Feldinhalte gesetzt

werden sollen (z.B. bei Arbeitsplätzen), oder wenn vor der Ausgabe einer Warnmeldung der neue Wert bereits in das betreffende Dynpro-Feld übertragen werden soll.

Die Aufrufschnittstelle des Funktionsbausteins ähnelt der des Funktionsbausteins DYNP_VALUE_READ, lediglich der Parameter TRANSLATE_TO_UPPER entfällt hier.

```
FUNCTION DYNP_VALUE_UPDATE
   IMPORTING
      DYNAME
      DYNUMB
   TABLES
      DYNPFIELDS
   EXCEPTIONS
      INVALID_ABAPWORKAREA = 01
      INVALID_DYNPROFIELD  = 02
      INVALID_DYNPRONAME   = 03
      INVALID_Dynpro-Nummer = 04
      INVALID_REQUEST      = 05
      NO_FIELDDESCRIPTION  = 06
      UNDEFINED_ERROR      = 07.
```

Der Tabellenparameter DYNPFIELDS referenziert auch bei diesem Funktionsbaustein eine interne Tabelle vom Typ DYNPREAD (siehe Tabelle 1.2). Der Inhalt des Feldes FIELDINP wird hierbei ignoriert.

Auch bei diesem Funktionsbaustein gilt, dass bei Verwendung von Step-Loops oder Table-Views vor Aufruf von DYNP_VALUE_UPDATE die gewünschte Loop-Zeile durch einen Aufruf der Funktion DYNP_GET_STEPL aktiviert werden muss (siehe oben).

1.3.4 Prüfungen und Feldsteuerung

Wenn während der Verarbeitung in der Ablaufsteuerung eine Fehlermeldung (E-Message) ausgegeben wird, werden alle Felder des Dynpros mit Ausnahme der gerade bearbeiteten Felder gegen Eingaben gesperrt. Lediglich das oder die Felder, die im aktuellen Kontext bearbeitet werden, sind eingabebereit. Aktueller Kontext meint bei einer einfachen FIELD-Anweisung das in der FIELD-Anweisung spezifizierte Feld, innerhalb einer CHAIN-ENDCHAIN-Kette alle durch FIELD-Anweisungen referenzierten Felder. Bei Ausgabe der E-Message steht der Cursor im ersten Feld des aktuellen Kontextes. Zu beachten ist, dass der Benutzer den aufgetretenen Fehler mit den Feldern des aktuellen Kontextes beheben können muss, da die einzige andere verbleibende Möglichkeit das Auslösen eines Exit-Commands ist, welches in der Regel jedoch das Dynpro ohne weitere Prüfung verlässt und keine Möglichkeit zur Bearbeitung von Feldinhalten zur Verfügung stellt.

Besonderes Augenmerk muss bei der Programmierung von Dialogprogrammen daher bei der Erstellung von Kontexten im Sinne der Feldsteuerung gelegt werden. Aufgrund der eingeschränkten Möglichkeiten bei der Feldsteuerung sollte immer der Grundsatz gelten, so wenige Felder wie möglich in einen Kontext zu integrieren, jedoch gleichzeitig so viele Felder wie notwendig, um die aufzutretenden Probleme zu beheben.

Soll beispielsweise auf einem Dynpro ein PM-Arbeitsplatz erfasst werden, der extern über den Arbeitsplatz plus dem dazugehörigen Werk dargestellt wird, könnte die Ablaufsteuerung folgenden Aufbau haben.

```
...
PROCESS AFTER INPUT.
...
  FIELD WERK MODULE CHECK_WERK ON REQUEST.

  CHAIN.
    FIELD: ARBPL,
           WERK.
    MODULE CHECK_ARBPL ON CHAIN-REQUEST.
  ENDCHAIN.
...
```

In der Implementierung des Moduls `CHECK_WERK` wird lediglich das Werk auf Gültigkeit überprüft. Die Werke sind in R/3 in der Tabelle `T001W` definiert. Primärschlüssel dieser Tabelle ist neben dem Mandanten ausschließlich das entsprechende Werk. Bei der Prüfung besteht daher keine Abhängigkeit zu einem anderen Dynpro-Feld.

Anders verhält es sich allerdings bei dem Feld `ARBPL`. Ein Arbeitsplatz wird in SAP in der Tabelle `CRHD` definiert. Der Primärschlüssel dieser Tabelle besteht jedoch nicht aus dem Arbeitsplatz, sondern aus der Objekt-Typ und dem Objekt-Schlüssel. Der Arbeitsplatz bildet zusammen mit dem Werk einen Sekundärschlüssel auf diese Tabelle. Es genügt demnach nicht, lediglich das Feld `ARBPL` in eine Feldprüfung aufzunehmen, obwohl im Beispiel auch der Wert des Werkes zum Zeitpunkt der Prüfung bereits ins ABAP übertragen wurde. Ist der Arbeitsplatz nicht im angegebenen Werk gültig, muss der Benutzer die Möglichkeit besitzen, sowohl Arbeitsplatz als auch Werk zu ändern, um einen gültigen Zustand herzustellen und die Feldprüfungen zu passieren. Aus diesem Grund wurde hier eine `CHAIN`-Anweisung über diese beiden Felder implementiert. Falls das eingegebene Werk gültig ist, der Arbeitsplatz jedoch in diesem Werk nicht existiert, stehen dem Benutzer beide Felder offen, um seine Eingaben zu korrigieren.

Wie bereits erwähnt, stehen dem Benutzer nach Auftreten einer Fehlermeldung nur die Felder des aktuellen Kontext zur Verfügung. Der Benutzer kann diesen Zustand nur verlassen, indem er über diese Felder einen gültigen Zustand herstellt oder ein Exit-Command auslöst. Daher sollte in jedem Dynpro zumindest

eine Abbrechen-Funktion als Exit-Command definiert werden, um dem Benutzer bei Bedarf zumindest das Abbrechen der Transaktion zu ermöglichen. Bedenken Sie hierbei jedoch, dass im Modul, in dem das Exit-Command verarbeitet wird, keinerlei Feldinhalte automatisch übernommen werden. Unabhängig davon, wo in der Ablaufsteuerung die Anweisung `MODULE <module> AT EXIT COMMAND` steht, wird dieses Modul vor jeder übrigen Verarbeitung des PAI verarbeitet. Unbenommen ist jedoch die oben beschriebene Möglichkeit, Dynpro-Feldinhalte über den Funktionsbaustein `DYNP_READ_VALUES` auszulesen und zu verarbeiten.

1.4 Benennen und verwenden von Variablen

Jede Variable in einem ABAP-Programm hat einen Gültigkeitsbereich. Man unterscheidet zwischen globalen und lokalen Variablen und hierbei wiederum zwischen statischen und nichtstatischen Variablen. Während globale Variablen – wie der Name schon sagt – in einem Report oder Modulpool globale Gültigkeit innerhalb der Instanz des Programms besitzen, d.h. von jeder Stelle im Programm aus beliebig zugreifbar sind, sind lokale Variablen lediglich in der `FORM`-Routine oder dem Funktionsbaustein, in der bzw. dem sie deklariert wurden, sichtbar und gültig.

Alle Variablen, die nicht innerhalb einer `FORM`-Routine oder einem Funktionsbaustein deklariert werden, haben globale Gültigkeit. Dies gilt auch für Variablen, die innerhalb eines Moduls (siehe unten) oder innerhalb der Implementierung eines Ereignisblocks deklariert wurden. `TABLES`-Anweisungen haben, egal wo im Programm deklariert, globale Gültigkeit. Aus diesem Grund ist es nicht ratsam, `TABLES`-Anweisungen innerhalb von Funktionsbausteinen oder `FORM`-Routinen zu stellen. Dies suggeriert lokale Gültigkeit, behindert aber die Programmierung unter Umständen erheblich, da für eine Data Dictionary-Struktur keine zwei Deklarationen in einem Programm stehen dürfen. `TABLES`-Anweisungen sollten daher stets am Anfang des Programms bzw. im TOP-Include platziert werden.

Statische Variablen sind Variablen, die ihren Wert über mehrere Aufrufe hin behalten. Es handelt sich hierbei immer um lokale Variablen. Sie werden analog zu nichtstatischen Variablen mit der Anweisung

```
STATICS feldname...
```

deklariert. Es gelten die gleichen Optionen wie bei der `DATA`-Anweisung.

Sinnvoll ist dieser Variablentyp einsetzbar, wenn eine Variable über mehrere Funktionsaufrufe hinweg ihren Wert behalten soll, jedoch aus Gründen des Designs nicht als globale Variablen deklariert werden sollen.

Bei der Namensgebung von Variablen kann es sinnvoll sein, den Gültigkeitsbereich der Variable im Namen deutlich zu machen. Dies kann beispielsweise

durch Verwendung von Präfixen bewerkstelligt werden. Tabelle 1.3 soll als Anregung für verschiedene Präfixe dienen, die in Programmen verwendet werden können.

Präfix	Verwendung
L_	lokale Variable
G_	globale Variable
LT_	lokale interne Tabellen
GT_	globale interne Tabellen
S_	lokale statische Variable
C_	Konstanten
T_	Typ-Deklarationen
P_	Parameter in FORM-Routinen, bzw. Variable, die über eine PARAMETERS-Anweisung erzeugt wurden.
SO_	Daten, die über eine SELECT-OPTIONS oder RANGES-Anweisung deklariert wurden.

Tabelle 1.3
Vorschlagswerte für Variablen-Präfixe

Obschon bei der Deklaration von Variablen die R/3-Datentypen verwendet werden können, ist es ratsam Variablen wo möglich mit Dictionary-Bezug zu deklarieren. Besonders für Felder, die auf Dynpros verwendet werden sollen, empfielt sich dieses Vorgehen, da auf diese Weise die F1-Hilfe, die Werthilfe F4 und etwaige Wertebereiche automatisch Anwendung finden.

Sollen interne Tabellen mit Dictionary-Bezug deklariert werden, kann genauso wie bei einfachen Datenfeldern die `LIKE`-Schreibweise verwendet werden. Zu beachten ist hier jedoch, dass nicht automatisch eine Kopfzeile für die interne Tabelle angelegt wird. Diese muss explizit durch den Zusatz `WITH HEADER LINE` bei der Tabellendeklaration angelegt werden.

Beispiel:

```
DATA GT_AUFTRAG LIKE AUFK OCCURS 0 WITH HEADER LINE.
```

Alternativ hierzu kann auch die Form:

```
DATA BEGIN OF GT_AUFTRAG OCCURS 0.
  INCLUDE STRUCTURE AUFK.
DATA END OF GT_AUFTRAG.
```

verwendet werden. Diese beiden Schreibweisen sind identisch, die erste ist jedoch aufgrund ihrer Kompaktheit zu bevorzugen, wenn keine weiteren Felder in die interne Tabelle eingefügt werden sollen. Beachtenswert ist hierbei, dass die INCLUDE-Anweisung eine eigene Anweisung ist. Die einleitende DATA-Anweisung muss daher unbedingt mit einem Punkt abgeschlossen werden. Für die END OF-Anweisung muss eine neue DATA-Anweisung eingefügt werden.

1.5 Unterprogramm-Techniken

In R/3 sind Unterprogramm-Techniken zur Strukturierung des Programmcodes vorgesehen. Im Einzelnen handelt es sich hierbei um FORM-Routinen im lokalen Kontext sowie Funktionsbausteine im globalen Umfeld.

Während FORM-Routinen in jedem ABAP-Programm verwendet werden können, müssen Funktionsbausteine in speziellen Modulen, den so genannten Funktionsgruppen definiert werden. Aber auch in anderer Hinsicht unterscheiden sich FORM-Routinen wesentlich von Funktionsbausteinen.

1.5.1 Unterprogramme als FORM-Routinen

FORM-Routinen werden direkt im jeweiligen Programm deklariert. Die Übergabeschnittstelle wird ebenfalls im ABAP-Code festgelegt. Da FORM-Routinen innerhalb eines anderen Programmes liegen, kann von FORM-Routinen auf alle globalen Variablen des jweiligen Programms zugegriffen werden.

Normalerweise werden FORM-Routinen nur in dem Programm verwendet, in dem sie implementiert werden. Sie können jedoch prinzipiell von jedem beliebigen ABAP-Programm extern aufgerufen werden. Dies macht jedoch nur Sinn, wenn die FORM-Routine keine globalen Variablen des Programms verwendet, denn diese können von Außen weder gesetzt noch nach Abschluss der Verarbeitung der Unterroutine ausgelesen werden.

Der Aufruf von FORM-Routinen erfolgt mit der PERFORM-Anweisung. Für programminterne Aufrufe gilt die Form:

```
PERFORM <name> [ TABLES   <tabellenparameter> ... ]
               [ USING    <in-parameter> ... ]
               [ CHANGING <in-out-parameter> ... ].
```

Analog dazu erfolgt der externe Aufruf einer FORM-Routine mit in der Form:

```
PERFORM <name>(programm) [ TABLES   <tabellenparameter> ... ]
                         [ USING    <in-parameter> ... ]
                         [ CHANGING <in-out-parameter> ... ].
```

Eine FORM-Routine selbst beginnt immer mit der Anweisung:

```
FORM <name> [ TABLES   <tabellenparameter> ]
            [ USING    <in-parameter> ]
            [ CHANGING <in-out-parameter> ].
```

Parameter, die im USING-Teil der Aufrufschnittstelle deklariert werden, sollten im Unterprogramm lediglich gelesen und nicht verändert werden. Tatsächlich funktioniert die Parameterübergabe in R/3 jedoch standardmäßig als Call-by-Reference. Daher besteht die Möglichkeit im Unterprogramm Parameter zu modifizieren. Diese Änderungen sind im aufrufenden Programm sichtbar. Hier unterscheiden sich die Parameter, die mit der USING-Klausel deklariert werden, nicht von den Parametern der CHANGING-Klausel, sie haben also lediglich erklärenden Wert. Sollen Parameter nicht veränderbar sein, müssen die Variablen als Call-by-Value deklariert werden. Diese Form ist lediglich bei den USING-Parametern sinnvoll und geschieht in der Form:

```
FORM <name> [ USING VALUE(<var>) [ LIKE <dd-variable> | TYPE <typ> ] ].
```

Folgende Grundsätze sollten bei der Verwendung von FORM-Routinen beachtet werden:

FORM-Routinen werden in der Regel innerhalb eines Modulpools oder Reports verwendet. Auch wenn Aufrufe von anderen Programmen aus möglich und in SAP-Programmen auch häufig anzutreffen sind, ist der übliche Geltungsbereich einer FORM-Routine lokal.

Falls FORM-Routinen extern aufgerufen werden sollen, kann dies nur geschehen, wenn bei der Implementierung darauf geachtet wird, dass die Routine keine Anforderungen an die Programmumgebung macht, d.h. im speziellen keinen Zugriff auf globale Variable durchführt. Konstanten, Typen und Tabellen können jedoch global verwendet werden, da sie unveränderliche Elemente des Programms darstellen. Aus dieser Forderung ergibt sich, dass sämtliche Daten, mit denen eine FORM-Routine mit der Außenwelt kommuniziert, über die Parameterschnittstelle der Routine definiert werden. Eine Ausnahme bilden hier lediglich Daten, die über andere Wege, wie z.B. das ABAP-Memory, im Unterprogramm aus einem systemglobalen Datenpool ermittelt werden.

Rückgabewerte aus FORM-Routinen sollten ausschließlich über die als CHANGING übergebenen Schnittstellenparameter oder die Variablen der Systemstruktur SY (z.B. SY-SUBRC) erfolgen.

1.5.2 Funktionsbausteine

Während die Deklaration und Verwendung von FORM-Routinen eher spontan erfolgen kann, verlangen Funktionsbausteine ein etwas formelleres Vorgehen. Funktionsbausteine verfügen, ähnlich wie FORM-Routinen, über eine wohldefinierte Aufrufschnittstelle, über die sowohl Parameter in den Funktionsbaustein hineingereicht, als auch Ergebnisparameter zurückgeliefert werden können.

Anders als FORM-Routinen, die im laufenden Programmquelltext an nahezu beliebiger Stelle eingefügt werden können, müssen Funktionsbausteine in speziellen ABAP-Programmen, den Funktionsgruppen (auch FUNCTION POOL), angelegt werden.

Funktionsgruppen sind ABAP-Programme, die keinen direkt ausführbaren Charakter haben, sondern lediglich Funktionsbausteine enthalten können. Funktionsgruppen können ebenfalls globale Datendeklarationen besitzen. Diese sind jedoch nur für die Elemente der Funktionsgruppe sichtbar und erhalten Ihren Wert solange das aufrufende Dialogprogramm/Report aktiv ist.

Da Funktionsbausteine in eigenen ABAP-Programmen abgelegt sind, haben sie – anders als interne FORM-Routinen – keine Möglichkeit, auf die globalen Daten, die im rufenden Programm deklariert wurden, zuzugreifen.

Im Gegensatz zu FORM-Routinen können Funktionsbausteine verwendet werden, um asynchrone Verarbeitungen zu bestimmten Zeitpunkten zu starten. Näheres hierzu wird in Kapitel 8 beschrieben.

Funktionsbausteine werden in R/3 in einer zentralen Funktionsbibliothek verwaltet. Aus diesem Grund müssen die Namen von Funktionsbausteinen systemweit eindeutig sein. Beim Aufruf eines Funktionsbausteins ist es nicht notwendig, die Funktionsgruppe, in der der gewünschte Funktionsbaustein definiert wurde, anzugeben.

Neben den von FORM-Routinen her bekannten Schnittstellenparametern können in der Funktionsbibliothek zu jedem Funktionsbaustein so genannte Ausnahmen (auch Exceptions) definiert werden, die einen fehlerhaften Abbruch der Verarbeitung des Funktionsbausteins anzeigen. Wird innerhalb eines Funktionsbausteins eine Exception ausgelöst, wird systemseitig ein Rollback durchgeführt und die Variablen der Aufrufschnittstelle auf den Wert vor dem Funktionsaufruf zurückgesetzt. Dies ist besonders bei der Verwendung von Funktionsbausteinen im Verbucher (siehe Kapitel 8) zu beachten. Löst ein Funktionsbaustein eine Exception aus, wird nicht nur der aktuelle Funktionsbaustein, sondern der gesamte Verbucherprozess abgebrochen und ein Rollback durchgeführt. Ausnahmen sind daher nicht geeignet, anwendungsbezogene Fehlersituationen anzuzeigen. Hierfür sollten stets Rückgabewerte der Aufrufschnittstelle genutzt werden.

Weiterhin kann in der Funktionsbibliothek zu jedem Funktionsbaustein eine Dokumentation abgelegt werden.

Als konsequente Weiterentwicklung des Funktionsgruppen-Konzeptes in R/3 ist die objektorierte Erweiterung von ABAP zu sehen. Während Funktionsbausteine lediglich Modularisierungseinheiten von ABAP-Code bilden und die Aufgabe von Funktionsgruppen die logische Zusammenfassung von Funktionsbausteinen ist, werden in ABAP-Objects umfassende objektorientierte Konzepte wie Vererbung, Kapselung und Polymorphie eingeführt. Eine umfassende Darstellung dieser Konzepte würde den Rahmen dieses Buches jedoch sprengen.

1.6 Typgruppen

Typgruppen sind spezielle ABAP-Programme, die, ähnlich wie Funktionsgruppen, für sich gesehen nicht alleine ablaufbar sind. Typgruppen dienen lediglich als Deklarationspool für Benutzerdatentypen und Konstantendefinitionen. Aus diesem Grund sind in Typgruppen nur TYPES- und CONSTANTS-Deklarationen erlaubt.

Ein Typpool wird innerhalb der ABAP-Workbench unter den Data Dictionary-Objekten eingeordnet. Eine Typgruppe kann im Object Navigator (Transaktion SE80) unter der Rubrik »Anderes Objekt« angelegt werden. In dem in Abbildung 1.4 dargestellten Dynpro muss der Name der neuen Typgruppe eingegeben werden. Dieser darf bis zu fünf Stellen lang sein und muss im Kundennamensraum liegen. Im Anschluss an das Einstiegsdynpro erscheint ein weiteres Dynpro, in dem eine Beschreibung zu der Typgruppe definiert werden muss. Bevor schließlich der Editor für die Typgruppe geöffnet wird, muss zuletzt noch die Entwicklungsklasse, in der die Typgruppe angelegt werden soll, festgelegt werden.

Sind alle diese Schritte durchlaufen, erscheint der ABAP-Editor. Jetzt können beliebig Typen und Konstanen deklariert werden. Zu beachten ist jedoch, dass alle innerhalb einer Typgruppe deklarierten Objekte den Namen der Typgruppe gefolgt von einem Unterstrich als Präfix im Namen tragen müssen. Die in Abschnitt 1.4 empfohlenen Präfixe müssen also noch einen weiteren Präfix erhalten.

Exemplarisch soll an dieser Stelle nochmals das Beispiel aus Abschnitt 1.2.2 (Eigene Datentypen) aufgegriffen werden.

Es wird eine Typgruppe ZBUCH definiert, die sowohl die Typ-Deklaration als auch die Definition der Programmkonstanten enthält. Der Quelltext der Typgruppe hat in diesem Fall folgendes Aussehen:

```
TYPE-POOL ZBUCH.

TYPES ZBUCH_T_ANSWER(1)  TYPE C.

CONSTANTS: BEGIN OF ZBUCH_C_ANSWER,
             YES  TYPE ZBUCH_T_ANSWER  VALUE 'J',
             NO   TYPE ZBUCH_T_ANSWER  VALUE 'N',
           END OF ZBUCH_C_ANSWER.
```

Das entsprechende Programmfragment aus Abschnitt 1.2.2 hätte dann folgenden Aufbau:

```
...
TYPE-POOLS ZBUCH.

DATA: G_ANSWER TYPE ZBUCH_T_ANSWER.
```

```
IF G_ANSWER = ZBUCH_C_ANSWER-YES.
  ...
ENDIF.
...
```

In diesem Programm ist es vollkommen unerheblich, wie der Datentyp ZBUCH_T_ANSWER tatsächlich definiert wurde. Dies wird zum einen durch die Deklaration des Datentyps, zum anderen aber auch durch die Definition der Konstanten an gleicher Stelle erreicht. Soll die technische Auslegung des Datentyps verändert werden, muss in diesem Fall lediglich der TYPE-POOL angepasst werden. Im Beispielprogramm wird nicht direkt Bezug zur technischen Repräsentation des Typs genommen. Es genügt demnach, die Konstante der Typgruppe dem neuen Datentyp anzupassen.

Die Lesbarkeit des abgebildeten Programmes kann nochmals gesteigert werden, wenn für Typgruppen kürzere Namen als der im Bespiel benutzte (ZBUCH) verwendet wird.

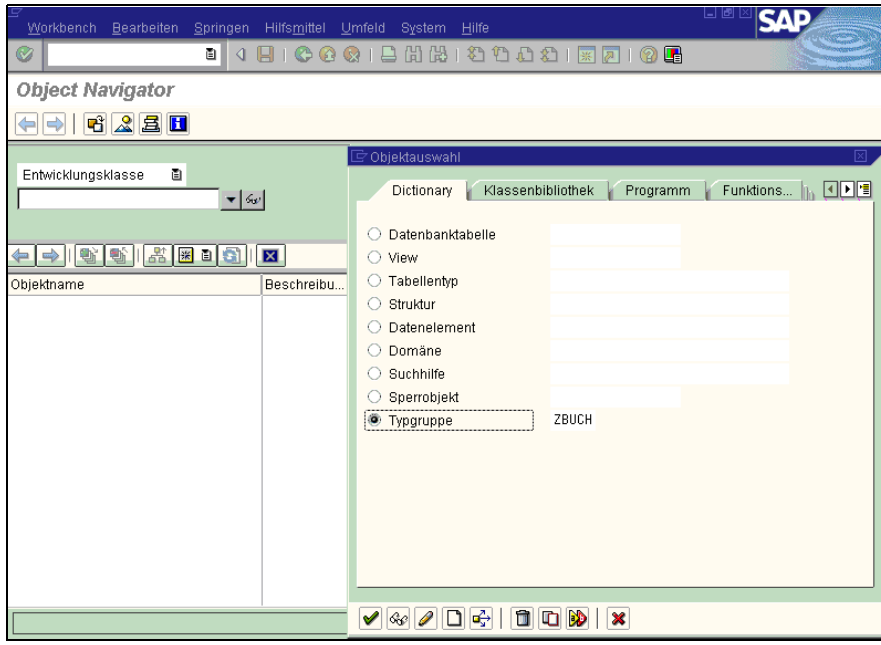

Abbildung 1.4
Einstieg in Typgruppe anlegen (© SAP AG)

Typgruppen können sinnvoll eingesetzt werden, wenn eigene Datentypen deklariert werden müssen, die in mehreren Programmen Verwendung finden. Sie sind insbesondere dann sinnvoll, wenn Daten über das ABAP-Memory ausgetauscht werden sollen und als Grundlage für die im ABAP-Memory abgelegten Daten keine Data Dictionary-Elemente verwendet werden können oder sollen.

1.7 Literale im Programmcode

Literale im Programmcode können unterteilt werden in Literale, die verwendet werden, um für den Benutzer sichtbar als Text angezeigt zu werden und Literale, die betriebswirtschaftliche Bedeutung innerhalb des Programmablaufs haben. Diese beiden Gruppen sollen im Folgenden einer kurzen Betrachtung unterzogen werden.

1.7.1 Textliterale

Es gibt verschiedene Arten von Literalen, die in Programmen vorkommen können. Text-Literale sind Strings oder Zeichenketten, die zur Ausgabe auf einem Dynpro oder in einer Liste verwendet werden. Aus Gründen der Übersetzbarkeit von Programmen ist es dringend angeraten, auf Text-Literale innerhalb des Programmcodes vollständig zu verzichten. Die Alternative zur Anweisung

```
WRITE 'Berechtigungen je Benutzer'.
```

könnte

```
WRITE TEXT-001.
```

lauten. Im zweiten Fall muss das Textelement `001` des Progamms entsprechend dem ersten Beispiel lauten. Eine Kombination beider Möglichkeiten stellt die Form

```
WRITE 'Berechtigungen je Benutzer'(001).
```

da. Ist das Textelemente `001` in der jeweiligen Sprache nicht definiert, kommt der im Programmcode abgelegte Text zur Ausgabe. Bei der Programmierung kann zunächst die zuerst genannte Form geschrieben werden. Bei einem Doppelclick auf das Textliteral erkennt R/3, dass hierzu noch kein Textelement definiert wurde und schlägt die automatische Erzeugung eines Textelements mit der nächsten freien Nummer vor. Wird hiervon Gebrauch gemacht, wird der Zusatz `(001)` anschließend automatisch in den Programmquelltext eingefügt.

1.7.2 Literale mit programmsteuernder Semantik

Anders als Textliterale, die lediglich zur Anzeige oder zum Ausdruck bestimmte, konstante Texte definieren, haben Programmliterale programminterne, zum Teil erhebliche steuernde Wirkung.

Anders als Textliterale sind sie als konstant innerhalb eines Programmkontextes anzusehen und sollten daher auch als Konstanten definiert werden. Konstanten können zusammen mit benutzerdefinierten Typen in Typgruppen (siehe Abschnitt 1.6) definiert werden und somit einen Wertevorrat für neue Typen bil-

den und die datentechnische Repräsentation von Typen von ihrer Verwendung in Programmen entkoppeln.

Die Verwendung von Konstanten ist im Programmbeispiel in Abschnitt 1.2.3 anschaulich dargestellt. Um die gültigen Funktionscodes der Drucktastenleiste nicht im Quelltext des Reports zu verstecken, werden am Beginn des Programms Konstanten innerhalb einer Struktur definiert. Über den Verwendungsnachweis können alle Stellen, an denen der Wert im Kontext ausgewertet wird, leicht nachvollzogen werden. Eine Möglichkeit, die bei Verwendung von Literalen – vor allem in großen Programmen mit mehreren Includes – nicht so einfach möglich wäre.

2 Parametrisieren von Programmen

2.1 Warum Programme parametrisierbar machen?

Der große Erfolg der Software R/3 auf dem Markt liegt darin begründet, dass *sämtliche* betriebswirtschaftlichen Abläufe in Unternehmen *beliebiger* Größe im System abgebildet werden können. Dies rührt nicht daher, dass die Prozesse in allen Unternehmen gleich sind oder im Rahmen der SAP-Einführung gleich gestaltet werden sollen, sondern vielmehr von der großen Flexibilität von R/3. Diese Anpassungsfähigkeit wurde von SAP über verschiedene Techniken realisiert. Im Einzelnen sind dies:

- Benutzerparameter
- Berechtigungen
- Customizing- und Steuertabellen
- Erweiterungskonzept

Über die genannten Möglichkeiten kann – wo vorgesehen - auf Programme im SAP-System Einfluss genommen werden, ohne den Programmcode selbst zu verändern. Diese Anforderung kann jedoch nicht nur aus den von SAP entwickelten Programmen heraus entstehen, auch in Kundenentwicklungen macht die Möglichkeit der Einflussnahme von außen großen Sinn. Hierbei stehen jedoch nur die ersten drei der oben genannten Möglichkeiten zur Verfügung, da seitens SAP keine Möglichkeit zur Verfügung gestellt wird, eigene Customer-Exits oder Menu-Exits im Rahmen des SAP-Erweiterungskonzeptes zu definieren. Da das Erweiterungskonzept von SAP im Zusammenhang dieses Kapitels nicht verwendbar ist, wird in diesem Abschnitt nicht weiter auf dieses Thema eingegangen. Dafür kann eine weitere Möglichkeit genutzt werden, den Programmlauf von außen beeinflussbar zu machen, indem Funktionscodes als Schalter definiert werden. Diese Methode eignet sich jedoch nur, um einfache

Schalter zu setzen, da einem Funktionscode keine Parameter angefügt werden können.

Die Erfahrung hat gelehrt, dass es stets einfacher ist, solche Einflussmöglichkeiten von vornherein in das Design eines Programmes einfließen zu lassen, als später den starr programmierten Programmcode dahingehend zu entflechten. Daher werden gute Programmierer stets versuchen, solche Möglichkeiten bei der Neuentwicklung von Programmen bereits an den Stellen vorzusehen, wo schon das Pflichtenheft wechselnde Anforderungen erkennen lässt oder andere Gründe (wie z.B. Debugging-Möglichkeiten) diesen Schritt nahelegen.

Wann welche Methode zur Parametrisierung angemessen ist, hängt von verschiedenen Faktoren ab und muss im Einzelfall entschieden werden. Im Folgenden sollen die vier skizzierten Möglichkeiten beispielhaft beschrieben werden. Dieses Kapitel soll einen Werkzeugkasten der Möglichkeiten anbieten und deren jeweilige Vor- und Nachteile gegeneinander aufzeigen. Es soll anregen, sich gedanklich mit dem Thema auseinanderzusetzen und auch eigene Lösungen zu finden. Eine vollständige Darstellung aller möglichen Methoden, um Programme parametrisierbar zu machen, ist daher hier nicht angestrebt.

2.2 Verwendung von Benutzerparametern

2.2.1 Was sind Benutzerparameter?

Die wohl einfachste Art, ein Programm beeinflussbar zu machen sind Benutzerparameter. SAP definiert diese als Weg, Daten transaktionsübergreifend zwischen Programmen zu übergeben. Benutzerparameter werden je Benutzer-Sitzung definiert, stehen also in allen Modi, die zu einer Anmeldung gehören, modusübergreifend zur Verfügung[1].

Benutzerparameter können je Benutzer im Benutzerstamm vordefiniert werden. Bei der Anmeldung werden diese aus dem Benutzerstamm gelesen und die Benutzerparameter für die neue Sitzung initialisiert. Während einer Benutzersitzung können die Inhalte der Benutzerparameter beliebig verändert werden. Dies geschieht entweder in Programmen über die jeweiligen ABAP Sprachelemente oder in Dynpros über den Schalter »SPA/GPA-Parameter setzen/lesen«[2]. Die aktuellen Werte für alle Benutzerparameter können nicht mit einer Standard-Transaktion angezeigt werden. Die Transaktion `SU52` zeigt lediglich die Initialwerte für die angegebenen Benutzerparameter an.

Ein Benutzerparameter besteht immer aus einem bis zu zwanzigstelligen Schlüssel (der so genannten Parameter-ID) und einem dazugehörigen Wert, der

1. siehe hierzu in »SAP-Bibliothek – BC – ABAP-Programmierung«, Version 4.6B, »Daten zwischen Programmen übergeben«
2. ebd., »Einstiegsbilder über SPA/GPA-Parameter füllen«

immer als Zeichenkette abgelegt ist. Parameter-Ids müssen vor ihrer Verwendung in Programmen oder Dynpros definiert werden. Dies geschieht entweder im Object Navigator (Transaktion SE80) oder über die Transaktion SM31 durch Erzeugung eines Eintrages in der Tabelle TPARAM, wobei hier der Transport des Tabelleneintrags in das Produktivsystem nicht vergessen werden darf. Die Parameter-Ids werden im Kundennamensraum angelegt.

In R/3 werden Benutzerparameter hauptsächlich als komfortabler, transaktionsübergreifender Zwischenspeicher genutzt. Sie sind die einzige Möglichkeit, Informationen über die verschiedenen Modi einer Anmeldung hinweg auszutauschen. Beispielsweise steht die Auftragsnummer eines neu erstellten Auftrags im SAP nach Beendigung der Standard-Transaktion in der Parameter-ID ANR zur Verfügung. In Dynpros kann definiert werden, ob Feldinhalte aus Parameter-Ids vorbelegt werden bzw. Feldeingaben den Wert von Benutzerparametern überschreiben.

An anderen Stellen verwendet jedoch auch SAP Benutzerparameter, um das Verhalten von Transaktionen zu beeinflussen. So steuert beispielsweise die Parameter-ID EKG die Feldattribute von Feldern in der Einkaufssicht einer Bedarfsanforderung (im Modul MM-Einkauf).

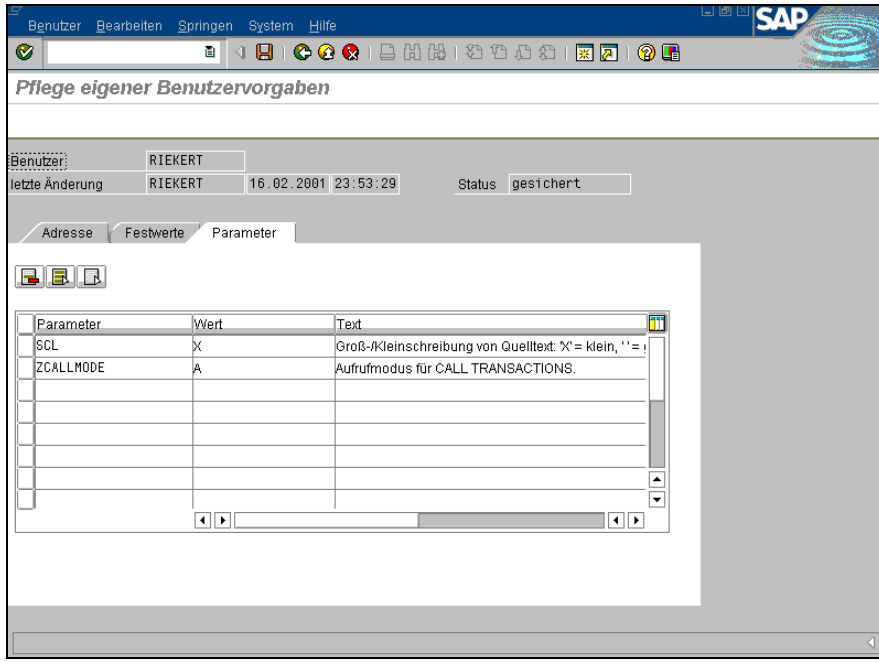

Abbildung 2.1
Benutzer pflegen: Parameter (© SAP AG)

Jeder Benutzer kann die Werte der Benutzerparameter nach der Anmeldung selbst festlegen. Abbildung 2.1 zeigt das Dynpro der Transaktion SU52 (»Benutzer pflegen: Parameter«). Durch Eintragen einer Parameter-ID in die linke Spalte und einen Wert in das entsprechenden Feld auf der rechten Seite können Initialwerte für Benutzerparameter festgelegt werden. Wie bereits erwähnt, können auf dieser Maske nicht die aktuellen Werte der Benutzerparameter und auch nicht alle gesetzten Benutzerparameter ersehen werden. Beim Speichern werden jedoch die hier dargestellten Benutzerparameter wieder auf den im Dynpro angezeigten Wert zurückgesetzt.

Eine entsprechende Berechtigung vorausgesetzt, können die Benutzerparameter für beliebige Benutzer in der Transaktion SU01 (Benutzer pflegen: Einstieg) angezeigt oder gepflegt werden.

2.2.2 Verwendung zur Parametrisierung

Auf Benutzerparameter wird in ABAP über die Befehle

```
SET PARAMETER ID <Parameter> FIELD <Field>.
```

schreibend und

```
GET PARAMETER ID <Parameter> FIELD <Field>.
```

lesend zugegriffen. Der Wert von <Parameter> muss hierbei in Hochkomma geschrieben werden. Eine Zuweisung von Werten ist ausschließlich über Variable möglich. Literale können – auch beim SET – nicht verwendet werden.

Ein Benutzerparameter ist eine Kombination aus einem Schlüssel und einem Wert, d.h. jeder Parameter-ID kann genau ein Wert zugewiesen werden, es können keine strukturierten Datentypen verwendet werden.

2.2.3 Beispiel zu Parametrisierungen über Benutzerparameter

Aufgrund ihrer Eigenschaften eignen sich Benutzerparameter besonders für Komforteinstellungen, die jeder Benutzer für sich ändern können soll. Beispielsweise könnte eine Transaktion so geschrieben sein, dass abhängig vom Wert eines Benutzerparameters bestimmte Felder eines Dynpros ein- oder ausgeblendet werden. Das folgende Programmfragment soll eine Lösung für dieses Beispiel skizzieren.

Im PBO können die Attribute der Felder auf einem Dynpro dynamisch verändert werden. Dies kann in einer Loop-Schleife über die vom System vordefinierte Tabelle SCREEN, die alle Controls auf dem aktuellen Dynpro enthält, realisiert werden. Zunächst muss hierzu im PBO des Dynpros ein zusätzliches MODULE aufgerufen werden.

```
...
PROCESS-BEFORE-OUTPUT
  ...
  MODULE 9000_OPEN_CLOSE_FIELDS.
...
```

Da auf den Wert von Benutzerparametern ausschließlich über Variable zugegriffen werden kann, eine in einem MODULE deklarierte Variable aber global gültig wäre (siehe auch Kapitel 1), wird die Implementierung der eigentlich Funktionalität in die FORM-Routine OPEN_CLOSE_FIELDS verlagert.

Das benötigte MODULE könnte demnach folgendermaßen programmiert sein:

```
...
MODULE 9000_OPEN_CLOSE_FIELDS.
  PERFORM OPEN_CLOSE_FIELDS.
ENDMODULE.
...
```

In der Form-Routine wird zunächst eine lokale Variable zur Aufnahme des Wertes aus dem Benutzerparameter definiert. Das eigentliche Auslesen des Benutzerparameters erfolgt in einer eigenen FORM-Routine GET_PARAMETER_VALUE. In den folgenden Beispielen dieses Kapitels wird lediglich die Implementierung dieser FORM-Routine ausgetauscht, die Implementierung des Moduls und der Routine OPEN_CLOSE_FIELDS bleibt im Folgenden unberührt.

Der Name des Benutzerparameters ist – wie bereits erwähnt – bis zu zwanzigstellig. Sollen eigene Benutzerparameter verwendet werden, so muss deren Name im Kundennamensraum liegen. In älteren Versionen von R/3 war die Länge der Parameter-Ids auf drei Stellen beschränkt.

```
...
FORM OPEN_CLOSE_FIELDS.
  DATA: L_MOREINFO(1) TYPE C.

* Benutzerparameter in lokale Variable auslesen
  PERFORM GET_PARAMETER_VALUE CHANGING L_MOREINFO.

* Zusatzfelder werden versteckt, wenn ZMI nicht gesetzt ist
  CHECK L_MOREINFO NE 'X'.

* Loop über Dynprofelder
  LOOP AT SCREEN.
*   Wenn Screen-Group 1 anzeigt, dass Feld betroffen ist
    IF SCREEN-GROUP1 = 'ZMI0'.
*     Feld ist kein Eingabefeld, kein Pflichtfeld und wird vesteckt
      SCREEN-INPUT    = 0.
      SCREEN-REQUIRED  = 0.
```

```
      SCREEN-OUTPUT    = 0.
      SCREEN-INVISIBLE = 1.
      MODIFY SCREEN.
    ENDIF.
  ENDLOOP.
...
ENDFORM.

* Die Routine Get_Parameter_Value liest den Parameter-ID aus.
FORM GET_PARAMETER_VALUE CHANGING P_MOREINFO TYPE C.
  GET PARAMETER ID 'ZMI' FIELD P_MOREINFO.
ENDFORM.
...
```

Im Anschluss an das Auslesen des Benutzerparameters wird überprüft, ob die zusätzlichen Felder ausgeblendet werden sollen oder nicht. Dieses Beispiel geht davon aus, dass im Dynpro-Editor alle Felder als sichtbar gekennzeichnet wurden und in diesem Schritt alle unerwünschten Felder ausgeblendet werden. Gleichwertig zu dieser Lösung wäre es, die betroffenen Felder im Dynpro-Editor auszublenden und nur auf Wunsch einzublenden. Für Online-Reports gelten diese Aussagen analog.

Die Entscheidung, welche Felder betroffen sind, wird im Beispiel über das Feld `GROUP1` der Struktur `SCREEN` getroffen. Ein Feld wird jedoch nur dann ausgeblendet, wenn es nicht als Eingabefeld (`INPUT`) und nicht als Pflichtfeld (`REQUIRED`) gekennzeichnet ist. Zusätzlich muss das Kennzeichen Ausgabefeld (`OUTPUT`) entfernt werden.

Um die Veränderung wirksam zu machen, muss zuletzt der Tabelleninhalt gespeichert werden. Dies geschieht mit der Anweisung `MODIFY SCREEN`.

Auf diese Weise kann durch Setzen des Benutzerparameters `ZMI` verhindert werden, dass bestimmte Felder auf einem Dynpro zur Laufzeit ausgeblendet werden.

2.3 Anpassung über Berechtigungen

Um unberechtigte Zugriffe auf die betriebswirtschaftlichen Daten eines R/3-Systems zu verhindern, existiert im System ein ausgefeiltes, auf Objekten basierendes Berechtigungswesen. Die Konzepte des Berechtigungswesens können jedoch nicht nur für diese Zwecke, sondern auch zur Parametrisierung von Programmen verwendet werden. Aufgrund des in der Regel hohen administrativen Aufwands, der bei der Zuweisung von Berechtigungen – zurecht – getätigt werden muss, wird diese Art der Parametrisierung jedoch in der Regel nur für wesentliche und sensible Programmfunktionalitäten gewählt werden.

2.3.1 Berechtigungswesen in R/3

Grundsätzlich gilt im R/3-System, dass ein Benutzer für nichts berechtigt ist, es sei denn, er oder sie hat eine Berechtigung explizit zugewiesen bekommen. Diese Berechtigungen und das Prüfen auf das Vorhandensein der entsprechenden Berechtigungen ist Gegenstand des Berechtigungswesens in R/3.

Definition von Berechtigungen

Die Architektur des Berechtigungswesens in R/3 ist mehrstufig. Grundlage des Konzeptes bilden so genannte Berechtigungsobjekte. Neben einer Beschreibung besteht ein Berechtigungsobjekt aus bis zu 10 so genannten Berechtigungsfeldern (siehe Abbildung 2.2) und bilden somit eine Schablone für alle konkreten Ausprägungen des Berechtigungsobjektes, den Berechtigungen. Die Felder eines Berechtigungsobjekts müssen hierbei immer Felder des Data Dictionaries referenzieren. In den Berechtigungen können für jedes Feld des zugrunde liegenden Berechtigungsobjeks Einzelwerte oder Wertebereiche definiert werden, für die die Berechtigung gilt. Die Berechtigungsobjekte werden organisatorisch zu so genannten Berechtigungsklassen zusammengefasst. Diese bilden lediglich eine Klammer, um die einzelnen Berechtigungsobjekte zu gruppieren. Funktional kommt den Berechtigungsklassen keine Bedeutung zu.

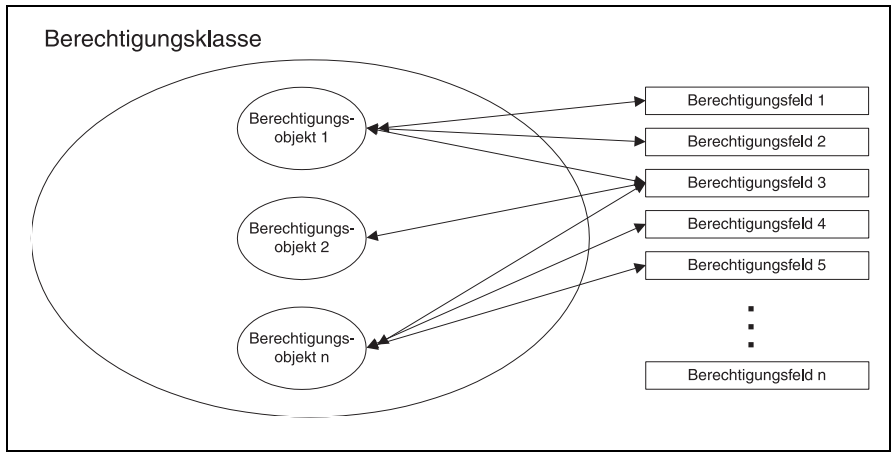

Abbildung 2.2
Zusammenfassung von Berechtigungsobjekten in Klassen.

Beispiel:
Das Berechtigungsobjekt `ZPARAM` soll ein Feld `ZTCODE` mit Bezug zur Dictionary-Domäne `TCODE` beinhalten. In einer konkreten Berechtigung könnte dieses Feld beispielsweise mit den Einzelwerten `ZFLIGHTGUI` und `ZFLIGHTSRV`, den Transaktionscodes von Dialogtransaktionen belegt werden. Zusätzlich könnte auch der

Wertebereich ZFLIGHT*, welcher ebenfalls mindestens die beiden genannten Transaktionen enthält, zugewiesen werden.

Diese Berechtigungen werden in so genannten Berechtigungsprofilen zusammengefasst. Diese Profile beinhalten – meist aufgabenbezogen – eine Reihe von Berechtigungen, die als Einheit einem oder mehreren Benutzerstammsätzen zugewiesen werden können. So könnten beispielsweise alle Berechtigungen, die benötigt werden, um Benutzerstammsätze zu bearbeiten, einem Profil zugeordnet werden, welches wiederum allen R/3-Benutzern, welche die Benutzeradministration in ihrem Aufgabenbereich haben, zugewiesen wird (siehe Abbildung 2.3). Eine Berechtigung, die einem Berechtigungsobjekt mit zugeordneten Wertebereichen für jedes Feld entspricht, kann hierbei in mehrere Profile aufgenommen werden.

Auf diesem Wege wird im System beschrieben, welche Berechtigungen ein Benutzer hat und wie diese technisch abgebildet werden. Die tatsächlichen Berechtigungen sind hierbei die Summe der Berechtigungen aller Profile, die dem Benutzerstammsatz zugeordnet sind.

Auf der anderen Seite besteht das Berechtigungwesen aus den Elementen, mit denen zur Programmlaufzeit festgestellt werden kann, ob der aktuelle Benutzer die, für eine Aktion benötigten, Berechtigungen hat.

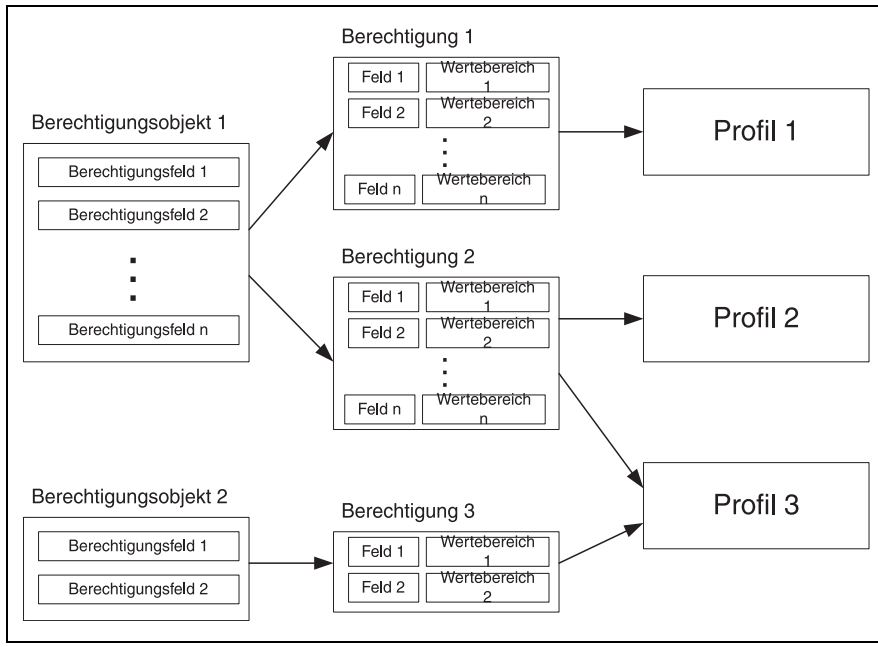

Abbildung 2.3
Zusammenfassung von Berechtigungen in Profile

Prüfen von Berechtigungen

Das Prüfen, ob ein Benutzer eine konkrete Berechtigung hat, erfolgt mit der ABAP-Anweisung AUTHORITY-CHECK. Mit Ausnahme von sehr wenigen Fällen (z.B. Dialogtransaktionen) muss das Prüfen von Berechtigungen immer explizit vom Programmierer realisiert werden. Dies geschieht hierbei ausschließlich auf der Ebene des in R/3 integrierten Berechtigungswesens. Entsprechende Mechanismen, wie sie die Hersteller der zugrunde liegenden Datenbanksysteme in der Regel ausliefern, finden hier keine Beachtung.

Das Berechtigungswesen in R/3 sieht vor, dass alles im System verboten ist, was nicht explizt durch Berechtigungen erlaubt wurde. Demnach kann das System in seiner Grundausrichtung als sicher betrachtet werden. Sicherheitslücken entstehen nur durch manuellen Eingriff von Administratoren. Umgekehrt gilt allerdings auch, dass alles erlaubt ist, wenn keine Berechtigungsprüfung es verhindert. Diese Aussage geht in Richtung der Programmierer, die angehalten sind, schützenswerte Sachverhalte mit entsprechenden Berechtigungen zu versehen.

Diese Prüfungen werden explizit mit der ABAP-Anweisung AUTHORITY-CHECK, deren Syntax im Folgenden dargestellt ist, realisiert.

```
AUTHORITY-CHECK <Berechtigungsobjekt>
            ID <Feldname 1> FIELD <Wert 1>
            ID <Feldname 2> FIELD <Wert 2>
            ...
            ID <Feldname n> FIELD <Wert n>.
```

Hierbei muss für jedes Feld, welches innerhalb des Berechtigungsobjekts definiert wurde, eine Überprüfung in Form eines ID <Feldname> FIELD <Wert>-Blockes stattfinden.

Eine Berechtigungsprüfung kann genau zwei Ergebnisse liefern. Entweder der Benutzer besitzt die geforderte Berechtigung oder nicht. Die geforderte Berechtigung besitzen bedeutet hier, dass der Benutzerstammsatz des aktuellen Benutzers eine Berechtigung enthält, für die jede einzelne Feldprüfung ergibt, dass der vom AUTHORITY-CHECK geforderte Wert im Wertebereich der Berechtigung liegt. Schlägt die Prüfung dieser Wertebereiche für mindestens ein Feld fehl, wird der gesamte AUTHORY-CHECK negativ beantwortet. Dies bedeutet, dass auch umfangreiche Berechtigungsobjekte nur einen dieser beiden Werte zurückliefern kann. Teilabfragen auf Berechtigungen sind nicht vorgesehen.

Die Prüfung ist erfolgreich durchlaufen, wenn der Benutzerstammsatz des Benutzers eine Berechtigung enthält, für die alle Felder des Berechtigungsobjekts ein positives Ergebnis liefert. Dabei spielt es keine Rolle, ob das betreffende Berechtigungsobjekt direkt dem Benutzerstammsatz zugeordnet ist oder in einem Profil enthalten ist.

2.3.2 Parametrisieren von Anwendungen mit Berechtigungen

Aufgrund der Komplexität und dem administrativen Aufwand, der notwendig ist, Berechtigungen einzuführen und entsprechend Benutzern zuzuweisen, wird dieses Verfahren bei Schaltern, die kurzfristig und schnell umgesetzt werden müssen, nicht zum Einsatz kommen. Vielmehr eignet sich das Verfahren, um dauerhaft Einstellungen am Verhalten von Programmen durchzuführen.

Obschon auf die Definition eigener Berechtigungsobjekte im Prinzip verzichtet werden kann, empfielt es sich auf jeden Fall, die semantische Bedeutung von Berechtigungsobjekten nicht zu sehr zu dehnen. Im Zweifelsfall sollte der Aufwand eigener Berechtigungsobjekte nicht gescheut werden, denn aufgrund der Komplexität des gesamten Themas sind nachträgliche Änderungen oftmals mit hohem Aufwand verbunden. Eine klare Trennung zwischen den zu schützenden Bereichen erleichtert im Zweifelsfall in der Zukunft Erweiterungen am Konzept vorzunehmen, ohne störenden Einfluss auf andere Bereiche auszuüben.

Über den Aufbau des Berechtigungsobjekts können die einflussnehmenden Faktoren auf die Parametrisierbarkeit gesteuert werden. Dies erfolgt über die Anzahl und Auswahl der Berechtigungsfelder des Berechtigungsobjektes. Dabei muss jedoch immer klar sein, dass über die Anzahl der Felder nicht geändert werden kann, dass eine Berechtigungsprüfung entweder positiv oder negativ ausfällt. Mehr Felder innerhalb eines Berechtigungsobjekts können daher auch hinderlich sein und eine Erweiterung des Konzeptes erschweren.

Weiterhin ist zu beachten, dass die Aufnahme des Benutzernamens als Feld in das Berechtigungsobjekt keinen Sinn macht, da Berechtigungen immer am Benutzerstamm festgemacht werden und der Bezug zum Benutzer daher automatisch gegeben ist.

2.3.3 Beispiel zur Parametrisierung mit Berechtigungen

Als Beispiel soll wieder der Sachverhalt aus dem vorigen Abschnitt herangezogen werden. In diesem Beispiel sollen Dynpro-Felder abhängig von einer Berechtigung `TPARAM` angezeigt oder versteckt werden. Hierzu wird zunächst ein Berechtigungsfeld `ZTCODE` angelegt. Dies geschieht in der Transaktion `SU20` (siehe Abbildung 2.4).

Das Berechtigungsfeld `ZTCODE` wird hierbei mit Bezug zum Datenelement `TCODE` angelegt.

Im Anschluss daran wird ein Berechtigungsobjekt `TPARAM` erzeugt. Das Berechtigungsobjekt enthält hierbei als einziges Feld das oben angelegte Feld `ZTCODE` (siehe Abbildung 2.5). Dies geschieht in der Transaktion `SU21`.

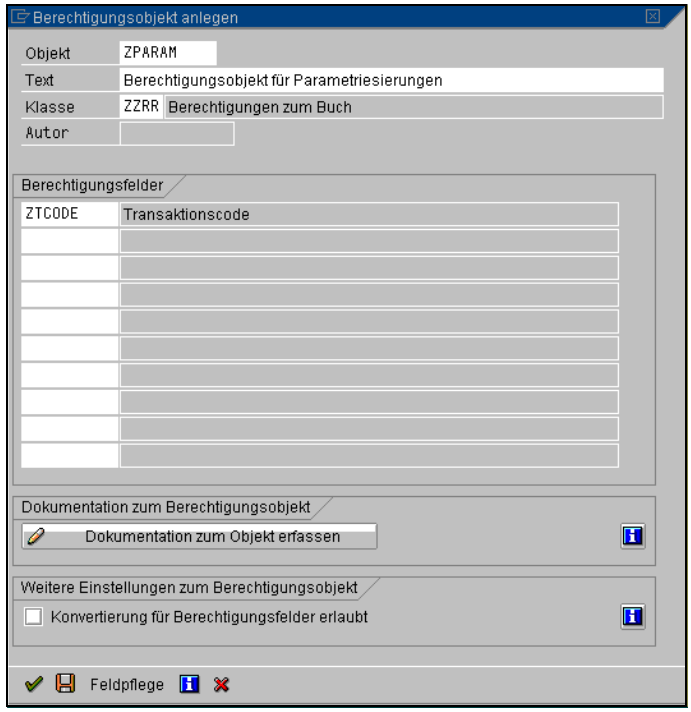

Abbildung 2.4
Anlegen eines Berechtigungsfeldes (© SAP AG)

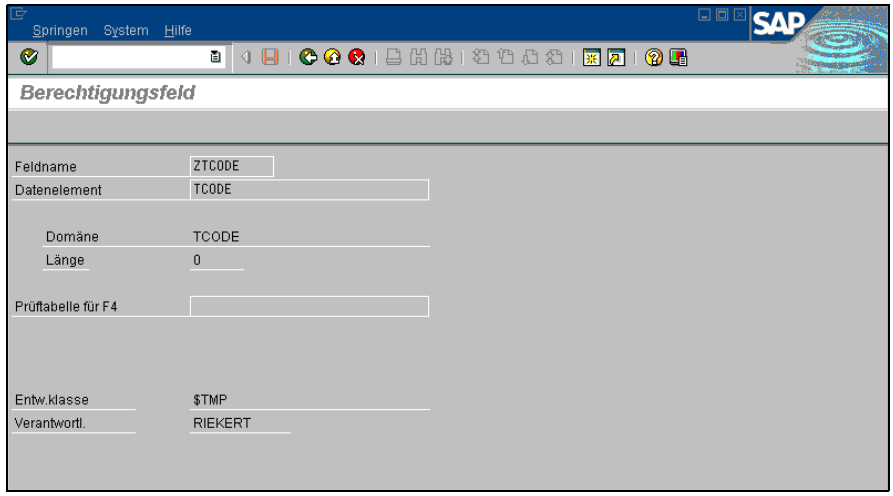

Abbildung 2.5
Anlegen eines Berechtigungsobjekts (© SAP AG)

Dieses Berechtigungsobjekt kann jetzt in Berechtigungsprofile aufgenommen werden und das Feld ZTCODE mit gültigen Werten belegt werden. Hierbei bestehen die gleichen Möglichkeiten, wie sie bei der Definition eines RANGES-Feldes für das Datenelement bestünden, d.h. es können sowohl Einzelwerte als auch Wertebereiche definiert werden.

Im Beispiel wird die Berechtigung mit den Einzelwerten ZFLIGHTGUI, ZFLIGHTSRV sowie dem Bereich ZFLIGHT* angelegt. Da Berechtigungen am Benutzerstammsatz abgelegt werden, besteht somit die Möglichkeit, sowohl benutzer- als auch transaktionsabhängig die Anzeige der Zusatzfelder zu steuern.

Im ABAP-Programm des vorigen Abschnitts muss in diesem Fall lediglich die Implementierung der Routine GET_PARAMETER_VALUE geändert werden.

```
* Die Routine Get_Parameter_Value liest den Parameter-Id aus.
FORM GET_PARAMETER_VALUE CHANGING P_MOREINFO TYPE C.
  CLEAR P_MOREINFO.

  AUTHORITY-CHECK 'ZPARAM'
       ID 'ZTCODE' FIELD SY-TCODE.

* RC=0: Berechtigung vorhanden
  IF SY-SUBRC EQ 0.
    P_MOREINFO = 'X'.
  ENDIF.
ENDFORM.
```

Wird der Programmcode in der Transaktion ZFLIGHTGUI oder ZFLIGHTSRV integriert, werden dem Benutzer die entsprechenden Felder angezeigt. In allen anderen Transaktionen (die nicht mit ZFLIGHT beginnen) würde die Berechtigungsprüfung fehlschlagen und die Felder versteckt werden.

2.4 Kundeneigene Customizing-Tabellen

Die in R/3 am häufigsten gebräuchliche Methode, um Programme parametrisierbar zu machen, ist die Verwendung von Steuertabellen.

Ein wesentlicher Grund für die große Flexiblität von R/3 liegt darin begründet, dass SAP reichlich Gebrauch von dieser Methode macht und hunderte Steuertabellen im System definiert hat. Es gibt keinen Grund, warum ein ABAP-Programmierer nicht ebenfalls dieses Werkzeug gebrauchen sollte und somit seine eigenen Entwicklungen im Rahmen des Absehbaren parametrisierbar macht. SAP hat hierfür einen eigenen Tabellentyp, die »Customizingtabelle, Pflege nur durch Kunden, kein SAP Import« (Tabellentyp »C«) definiert. Diese Art Customizing-Tabellen werden von Patches oder Releasewechseln von SAP nicht berührt.

Beim Anlegen von Steuertabellen sollte nicht vergessen werden, die dazugehörigen Pflegedialoge zu generieren. Ohne die Pflegedialoge können keine Einträge in die Tabellen mit der Transaktion SM30 (bei Tabellen mit längeren Tabellennamen) bzw. SM31 (bei Tabellennamen bis 5 Zeichen Länge) vorgenommen werden.

Da Customizing-Tabellen in der Regel nicht viele Sätze enthalten, können die Zugriffszeiten auf die Einstellungen in der Regel vernachlässigt werden. Gegebenenfalls können Zugriffe durch Verwendung von Pufferung nochmals optimiert werden. Aus Sicht der Programm-Ablaufgeschwindigkeit spielt die Verwendung von Customizing-Tabellen demnach keine entscheidende Rolle.

Customizing-Tabellen können grob in zwei Typen unterschieden werden. Der erste Typ, im Folgenden generische Steuertabellen genannt, definieren eine Struktur, in der Einstellungen verschiedenster Arten abgelegt werden können. Die zweite Art von Customizing-Tabellen definieren Einstellungen für einen speziellen Sachverhalt. Sowohl der Schlüsselbereich als auch die Datenfelder der Customizing-Tabelle sind dabei speziell auf eine Problemstellung abgestimmt. Dieser Tabellentyp wird im Folgenden spezielle Steuertabelle genannt.

Es ist sehr zu empfehlen, Customizing-Tabellen mandantenabhängig zu definieren. Oft sind unterschiedliche Voraussetzungen in den verschiedenen Mandanten im vornhinein nicht abschätzbar. Zudem erschweren die Standard-Mechanismen in R/3 die Pflege mandantenübergreifender Objekte.

2.4.1 Generische Customizing-Tabellen

Generische Customizing-Tabellen dienen als Pool für verschiedenste Einstellungen. Die Sätze der Tabelle müssen nicht in einem direkten Bezug zueinander stehen. Dieser Tabellentyp ist dann sinnvoll, wenn viele nicht komplexe Parameter eines Programms in Tabellen definiert werden sollen.

Tabelle 2.1 zeigt eine einfache Form einer generischen Steuertabelle. Sie deckt im Wesentlichen die gleiche Funktionalität ab, wie Benutzerparameter (siehe Abschnitt 2.2). Im Unterschied zu Benutzerparametern sind die Einstellungen im dargestellten Beispiel jedoch nicht benutzerspezifisch definierbar.

Zu einem eindeutig definierten Schlüssel kann hier genau ein Wert definiert werden. Sowohl der Schlüssel (KEY) als auch der Wert (VALUE) sind hier als Character-Felder definiert. Es ist nicht möglich, komplexe Daten, insbesondere Fließkommazahlen in dieser Tabelle abzulegen.

Feldname	Key	Datenelement	Typ	Länge	Inhalt
MANDT	X	MANDT	CLNT		Mandant
KEY	X	ZZKEY	CHAR	10	Schlüsselfeld (selbst definiert, hier nicht gezeigt)
VALUE		ZZVALUE	CHAR	30	Wertfeld für die Einstellung (selbst definiert, hier nicht gezeigt)

Tabelle 2.1
Aufbau einer einfachen generischen Steuertabelle

Durch Erweiterung dieser Tabellenstruktur um weitere Wert-Felder anderen oder gleichen Datentyps können weitere Einstellungen zu einem Schlüssel gespeichert werden. Je nach Schlüsselwert können unterschiedliche Felder der Tabelle verwendet werden.

Weiterhin ist es oft gewünscht, die Einstellungen in solchen Steuertabellen benutzerabhängig definieren zu können. Die Struktur der Tabelle muss in diesem Fall noch um ein Feld `USER` im Bereich des Primärschlüssels erweitert werden.

Eine ebenfalls häufig gestellte Anforderung an derartige Tabellen ist es, Einträge mit einem Gültigkeitszeitraum, welcher ebenfalls im Primärschlüssel definiert werden muss, einzurichten. Da jeder Zugriff auf die Steuertabelle explizit im Programm realisiert werden muss, ist es beliebig möglich, im Einzelfall Felder oder Feldinhalte in unterschiedlichen Kontexten zu verwenden.

Abschnitt 2.4.4 enthält nochmals ein vollständiges Beispiel für eine generische Steuertabelle mit dazugehörigem Programmcode.

2.4.2 Spezielle Customizing-Tabellen

Bei der Erzeugung von Steuertabellen ist immer abzuwägen, inwieweit nicht zusammengehörige Sachverhalte vermischt werden sollen und können. Im Fall der Verwendung von generischen Steuertabellen können alle Benutzer, die die Pflegeberechtigung für die Tabelle haben, beliebig Sätze in der Tabelle anfügen, ändern oder löschen. Eine weitere Einschränkung auf die Art des Customizings kann also nicht vorgenommen werden. Soll dies verhindert werden, oder sind die Einstellungen, die über die Steuertabellen gemacht werden können, sehr unterschiedlich, bietet sich an, für verschiedene Sachverhalte eigene Steuertabellen anzulegen, die den genauen Erfordernissen der Parametrisierung entsprechen. Es ist nicht sinnvoll, das Streben nach hoher Programmqualität dadurch zu torpedieren, dass im Bereich der Steuertabellen Kompromisse eingegangen werden.

Spezielle Customizing-Tabellen sollten dabei nach den gleichen Grundsätzen angelegt werden, wie generische Steuertabellen. Die Tabellen sollten nach Möglichkeit mandantenabhängig definiert werden. Um eine kontrollierte Pflege der Tabelleninhalte zu ermöglichen, ist die Generierung von Pflegedialogen unbedingt zu empfehlen.

2.4.3 Namen von Steuertabellen

Im SAP-Standard sind die Namen von Customizing-Tabellen in der Regel vier oder fünfstellig. Sie beginnen mit einem Präfix, der den Kontext der Tabelle erklären soll, gefolgt von einer Nummer und ggf. einem erläuternden Suffix. Die Tabellen, die das Data Dictionary beschreiben, beginnen bespielsweise alle mit dem Präfix `DD`. Ihm folgt eine zweistellige Zahl gefolgt von einem erläuternden Buchstaben, falls in einem Kontext mehrere Tabellen sinnvoll sind (z.B. Texttabellen o.Ä.).

Da sich dieses Prinzip in vielen Jahren bewährt hat, liegt der Gedanke nahe, eigene Customizing-Tabellen nach ähnlichem Muster zu benennen.

Beispielsweise könnte man Customizing-Tabellen zu einer selbst entwickelten Transaktion mit dem Transaktionscode einleiten und anschließend durchnummerieren. Sehr bewährt hat sich auch das Konzept, Tabellen, die sprachabhängig Texte definieren, mit dem Suffix »`T`« zu benennen. So heißt die Tabelle, in der die Buchungskreise im SAP-System definiert werden, `T001` und die Tabelle, die die Bezeichnungen dieser Buchungskreise sprachabhängig enthält, `T001T`. Um Redundanzen in den Customizing-Tabellen zu vermeiden, ist es in der Regel nicht sinnvoll, inhaltliche Aspekte und sprachabhängige Einstellungen in einer Tabelle zu mischen.

Neben der Möglichkeit, auch komplexe Einstellungen zu definieren, ist ein klarer Vorteil von Customizing-Tabellen gegenüber Benutzerparametern die Tatsache, dass der Zugriff auf die Tabellen über Standard-Berechtigungsobjekte eingeschränkt werden kann. Damit besteht die Möglichkeit, einem genau definierten Benutzerkreis die Möglichkeit zu geben, die Customizing-Einstellungen zu ändern. Je nachdem, ob im Schlüssel der Customizing-Tabelle der Benutzername enthalten ist oder nicht, können Parametrisierungen für einzelne Benutzer, Benutzergruppen oder benutzerunabhängig über andere Parameter definiert werden.

2.4.4 Beispiele zur Verwendung von Steuertabellen

Anwendung generischer Steuertabellen im Beispiel

Zunächst soll mit Hilfe einer generischen Steuertabelle das Beispiel aus dem vorigen Abschnitt reproduziert werden. Hierzu wird die folgende Tabelle (im Beispiel mit dem Namen `ZBSP01`) definiert.

Feldname	Key	Datenelement	Typ	Länge	Inhalt
MANDT	X	MANDT	CLNT		Mandant
KEY	X	ZZKEY	CHAR	10	Schlüsselfeld (selbst definiert, hier nicht gezeigt)
USER	X	SYUNAME	CHAR	12	Benutzer
VALUE		ZZVALUE	CHAR	30	Wertfeld für die Einstellung (selbst definiert, hier nicht gezeigt)

Tabelle 2.2
Struktur für generische Steuertabelle

Wie erwähnt, wird diese Tabelle mandantenabhängig definiert, um eine saubere Trennung der betriebswirtschaftlichen Einheiten des Unternehmens zu ermöglichen. Das Feld KEY wird den selbstdefinierten Schlüssel, über den auf die Tabelle zugegriffen werden soll, enthalten. Das Datenelement ZZKEY wird in diesem Rahmen nicht genauer definiert, es sollte jedoch eine zeichenbasierte Domäne referieren. Die Länge dieses Feldes muss den jeweiligen Gegebenheiten angepasst werden, eine Größe von ca. 10 Zeichen ist in den meisten Fällen ausreichend.

Über das Feld USER kann eine Einschränkung auf einen einzelnen SAP-Benutzer vorgenommen werden. Eine Fremdschlüsselbeziehung zum Benutzerstamm wird hier nicht gepflegt, um beispielsweise einen '*' als Platzhalter für alle Benutzer in der Tabelle anlegen zu können.

Schließlich enthält diese Tabelle das Feld VALUE, in dem der Wert des Steuerparameters hinterlegt wird. Für dieses Feld gelten die gleichen Vorraussetzungen wie für das KEY-Feld. Falls gewünscht, können hier auch andere Datentypen verwendet werden.

Falls zu einem Schlüsselwert mehrere gleichartige Daten möglich sind (z.B. eine Liste aller Werke, mit denen ein Benutzer arbeiten kann), können entweder mehrere Sätze zum gleichen KEY angelegt werden (in diesem Fall müsste der Primärkey der Tabelle um ein weiteres Feld, z.B. ZAEHLER, erweitert werden) oder es müssen außerhalb des Key-Bereichs der Tabelle mehrere Wert-Felder angelegt werden.

Sollen mehrere Werte gleichen Typs definiert werden, ist vermutlich die erste Lösung zu bevorzugen. Fallen zu einem Schlüssel mehrere Werte verschiedenen Charakters an, ist es sinnvoll, eher den nicht-Key-Bereich der Tabelle zu erweitern.

Zu dieser Tabelle werden die üblichen Pflegedialoge generiert und folgende Einträge vorgenommen:

MANDT	KEY	USER	VALUE
SY-MANDT	MOREINFO	*	
SY-MANDT	MOREINFO	RIEKERT	X

Tabelle 2.3
Inhalt aus generischer Steuertabelle für MOREINFO

Die Einträge dieser Tabelle sollen bewirken, dass die zusätzlichen Felder für alle Benutzer ausgeblendet werden, außer für den Benutzer »RIEKERT«. In der vorliegenden Implementierung könnte der erste Satz der Tabelle auch entfallen, da als Default angenommen wird, dass das Flag nicht gesetzt ist (siehe Programmcode).

Im Beispiel aus Abschnitt 2.2.3 muss in diesem Fall lediglich der Teil der FORM-Routine OPEN_CLOSE_FIELDS verändert werden, in dem die Variable L_MOREINFO belegt wird.

Die GET PARAMETER-Anweisung der Routine muss entsprechend dem abgebildeten Quellcode-Fragment verändert werden.

```
...
FORM GET_PARAMETER_VALUES CHANGING P_MOREINFO TYPE C.

* Kopfzeile der Tabelle löschen
  CLEAR ZBSP01.

* Steuertabelle mit exaktem Key abfragen
  SELECT SINGLE *
         FROM   ZBSP01
         WHERE  KEY  = 'MOREINFO' AND
                USER = SY-UNAME.

* Wenn kein Eintrag für diesen User, vielleicht generelle
* Festlegung?
  IF SY-SUBRC NE 0.
    SELECT SINGLE *
           FROM   ZBSP01
           WHERE  KEY  = 'MOREINFO' AND
                  USER = '*'.
  ENDIF.

* Wert an Übergabeparameter zuweisen
  P_MOREINFO = ZBSP01-VALUE.
ENDFORM.
...
```

Das Beispiel geht davon aus, dass die TABLES-Anweisung zur Tabelle ZBSP01 im Deklarationsteil des Programms enthalten ist. Im ersten Schritt wird versucht, auf die Tabelle mit dem korrekten Benutzernamen zuzugreifen. Ist für diesen Schlüssel kein Satz in der Tabelle enthalten, wird in einem zweiten Schritt ein Platzhalter ('*') für den Benutzernamen verwendet. Dieses Vorgehen ist sinnvoll, um auch Default-Einstellungen in der Steuertabelle dokumentieren zu können. Je nach Umfang des Schlüssels, können ein oder mehrere Felder mit Platzhaltern versehen abgefragt werden.

Zuletzt wird der Wert dem Übergabeparameter der FORM-Routine zugewiesen. Sind beide Zugriffe erfolglos, wird der Wert aus der leeren Kopfzeile übernommen.

Beispiel zu speziellen Steuertabellen

Die Struktur einer speziellen Steuertabelle wird genau für eine Problemstellung definiert und kann dementsprechend wesentlich exakter auf ein Thema zugeschnitten sein. Im vorliegenden Fall soll nicht nur ein Kennzeichen, ob Felder ausgeblendet werden sollen, in der Tabelle verankert werden, sondern allgemein die Möglichkeit bestehen, die Attribute von Feldern oder Feldgruppen zu verändern.

Zu diesem Zweck soll die Customizing-Tabelle ZBSP02 in der dargestellten Struktur verwendet werden.

Feldname	Key	Datenelement	Typ	Länge	Inhalt
MANDT	X	MANDT	CLNT		Mandant
USER	X	SYUNAME	CHAR	12	Benutzer
DYNNR	X	SCRFDYNNR	CHAR	4	Dynpro-Nummer
FNAME	X	SCRFNAME	CHAR	132	Name eines Dynpro-Elements
GRP1	X	SCRFGRP1	CHAR	3	Modifikationsgruppe 1
EIN		SCRFEIN	CHAR	1	Eingabe im Feld vorgesehen
OUT		SCRFOUT	CHAR	1	Ausgabe im Feld vorgesehen
OBL		SCRFOBL	CHAR	1	Muß-Feld (obligatorisches Eingabefeld)
UNSI		SCRFUNSI	CHAR	1	Anzeige unsichtbar

Tabelle 2.4
Struktur für spezielle Steuertabelle

Über das Feld USER soll, wie im vorigen Beispiel, eine Einschränkung auf Benutzerebene möglich sein. Weiterhin kann das gewünschte Feld über die Kombination von Dynpro-Nummer und der ersten Feldgruppe (SCREEN-GROUP1) spezifiziert werden. Für einen realen Einsatz könnte der Schlüssel noch wesentlich ausgefeilter definiert werden, um weitere Steuermöglichkeiten, z.B. auf Feldebene, zuzulassen.

In der Tabelle 2.5 werden im vorliegenden Beispiel die in der Tabelle abgebildeten Sätze eingefügt.

MANDT	USER	DYNNR	GRP1	EIN	OUT	OBL	UNSI
SY-MANDT	*	1000	ZMI0	0	0	0	1
SY-MANDT	RIEKERT	1000	ZMI0	0	1	0	0

Tabelle 2.5
Datensätze für Tabelle ZBSP02

Die Einträge sollen bewirken, dass für alle Benutzer die Felder, bei denen der Wert des Feldes SCREEN-GROUP1 »ZMI0« ist, ausgeblendet werden. Für den Benutzer »RIEKERT« wird davon abweichended definiert, dass die Felder sichtbar aber nicht eingabebereit sein sollen.

Für dieses Beispiel muss die Implementierung der FORM-Routine OPEN_CLOSE_FIELDS aus dem Abschnitt 2.2.3 neu implementiert werden.

```
...
FORM OPEN_CLOSE_FIELDS.
  DATA: L_ZBSP02 LIKE ZBSP02 OCCURS 0 WITH HEADER-LINE.

* Steuertabelleneinträge einlesen
  SELECT *
         INTO  TABLE L_ZBSP02
         FROM  ZBSP02
         WHERE USER   IN (SY-UNAME, '*') AND
               DYNNR  = SY-DYNNR.

* Wurden Sätze gefunden?
  CHECK SY-SUBRC EQ 0.

* Loop über Dynprofelder
  LOOP AT SCREEN.
*   Satz für User in Tabelle vorhanden
    READ TABLE L_ZBSP02 WITH KEY USER = SY-UNAME
                                 GRP1 = SCREEN-GROUP1.

*   Wenn dies Fehlschlägt, ohne User probieren.
```

```
    IF SY-SUBRC NE 0.
      READ TABLE L_ZBSP02 WITH KEY USER = '*'.
                                   GRP1 = SCREEN-GROUP1.
    ENDIF.

*   Wenn kein Satz gefunden wurde, keine Verarbeitung
    CHECK SY-SUBRC EQ 0.

*   Feldattribute modifizieren
    SCREEN-INPUT     = L_ZBSP02-EIN.
    SCREEN-REQUIRED  = L_ZBSP02-OBL.
    SCREEN-OUTPUT    = L_ZBSP02-OUT.
    SCREEN-INVISIBLE = L_ZBSP02-UNSI.
    MODIFY SCREEN INDEX SY-TABIX.
  ENDLOOP.
...
ENDFORM.
...
```

Zunächst werden alle Sätze der Steuertabelle, die den aktuellen User und das aktuelle Dynpro betreffen in die interne Tabelle L_BSP02 eingelesen. Wenn dabei keine Sätze gefunden wurden, wird die Verarbeitung abgebrochen.

Anschließend wird in einer Schleife über alle Dynpro-Elemente geprüft, ob die Darstellungsattribute für das Feld verändert werden sollen oder nicht. Ist dies der Fall, werden die Screen-Attribute gemäß den Einträgen der Steuertabelle verändert.

Da es sich hier um eine spezielle Steuertabelle handelt, kann hier nicht nur ein Flag abgelegt werden, sondern es werden alle wesentlichen Attribute mit dem tatsächlichen Wert hinterlegt. Diese Variante bietet also bezüglich der Steuerbarkeit eine erheblich höhere Flexibilität als die zuvor dargestellte Methode mit der generischen Steuertabelle. Allerdings können in dieser Tabelle ausschließlich Darstellungsattribute für Dynpro-Elemente definiert werden, während in der generischen Variante auch völlig andersartige Einstellungen abgelegt werden könnten.

2.5 Funktionscodes als Programmschalter

Zuletzt bietet sich noch die Möglichkeit, den Programmlauf durch Definition von Funktionscodes zu beeinflussen, über die der Benutzer zur Laufzeit Programmschalter setzten oder löschen kann.

Da Funktionscodes keine Parameter mitgegeben werden können, eignet sich diese Methode in seiner Grundform allerdings allenfalls zum Setzen einfacher Schalter. Eine erweiterte Version dieses Konzepts könnte so aussehen, dass über einen Funktionscode beispielsweise ein Pop-Up angezeigt werden kann, über

welches auch komplexe Parameter gesetzt werden könnten. Diese Möglichkeit, die ein erhebliches Maß an Programmieraufwand mit sich bringt, soll an dieser Stelle jedoch nicht weiter dargestellt werden.

Bei der Verwendung von Funktionscodes zum Steuern von Transaktionen erfolgt die Implementierung zweistufig. Im Command-Handler des Programms müssen die verwendeten Funktionscodes ausgewertet werden und im globalen Datenbereich des Programms in Variablen hinterlegt werden. Das Abfragen des Schalters erfolgt, wie bei den übrigen Methoden zur Parametrisierung auch, an den jeweiligen beeinflussten Stellen im Programm, im Beispiel also in der FORM-Routine GET_PARAMETER_VALUES.

Wird das Beispiel aus den vorigen Abschnitten aufgegriffen, genügt es also nicht, lediglich die Implementierung der Routine GET_PARAMETER_VALUES anzupassen, da hier lediglich die Schalter ausgewertet werden können. Unter der Annahme, dass ein globales Datenfeld G_MOREINFO angelegt wurde, sähe die Implementierung der Routine bei Verwendung dieser Methode wie dargestellt aus.

```
...
FORM GET_PARAMETER_VALUES CHANGING P_MOREINFO TYPE C.

* Es genügt, den Wert der globalen Variable zu übergeben.
  P_MOREINFO = G_MOREINFO.

ENDFORM.
...
```

Auch wenn es in diesem Beispiel überzogen wirkt, für die einfache Zuweisung eine eigene FORM-Routine zu definieren, ist dieses Vorgehen generell zu empfehlen. Hierdurch kann das verwendete Konzept zur Parametrisierung bei Bedarf leicht erweitert oder geändert werden. Eine Kapselung darartiger Funktionalitäten macht daher durchaus Sinn.

Durch das obige Programmfragment wird jedoch lediglich der Schalter ausgelesen. Das Setzten der Variable muss an anderer Stelle, dem Command-Handler des Programms, erfolgen. Im Beispiel sollen zwei Funktionscodes MOREINFO_ON und MOREINFO_OFF definiert werden, die den Schalter entsprechend ein- oder ausschalten. Da der Command-Handler in der Regel in Form eines CASE-Konstrukts realisiert ist, genügt es demnach, hier zwei Zweige für die neuen Funktionscodes einzufügen.

```
...
  CASE SAVE_OKCODE.
    ...
    WHEN 'MOREINFO_ON'.
      G_MOREINFO = 'X'.

    WHEN 'MOREINFO_OFF'.
```

```
      G_MOREINFO = SPACE.
    ...
ENDCASE.
```

In diesem Zusammenhang sei nochmals darauf verwiesen, dass Funktionscodes in R/3 nicht mehr auf die Länge von vier Stellen beschränkt sind, sondern bis zu 16 Stellen umfassen können. Es spricht also nichts dagegen, Funktionscodes – wie im Beispiel gezeigt – sprechend zu benennen.

2.6 Welche Methode verwenden

In den bisherigen Abschnitten dieses Kapitels wurden die drei Möglichkeiten beschrieben, wie in eigenentwickelten Programmen Steuerungsmöglichkeiten außerhalb der ABAP-Programmebene definiert werden können. Die Methoden unterscheiden sich nicht nur vom Aufwand der Realisierung her erheblich. Bei der Entscheidung, welcher Weg im konkreten Fall eingeschlagen werden soll, um Programme beeinflussbar zu machen, sollten folgende Aspekte in Betracht gezogen werden.

- Veränderbarkeit durch den Benutzer
- Abgrenzbarkeit
- Realisierungsaufwand

Oftmals ist es aufgrund unterschiedlicher Anforderungen sinnvoll mehrere der genannten Möglichkeiten in einem Programm zu vereinen. Als Grundsatz sollte hier wie überall die Regel gelten, die auf das jeweilige Problem am besten passende Lösung anzuwenden.

2.6.1 Veränderbarkeit durch den Benutzer

Ein wichtiger Aspekt bei der Auswahl der geeigneten Methode zur Einflussnahme von Außen ist die Frage, inwieweit jeder einzelne Benutzer in den Programmablauf eingreifen können soll.

Der Zugriff der Benutzer auf die eigenen Benutzerparameter ist nicht steuerbar, bieten hier also den geringsten Schutz. Das Gleiche gilt für die Verwendung von Funktionscodes, wobei hier bei der Implementierung gegebenenfalls auch auf Konzepte des Berechtigungswesens zurückgegriffen werden kann. Die übrigen Methoden hingegen zeichnen sich besonders durch gute Schutzmechanismen mit SAP-Standardmitteln aus.

Benutzerparameter eignen sich daher in der Regel ausschließlich für Komfort-Funktionen, die nicht unmittelbar das betriebswirtschaftliche Ergebnis eines Programmes beeinflussen. Beispielsweise könnte die Menge anzuzeigender Daten in einem Dynpro oder einer Liste über Benutzerparameter einfach gesteuert

werden. Die SAP-Basis kann für jeden Benutzer individuell einen Initialwert für den Benutzerparameter zuweisen, den der Benutzer aber nach Belieben verändern kann.

Da Funktionscodes nicht von vornherein gesetzt werden können, bietet sich diese Methode vor allen Dingen an, um Debugging-Möglichkeiten zur Fehlersuche ein- bzw. auszuschalten. Sie eignen sich nicht, um Transaktionen generell parametrisierbar zu machen.

Für Einstellungen, die das Ergebnis eines Programmes beeinflussen, sind Steuermöglichkeiten über Benutzerparameter keinesfalls zu empfehlen, da im nachhinein keine Möglichkeit besteht, die Benutzerparameter eines Benutzers zum Zeitpunkt des Programmlaufs nachzuvollziehen (Nachweismöglichkeit!).

Berechtigungsobjekte werden normalerweise im Bereich der Basis-Betreuung angesiedelt. Ein Benutzer hat demnach nie die Möglichkeit, selber Einfluss auf den Programmlauf zu nehmen. Bei der Verwendung von Steuertabellen kann die Pflege der Tabellen an das SAP-Berechtigungskonzept gekoppelt werden. Somit sind nahezu beliebige Konfigurationen der Pflegeberechtigungen für die Steuerinformationen möglich.

2.6.2 Abgrenzbarkeit

Wie bereits erwähnt, sind die Werte von Benutzerparametern für jeden User getrennt abgelegt. Es gibt hierbei weder Konzepte, um eine Einstellung für mehrere Benutzer in einem Schritt zu tätigen, noch Einstellungen an andere Kriterien als eine Benutzerkennung zu binden. Einstellungen, die das gesamte System oder eine Gruppe von Benutzern betreffen sollen, können demnach nicht mit Benutzerparametern abgebildet werden. Benutzerberechtigungen sind ein Weg, um derartige Parameter einzustellen, wenn das Kriterium für die Abgrenzung der Gültigkeit eines Steuerungsparameters Benutzer oder Benutzergruppen sind.

Sollen Parameter über andere Kriterien als Benutzer oder Benutzergruppen einstellbar sein, bleibt als letzte Möglichkeit die Verwendung von Steuertabellen. Hier können beliebige Schlüssel gewählt werden, um Einfluss auf das laufende Programm zu nehmen. Allerdings muss das Design der Steuertabellen mit Bedacht durchgeführt und die Zugriffe jeweils manuell programmiert werden.

Da die Methode von Funktionscodes lediglich Schalter-Charakter besitzt, stellt sich hier das Problem der Abgrenzbarkeit nicht unmittelbar. Ein Benutzer kann einen Schalter ein- und ausschalten. Auf andere Benutzer hat dies in der Regel keinen Einfluss. Der Schalter wird in der Regel nicht persistent abgelegt, ist also auch nach einem Neustart des Programms wieder zurückgesetzt.

2.6.3 Realisierungsaufwand

Wie schon aus den bisherigen Ausführungen erkennbar, unterscheiden sich die Methoden zur Parametrisierung nicht zuletzt durch den damit verbundenen Aufwand bei der Programmerstellung. Aufwand meint hierbei nicht nur den tatsächlichen Programmieraufwand, sondern umfasst auch notwendige Abstimmungsprozesse im Unternehmen usw.

Während dieser bei der Verwendung von Benutzerparametern und Funktionscodes minimal ist, steigert er sich bei Verwendung von Berechtigungsobjekten oder Steuertabellen erheblich. Welche dieser beiden Methoden einen höheren Aufwand bedeutet, kann nicht pauschal gesagt werden. Bei der reinen Programmierung ist der Aufwand von Berechtigungsobjekten ähnlich niedrig wie bei der Verwendung von Benutzerparametern, während er bei Steuertabellen entsprechend hoch ist. Der Abstimmungsaufwand in den Unternehmen wird allerdings in der Regel bei den Berechtigungsobjekten wesentlich höher liegen, da Berechtigungen üblicherweise nach dem 4- oder gar 6-Augen-Prinzip vergeben und aktiviert werden[3].

3. siehe auch »R/3 Sicherheitsleitfaden«, Band 2, S. 2-18

2.7 Zusammenfassung

Es ist häufig sehr sinnvoll, auf den Ablauf eigener Programme Einfluss nehmen zu können, ohne den Programmquelltext zu ändern. Programmänderungen führen in der Regel neue Tests und Abnahmeprozeduren nach sich, die selbst eine kleine Modifikation sehr teuer werden lässt.

Tabelle 2.6 soll nochmals im Überblick die Vor- und Nachteile der hier vorgestellten Methoden gegenübergestellen.

	Benutzer-parameter	**Berechti-gungen**	**Steuer-tabellen**	**Funktions-codes**
Veränderbarkeit	Jeder Benut-zer	SAP Basis	Beliebig über Berechtigung der Tabellen-pflege ein-stellbar	Jeder Benutzer zur Laufzeit
Eingrenzung	Je Benutzer	Auf Benutzer einschränkbar	Beliebige Gül-tigkeitsbereiche können realisiert wer-den.	Je Benutzer
Progarmmierauf-wand	niedrig	Niedrig	hoch	niedrig
Abstimmungsauf-wand im Unter-nehmen	i.d.R. niedrig	Hoch	niedrig	niedrig

Tabelle 2.6
Eigenschaften von Methoden der Parametrisierung

Spezielle Dialogelemente

3

3.1 Tree-Views nach der »alten« Methode

Die Verwendung von Baumdarstellungen zur strukturierten Darstellung hierarchischer Informationen wird seit jeher in der Datenverarbeitung häufig verwendet. Auch in R/3 ist diese Darstellungsmöglichkeit seit längerer Zeit verfügbar.

In R/3 werden Baumdarstellungen (Tree-Views) an verschiedensten Stellen im System verwendet, um hierarchisch strukturierte Informationen übersichtlich zu präsentieren. Offenkundigstes Beispiel im R/3-Release 4.6C ist das Easy Access-Menü, welches unmittelbar nach der Anmeldung an das System erscheint. Diese Art Bäume darzustellen, hat erst in neuere Releasestände von R/3 Einzug gehalten. In älteren Versionen von R/3 war eine andere Darstellung von Bäumen üblich, die dem ABAP-Programmierer sicherlich in Form der Transport-Workbench (Transaktion SE09) am geläufigsten sind. Diese älteren Tree-Views sind – im Gegensatz zu der modernen Variante - keine Dialogelemente im engeren Sinne in R/3 sondern lediglich eine besondere Variante des interaktiven Reportings. Demzufolge können diese Tree-Views nicht im Dynpro-Editor angelegt werden und können auch nicht mit anderen Dialogelementen kombiniert werden. Die Programmierung von Bäumen in der Form, wie sie in Reports am geläufigsten sind, wird in diesem Abschnitt ausführlich dargestellt. Die neue Variante, Bäume zu verwenden wird in Abschnitt 3.3 gezeigt. Diese Teilung wurde vorgenommen, da im dazwischen liegenden Abschnitt ein Beispielprogramm für Tab-Strips angelegt wird, welches als Trägerprogramm für die Dialogelemente in den darauffolgenden Abschnitten dient.

Die Dokumentation seitens SAP zu der älteren Variante ist nicht sehr ausführlich. Dies liegt sicherlich nicht zuletzt daran, dass diese Tree-Views keine Funktionalität der Sprache ABAP an sich darstellen, sondern auf einer Reihe von Funktionsbausteinen beruhen, die in der Funktionsgruppe SEUT zusammenge-

fasst sind. Es handelt sich also nicht um Sprachelemente, sondern um eine fortgeschrittene Programmiertechnik. In diesem Kapitel soll die Verwendung der wesentlichen Funktionsbausteine dieser Funktionsgruppe anhand eines Beispiels gezeigt werden. Eine detaillierte Beschreibung jeder einzelnen Funktion findet sich im Abschnitt 3.1.8 wieder. Das hier beschriebene Beispiel wird auch in den folgenden Abschnitten dieses Kapitels verwendet werden.

Ein Baum besteht aus Knoten (Nodes), die andere Knoten miteinander verknüpfen (hier Verbindungsknoten genannt) und anderen, die am Ende der Hierarchie stehen und Blätter (Leafs) genannt werden. Dem Knoten auf der obersten Ebene (Wurzel oder Root genannt) kommt hierbei eine besondere Rolle zu. Er ist der einzige Knoten im gesamten Baum, der keine Beziehung zu einem übergeordneten Knoten (Parent bzw. Vater) unterhält (siehe Abbildung 3.1). Alle in der Hierarchie unterhalb des Wurzelknotens stehenden Knoten haben genau eine Verbindung zu ihrem Elternknoten (Parent) und keine, eine oder mehrere Verbindungen zu wiederum untergeordneten Knoten (Child, Kinder). Knoten auf gleicher Ebene, die einen gemeinsamen Parent haben, werden Geschwisterknoten genannt. Bei der SAP Realisierung von Bäumen ist die Unterscheidung zwischen Knoten und Blättern lediglich eine aus dem Datenbestand resultierende Eigenschaft eines Knotens. Ein Blatt kann jederzeit zum Verbindungsknoten werden, indem ihm untergeordnete Knoten zugewiesen werden und umgekehrt kann ein Verbindungsknoten jederzeit zum Blatt werden, wenn der darunter liegende Teilbaum entfernt wird.

Neben der eindeutigen ID, spielt bei der Verwendung von Knoten der Knotenname eine wesentliche Rolle. Während die ID wie gesagt eindeutig im gesamten Baum ist, muss dies bei den Knotennamen nicht unbedingt der Fall sein. Über einen Schalter des Funktionsbausteins `RS_TREE_LIST_DISPLAY` kann festgelegt werden, dass während der Baumdarstellung angelegte oder umbenannte Knoten keine bereits verwendeten Namen haben dürfen oder automatisch zu Referenzknoten werden. Beim programmgesteuerten Manipulieren in den Knoten eines Baumes existiert diese Einschränkung nicht. Trotzdem sollten Knoten mit dem gleichen Namen auch dieselben semantischen Knoten beschreiben. Dies wird realisiert, indem diese Knoten als Referenzknoten (Link-Knoten) angelegt werden (siehe hierzu auch Abschnitt 3.1.5).

Die folgenden Abschnitte sollen die wesentlichen Aufgaben bei der Programmierung von Baumdarstellungen aufzeigen. Dies soll mit einem Beispielprogramm exemplarisch gezeigt werden.

Das Beispiel verwendet Tabellen, die in jedem SAP-System vorhanden sind. Zu Schulungszwecken hat SAP eine Reihe von Tabellen sowie deren Inhalt definiert, die Flugbewegungen von Fluggesellschaften abbilden. Der Datenbestand der Tabellen ist übersichtlich. SAP verwendet die Tabellen in den Programmierbeispielen der ABAP-Schulungen.

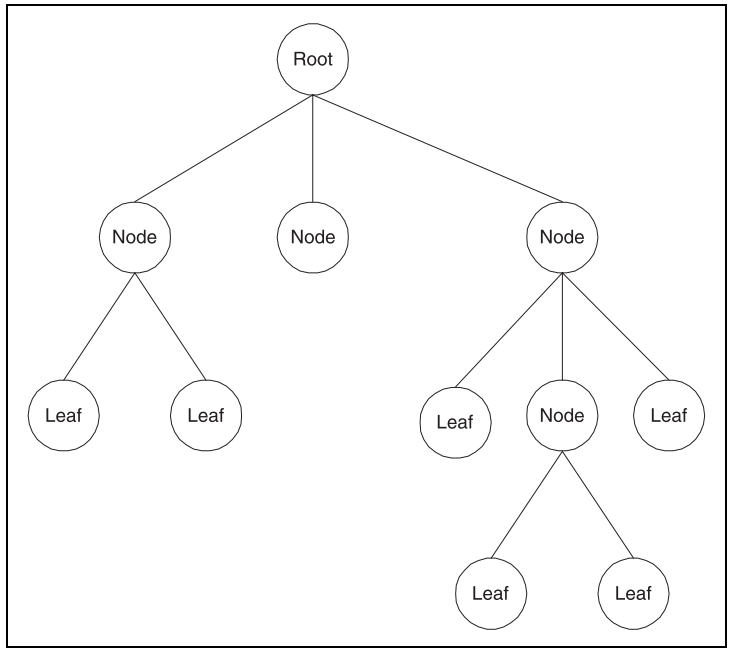

Abbildung 3.1
Schematische Darstellung eines Baumes

In Beispiel sollen alle Flüge je Fluggesellschaft angezeigt werden. Ein Flug kann hierbei zu verschiedenen Terminen stattfinden. Dies soll in einer dritten Ebene des Baumes dargestellt werden. Die Buchungen zum jeweiligen Flug können als Zusatzinformationen für jeden Flug individuell angezeigt werden. Diese Zusatzinformationen sind nur an den Knoten der dritten Ebene, den konkreten Flugereignissen, sinnvoll.

Die Ausgabe des fertigen Programms dieses Kapitels könnte beispielsweise das in Abbildung 3.2 gezeigte Aussehen haben. Bei den konkreten Flugereignissen (3. Ebene der Darstellung) werden sowohl die verfügbaren als auch die freien Plätze angezeigt. Falls weniger als 70% der verfügbaren Plätze belegt sind, wird die Zahl auf grünem Grund angezeigt. Bei mehr als 95% Belegung ist der Hintergrund rot, die Belegungsquoten dazwischen werden mit gelber Darstellung repräsentiert. Dadurch soll die Verwendung der Farbattribute der Knotenelemente demonstriert werden. Diese Farben werden hinter der Anzahl freier Plätze in Form einer Ampeldarstellung erneut aufgegriffen. Weiterhin werden eine Reihe eigener Funktionscodes definiert, um die interaktiven Manipulationmöglichkeiten während der Baumdarstellung zu demonstrieren. Aus dem gleichen Grund werden für die Knoten kontextabhängige Menüs definiert.

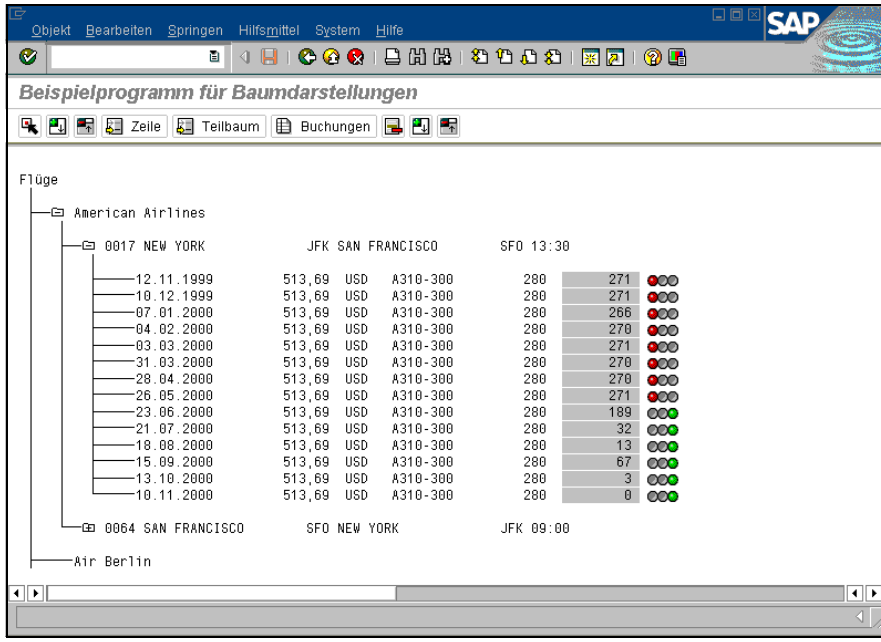

Abbildung 3.2
Ausgabe des Beispielprogramms (© SAP AG)

3.1.1 Erzeugung des Wurzelknotens

Beim Anlegen eines Tree-Views wird in R/3 der Wurzelknoten automatisch mit angelegt. Ein Tree-View besteht daher zumindest aus dem Wurzelknoten. Der Baum wird mit einem Aufruf des Funktionsbausteins `RS_TREE_CREATE` angelegt. Die Schnittstelle dieses Funktionsbausteins hat den unten dargestellten Aufbau:

```
FUNCTION 'RS_TREE_CREATE'
  IMPORTING
    ROOT_NAME
    ROOT_TYPE
    DISPLAY_ATTRIBUTES
  EXPORTING
    ROOT_ID
  EXCEPTIONS
    OTHERS
```

Über den Parameter `ROOT_NAME` wird der Wurzelknoten benannt. Dieser Name kann frei gewählt werden und erscheint standardmäßig als Text in der Darstellung des Knotens (siehe unten). Weiterhin kann der Typ des Knotens definiert werden. Dieser kann selbst vergeben werden und dient später, bei der Verarbeitung von Benutzeraktionen, zur Identifikation eines Knotentyps (siehe Ab-

schnitt 3.1.3). Daher ist es in der Regel ratsam, jeder Art von Knoten einen eigenen Typ zuzuweisen.

Wie der Knoten auf dem Bildschirm des Benutzers dargestellt werden soll, wird dem Funktionsbaustein in Form einer optionalen Struktur übergeben. Diese Struktur (die wichtigsten Felder dieser Struktur sind in Tabelle 3.1 dargestellt) – sie ist im Data Dictionary mit dem Namen STREEATTR abgelegt - definiert neben den allgemeinen Eigenschaften des Knotens (in den Feldern NLENGTH, COLOR und INTENSIV) zehn Darstellungsbereiche für den Knoten. Jedes dieser Knotenelemente wird durch die Attribute TEXT, TLENGHT, TCOLOR, TINTENSIV, TPOS sowie KIND definiert.

Feldname	Datentyp	Bedeutung
NLENGTH	NUMC 2	Ausgabelänge des Hierarchieknotens
COLOR	NUMC 1	Farbdarstellung des Hierarchieknotens
INTENSIV	CHAR 1	Intensität der Darstellung
TEXT	CHAR 75	Text
TLENGTH	NUMC 2	Länge des Knotenelements
TCOLOR	NUMC 1	Farbdarstellung des Knotenelements
TINTENSIV	CHAR 1	Intensität der Darstellung
TPOS	CHAR 3	Spaltenposition des Knotenelementss
KIND	CHAR 1	Textart des Knotenelements Space = Normaler Text N = Text ohne Lücke zum vorhergehenden Feld I = Ikone J = Ikone 2 K = Ikone 3 S = Symbol T = Symbol 2 R = Symbol 3 C = Ankreuzfeld B = Ankreuzfeld mit Text
TEXT1	CHAR 75	Text
TLENGTH1	NUMC 2	Länge des Knotenelements
TCOLOR1	NUMC 1	Farbdarstellung des Knotenelements

Feldname	Datentyp	Bedeutung
TINTENSIV1	CHAR 1	Intensität der Darstellung
TPOS1	CHAR 3	Spaltenposition des Knotenelements
TKIND1	CHAR 1	Knotentyp (siehe oben)
...	...	...
TEXT9	CHAR 75	Text
TLENGTH9	NUMC 2	Länge des Knotenelements
TCOLOR9	NUMC 1	Farbdarstellung des Knotenelements
TINTENSIV9	CHAR 1	Intensität des Knotenelements
TPOS9	CHAR 3	Spaltenposition des Knotenelements
TKIND9	CHAR 1	Knotentyp (siehe oben)
HIDE	CHAR 75	Unsichtbarer Bereich des Knotens für interne Informationen
...	...	...
NODISPLAY	CHAR 1	Flag, um Knoten bei Bedarf nicht anzuzeigen
MOREINFO	CHAR 1	Flag, ob zum Knoten erweiterte Informationen angezeigt werden sollen (siehe Abschnitt 0)

Tabelle 3.1
Darstellungsattribute eines Knotens (STREEATTR)

Wird diese Struktur nicht an den Funktionsbaustein übergeben, wird der Knoten mit dem Knotennamen angezeigt. Es wird hierbei eine Default-Farbe und eine Default-Länge angenommen. Diese kann bei der Anzeige des Baumes global festgelegt werden. Wird die STREEATTR-Struktur übergeben, wird der Knoten auf jeden Fall mit den darin definierten Werten angezeigt. Der Knotenname dient in diesem Fall nur zur Identifikation des Knotens.

Ist das jeweilige Feld KINDx leer (Space), wird der Inhalt des Feldes TEXT als Bezeichnung interpretiert. Bei den Knotentypen »I«, »J« und »K« wird der Inhalt dieses Feldes als Name der Ikone, bei »S«, »T« sowie »R« als Name des Symbols ausgelegt. Die Knotentypen »C« und »B« repräsentieren Ankreuzfelder ohne bzw. mit Text. Im letzteren Fall wird der Inhalt des Feldes TEXT als Beschriftung des Feldes verwendet.

Über TLENGTH, TCOLOR sowie TINTENSIV kann die Darstellung des Knotenelements auf dem Bildschirm definiert werden. Zu beachten ist hierbei, dass nicht alle

Symbole und Ikonen bei der Darstellung nur ein Zeichen benötigen. Verschiedene Elemente sind bis zu 4 Zeichen breit. Genauere Informationen über die Breite des jeweiligen Symbols oder der jeweiligen Ikone sind im Include <SYMBOL> bzw. <ICON> definiert.

Das Feld TPOS kann zur spaltengenauen Platzierung des Knotenelements verwendet werden. Nachfolgende Knotenelemente überschreiben hierbei ggf. vorhergehende Knotenelemente, falls eine Überschneidung definiert wurde. Der Inhalt des Feldes TPOS gibt die Startposition des Textelements vom linken Bildschirmrand wieder. Die Hierarchieebene des Knotens spielt hierbei keine Rolle. Wird das Feld TPOS nicht belegt, werden die Knoten hintereinander angeordnet. Dieses Vorgehen ist in der Regel günstiger, als das explizite Platzieren von Knotenelementen.

Je nach Anwendungsfall wird eine unterschiedliche Anzahl Knotenelemente zur Darstellung eines Knotens benötigt. Knotenelemente, deren Längenfeld 0 enthält, werden nicht angezeigt.

Das Ergebnis des Funktionsbausteins RS_TREE_CREATE ist eine ID, mit welcher der neu erzeugte Wurzelknoten eindeutig angesprochen werden kann. Während die Namen der Knoten eines Baumes ggf. nicht eindeutig sind (siehe hierzu Kapitel 3.1.5), ist die ID eines Knotens auf jeden Fall eindeutig.

Die erste Version des Beispielprogramms legt lediglich einen Wurzelknoten an und zeigt diesen Minimalbaum an.

```
REPORT ZFLIGHTTREE.

*
* Type-Pools
*
TYPE-POOLS: STREE.

*
* globale Daten
*
DATA:   G_ROOT       LIKE SNODE-ID.

START-OF-SELECTION.
* Erzeuge Wurzelknoten
  PERFORM CREATE_TREE CHANGING G_ROOT.

* Zeige Baum an
  PERFORM SHOW_TREE.



*&---------------------------------------------------------------------*
*&      Form  CREATE_TREE
```

```
*&---------------------------------------------------------------------*
*       Erzeugt den Wurzelknoten
*---------------------------------------------------------------------*
*      <--P_ROOT   ID des Wurzelknotens
*---------------------------------------------------------------------*
FORM CREATE_TREE CHANGING P_ROOT LIKE SNODE-ID.
  CALL FUNCTION 'RS_TREE_CREATE'
       EXPORTING
            ROOT_NAME          = 'Flüge'(001)
*           ROOT_TYPE          =
*           DISPLAY_ATTRIBUTES =
       IMPORTING
            ROOT_ID            = P_ROOT.
ENDFORM.                               " CREATE_TREE

*&---------------------------------------------------------------------*
*&      Form  SHOW_TREE
*&---------------------------------------------------------------------*
*       Zeigt den Baum an
*---------------------------------------------------------------------*
FORM SHOW_TREE.
  CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
*      EXPORTING
*           CALLBACK_PROGRAM          =
*           CALLBACK_USER_COMMAND     =
*           CALLBACK_TEXT_DISPLAY     =
*           CALLBACK_MOREINFO_DISPLAY =
*           CALLBACK_COLOR_DISPLAY    =
*           CALLBACK_TOP_OF_PAGE      =
*           CALLBACK_GUI_STATUS       =
*           CALLBACK_CONTEXT_MENU     =
*           STATUS                    = 'IMPLICIT'
*           CHECK_DUPLICATE_NAME      = '1'
*           COLOR_OF_NODE             = '4'
*           COLOR_OF_MARK             = '3'
*           COLOR_OF_LINK             = '1'
*           COLOR_OF_MATCH            = '5'
*           LOWER_CASE_SENSITIVE      = ' '
*           MODIFICATION_LOG          = ' '
*           NODE_LENGTH               = 30
*           TEXT_LENGTH               = 75
*           TEXT_LENGTH1              = 0
*           TEXT_LENGTH2              = 0
*           RETURN_MARKED_SUBTREE     = ' '
*           SCREEN_START_COLUMN       = 0
*           SCREEN_START_LINE         = 0
*           SCREEN_END_COLUMN         = 0
```

```
*           SCREEN_END_LINE           = 0
*           SUPPRESS_NODE_OUTPUT      = ' '
*           LAYOUT_MODE               = ' '
*           USE_CONTROL               = STREE_USE_LIST
*       IMPORTING
*           F15                       =
.
ENDFORM.                               " SHOW_TREE
```

Da beim Erzeugen des Wurzelknotens lediglich der Name des Knotens angegeben wird, ohne eine Darstellungsstruktur zu definieren, wird der Knoten mit dem als ROOT_NAME übergebenen String mit den Standard-Attributen angezeigt.

Über den Funktionsbaustein RS_TREE_LIST_DISPLAY wird der zunächst im Speicher definierte Baum auf dem Präsentations-Server (dem GUI des Benutzers) angezeigt. Dieser Funktionsbaustein definiert eine ganze Reihe von Parametern, über die das Verhalten und die Darstellung des Baumes beeinflusst werden können. Die meisten Parameter werden in den folgenden Abschnitten genauer betrachtet. Ansonsten sei an dieser Stelle auf die Online-Dokumentation zum Funktionsbaustein RS_TREE_LIST_DISPLAY verwiesen.

3.1.2 Erzeugung von Knoten

Im zweiten Schritt sollen in den erzeugten Baum Knoten eingefügt werden. Im Endzustand sollen unterhalb des Wurzelknotens drei Ebenen existieren. In der ersten Ebene werden die Fluggesellschaften angezeigt. Die zweite Ebene stellt die Flüge der jeweiligen Fluggesellschaften dar und in der dritten Ebene die Termine, zu denen die Flüge der zweiten Ebene stattfinden mit einer Belegungsanzeige. Von Ebene zu Ebene werden die Knoten etwas aufwendiger erzeugt. Während in der ersten Ebene noch keine Attribute für den Knoten definiert werden, wird in der zweiten Ebene Gebrauch von dieser Struktur gemacht. In der dritten Ebene werden dann auch die Farbinformationen für Knotenfelder verwendet und ein Knotenfeld als Symbol definiert (siehe Abbildung 3.2).

Knoten ohne Attribute

Zunächst sollen nun die Knoten für die Fluggesellschaften hinzugefügt werden. Diese werden aus der Tabelle SCARR ausgelesen. Der Einfachheit halber werden alle Fluggesellschaften ausgelesen und als Knoten angelegt. Hierzu muss zunächst eine TABLES-Anweisung für die Tabelle SCARR am Anfang des Programmcodes eingefügt werden.

Weiterhin wird im START-OF-SELECTION-Block zwischen den Aufrufen der Routine CREATE_TREE und SH560W_TREE eine neue FORM-Routine mit dem Namen CREATE_TREE_NODES eingefügt. Der neue Ereignisblock START-OF-SELECTION, sowie

die erste Implementierung der Routine CREATE_TREE_NODES sind im untenstehenden Programmfragment dargestellt.

```
...
START-OF-SELECTION.
* Erzeuge Wurzelknoten
  PERFORM CREATE_TREE CHANGING G_ROOT.

* Füge Knoten in Baum ein
  PERFORM CREATE_TREE_NODES USING G_ROOT.

* Zeige Baum an
  PERFORM SHOW_TREE.
...

*&---------------------------------------------------------------------*
*&      Form  CREATE_TREE_NODES
*&---------------------------------------------------------------------*
*       Erzeugt alle Knoten unterhalt des Wurzelknotens
*----------------------------------------------------------------------*
*      -->P_ROOT  Id des Wurzelknotens
*----------------------------------------------------------------------*
FORM CREATE_TREE_NODES USING P_ROOT  LIKE SNODE-ID.
  DATA: LT_SCARR       LIKE  SCARR   OCCURS 0 WITH HEADER LINE.
  DATA: L_AIRLINE_NODE LIKE  SNODE-ID.

* Airlines lesen
  SELECT *
         INTO TABLE LT_SCARR
         FROM SCARR.

* Umgekehrt sortieren für Baum
  SORT LT_SCARR BY CARRID DESCENDING.

  LOOP AT LT_SCARR.
*   Knoten für Airline anlegen
    PERFORM CREATE_AIRLINE_NODE USING    LT_SCARR
                                         P_ROOT
                                CHANGING L_AIRLINE_NODE.
  ENDLOOP.
ENDFORM.                               " CREATE_TREE_NODES


*&---------------------------------------------------------------------*
*&      Form  CREATE_AIRLINE_NODE
*&---------------------------------------------------------------------*
*       Legt einen neuen Knoten an
*----------------------------------------------------------------------*
```

```
*      -->P_SCARR    Struktur mit Airline-Information
*      -->P_PARENT   Knoten-Id des Parentnodes
*      <--P_NODE     Knoten-Id des neuen Knotens
*---------------------------------------------------------------------*
FORM CREATE_AIRLINE_NODE USING    P_SCARR   LIKE SCARR
                                  P_PARENT  LIKE SNODE-ID
                         CHANGING P_NODE    LIKE SNODE-ID.
  CALL FUNCTION 'RS_TREE_ADD_NODE'
       EXPORTING
            NEW_NAME           = P_SCARR-CARRNAME
            INSERT_ID          = P_PARENT
            RELATIONSHIP       = STREE_RELTYPE_CHILD
*           LINK               = ' '
            NEW_TYPE           = C_NODETYPE-AIRLINE
*           DISPLAY_ATTRIBUTES = ' '
       IMPORTING
            NEW_ID             = P_NODE
*           NODE_INFO          =
       EXCEPTIONS
            ID_NOT_FOUND       = 1
            OTHERS             = 2.
ENDFORM.                               " CREATE_AIRLINE_NODE
```

Im Beispiel wurde die Datenermittlung von der Definition der Darstellung des Knotens getrennt. Dies empfielt sich, um den Programmcode übersichtlich zu halten und ggf. an anderer Stelle gleiche Knoten einfügen zu können.

Um Knoten in den Baum einzufügen, wird der Funktionsbaustein `RS_TREE_ADD_NODE` verwendet. Der Funktionsbaustein erhält als Parameter den Namen des Knotens (`NEW_NAME`), die Id des referenzierten Knotens (`INSERT_ID`) sowie die Beziehung des neuen Knotens zum Bezugsknoten. Im Beispiel wird die Beziehung `STREE_RELTYPE_CHILD` angewendet, d.h. der neue Knoten wird als Unterknoten des referenzierten Knotens angelegt. Da der neue Child-Knoten immer als erster in die Hierarchie eingetragen wird, ist es notwendig vor dem Durchlaufen der `LOOP`-Schleife in der `FORM`-Routine `CREATE_TREE_NODES` die Sätze in umgekehrter Reihenfolge zu sortieren.

Eine andere Möglichkeit, die Knoten in den Baum einzufügen wäre, den ersten untergeordneten Knoten mit `STREE_RELTYPE_CHILD` und dem Parent als Referenz und die folgenden Knoten mit der Beziehung `STREE_RELTYPE_NEXT` und dem vorhergehenden Knoten als Referenz anzulegen. Allein die Beschreibung zeigt, dass dieses Verfahren ein wenig aufwendiger ist als einfach die Liste umzusortieren. Wenn möglich wird daher in der Regel die erste Methode verwendet.

Auch in diesem Beispiel werden keine Anzeige-Attribute für den Knoten definiert (`DISPLAY_ATTRIBUTES`). Die Darstellung des Knotens beschränkt sich daher auf den Knotennamen mit den Standard-Attributen. Die erzeugten Knoten erhalten alle den Knotentyp `C_NODETYPE-AIRLINE`, welcher am Anfang des Programms als Konstanten-

definition angelegt wird. Auch für die in den späteren Abschnitten erzeugten Knoten, sowie den Wurzelknoten werden Knotentypen definiert, so dass folgende Definition am Anfang des Programms notwendig wird. Der Aufruf des Funktionsbausteins RS_TREE_CREATE wird in diesem Schritt ebenfalls entsprechend angepasst, um den definierten Knotentyp C_NODETYPE-ROOT zu verwenden.

```
CONSTANTS: BEGIN OF C_NODETYPE,
             ROOT          LIKE SNODE-TYPE  VALUE 'ROOT',
             AIRLINE       LIKE SNODE-TYPE  VALUE 'AIRL',
             FLIGHT        LIKE SNODE-TYPE  VALUE 'PFLI',
             FLIGHTEVENT   LIKE SNODE-TYPE  VALUE 'SFLI',
           END OF C_NODETYPE.
```

In Abschnitt 3.1.3 wird gezeigt, wofür die Definition von Knotentypen für verschiedene Knoten sinnvoll ist und wie sie verwendet werden.

Der Funktionsbaustein RS_TREE_ADD_NODE liefert eine Struktur für den neu erzeugten Knoten zurück, in der die internen Informationen zum Knoten enthalten sind. Der Aufbau dieser Struktur ist im Data Dictionary mit dem Namen SNODE definiert (siehe Tabelle 3.2).

Feldname	Datentyp	Bedeutung
ID	NUMC 6	ID des Knotens
TYPE	CHAR 4	Knotentyp
NAME	CHAR 30	Knotenname
PARENT	NUMC 6	ID des Vaterknotens
CHILD	NUMC 6	ID des ersten Kindknotens
NEXT	NUMC 6	ID des nächsten Geschwisterknotens
TLEVEL	NUMC 2	Hierarchieebene auf dem sich der Knotens befindet
TLOCK	CHAR 1	Markierungskennzeichen
STATUS	CHAR 4	interne Verwendung von SAP
INCLUDE	CHAR 8	Name des ABAP-Programms
LINK	CHAR 1	Linkkennzeichen

Tabelle 3.2
Aufbau der Struktur SNODE

Diese Struktur beschreibt sowohl den Knoten (nicht dessen Darstellung) als auch die Position des Knotens innerhalb des Baumes.

Knoten mit Textattributen

In der nächsten Ebene des Baumes soll ein Knoten durch mehrere Felder dargestellt werden. Auch hier wird die Ermittlung der Daten von der Erzeugung des Knotens getrennt. Die Datenermittlung wird in die FORM-Routine CREATE_TREE_NODES integriert. Das Programmfragment zeigt die geänderte Version dieser FORM-Routine.

```
FORM CREATE_TREE_NODES USING P_ROOT  LIKE SNODE-ID.
  DATA: LT_SCARR       LIKE  SCARR   OCCURS 0 WITH HEADER LINE,
        LT_SPFLI       LIKE  SPFLI   OCCURS 0 WITH HEADER LINE.
  DATA: L_AIRLINE_NODE LIKE  SNODE-ID,
        L_FLIGHT_NODE  LIKE  SNODE-ID.


  ...
  LOOP AT LT_SCARR.
    ...
*   Flüge zu Airline lesen
    SELECT *
           INTO  TABLE LT_SPFLI
           FROM  SPFLI
           WHERE CARRID = LT_SCARR-CARRID.

*   Umgekehrt sortieren für Baum
    SORT LT_SPFLI BY CONNID DESCENDING.

    LOOP AT LT_SPFLI.
*     Knoten für Flug anlegen
      PERFORM CREATE_FLIGHT_NODE USING    LT_SPFLI
                                          L_AIRLINE_NODE
                                 CHANGING L_FLIGHT_NODE.
    ENDLOOP.
  ENDLOOP.
ENDFORM.                                  " CREATE_TREE_NODES
```

Innerhalb der LOOP-Schleife der Fluggesellschaften werden aus der Tabelle SPFLI alle Flüge zur Airline ausgelesen, umsortiert (wie oben) und mit der Routine CREATE_FLIGHT_NODE angezeigt. Die eigentlichen Unterschiede ergeben sich im Unterprogramm CREATE_FLIGHT_NODE, welches folgenden Aufbau hat:

```
*&---------------------------------------------------------------------*
*&      Form  CREATE_FLIGHT_NODE
*&---------------------------------------------------------------------*
*       Erstellt einen Knoten für einen Flug
*----------------------------------------------------------------------*
*      -->P_SPFLI    Datenstruktur des Fluges
*      -->P_PARENT   ID des Parent-Knotens
*      <--P_NODE     ID des neu erzeugten Knotens
```

```
*---------------------------------------------------------------------*
FORM CREATE_FLIGHT_NODE USING    P_SPFLI     LIKE SPFLI
                                 P_PARENT    LIKE SNODE-ID
                       CHANGING P_NODE      LIKE SNODE-ID.
  DATA: L_STREEATTR  LIKE STREEATTR.

  WRITE: P_SPFLI-CONNID    TO L_STREEATTR-TEXT,
         P_SPFLI-CITYFROM  TO L_STREEATTR-TEXT1,
         P_SPFLI-AIRPFROM  TO L_STREEATTR-TEXT2,
         P_SPFLI-CITYTO    TO L_STREEATTR-TEXT3,
         P_SPFLI-AIRPTO    TO L_STREEATTR-TEXT4,
         P_SPFLI-DEPTIME   TO L_STREEATTR-TEXT5.

  L_STREEATTR-TLENGTH  = 4.
  L_STREEATTR-TLENGTH1 = 20.
  L_STREEATTR-TLENGTH2 = 3.
  L_STREEATTR-TLENGTH3 = 20.
  L_STREEATTR-TLENGTH4 = 3.
  L_STREEATTR-TLENGTH5 = 5.

  CALL FUNCTION 'RS_TREE_ADD_NODE'
       EXPORTING
            NEW_NAME           = P_SPFLI-CONNID
            INSERT_ID          = P_PARENT
            RELATIONSHIP       = STREE_RELTYPE_CHILD
*           LINK               = ' '
            NEW_TYPE           = C_NODETYPE-FLIGHT
            DISPLAY_ATTRIBUTES = L_STREEATTR
       IMPORTING
            NEW_ID             = P_NODE
*           NODE_INFO          =
       EXCEPTIONS
            ID_NOT_FOUND       = 1
            OTHERS             = 2
            .
ENDFORM.                              " CREATE_FLIGHT_NODE
```

Zunächst werden über eine Kette von WRITE-Anweisungen die Feldinhalte in die entsprechenden Felder der Anzeige-Attribute übertragen. Die WRITE-Anweisung wird verwendet, damit etwaige Konvertierungsexits bei der Ausgabe berücksichtigt werden. Im Anschluss daran werden die Längenfelder der einzelnen Bereiche gesetzt. Im dargestellten Beispiel werden lediglich 6 der 10 möglichen Bereiche genutzt. Da das Längenfeld der übrigen Felder »0« enthält, werden diese bei der Ausgabe nicht berücksichtigt. Die so erzeugte Struktur wird im anschließenden Aufruf von RS_ADD_TREE_NODE im Parameter DISPLAY_ATTRIBUTES an den Funktionsbaustein übergeben.

Erweiterte Möglichkeiten von Knoten

Die Knoten der dritten Ebene, welche die konkreten Flugereignisse zeigen sollen, werden mit weiteren Attributen ausgestattet. Bei der Anzeige der freien Plätze wird eine Farbcodierung eingeführt. Flüge mit weniger als 70% Belegung werden hierbei grün dargestellt, Flüge mit mehr als 95% Sitzbelegung rot. Dazwischen soll die Anzeige der freien Plätze eines Fluges in gelb erfolgen. Analog zu diesen Grenzwerten soll hinter den Datenspalten eine Ampel mit der gleichen Farbcodierung angezeigt werden.

Die angezeigten Informationen werden aus der Tabelle SFLIGHT zum jeweiligen Flug ermittelt. Hierzu wird erneut die FORM-Routine CREATE_TREE_NODES erweitert. Die Erzeugung des Knotens selbst wird – wie in den vorigen Programmversionen auch – in eine eigene FORM-Routine CREATE_FLIGHT_EVENT_NODE ausgelagert.

Am Programmanfang wird eine konstante Struktur definiert, in der die Schwellwerte für die Darstellung bei den freien Plätzen festgelegt wird. Diese Struktur hat den Aufbau:

```
CONSTANTS: BEGIN OF C_BELEGUNG,
             WARNING   TYPE I   VALUE 70,
             ALERT     TYPE I   VALUE 95,
           END OF C_BELEGUNG.
```

In der FORM-Routine CREATE_TREE_NODES werden in diesem Schritt die Daten zu den konkreten Flugereignissen ausgelesen und durch die Routine CREATE_FLIGHT_EVENT_NODE in die Baumstruktur eingefügt.

```
FORM CREATE_TREE_NODES USING P_ROOT  LIKE SNODE-ID.
  DATA: LT_SCARR       LIKE  SCARR   OCCURS 0 WITH HEADER LINE,
        LT_SPFLI       LIKE  SPFLI   OCCURS 0 WITH HEADER LINE,
        LT_SFLIGHT     LIKE  SFLIGHT OCCURS 0 WITH HEADER LINE.
  DATA: L_AIRLINE_NODE LIKE  SNODE-ID,
        L_FLIGHT_NODE  LIKE SNODE-ID,
        L_EVENT_NODE   LIKE SNODE-ID.
  ...
  LOOP AT LT_SCARR.
    ...
    LOOP AT LT_SPFLI.
      ...
*     Fluginformationen lesen
      SELECT *
             INTO  TABLE LT_SFLIGHT
             FROM  SFLIGHT
             WHERE CARRID = LT_SCARR-CARRID AND
                   CONNID = LT_SPFLI-CONNID.

*     Umgekehrt sortieren
```

```
      SORT LT_SFLIGHT BY FLDATE DESCENDING.

      LOOP AT LT_SFLIGHT.
*       Flugknoten anzeigen
        PERFORM CREATE_FLIGHT_EVENT_NODE USING    LT_SFLIGHT
                                                  L_FLIGHT_NODE
                                         CHANGING L_EVENT_NODE.
      ENDLOOP.
    ENDLOOP.
  ENDLOOP.
ENDFORM.                               " CREATE_TREE_NODES

*&---------------------------------------------------------------------*
*&      Form  CREATE_FLIGHT_EVENT_NODE
*&---------------------------------------------------------------------*
*       Erzeugt einen Knoten für einen Flugevent
*----------------------------------------------------------------------*
*      -->P_SFLIGHT  Datenstruktur für Flug
*      -->P_PARENT   Id des Parent-Knotens
*      <--P_NODE     Id des neuen Knotens
*----------------------------------------------------------------------*
FORM CREATE_FLIGHT_EVENT_NODE USING    P_SFLIGHT  LIKE SFLIGHT
                                       P_PARENT   LIKE SNODE-ID
                              CHANGING P_NODE     LIKE SNODE-ID.
  DATA: L_STREEATTR  LIKE STREEATTR.
  DATA: L_BELEGUNG   TYPE F.

  WRITE: P_SFLIGHT-FLDATE    TO L_STREEATTR-TEXT,
         P_SFLIGHT-PRICE     TO L_STREEATTR-TEXT1(15)
                               CURRENCY P_SFLIGHT-CURRENCY,
         P_SFLIGHT-CURRENCY  TO L_STREEATTR-TEXT2,
         P_SFLIGHT-PLANETYPE TO L_STREEATTR-TEXT3,
         P_SFLIGHT-SEATSMAX  TO L_STREEATTR-TEXT4(10),
         P_SFLIGHT-SEATSOCC  TO L_STREEATTR-TEXT5(10).

  L_STREEATTR-TLENGTH  = 10.
  L_STREEATTR-TLENGTH1 = 15.
  L_STREEATTR-TLENGTH2 = 5.
  L_STREEATTR-TLENGTH3 = 10.
  L_STREEATTR-TLENGTH4 = 10.
  L_STREEATTR-TLENGTH5 = 10.
  L_STREEATTR-TLENGTH6 = 4.

  L_STREEATTR-TCOLOR   =
  L_STREEATTR-TCOLOR1  =
  L_STREEATTR-TCOLOR2  =
  L_STREEATTR-TCOLOR3  =
```

```
  L_STREEATTR-TCOLOR4  = 0.

* Typ des 7. Feldes auf Symbol setzten
  L_STREEATTR-KIND6  = 'I'.

  IF P_SFLIGHT-SEATSMAX NE 0.
    L_BELEGUNG = ( ( P_SFLIGHT-SEATSOCC / P_SFLIGHT-SEATSMAX ) * 100 ).
  ELSE.
    L_BELEGUNG = 0.
  ENDIF.

* Defaultwerte setzten
  L_STREEATTR-TCOLOR5 = 5.
  L_STREEATTR-TEXT6   = 'ICON_GREEN_LIGHT'.

  IF L_BELEGUNG >= C_BELEGUNG-WARNING.
    L_STREEATTR-TCOLOR5 = 7.
    L_STREEATTR-TEXT6   = 'ICON_YELLOW_LIGHT'.
  ENDIF.

  IF L_BELEGUNG >= C_BELEGUNG-ALERT.
    L_STREEATTR-TCOLOR5 = 6.
    L_STREEATTR-TEXT6   = 'ICON_RED_LIGHT'.
  ENDIF.

* Datenbank-Record im HIDE-Bereich merken
  L_STREEATTR-HIDE = P_SFLIGHT.

  CALL FUNCTION 'RS_TREE_ADD_NODE'
       EXPORTING
            NEW_NAME           = P_SFLIGHT-FLDATE
            INSERT_ID          = P_PARENT
            RELATIONSHIP       = STREE_RELTYPE_CHILD
*           LINK               = ' '
            NEW_TYPE           = C_NODETYPE-FLIGHTEVENT
            DISPLAY_ATTRIBUTES = L_STREEATTR
       IMPORTING
            NEW_ID             = P_NODE
*           NODE_INFO          =
       EXCEPTIONS
            ID_NOT_FOUND       = 1
            OTHERS             = 2.
ENDFORM.                               " CREATE_FLIGHT_EVENT_NODE
```

Im Unterprogramm CREATE_FLIGHT_EVENT_NODE werden, wie in den vorigen Beispielen auch, zunächst die Textfelder für die Bezeichungen sowie die Längenfelder belegt. Neu hinzugekommen ist die Versorgung der Farbattribute der nicht

hervorgehobenen Felder, sowie die Kennzeichnung des 7. Feldes als Anzeigefeld einer Ikone.

Im Anschluss daran wird die aktuelle Sitzbelegung aus den Feldern SEATS_MAX und SEATS_OCC ermittelt und als Fließkommazahl abgelegt und in einer Fallunterscheidung das Setzen der Farbe sowie die Auswahl des jeweils passenden Symbols realisiert. Die Symbole werden über Konstanten angesprochen, die im ABAP-Include <ICON> von SAP definiert wurden. Im Anschluss daran wird der von den anderen Ebenen bekannte Funktionsbaustein RS_TREE_ADD_NODE aufgerufen.

In der Struktur STREEATTR existiert weiterhin das Feld mit dem Namen HIDE. Die in diesem Feld abgelegten Informationen werden nicht zur Anzeige des Knotens verwendet, sondern stellen eine interne Ablagemöglichkeit für Informationen zur Verfügung. Im Beispiel wird in diesem Feld zumindest der Anfang des Datensatzes der Tabelle SFLIGHT abgelegt. Hierdurch kann später über den Knoten wieder eindeutig auf den angezeigten Datensatz der Tabelle SFLIGHT geschlossen werden. Es wird im Beispiel billigend in Kauf genommen, dass die Informationen der Tabelle SFLIGHT im Feld abgeschnitten werden, da für den späteren Zugriff lediglich der Primärschlüssel am Anfang der Struktur benötigt wird.

3.1.3 Interaktion in der Baumdarstellung

Das Beispielprogramm in seiner aktuellen Form zeigt die Hierarchie der Flüge mit drei Ebenen unterhalb des Wurzelknotens an. Bei der Anzeige des Baumes setzt das System einen vordefinierten GUI-Status (LD_TREE), der eine Reihe von Möglichkeiten zur Manipulation des Baumes ermöglicht. Diese sind nicht in jedem Fall erwünscht. Da in unserem Beispiel lediglich Daten aus der Datenbank angezeigt werden, sollen die Möglichkeiten des Benutzers zur Veränderung der dargestellten Informationen eingeschränkt werden. Zusätzlich sollen eine Reihe von eigenen Funktionscodes definiert werden.

GUI-Status während der Baumdarstellung

Besondere Beachtung findet hier die Option STATUS des Funktionsbausteins RS_TREE_LIST_DISPLAY. Standardmäßig wird hier der Wert »IMPLICIT« übergeben. Hier sind Werte

- OWN
- STANDARD
- IMPLICIT

möglich. Wird der Wert »OWN« an den Funktionsbaustein übergeben, wird während der Baumdarstellung der aktuell gesetzte Status des Programms beibehalten. Bei Verwendung dieser Option wird demnach in der Regel unmittelbar vor Aufruf von RS_TREE_LIST_DISPLAY der gewünschte Status mit der Anweisung SET PF-STATUS gesetzt. Dieser Status sollte eine Kopie des systemseitig verwendeten

Status sein und lediglich um eigene Funktionen erweitert werden. Nicht benötigte oder unerwünschte Systemfunktionalitäten können aus dem Status gelöscht werden oder – besser noch – über die Option EXCLUDING <FCodes> beim Setzten des Status deaktiviert werden (siehe Beispiel).

Der Wert STANDARD weist den Funktionsbaustein an, auf jeden Fall den Systemstatus LD_TREE zu setzten. Der Wert IMPLICIT setzt entweder den Standard-Status oder behält den aktuellen GUI-Status bei. Der Systemstatus wird gesetzt, wenn mindestens einer der Parameter CALLBACK_PROGRAMM oder CALLBACK_USER_COMMAND nicht übergeben werden. Sind beide Parameter versorgt, wird der aktuelle GUI-Status beibehalten.

Seit Release 4 wurde eine weitere Möglichkeit definiert, um den GUI-Status während der Anzeige zu beeinflussen. Über den Parameter CALLBACK_GUI_STATUS des Funktionsbausteins RS_TREE_LIST_DISPLAY kann eine FORM-Routine definiert werden, die vom System vor Anzeige des Baumes aufgerufen wird, um den GUI-Status zu setzen. Ist dieser Parameter (und der Parameter CALLBACK_PROGRAM) versorgt, erfolgt keine Auswertung des Parameters STATUS. Die Aufrufschnittstelle dieser FORM-Routine definiert keine Parameter, d.h. es werden weder Parameter an die Routine übergeben, noch werden Werte an das aufrufende Programm zurückgeliefert.

Verarbeitung von Funktionscodes

Der Standard-GUI-Status, den R/3 bei der Baumdarstellung verwendet, definiert eine Reihe von Funktionscodes. Diese werden in Tabelle 3.4 zusammengefasst. Die Schnittstelle des Funktionsbausteins RS_TREE_LIST_DISPLAY erlaubt die Definition einer FORM-Routine, in der die Funktionscodes, die während der Baumdarstellung ausgelöst werden, ausgewertet werden können. Diese Routine wird über die Parameter CALLBACK_PROGRAM, welches den Namen des ABAP-Programms enthält, in dem alle Callback-Routinen für die Baumdarstellung enthalten sind, und den Parameter CALLBACK_USER_COMMAND definiert. Wenn die so definierte FORM-Routine nicht vorhanden ist, wird der Aufruf ignoriert[1]. Dies beeinflusst jedoch nicht das oben beschriebene Verhalten bzgl. des Setzens des GUI-Status. Diese FORM-Routine muss mit folgender Schnittstelle deklariert sein:

```
FORM <Form-Name> TABLES   <Node-Tab>     STRUCTURE SEUCOMM
                 USING    <Command>
                 CHANGING <Exit>
                          <List-Refresh>.
```

Im Parameter <Node-Tab> werden dem Unterprogramm die zu bearbeitenden Knoten übergeben. Die Struktur SEUCOMM hat hierbei den in Tabelle 3.3 definierten Aufbau. Der Parameter <Command> enthält den ausgewählten Funktionscode.

1. In früheren R/3-Releases wurde das Programm in diesem Fall mit einem Kurzdump abgebrochen. In neueren Releaseständen wird der Aufruf ignoriert.

Für die Funktionscodes des Standard-GUI-Status sind im TYPE-POOL STREE Konstanten deklariert. Weiterhin werden die Rückgabeparameter <Exit> sowie <List-Refresh> deklariert, mit dem die Folgeverarbeitung im aufrufenden Programm definiert werden kann. Enthält das Feld <Exit> nach Verlassen der Routine ein »X«, wird die Verarbeitung in der übergeordneten Routine ohne weitere Verarbeitung sofort beendet. Der Parameter <List-Refresh> kann die Werte »X« (Liste wird neu ausgegeben, Defaultwert), »M« (die Liste wird neu ausgegeben und die Markierungen bleiben erhalten), sowie eine Leerstelle (die Liste wird nicht neu aufgebaut) enthalten.

Feldname	Daten-typ	Bedeutung
NAME	CHAR 30	Name des ausgewählten Knotens
ID	NUMC 6	ID des ausgewählten Knotens
TYPE	CHAR 4	Typ des ausgewählten Knotens
PARENT	NUMC 6	ID des Vaterknotens
CHILD	NUMC 6	ID des ersten Kindknotens des ausgewählten Knotens
NEXT	NUMC 6	ID des nächsten Bruderknotens
NEWNAME	CHAR 30	Neuer Knotenname (nur bei Funktionscodes TRAD, TRRN, TRMV (siehe unten))
NEWCHILD	CHAR 1	Enthält bei den Funktionscodes TRAD sowie TRMV ein X, falls der neue Knoten als Kindknoten an den ausgewählten Knoten angehangen werden soll
NEWTYPE	CHAR 4	Typ des neuen Knotens (Funktioscode TRAD)
TEXT	CHAR 30	Bezeichnung des Knotens
LINK_ID	NUMC 6	ID des gleichnamigen Bezugsknotens (siehe auch Abschnitt 3.1.5).
LINK_TYPE	CHAR 4	Typ des gleichnamigen Bezugsknotens
OK	CHAR 1	Bestätigungsflag für Callback-Routine

Tabelle 3.3
Aufbau der Struktur SEUCOMM

Neben den systemseitig definierten Funktionscodes (siehe Tabelle 3.4) können in einem eigenen GUI-Status beliebige andere Funktionscodes definiert werden, um das Verhalten der Baumdarstellung den eigenen Bedürfnissen anzupassen.

Funktionscode	Bedeutung
TRRF	Auffrischen der Baumdarstellung
TRSL	Auswählen eine Knotens
TREP	Der Teilbaum, auf dem der Cursor steht, wird vollständig expandiert. Steht der Cursor auf keinem Knoten, wird der gesamte Baum expandiert.
TRCM	Der Baum wird ab dem Knoten, auf dem der Cursor steht vollständig komprimiert. Steht der Cursor auf keinem Knoten, wird der gesamte Baum komprimiert.
TRN+	Der nächste Bruderknoten in der Hierarchie wird um eine Stufe expandiert. Der aktuelle Knoten wird komprimiert.
TRN-	Der vorige Bruderknoten der Hierarchie wird um eine Stufe expandiert. Der aktuelle Knoten wird komprimiert.
TRG+	Von einem Referenzknoten wird zum Originalknoten gesprungen. Die Funktion kann nur sinnvoll angewendet werden, wenn der Parameter CHECK_DUPLIKATE_NAME auf 3 (siehe auch gesetzt wurde Abschnitt 3.1.5).
TRG-	Rücksprung vom Originalknoten auf Referenzknoten. Die Aufrufe von TRG+ werden im System gespeichert und sind daher schrittweise rückwärts abrufbar.
TRZM	Der Knoten, auf dem der Cursor aktuell steht, wird in der ersten Bildschirmzeiel dargestellt. Über der Baumdarstellung wird der Pfad im Baum bis zum dargestellten Knoten ausgegeben.
TRMK	Der aktuelle Knoten und der ggf. darunter liegende Teilbaum werden markiert bzw. entmarkiert.
TRAD	Knoten hinzufügen. Vor Aufruf der FORM wurde vom System ein Dynpro angezeigt, in dem der Benutzer den neuen Knoten definieren konnte. Der FORM-Routine zur Auswertung der Funktionscodes wird eine Struktur vom Typ SEUCOMM übergeben. Die FORM-Routine muss den neuen Knoten über das Feld OK der Struktur als gültig markieren.
TRDL	Knoten löschen. Die FORM-Routine muss die Löschung des Knotens bestätigen, indem das Feld OK der Struktur SEUCOMM auf X gesetzt wird.
TRRN	Knoten umbenennen. Die FORM-Routine muss den neuen Knotennamen, der im Feld NEWNAME der Struktur SEUCOMM übergeben wird, im Feld OK bestätigen.
TRMV	Knoten umhängen

Funktionscode	Bedeutung
TRCL	Anzeigen der Farblegende (siehe Abschnitt 3.1.4)
TRRT	Zurück F3
BACK	Der Funktionsbaustein wird ohne Aufruf der FORM-Routine für die Funktionscodeauswertung verlassen.
TREX	Beenden F15. Wird der Parameter <Exit> der Schnittstelle zur Behandlung der Funktion auf X gesetzt, wird die Baumdarstellung verlassen und der Rückgabeparameter F15 des Funktionsbausteins RS_TREE_LIST_DISPLAY auf X gesetzt.
EXIT	Die Baumdarstellung wird ohne Aufruf der benutzerdefinierten Behandlungsroutine für Funktionscodes verlassen. Der Ausgabeparameter F15 des Funktionsbausteins RS_TREE_LIST_DISPLAY wird gesetzt.
%SC	Suchen in angezeigten Knoten
%PRI	Drucken der Baumdarstellung
%PC	Download

Tabelle 3.4
Funktionscodes im Standard-Status bei Baumdarstellungen

Da im Beispielprogramm interaktive Manipulationen an der Baumdarstellung nicht erwünscht sind, sollen die Funktionscodes TRAD, TRDL, TRRN sowie TRMV nicht angeboten werden. Es soll ein neuer Funktionscode BOOK definiert werden, mit dem zu einem konkreten Flugevent weitere Informationen angezeigt werden können, sowie DELE zum Löschen von Knoten/Teilbäumen (nur in der Anzeige) und EXPA und COMP zum Expandieren und Komprimieren von Teilbäumen.

Hierzu wird eine Kopie des im Programm SAPLSEUT definierten GUI-Status LD_TREE im Beispielprogramm angelegt. Dieser Kopie werden die beschriebenen Funktionscodes hinzugefügt. Um den neuen Status in der Anzeige zu verwenden, muss die Routine SHOW_TREE angepasst werden:

```
FORM SHOW_TREE.
  PERFORM SET_PF_STATUS.

  CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
       EXPORTING
            CALLBACK_PROGRAM          = SY-CPROG
            CALLBACK_USER_COMMAND     = 'COMMAND_HANDLER'
            ...
            STATUS                    = 'OWN'
            ...
ENDFORM.
```

Bevor der Baum auf dem Bildschirm ausgegeben wird, wird durch Aufruf der neuen FORM-Routine SET_PF_STATUS explizit der Benutzerstatus gesetzt. Um zu verhindern, dass dieser bei der Anzeige des Baums wieder durch den Standard-Status ersetzt wird, wird der Parameter STATUS mit dem Wert »OWN« belegt. Da im dargestellten Beispiel auch die Parameter CALLBACK_PROGRAMM, sowie CALLBACK_USER_COMMAND versorgt wurden, hätte der Parameter STATUS auch mit dem Wert IMPLICIT belegt werden können.

Das gleiche Ergebnis wie oben dargestellt kann außerdem auch über die Verwendung einer Callback-Routine zum Setzen des GUI-Status realisisert werden. Da die erwartete FORM-Routine bei Verwendung dieses Verfahrens keine Parameterschnittstelle definiert, könnte das gleiche Verhalten wie im obigen Programmfragment auch durch folgenden Code erreicht werden.

```
FORM SHOW_TREE.
  CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
       EXPORTING
            CALLBACK_PROGRAM          = SY-CPROG
            CALLBACK_USER_COMMAND     = 'COMMAND_HANDLER'
            ...
            CALLBACK_GUI_STATUS       = 'SET_PF_STATUS'
            ...
ENDFORM.
```

Die Implementierung der Routine SET_PF_STATUS definiert hierbei eine interne Tabelle der auszuschließenden Funktionscodes und setzt anschließend den GUI-Status unter Verwendung dieser Tabelle.

```
FORM SET_PF_STATUS.
  DATA: BEGIN OF LT_EX_FCODES OCCURS 0,
          FCODE LIKE STREE_UCOMM,
        END OF LT_EX_FCODES.

* Tabelle zu exkludierender Funktionscodes aufbauen
  REFRESH LT_EX_FCODES.

  LT_EX_FCODES-FCODE = STREE_CMD_ADD_TYPE.
  APPEND LT_EX_FCODES.
  LT_EX_FCODES-FCODE = STREE_CMD_DELETE.
  APPEND LT_EX_FCODES.
  LT_EX_FCODES-FCODE = STREE_CMD_RENAME.
  APPEND LT_EX_FCODES.
  LT_EX_FCODES-FCODE = STREE_CMD_MOVE.
  APPEND LT_EX_FCODES.

  SET PF-STATUS 'LD_TREE' EXCLUDING LT_EX_FCODES.
ENDFORM.
```

Nachdem nun der gewünschte GUI-Status bei der Anzeige verwendet wird, fehlt lediglich noch die Implementierung der Routine zur Behandlung der Funktionscodes. Im ersten Schritt genügt es, diese ohne Verarbeitungen im Programm einzufügen.

```
FORM COMMAND_HANDLER  TABLES   NODES        STRUCTURE SEUCOMM
                      USING    COMMAND
                      CHANGING EXIT
                               LIST_REFRESH.

ENDFORM.
```

Analog zum GUI-Status kann mit der Routine SET_PF_STATUS auch der Programmtitel beeinflusst werden.

Bis hierher wurde gezeigt, wie der verwendete GUI-Status während der Anzeige eines Tree-Views beeinflusst werden kann. Durch die Verwendung einer Kopie des vom System verwendeten GUI-Status LD_TREE des Programms SAPLSEUT können eigene Funktionscodes definiert werden. System-Funktionscodes können entweder durch explizites Ausschließen beim Setzten des GUI-Status (SET PF-STATUS ... EXCLUDING ...) oder durch Entfernen der unerwünschten Funktionen aus der Kopie des GUI-Status realisiert werden. Die Implementierung der selbstdefinierten Funktionscodes erfolgt im Abschnitt 3.1.5.

3.1.4 Definition einer Farblegende zur Darstellung

In der Darstellung des Baumes wurde für die Belegung der Plätze eines Fluges eine Farbcodierung verwendet. Um dem Benutzer das verwendete Farbschema zu erklären, besteht die Möglichkeit, dieses in Form einer Farblegende anzeigen zu lassen. SAP hat für diesen Zweck im Standard-GUI für Baumdarstellungen (LD_TREE) den Funktionscode TRCL (STREE_CMD_COLOR_LEGEND) definiert.

Wird der Funktionscode ausgelöst, zeigt R/3 das verwendete Farbschema in Form eines Pop-Up-Fensters an. Für den Programmierer besteht hier die Möglichkeit einzugreifen und das eigene Farbschema zu erklären.

Für diesen Zweck definiert der Funktionsbaustein RS_TREE_LIST_DISPLAY den Parameter CALLBACK_COLOR_DISPLAY, der in Zusammenhang mit dem Parameter CALLBACK_PROGRAM eine FORM-Routine mit folgender Schnittstellenbeschreibung definiert.

```
FORM <Form-Name> TABLES <Farb-Tab> STRUCTURE SEUCOLOR.
```

Die Routine liefert im Parameter <Farb-Tab> eine Tabelle der verwendeten Farbdefinitionen zurück Die Struktur SEUCOLOR hat den in Tabelle 3.5 gezeigten Aufbau.

Feldname	Datentyp	Bedeutung
COLOR	NUMC 1	Farbdarstellung des Knotens
INTENSIV	CHAR 1	Intensität der Darstellung
TEXT	CHAR 75	Erläuterung der Farbverwendung
ICON	CHAR 4	Ikone in Textfeldern
SYMBOL	CHAR 2	Symbol in Textfeldern

Tabelle 3.5
Aufbau der Struktur SEUCOLOR

In den Feldern COLOR und INTENSIV wird der jeweilige Farbwert definiert. Die übrigen Felder dienen der Erklärung der Farbcodierung.

Der Tabellenparameter <Farb-Tab> ist beim Betreten der FORM-Routine mit den, dem System bekannten, Farbwerten vorbelegt. Diese können in der Routine um eigene Definition ergänzt werden.

Im Beispiel wird der Aufruf des Funktionsbausteins RS_TREE_LIST_DISPLAY wie dargestellt erweitert:

```
CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
     EXPORTING
          CALLBACK_PROGRAM          = SY-CPROG
          ...
          CALLBACK_COLOR_DISPLAY    = 'COLOR_DISPLAY'
          ...
```

Die Implementierung mit dem verwendeten Farbschema könnte folgendes Aussehen besitzen:

```
FORM COLOR_DISPLAY TABLES P_COLOR_TAB STRUCTURE SEUCOLOR.
  CLEAR   P_COLORTAB.

* Flüge mit weniger als 70% Buchungsquote
  P_COLOR_TAB-COLOR    = 5
  P_COLOR_TAB-INTENSIV = SPACE.
  P_COLOR_TAB-TEXT     = 'Flüge mit weniger als 70% Auslastung'(002).
  APPEND P_COLORTAB.

* Flüge mit 70-95% Buchungslage
  P_COLOR_TAB-COLOR    = 3
  P_COLOR_TAB-INTENSIV = SPACE.
  P_COLOR_TAB-TEXT     = 'Flüge mit 70%-95% Auslastung'(003).
  APPEND P_COLORTAB.
```

```
* Flüge mit mehr als 95% Auslastung
  P_COLOR_TAB-COLOR    = 6
  P_COLOR_TAB-INTENSIV = SPACE.
  P_COLOR_TAB-TEXT     = 'Flüge mit mehr als 95% Auslastung'(004).
  APPEND P_COLORTAB.
ENDFORM.
```

Wird in der Anzeige der Funktionscode `TRCL` ausgewählt, erscheint das in Abbildung 3.3 dargestellte Fenster.

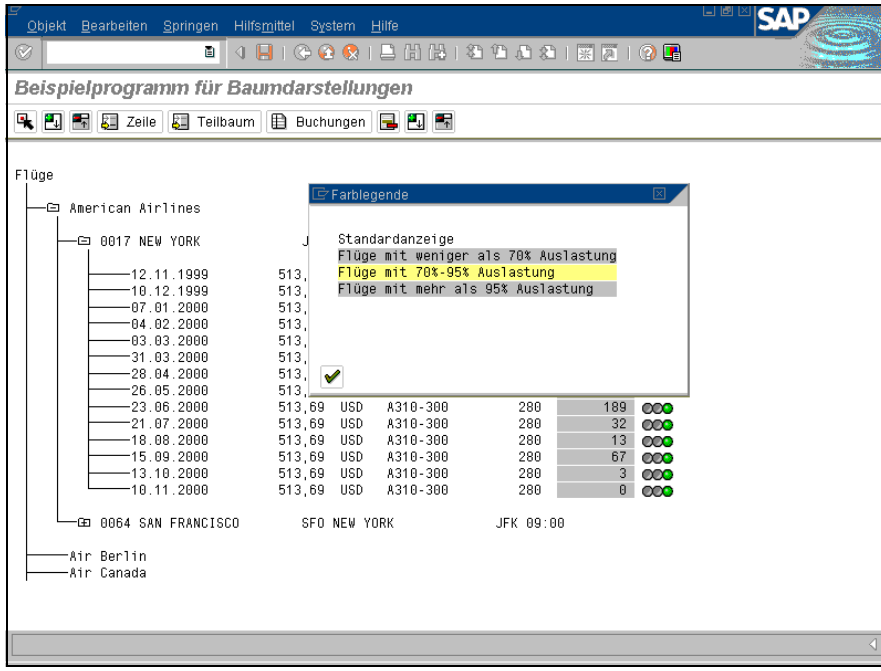

Abbildung 3.3
Legende zum verwendeten Farbschema (© SAP AG)

Das PopUp enthält die Farbdefinitionen für alle vorkommenden Knotentypen sowie Einträge für das selbstdefinierte Farbschema.

3.1.5 Weitere Gestaltungsmöglichkeiten der Darstellung

Zusatzinformationen zu Knoten anzeigen

Wie bereits in Abschnitt 3.1.3 erwähnt, besteht die Möglichkeit, zu jedem beliebigen Knoten weitere Informationen anzuzeigen. Um zu erkennen, für welche Knoten weitere Daten dargestellt werden sollen, existiert in den Knotenattributen (Struktur STREEATTR, siehe Tabelle 3.1) das Feld MOREINFO. Bei der Anzeige des Baumes wird für jeden Knoten, der in diesem Feld den Wert »X« enthält, die, dem Funktionsbaustein RS_TREE_LIST_DISPLAY im Parameter CALLBACK_MOREINFO übergebene, FORM-Routine aufgerufen und kann weitere Informationen zum jeweiligen Knoten ausgeben. Die so übergebene Routine muss die unten stehende Aufrufschnittstelle haben.

```
FORM <Form-Name> TABLES <MoreInfo-Tab> STRUCTURE STREEATTR
                 USING  <Node>         LIKE      STREENODE.
```

Im Beispiel sollen die Buchungen zu den jeweiligen Flügen angezeigt werden. Dies ist nur auf der untersten Baumebene, der Ebene der tatsächlichen Flüge sinnvoll. Aus diesem Grund erfolgt am Beginn der Routine MORE_INFO eine Abfrage, ob der jeweilige Knoten vom gewünschten Typ ist (zu Knotentypen siehe auch Abschnitt 3.1.1).

```
FORM MORE_INFO TABLES P_MOREINFO_TAB STRUCTURE STREEATTR
               USING  P_NODE         LIKE      STREENODE.
  DATA: L_NODETEXT  LIKE SNODETEXT,
        L_SFLIGHT   LIKE SFLIGHT.

* Zusatzinfos nur für konkrete Flüge
  CHECK P_NODE-TYPE EQ C_NODETYPE-FLIGHTEVENT.

* Tabellenparameter löschen
  REFRESH P_MOREINFO_TAB.

* Knotenattribute ermitteln
  CALL FUNCTION 'RS_TREE_GET_NODE'
       EXPORTING
            NODE_ID      = P_NODE-ID
       IMPORTING
            NODE_INFO    = L_NODETEXT
       EXCEPTIONS
            ID_NOT_FOUND = 1
            OTHERS       = 2.

* Flug aus HIDE-Bereich ermitteln
  L_SFLIGHT = L_NODETEXT-HIDE.
```

```
* Buchungen lesen und Anzeigeinformationen erzeugen
  SELECT *
         FROM  SBOOK
         WHERE CARRID  = L_SFLIGHT-CARRID AND
               CONNID  = L_SFLIGHT-CONNID AND
               FLDATE  = L_SFLIGHT-FLDATE.
*   Kunde ermitteln
    SELECT SINGLE *
           FROM   SCUSTOM
           WHERE  ID   = SBOOK-CUSTOMID.

*   Information aufbereiten
    WRITE: SCUSTOM-NAME  TO P_MOREINFO_TAB-TEXT,
           SBOOK-CLASS   TO P_MOREINFO_TAB-TEXT1.

    P_MOREINFO_TAB-TLENGTH  = 30.
    P_MOREINFO_TAB-TLENGTH1 = 8.

    APPEND P_MOREINFO_TAB.
  ENDSELECT.
ENDFORM.
```

Um die Buchungen zum jeweiligen Flug aus der Tabelle SBOOK ermitteln zu können, muss der Schlüssel des Fluges bekannt sein. Dieser wurde im HIDE-Bereich des Knotens abgelegt (siehe Abschnitt 3.1.2). Die FORM-Routine MORE_INFO erhält jedoch lediglich die Struktur STREENODE zum jeweiligen Knoten. In diesem sind die benötigten Informationen nicht enthalten. Mit dem Funktionsbaustein RS_TREE_GET_NODE können jedoch bei bekannter Knoten-ID die übrigen Attribute (siehe Tabelle 3.6) zum Knoten ermittelt werden. Aus dem HIDE-Bereich des Knotens wird der Datensatz des zugrunde liegenden Satzes der Tabelle SFLIGHT rekonstruiert.

Jetzt können aus der Tabelle SBOOK die Buchungen zum Flug gelesen werden. Über die Kundennummer wird aus der Tabelle SCUSTOM der zur Buchung gehörige Kunde ausgelesen.

Die FORM-Routine MORE_INFO liefert an das rufende Programm eine Tabelle vom Typ STREEATTR zurück. Innerhalb der beiden SELECT-Statements werden daher die gewünschten Informationen in die Knontenattribute-Tabelle übertragen. Zur Demonstration soll hier der Name des Kunden und die gebuchte Klasse genügen.

Textausgaben am Seitenanfang

Analog zum normalen Reporting können auch bei der Darstellung von Bäumen Informationen am Anfang der Seite ausgegeben werden. Da hier jedoch nicht mit den Standardereignissen des Reportings gearbeitet werden kann – der Re-

portingteil des Programms befindet sich in der Funktionsgruppe SEUT – besteht die Möglichkeit, eine FORM-Routine zu definieren, die von der Funktionsgruppe SEUT aufgerufen wird, wenn in dieser das Ereigniss TOP-OF-PAGE eintritt.

Hierzu muss eine FORM-Routine ohne Parameter definiert werden. Der Name dieser FORM-Routine wird dem Funktionsbaustein RS_TREE_LIST_DISPLAY im Parameter CALLBACK_TOP_OF_PAGE zusammen mit dem Programmnamen im Parameter CALLBACK_PROGRAM übergeben. Die FORM-Routine wird dann während der Darstellung des Baumes und beim Druck am Anfang jeder Seite aufgerufen und kann beliebige Informationen mit der Anweisung WRITE in die Liste ausgeben.

Textausgaben vor der Baumdarstellung

Genauso, wie im vorigen Abschnitt am Anfang jeder Seite, kann eine FORM-Routine definiert werden, die einmalig, vor der Ausgabe des Baumes, aufgerufen wird. Anders als die Routine, die in CALLBACK_TOP_OF_PAGE deklariert wird, muss die über CALLBACK_TEXT_DISPLAY definierte Routine über folgendes Interface verfügen.

```
FORM <Form-Name> TABLES <Node-Tab> STRUCTURE SNODETEXT.
```

Die Routine erhält für jeden darzustellenden Knoten einen Eintrag in der Tabelle <Node-Tab>. Die mit der WRITE-Anweisung getätigten Listenausgaben dieses Funktionsbausteins werden vor dem Wurzelknoten auf dem Bildschirm des Benutzers angezeigt bzw. dem Ausdruck gedruckt.

Die Struktur SNODETEXT (siehe Tabelle 3.6) enthält die vollständigen Informationen zu jedem Knoten. Dies beinhaltet zum einen die Informationen über die Position des Knotens in der Baumhierarchie sowie die Darstellungsattribute des Knotens.

Feldname	Datentyp	Bedeutung
INCLUDE STREENODE		Knotenattribute
ID	NUMC 6	ID des Knotens
TYPE	CHAR 4	Knotentyp
NAME	CHAR 30	Name des Knotens
TLEVEL	NUMC 2	Level des Knotens (Hierarchieebene)
LINK	CHAR 1	Kennzeichen, ob es sich um einen Link handelt
COMPRESS	CHAR 1	Komprimierungskennzeichen
LEAF	CHAR 1	Kennzeichen, ob der Knoten ein Blatt (Leaf) ist
TLOCK	CHAR 1	Kennzeichen, ob der Knoten markiert ist

Feldname	Datentyp	Bedeutung
PARENT	NUMC 6	ID des Vaterknotens
CHILD	NUMC 6	ID des ersten Kindknotens
NEXT	NUMC 6	ID des nächsten Bruderknotens
INCLUDE STREEATTR		Darstellungsattribute (siehe Tabelle 3.1)
INCLUDE STREECNTL		Knoteneigenschaften
PROPFAW0	CHAR 1	Flag für Proportionalschrift für das erste Knotenfeld
...		
PROPFAW9	CHAR 1	Flag für Proportionalschrift für das 10. Knotenfeld
STYLE0	NUMC 2	Tree Control Style für das erste Knotenfeld
...		
STYLE9	NUMC 2	Tree Control Style für das 10. Knotenfeld
NODEICON	CHAR 6	Ikone/eingebaute Bitmap/benutzerdef. Bitmap
LASTHTEXT	CHAR 5	Textfeld aus STREEATTR

Tabelle 3.6
Aufbau der Struktur SNODETEXT

Die Struktur SNODETEXT wird in den folgenden Kapiteln noch häufiger verwendet. Sie wird generell verwendet, um Informationen über Knoten zu ermitteln bzw. zu setzten.

Verwendung von Referenzknoten

Es wurde bereits erwähnt, dass die Informationen in Baumdarstellungen zwar nur hierarchisch dargestellt werden können, aber durchaus auch netzartige Strukturen haben können. Zu diesem Zweck steht das Konstrukt der Referenzknoten und der Link-Knoten zur Verfügung. Die Idee dabei ist, dass an Stellen, an denen die streng hierarchische Datenstruktur aufgebrochen wird, Knoten als Platzhalter für den Originalknoten eingefügt werden können. Über den Funktionscode TRG+ des Standard-Status kann von diesem Platzhalter zum jeweiligen Originalknoten verzweigt werden, über den Funktionscode TRG- des Standard-GUI-Status entsprechend wieder zurück.

Der Originalknoten wird hierbei wie in Abschnitt 3.1.2 dargestellt angelegt. Der Parameter LINK des Funktionsbausteins RS_TREE_ADD_NODE muss – wie als Defaultwert definiert – leer übergeben werden.

```
CALL FUNCTION 'RS_TREE_ADD_NODE'
       EXPORTING
            NEW_NAME           = P_SFLIGHT-FLDATE
            INSERT_ID          = P_PARENT
            RELATIONSHIP       = STREE_RELTYPE_CHILD
            LINK               = 'X'
            NEW_TYPE           = C_NODETYPE-FLIGHTEVENT
            DISPLAY_ATTRIBUTES = L_STREEATTR
       IMPORTING
            NEW_ID             = P_NODE
            NODE_INFO          =
       EXCEPTIONS
            ID_NOT_FOUND       = 1
            OTHERS             = 2.
```

Für Referenzknoten wird der gleiche Funktionsbaustein verwendet. Lediglich der Parameter `LINK` muss hier den Wert »`X`« enthalten (siehe Beispiel). Die Verbindung von Link-Knoten zum Original erfolgt hierbei über den Knotennamen. Beim Anlegen eines Link-Knotens überprüft R/3 nicht, ob ein entsprechender Originalknoten exisitert. Genausowenig wird beim Anlegen eines »normalen« Knotens nicht überprüft, ob bereits ein Originalknoten mit dem gleichen Namen existiert. Auch beim Löschen von Knoten oder Teilbäumen wird standardmäßig nicht überprüft, ob Originalknoten, zu denen Referenzen existieren, betroffen sind. Für die Konsistenz von Original- und Link-Knoten ist also der Programmierer selbst verantwortlich. Bei Verwendung der im Standard-GUI-Status seitens SAP definierten Funktionscodes kann jedoch über den Parameter `CHECK_DUPLICATE_NAME` des Funktionsbausteins `RS_TREE_LIST_DISPLAY` festgelegt werden, dass die Knotennamen im gesamten Baum (»`1`«) oder innerhalb der Geschwister (»`2`«) eindeutig sein müssen, bei doppelten Knotennamen automatisch Referenzknoten angelegt werden sollen (»`3`«) oder keine Prüfung bezüglich doppelter Knotennamen vorgenommen werden soll (»`0`«). Diese Prüfungen erfolgen jedoch nur, wenn der Parameter `SUPRESS_DUPLICATE_NAME` leer (Space) übergeben wird. Auch bei Verwendung dieser Option hat das Löschen von Originalknoten keinen Einfluss auf etwaige Referenzknoten.

```
CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
       EXPORTING
            ...
            CHECK_DUPLICATE_NAME      = '1'
            ...
            COLOR_OF_LINK             = '1'
            ...
            SUPPRESS_NODE_OUTPUT      = ' '
            ...
.
```

Den Referenzknoten kann in der Baumdarstellung zur optischen Unterscheidung eine eigene Farbe zugewiesen werden. Diese wird im Aufruf des Funktionsbausteins RS_TREE_LIST_DISPLAY im Parameter COLOR_OF_LINK definiert. Das dargestellte Codefragment für den Aufruf von RS_TREE_LIST_DISPLAY zeigt die Standardwerte für die hier relevanten Parameter.

Ausgabe des Baumes in einem modalen Fenster

Bisher wurde die Ausgabe des Beispielprogramms im Hauptbereich des Fensters dargestellt. Oftmals ist es aus Anwendungssicht jedoch notwendig, die hierarchischen Daten in einem modalen Fenster oberhalb des eigentlichen Anwendungsfensters darzustellen. Hierzu bietet der Funktionsbaustein RS_TREE_LIST_DISPLAY die Parameter SCREEN_START_COLUMN, SCREEN_START_ROW, SCREEN_END_COLUMN sowie SCREEN_END_ROW an. Über diese Parameter können die Eckpunkte eines Fensters definiert werden, in welchem der Baum angezeigt wird. Bei Verwendung eines eigenen GUI-Status ist hierbei wichtig, dass es sich um einen Dialogstatus handelt, der am unteren Rand eines modalen Fensters angezeigt werden kann.

Wird die Anweisung RS_TREE_LIST_DISPLAY des Beispielprogramms wie dargestellt geändert, ergibt sich die in Abbildung 3.4 dargestellte Bildschirmausgabe.

```
...
CALL FUNCTION 'RS_TREE_LIST_DISPLAY'
       EXPORTING
            ...
            SCREEN_START_COLUMN       = 5
            SCREEN_START_LINE         = 5
            SCREEN_END_COLUMN         = 80
            SCREEN_END_LINE           = 25
            ...
```

Sämtliche Möglichkeiten, die bei der Baumdarstellung als gesamtes Fenster zur Verfügung standen bleiben, bestehen. Die Baumdarstellung kann somit auch als Auswahldialog verwendet werden. Hierzu muss dann allerdings ein Funktionscode definiert werden, der das Fenster ordnungsgemäß schließt und die aktuelle Selektion an das übergeordnete Programm weitergibt.

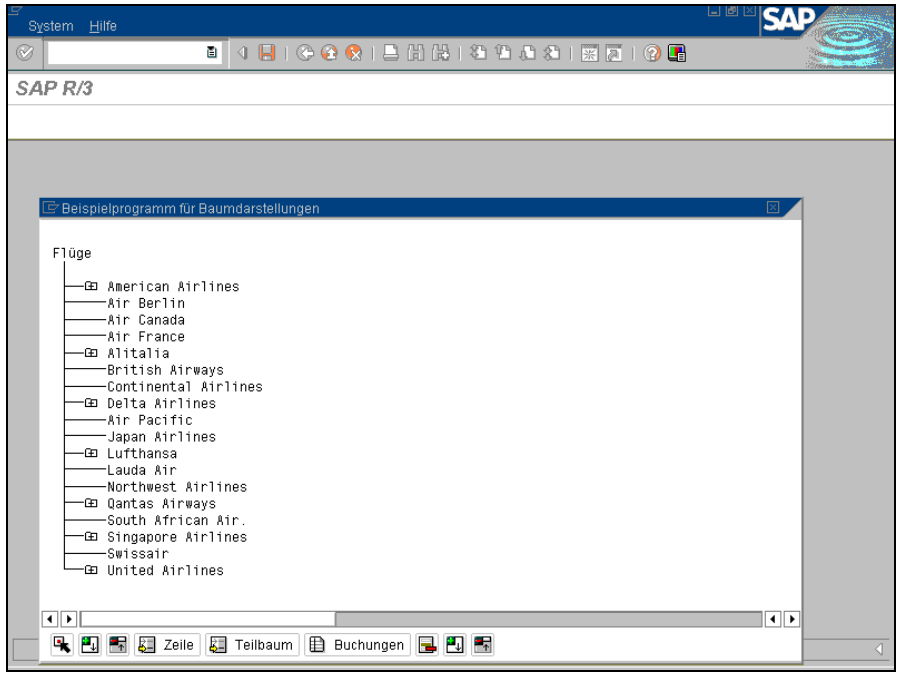

Abbildung 3.4
Baumdarstellung als modaler Dialog (© SAP AG)

3.1.6 Programmgesteuerte Veränderungen des Baums

Um das programmgesteuerte Verändern von Programmen zu demonstrieren, wurden in Abschnitt 3.1.3 die Funktionscodes `BOOK` zum Anzeigen weiterer Knoteninformationen, `DELE` zum Löschen eines Knotens/Teilbaums sowie `EXPA` und `COMP` zum Ausklappen bzw. Komprimieren eines Teilbaums definiert. Ebenfalls in Abschnitt 3.1.3 wurde eine Callback-Routine `COMMAND_HANDLER` erzeugt und als Parameter an den Funktionsbaustein `RS_TREE_LIST_DISPLAY` übergeben. Diese Routine erhält jetzt für alle auftretenden Funktionscodes die Kontrolle. Bei den System-Funktionscodes ist dies im Beispiel irrelevant. Für sie ist hier keine Aktion erforderlich. Anders sieht es für die selbstdefinierten Funktionscodes aus, die ohne eigene Implementierung keine Auswirkung im Programm zeigen würden.

Anzeige weiterer Knotenattribute

Wie bereits in Abschnitt 3.1.5 erläutert, können weitere Informationen zu den Knoten im Baum angezeigt werden. Dies erfolgt in der, dem Funktionsbaustein

RS_TREE_LIST_DISPLAY im Parameter CALLBACK_MOREINFO definierten FORM-Routine für die Knoten, in deren Attributen das Feld MOREINFO mit dem Wert X belegt ist.

Im GUI-Status des Beispiels wurde ein Funktionscode BOOK vorgesehen, mit dem dieses Flag ein- bzw. ausgeschaltet werden soll. Die Implementierung muss in der Callback-Routine COMMAND_HANDLER realisiert werden. Analog zur Dialogprogrammierung wird hier eine CASE-Schleife für die möglichen Funktionscodes realisiert.

Die Aufgabe beim Funktionscode BOOK besteht also darin, das MOREINFO-Flag der betroffenen Knoten umzusetzen. Die Knotentabelle, welche der FORM-Routine übergeben wird, enthält jedoch nicht die Knotenattribute selbst, sondern lediglich Informationen über die Position des Knotens sowie die ID des Knotens. Über diese ID wird mit Hilfe des Funktionsbausteins RS_TREE_GET_NODE, welcher bereits in Abschnitt 3.1.5 verwendet wurde, die vollständigen Knotenattribute ermittelt. Nachdem das Flag entsprechend angepasst wurde, müssen die Knoteninformationen über einen Aufruf des Funktionsbausteins RS_TREE_SET_NODE aktualisiert werden.

```
FORM COMMAND_HANDLER  TABLES   P_NODES        STRUCTURE SEUCOMM
                      USING    P_COMMAND
                      CHANGING P_EXIT
                               P_LIST_REFRESH.
  DATA: L_NODETEXT  LIKE SNODETEXT.

  CASE P_COMMAND.
    WHEN C_CMD-MORE_INFO.
      LOOP AT P_NODES.
*       Nur konkrete Flüge
        CHECK P_NODES-TYPE EQ C_NODE_TYPE-FLIGHTEVENT.

*       Knoten ermitteln
        CALL FUNCTION 'RS_TREE_GET_NODE'
             EXPORTING
                 NODE_ID       = P_NODES-ID
             IMPORTING
                 NODE_INFO     = L_NODETEXT
             EXCEPTIONS
                 ID_NOT_FOUND  = 1
                 OTHERS        = 2.

*       More-Info-Flag aktualisieren
        IF L_NODETEXT-MOREINFO EQ SPACE.
          L_NODETEXT-MOREINFO = 'X'.
        ELSE.
          CLEAR L_NODETEXT-MOREINFO.
        ENDIF.
```

```
*       Knoten aktualisieren
        CALL FUNCTION 'RS_TREE_SET_NODE'
             EXPORTING
                 NODEINFO        = L_NODETEXT
             EXCEPTIONS
                 ID_NOT_FOUND    = 1
                 OTHERS          = 2.
      ENDLOOP.

  ENDCASE.
ENDFORM.
```

Knoten/Teilbäume löschen

Für die Realisierung des Funktionscodes DELE zum Löschen von Knoten bzw. Teilbäumen muss in der im vorigen Abschnitt dargestellten FORM-Routine eine weitere WHEN-Klausel implementiert werden. Für das Löschen von Knoten/Teilbäume aus einem Baum steht der Funktionsbaustein RS_TREE_DELETE_NODE zur Verfügung. Der Funktionsbaustein löscht den Teilbaum unterhalb des gegebenen Knotens und – auf Wunsch – auch den Knoten selbst. Der Ausgangsknoten wird hierbei im Parameter NODE_ID übergeben. Über das Flag WITHOUT_ROOT kann gesteuert werden, ob der Ausgangsknoten ebenfalls gelöscht wird (Space), oder nur der darunter liegende Teilbaum (»X«).

Auch in diesem Beispiel besteht die Implementierung aus einer LOOP-Schleife über die selektierten Knoten mit einem Aufruf des genannten Funktionsbausteins.

```
FORM COMMAND_HANDLER  TABLES   P_NODES        STRUCTURE SEUCOMM
                      USING    P_COMMAND
                      CHANGING P_EXIT
                               P_LIST_REFRESH.
  DATA: L_NODETEXT  LIKE SNODETEXT,
        L_ROOT      TYPE C.

  CASE P_COMMAND.
    ...
    WHEN C_CMD-DELETE.
      LOOP AT P_NODES.
        IF P_NODES-TYPE EQ C_NODE_TYPE-ROOT.
          L_ROOT = 'X'.
        ELSE.
          L_ROOT = SPACE.
        ENDIF.

        CALL FUNCTION 'RS_TREE_DELETE_NODE'
             EXPORTING
```

```
                NODE_ID       = P_NODES-ID
                WITHOUT_ROOT  = L_ROOT
            EXCEPTIONS
                ID_NOT_FOUND  = 1
                OTHERS        = 2.
      ENDLOOP.
    ...
  ENDCASE.
ENDFORM.
```

Da im Beispiel nicht gewünscht ist, dass der Wurzelknoten gelöscht werden kann, wird für diesen der Parameter WITHOUT_ROOT mit dem Wert X belegt. Für alle anderen Knoten wird an dieser Stelle eine Leerstelle übergeben. Ob es sich bei einem konkreten Knoten um den Wurzelknoten handelt, wird anhand des benutzerdefinierten Knotentyps festgestellt. Beim Anlegen des Knotens wurde im Beispiel für den Wurzelknoten der in der Konstante C_NODE_TYPE-ROOT definierte Typ mit dem Wert ROOT definiert.

Teilbäume anzeigen/verstecken

Bleibt zuletzt noch die Realisierung der Funktionscodes EXPA sowie COMP. Mit EXPA soll der Teilbaum unterhalb des selektierten Knotens um eine Ebene ausgeklappt werden. Mit COMP soll der Teilbaum unterhalb des aktuellen Knotens vollständig versteckt werden. Hierfür stehen die Funktionsbausteine RS_TREE_EXPAND sowie RS_TREE_COMPRESS zur Verfügung. Die Implementierung erfolgt abermals in Form von WHEN-Klauseln innerhalb der Routine COMMAND_HANDLER.

```
FORM COMMAND_HANDLER  TABLES   P_NODES        STRUCTURE SEUCOMM
                      USING    P_COMMAND
                      CHANGING P_EXIT
                               P_LIST_REFRESH.

  ...
  CASE P_COMMAND.
    ...
    WHEN C_CMD-EXPAND.
      LOOP AT P_NODES.
        CALL FUNCTION 'RS_TREE_EXPAND'
             EXPORTING
                 NODE_ID      = P_NODES-ID
*                ALL          = ' '
*                LIST_SCROLL  =
*                DEPTH        = 1
             EXCEPTIONS
                 NOT_FOUND    = 1
                 OTHERS       = 2.
      ENDLOOP.
```

```
    WHEN C_CMD-COMPRESS.
      LOOP AT P_NODES.
        CALL FUNCTION 'RS_TREE_COMPRESS'
             EXPORTING
                 NODE_ID      = P_NODES-ID
             EXCEPTIONS
                 NOT_FOUND    = 1
                 OTHERS       = 2.
      ENDLOOP.
  ENDCASE.
ENDFORM.
```

Dem Funktionsbaustein RS_TREE_EXPAND wird im Parameter NODE_ID die ID des Ausgangsknotens übergeben. Weiterhin werden drei optionale Parameter definiert. Wird ALL mit dem Wert »X« belegt, werden alle Baumebenen unterhalb des Ausgangsknotens angezeigt. Mit dem Parameter LIST_SCROLL kann gesteuert werden, dass der Ausgangsknoten nach dem Expandieren in der ersten Bildschirmzeile dargestellt wird. Über den letzten Parameter DEPTH schließlich kann gesteuert werden, wieviele Baumebenen expandiert werden sollen. Dieser Parameter bezeichnet die Anzahl Ebenen, ausgehend von dem in NODE_ID gegebenen Knoten, die angezeigt werden sollen. Der Default von »1« bedeutet hier, dass der Baum um eine Ebene erweitert dargestellt wird.

Einfacher als beim Baustein RS_TREE_EXPAND ist die Aufrufschnittstelle des Funktionsbausteins RS_TREE_COMPRESS. Da festgelegt ist, dass beim Aufruf alle Baumebenen unterhalb des Ausgangsknotens versteckt werden sollen, genügt es, dem Funktionsbaustein diesen Knoten mitzuteilen. Dies erfolgt im Parameter NODE_ID.

Für beide Funktionsbausteine gilt, dass der Aufruf ignoriert wird, wenn es sich beim gegebenen Knoten um einen Blattknoten (Leaf) handelt.

3.1.7 Verwendung benutzerdefinierter Kontextmenüs

Wenn in der Baumdarstellung auf einen Knoten mit der rechten Maustaste geklickt wird, erscheint – wie in der normalen Windows-Oberfläche – ein kontextabhängiges Menü, in dem der Benutzer eine Aktion für den aktuellen Knoten durchführen kann. Mit der Einführung von ABAP OBJECTS in R/3 Release 4 ist dieses Kontextmenü vom Programmierer beeinflussbar. Hierzu wurde die Schnittstelle des Funktionsbausteins RS_TREE_LIST_DISPLAY um eine weitere Callback-Routine, die im Parameter CALLBACK_CONTEXT_MENU übergeben wird, erweitert. Diese FORM-Routine muss die dargestellte Schnittstelle haben.

```
FORM <Form-Name> USING <Node> LIKE SNODETEXT
                       <Menu> TYPE REF TO CL_CTMENU.
```

Neben dem Knoten, für den das Menü angezeigt werden soll, bekommt diese Routine als Parameter eine Referenz auf eine Instanz eines ABAP-Objekts vom Typ CL_CTMENU übergeben. Die Aufgabe dieser Routine besteht darin, in diesem Menü die gewünschten Menüeinträge anzulegen. Das so erzeugte Menü kann neben normalen Menüeinträgen auch Untermenüs sowie horizontale Trennlinien enthalten. Einzelne Einträge des Menüs können deaktiviert werden, falls der Menüpunkt im jeweiligen Kontext nicht möglich ist. Für diese Aufgaben stellt die Klasse CL_CTMENU die Methoden ADD_FUNCTION, ADD_SEPARATOR sowie ADD_SUBMENU zur Verfügung. Diese und die übrigen, hier nicht genannten Methoden der Klasse, geben dem Programmierer weitreichende Gestaltungsmöglichkeiten für Kontextmenüs an die Hand.

Die hier beschriebene Funktionalität setzt voraus, dass die Funktionstaste ⇧ + F10, welche für die Anzeige von Kontext-Menüs reserviert ist, im aktuellen GUI-Status belegt ist. Wird der Standard-Status verwendet, ist dies der Fall, andernfalls muss der Programmierer hierfür Sorge tragen.

Anlegen eines Menüeintrags

Einfache Menüeinträge werden über die Methode ADD_FUNCTION erzeugt. Ein Menüeintrag besteht hierbei mindestens aus einem Funktionscode und dem Text, mit dem der Menüeintrag im Menü aufgeführt werden soll. Die Schnittstelle der Methode lässt jedoch noch weitere Möglichkeiten zu.

```
METHOD ADD_FUNCTION
       IMPORTING
           FCODE
           TEXT
           ICON
           FTYPE
           DISABLED
           HIDDEN
           CHECKED
           ACCELERATOR
```

Im Parameter FCODE wird der auszuwertende Funktionscode übergeben, TEXT enthält die Bezeichnung des Menüeintrags. Die übrigen Parameter der Methode sind optional. ICON definiert eine Ikone, die mit dem Menüeintrag ausgegeben werden soll. In FTYPE wird der Funktionstyp analog zu den Funktionstypen im GUI-Status definiert. Funktionen, die im Parameter DISABLED den Wert »X« übergeben bekommen, werden nicht anwählbar im Menü dargestellt, Menüeinträge mit »X« im Parameter HIDDEN werden nicht dargestellt. Über den Parameter CHECKED kann der Menüpunkt mit einem Häckchen im Menü dargestellt werden. ACCELERATOR legt ein Zeichen für die Direktanwahl des Menüpunkts fest.

Für die Knoten im Baum des Beispiels dieses Kapitels sollen Kontextmenüs angeboten werden. Hierbei sollen die benutzerdefinierten Funktionscodes ange-

boten werden, d.h. für alle Knoten soll die Möglichkeit bestehen, die darunter liegende Ebene - sofern vorhanden - auszuklappen sowie die Knoten mit ihren darunter liegenden Teilbäumen zu löschen. Für die Knoten der dritten Ebene soll weiterhin die Möglichkeit bestehen, zusätzliche Informationen anzuzeigen bzw. die Anzeige dieser Informationen zu unterdrücken.

Hierzu wird die FORM-Routine CONTEXT_MENU mit der oben dargestellten Schnittstelle definiert.

```
FORM CONTEXT_MENU USING P_NODE LIKE SNODETEXT
                        P_MENU TYPE REF TO CL_CTMENU.
  DATA: L_CUA_ACTIVE  TYPE C.

* Aufklappen bzw. Einklappen wo möglich
  IF P_NODE-LEAF NE SPACE.
*   COMPRESS-Flag für EXPAND ermitteln
    IF P_NODE-COMPRESS EQ SPACE.
      L_CUA_ACTIVE = 'X'.
    ELSE.
      CLEAR L_CUA_ACTIVE.
    ENDIF.

    CALL METHOD P_MENU->ADD_FUNCTION
                EXPORTING
                   FCODE    = C_CMD-EXPAND
                   TEXT     = 'Ausklappen'
                   DISABLED = L_CUA_ACTIVE.

    CALL METHOD P_MENU->ADD_FUNCTION
                EXPORTNG
                   FCODE    = C_CMD-COMPRESS
                   TEXT     = 'Komprimieren'
                   DISABLED = P_NODE-COMPRESS.
  ENDIF.

* Knoten/Teilbaum löschen
  CALL METHOD P_MENU->ADD_FUNCTION
              EXPORTNG
                 FCODE  = C_CMD-DELETE
                 TEXT   = 'Löschen'.

* More-Info!
  IF P_NODE-TYPE EQ C_NODETYPE-FLIGHTEVENT.
    CALL METHOD P_MENU->ADD_FUNCTION
                EXPORTING
                   FCODE   = C_CMD_MORE_INFO
                   TEXT    = 'Buchungen'
```

```
                    CHECKED = P_NODE-MOREINFO.
  ENDIF.
ENDFORM.
```

In dieser Routine wird zunächst ein Eintrag zum Aufklappen bzw. Komprimieren des Knotens angelegt, soweit es sich bei dem Knoten nicht um ein Blatt (ohne Unterknoten) handelt. Je nachdem, ob der unter dem gewählten Knoten liegende Teilbaum bereits angezeigt wird oder nicht, wird der jeweils passende Menüeintrag aktiviert bzw. deaktiviert dargestellt. Hierzu wird der Parameter DISABLED der Methode ADD_FUNCTION verwendet.

Im zweiten Schritt wird ein Menüeintrag zum Löschen des Knotens bzw. Teilbaums angelegt. Daran folgt für Knoten, die konkrete Flüge repräsentieren, ein Eintrag, um weitere Informationen zum Flug (in diesem Fall die Buchungen) anzuzeigen. Dieser Menüeintrag ist je nach Inhalt des Feldes MOREINFO der Knotenattribute markiert bzw. nicht markiert.

Die Auswertung der Funktionscodes erfolgt wie bei den Funktionscodes des GUI-Status in der im Parameter CALLBACK_COMMAND an den Funktionsbaustein RS_TREE_LIST_DISPLAY definierten FORM-Routine (siehe Abschnitt 3.1.3 bzw. Abschnitt 3.1.6), im Beispiel der Routine COMMAND_HANDLER.

Horizontale Trennlinien im Kontextmenü

Auch in benutzerdefinierten Kontextmenüs besteht die Möglichkeit, Menüeinträge aus Gründen der Übersichtlichkeit in – durch horizontale Linien – getrennten Gruppen einzuteilen. Um in ein Menü eine solche Trennlinie einzufügen, steht die Methode ADD_SEPARATOR der Klasse CL_CTMENU zur Verfügung. Die Methode definiert keine Parameter. Um eine Trennlinie in ein Menü einzufügen, genügt daher der Aufruf:

```
CALL METHOD P_MENU->ADD_SEPARATOR.
```

Diese Aufrufe sollen im Beispielprogramm hinter den Menüeinträgen zum Ausklappen/Komprimieren von Teilbäumen sowie nach dem Menüeintrag zum Löschen eingesetzt werden. Wegen der Einfachheit des Methodenaufrufes wird auf eine Darstellung der veränderten FORM-Routine hier verzichtet.

Untermenüs erzeugen

Wie bereits erwähnt, besteht bei Kontextmenüs die Möglichkeit, weitere Menüebenen einzufügen. Hierzu wird die Methode ADD_SUBMENU der Klasse CL_CTMENU, deren Schnittstelle wie dargestellt definiert ist, verwendet.

```
METHOD ADD_SUBMENU
       IMPORTING
          MENU
          TEXT
```

```
ICON
DISABLED
HIDDEN
ACCELERATOR
```

Im Gegensatz zur Methode ADD_FUNCTION wird hier als erster Parameter eine Referenz auf eine Instanz der Klasse CL_CTMENU, d.h. ein anderes Kontextmenü, erwartet. Ansonsten gleichen sich die Schnittstellen der beiden Methoden bis auf die Tatsache, dass der Parameter CHECKED bei Untermenüs keinen Sinn macht und daher nicht definiert ist.

Zu bemerken ist hierbei, dass die Instanz des Untermenüs bei Aufruf der Methode ADD_SUBMENU bereits existieren muss. Dies bedeutet, dass die Untermenüstrukturen in Kontextmenüs nicht Top-Down (d.h. von oben nach unten), sondern Bottom-Up (d.h. von der tiefsten Menüebene nach oben) angelegt werden. Zum Zeitpunkt des Aufrufs muss jedoch lediglich die Instanz des Untermenüs selbst existieren. Die Menüeinträge in den Instanzen können hingegen auch zu einem späteren Zeitpunkt angelegt werden.

3.1.8 Verwendete und wichtige Funktionsbausteine

RS_TREE_ADD_NODE

Mit der Funktion RS_TREE_ADD_NODE wird zum aktuellen Baum ein Knoten hinzugefügt.

Import-Parameter	
NEW_NAME	Name des neuen Knotens
INSERT_ID	ID des Knotens, zu dem der neue Knoten angelegt werden soll
RELATIONSHIP	Beziehung des neuen Knotens zum Knoten in INSERT_ID
LINK	Knoten ist ein Referenzknoten (LINK)
NEW_TYPE	(Benutzerdefinierter) Typ des neuen Knotens
DISPLAY_ATTRIBUTES	Darstellungsattribute des neuen Knotens (STREEATTR)
Export-Parameter	
NEW_ID	ID des neu erzeugten Knotens
NODE_INFO	Vollständige Knoteninformationen des neu erstellten Knotens
Exceptions	
ID_NOT_FOUND	Knoten INSERT_ID nicht vorhanden

RS_TREE_COMPRESS

Verbergen der Unterknotenstrukturen zu einem Knoten.

Import-Parameter	
NODE_ID	Knoten-ID, dessen Unterstrukturen nicht angezeigt werden sollen
Exceptions	
NOT_FOUND	Knoten nicht vorhanden

RS_TREE_CONSTRUCT

Erzeugen von Knoten aus einer Tabelle von Knotenelementen. Wird keine `INSERT_ID` übergeben, wird ein neuer Baum erzeugt, ansonsten werden die Knoten aus der Tabelle `NODETAB` in den bestehenden Baum eingefügt.

Import-Parameter	
INSERT_ID	ID des Knotens, zu dem die neuen Knoten eingefügt werden sollen
RELATIONSHIP	Beziehung, mit der die neuen Knoten angelegt werden sollen
Tabellen	
NODETAB	Tabelle mit Knoteninformationen zu den neuen Knoten
Exceptions	
TREE_FAILURE	Knoten konnten nicht erzeugt werden
ID_NOT_FOUND	ID des Einfügeknotens existiert nicht
WRONG_RELATIONSHIP	Ungültige Beziehung angegeben

RS_TREE_CREATE

Erzeugen eines neuen, leeren Baumes.

Import-Parameter	
ROOT_NAME	Name des Wurzelknotens
ROOT_TYPE	(Benutzerdefinerter) Typ des Wurzelknotens
DISPLAY_ATTRIBUTES	Darstellungsattribute des Knotens
Export-Parameter	
ROOT_ID	ID des Wurzelknotens des neu erstellten Baumes

RS_TREE_DELETE_NODE

Löschen eines Knotens mit allen darunter liegenden Unterknoten.

Import-Parameter	
NODE_ID	ID des zu löschenden Knotens
WITHOUT_ROOT	Kennzeichen, ob der Knoten selbst oder nur dessen Unterstrukturen zu löschen sind
Exceptions	
ID_NOT_FOUND	Knoten nicht vorhanden

RS_TREE_EXPAND

Ausklappen der Unterknoten eines gegebenen Knotens. Die darzustellende Tiefe der Unterstrukturen kann über Parameter gesteuert werden.

Import-Parameter	
NODE_ID	Knoden-ID, dessen Unterstrukturen angezeigt werden sollen
ALL	Kennzeichen, ob alle darunter liegenden Ebenen angezeigt werden sollen (X) oder nur die in DEPTH definierte Anzahl Unterebenen (SPACE)
LIST_SCROLL	Kennzeichen, ob der in NODE_ID übergebene Knoten als erster Knoten auf dem Bildschirm dargestellt werden soll
DEPTH	Tiefe der darzustellenden Baumebenen (Default = 1). Parameter hat nur Wirkung, wenn ALL = SPACE
Exceptions	
NOT_FOUND	Knoten ist nicht vorhanden

RS_TREE_GET_NODE

Ermitteln der Knotenattribute zu einer Knoten-ID.

Import-Parameter	
NODE_ID	ID des Knotens
Export-Parameter	
NODE_INFO	Knotenattribute des Knotens
Exceptions	
ID_NOT_FOUND	Knoten ist nicht vorhanden

RS_TREE_LIST

Rückgabe der Knoten eines Baumes in Form einer internen Tabelle. Diese kann erneut zur Erzeugung eines Baumes mit der Funktion `RS_TREE_CONSTRUCT` verwendet werden.

Import-Parameter	
NODE_ID	ID des Ausgangsknotens
ALL	Kennzeichen, ob alle Knoten zu ermitteln sind
WITH_ATTRIBUTES	Kenzeichen, ob Knoten mit vollständigen Knotenattributen ausgelesen werden sollen
HIDDEN_NODES	Kennzeichen, ob auch versteckte Knoten ausgelesen werden sollen
Tabellen	
LIST	Tabelle mit Knotendefinitionen (STREEATTR)
Exceptions	
CYCLE_DETECTED	Zyklische Knotenstruktur gefunden. Die LINK-Knoten des Baumes ergeben eine zyklische Stuktur und können deshalb auf diese Weise nicht ausgelesen werden
NODE_NOT_FOUND	Knoten ist nicht vorhanden

RS_TREE_LIST_DISPLAY

Anzeige des aktuellen Baumes.

Import-Parameter		
CALLBACK_PROGRAMM	Name des ABAP-Programms, in dem die Callback-Routinen abgelegt sind	
CALLBACK_USER_COMMAND	Name der Callback-Routine für die Auswertung von Funktionscodes	
CALLBACK_TEXT_DISPLAY	Name der Callback-Routine um Text vor der Baumdarstellung auszugeben	
CALLBACK_MOREINFO_DISPLAY	Name der Callback-Routine, um erweiterte Informationen zu Knoten auszugeben	
CALLBACK_COLOR_DISPLAY	Name der Callback-Routine, die die Farblegende definiert	
CALLBACK_TOP_OF_PAGE	Name der Callback-Routine, die am Anfang jeder Ausgabeseite aufgerufen wird	
CALLBACK_GUI_STATUS	Name der Callback-Routine, mit der der GUI-Status sowie die Überschrift gesetzt werden können	
CALLBACK_CONTEXT_MENU	Name der Callback-Routine, die zu einem Knoten das kontextsensitive Menü erstellt	
STATUS	Definiert, wie der GUI-Status während der Darstellung des Baumes gesetzt wird. Mögliche Werte:	
	OWN	Der vom Programm gesetzte Status wird beibehalten
	STANDARD	Der von SAP definierte Standard-Status wird gesetzt
	IMPLICIT	Falls eine Callback-Routine für Funktionscodes definiert ist wie OWN, sonst wie STANDARD
CHECK_DUPLICATE_NAME	Prüfung, ob beim Umbenennen und neu Anlegen von Knoten doppelte Knotennamen vorkommen können	
	‚0'	Keine Prüfung
	‚1'	Name muss in Gesamthierarchie eindeutig sein.

	‚2'	Name muss innerhalb der Hierarchiestufe eindeutig sein.
	‚3'	Wenn ein Knoten mit diesem Namen bereits existiert, wird eine Referenz auf diesen Knoten erstellt (LINK).
COLOR_OF_NODE	Standardfarbe eines normalen Knotens	
COLOR_OF_MARK	Knotenfarbe eines markierten Knotens	
COLOR_OF_LINK	Knotenfarbe eines Referenzknotens (LINK)	
COLOR_OF_MATCH	Knotenfarbe eines mit Suchen gefundenen Knotens	
LOWER_CASE_SENSITIVE	Kennzeichen, ob Groß- und Kleinschreibung unterstützt werden soll (Default X)	
MODIFICTION_LOG	Kennzeichen, ob Änderungsprotokoll geführt werden soll	
NODE_LENGTH	Allgemeine Knotenlänge	
TEXT_LENGTH	Standardlänge des ersten Textblocks	
TEXT_LENGTH1	Standardlänge des zweiten Textblocks	
TEXT_LENGHT2	Standardlänge des dritten Textblocks	
RETURN_MARKED_SUBTREE	Auch markierte Unterknoten berücksichtigen Hiermit wird gesteuert, ob an die Routine, die die Funktionscodes verarbeitet nur der markierte Knoten übergeben wird oder auch alle Unterknoten des markierten Knotens.	
SCREEN_START_COLUMN	Startspalte des PopUps, in dem der Baum dargestellt wird	
SCREEN_START_LINE	Startzeile des PopUps, in dem der Baum darstellt wird	
SCREEN_END_COLUMN	Endspalte des PopUps, in dem der Baum dargestellt wird	
SCREEN_END_LINE	Endzeile des PopUps, in dem der Baum dargestellt wird	
SUPRESS_NODE_OUTPUT	Unterdrückt die Knotenausgabe	
LAYOUT_MODE	Baumdarstellung ein-/zweizeilig je Knoten	

USE_CONTROL	
Export-Paramter	
F15	Kennzeichen, ob Baumdarstellung über F15 beendet wurde

RS_TREE_POP

Laden des aktuellen Baumes vom Stack. Über die Funktion RS_TREE_PUSH kann ein kompletter Baum auf einem Stack abgelegt werden. Die Funktion RS_TREE_POP lädt diesen wieder vom Stack herunter und macht den geladenen Baum zum aktiven Baum.

Exceptions	
EMPTY_STACK	Stack ist leer. Es wurde kein Baum auf dem Stack abgelegt.

RS_TREE_PUSH

Speichert den aktuellen Baum auf dem Stack.

Der Funktionsbaustein definiert keine Schnittstelle, d.h. weder Import- noch Export-Parameter. Auch Exceptions sind nicht definiert, d.h. der Funktionsbaustein liefert in keinem Fall ein Ergebnis.

RS_TREE_SET_NODE

Aktualisiert die Knotenattribute eines Knotens.

Import-Parameter	
NODE_INFO	Knotenattribute des zu aktualisierenden Knotens als Struktur mit Aufbau SNODETEXT (ID des Knotens ist in Struktur enthalten)
Exceptions	
ID_NOT_FOUND	Knoten nicht vorhanden

3.2 Tab-Strips

Seit dem Releasestand 4.0 besteht die Möglichkeit, bei der Programmierung von Oberflächen in R/3-Systemen so genannte Tab-Strips zu verwenden. Dieses Control, welches sich seit einigen Jahren in den grafischen Benutzeroberflächen der Betriebssysteme bewährt hat, wird verwendet, um Informationen, die den

Rahmen einer Maske sprengen würden, jedoch inhaltlich strukturiert sind, darzustellen. Anders als bisher, wo in solchen Fällen mehrere Masken, die über nicht standardisierte Wege hintereinandergeschaltet werden mussten, bieten Tab-Strips hier eine elegante und einheitliche Art der Darstellung. Bei der Programmierung von Tab-Strips wird ein Bereich auf der Maske definiert, in dem die unterschiedlichen Teilbereiche (oder Sichten) auf eine Informationsmenge angezeigt werden sollen. Diese Teilbereiche werden benannt. Dieser Name wird oberhalb des Anzeigebereiches auf einer Schaltflächen (auch Reiter genannt) angezeigt. Der Benutzer kann über diese Schaltflächen zwischen den einzelnen Seiten umschalten. Das Control versucht über optische Mittel eine Stapelung der einzelnen Reiter anzudeuten. Die aktuell dargestellte Seite wird dabei als die vorne liegende Karte dargestellt. Technisch gesehen bildet ein Tab-Strip eine Reihe von hintereinander angeordneten Subscreens. Jeder dieser Subscreens ist mit einer darüberliegenen Schaltfläche verbunden, über die der Benutzer zwischen den Ansichten wechseln kann. Abbildung 3.5 zeigt exemplarisch die Darstellung eines Tab-Strips. Es handelt sich hierbei um einen Screen-Shot des in diesem Abschnitt erstellten Beispielprogramms.

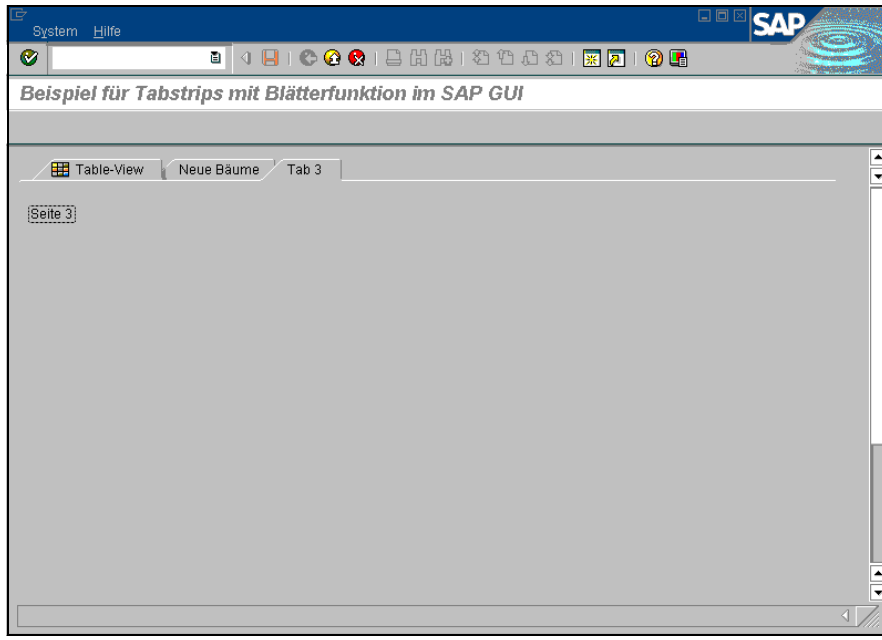

Abbildung 3.5
Beispiel für die Darstellung von Tab-Strips im R/3 (© SAP AG)

Über die Reiter kann der Benutzer die aktuell anzuzeigende Seite auswählen. Falls die Fensterbreite nicht ausreicht, um die Reiter für alle Tab-Seiten darzustellen, zeigt R/3 am rechten Rand der Reiterleiste Schaltflächen zum Blättern innerhalb der Reiter an.

Es ist sehr wichtig, hervorzuheben, wann diese Art der Darstellung gewählt werden kann. Da der Benutzer die Seiten des Tab-Strips in beliebiger Reihenfolge anspringen kann, können so keine Seitenfolgen dargestellt werden, die aufeinander aufbauend eine feste Reihenfolge der Darstellung erzwingen. Während sich Tab-Strips zum Beispiel für die Darstellung der verschiedenen Sichten des Materialstamms, die voneinander jeweils unabhängig sind, sehr gut eignen, ist deren Verwendung z.B. bei der Auftragseröffnung, wo aufeinander aufbauende Informationen erfasst werden sollen, nicht sinnvoll. Ein weitere Eigenschaft von Tab-Strips, die Einfluss auf die Benutzbarkeit des Controls nehmen können, besteht darin, dass alle Tab-Seiten den gleichen GUI-Status verwenden müssen. Bei der Programmierung von Sequenzen voneinander unabhängiger Bildschirme besteht diese Einschränkung nicht, es kann auf jedem Dynpro neu festgelegt werden, welcher GUI-Status und Titel verwendet werden soll.

Das Konzept von R/3 bietet bei der Verwendung von Tab-Strips zwei Möglichkeiten an, die den Wechsel der aktuell angezeigten Seite betreffen. Die eine Möglichkeit besteht darin, den Tab-Wechsel ausschließlich im SAPGUI des Benutzers vorzunehmen. Bei dieser Lösung müssen vor der Anzeige des Dynpros die Informationen für alle Tab-Seiten – auch die verdeckten – vom Applikationsserver übertragen werden. Wechseln zwischen den Seiten bedeutet hier, dass sich lediglich die Anzeige im SAPGUI des Benutzers verändert. Ein Wechsel der aktuellen Seite löst nicht das Ereignis PAI aus, es besteht demnach keine Möglichkeit, auf den Programmablauf Einfluss zu nehmen. Der andere von SAP angebotene Lösungsweg sieht das Blättern auf dem Applikationsserver vor. Bei dieser Lösung wird bei betätigen der Reiter das Ereignis PAI ausgelöst. In der Ablauflogik des Dynpros muss das Blättern, als Reaktion auf den dem Reiter zugeordneten Funktionscode, programmgesteuert erfolgen (siehe Abschnitt 3.2.3).

Die Vorteile der letzt genannten Methode liegen auf der Hand. Beim Wechsel der aktuellen Tab-Seite wird die vollständige Ablauflogik sowohl der dargestellten Seite als auch des Träger-Dynpros abgearbeitet. Plausibilisierungen und Ableitungen von Feldwerten sind hierbei leicht möglich. Nachteilhaft wirkt sich – neben dem erhöhten Programmieraufwand – die Tatsache aus, dass jeder Tab-Wechsel eine Kommunikation mit dem Applikationsserver bewirkt, was neben erhöhter Netzwerklast auch eine höhere Belastung des Applikationsservers mit sich bringt. Netzwerklast tritt beim Blättern im SAPGUI nur bei der Anzeige des Dynpros insgesamt auf. Es werden die Informationen für alle Tab-Seiten auf einmal an den Präsentationsrechner übertragen. Blättern erfolgt ohne Beteiligung des Applikationsservers.

Welche dieser Varianten schließlich verwendet wird, hängt von der Art der Anwendung ab und muss im Einzelfall entschieden werden.

In diesem Abschnitt sollen beide Lösungsansätze gezegt werden. Es wird ein Modulpool `SAPMZFLIGHT` erzeugt, in dem Tab-Strips mit drei Tab-Seiten angelegt werden. Während die dritte Tab-Seite nur angelegt wird, um die Möglichkeit zu zeigen, Tab-Seiten dynamisch auszublenden, werden die ersten beiden Seiten in

den folgenden Abschnitten verwendet, um die Erstellung von Drop-Down-Listen (Abschnitt 3.4) Table-Views (Abschnitt 3.5) sowie Baumdarstellungen mit der »neuen« Methode (Abschnitt 3.3) zu demonstrieren.

3.2.1 Anlegen von Tab-Strip-Controls

Anlegen des Träger-Dynpros

Für beide Programmiermodelle identisch ist das Anlegen des Controls im Dynpro-Editor selbst. Für das hier dargestellte Beispiel wird in der Entwicklungsumgebung der Modulpool `SAPMZFLIGHT` angelegt. In diesem Modulpool soll das Dynpro mit der Nummer `9000` als Träger-Dynpro für das Tab-Strip dienen. Das Tab-Strip soll hier mit drei Seiten angelegt werden. Diese werden in den Subscreen `9100`, `9200` sowie `9300`, die im Folgenden Abschnitt erzeugt werden, repräsentiert.

Zunächst jedoch muss das Träger-Dynpro mit dem Tab-Strip selbst erzeugt werden. Im grafischen Dynpro-Editor muss hierzu das Icon für Tab-Strips am linken Fensterrand angeklickt werden. Anschließend kann auf der Maske der Bereich, der vom Tab-Strip auf dem Dynpro eingenommen werden soll, aufgezogen werden. Die Größe und Positon des Tab-Strips kann auch im Nachhinein mit der Maus auf der Maske oder in den Eigenschaften des Tab-Strips verändert werden.

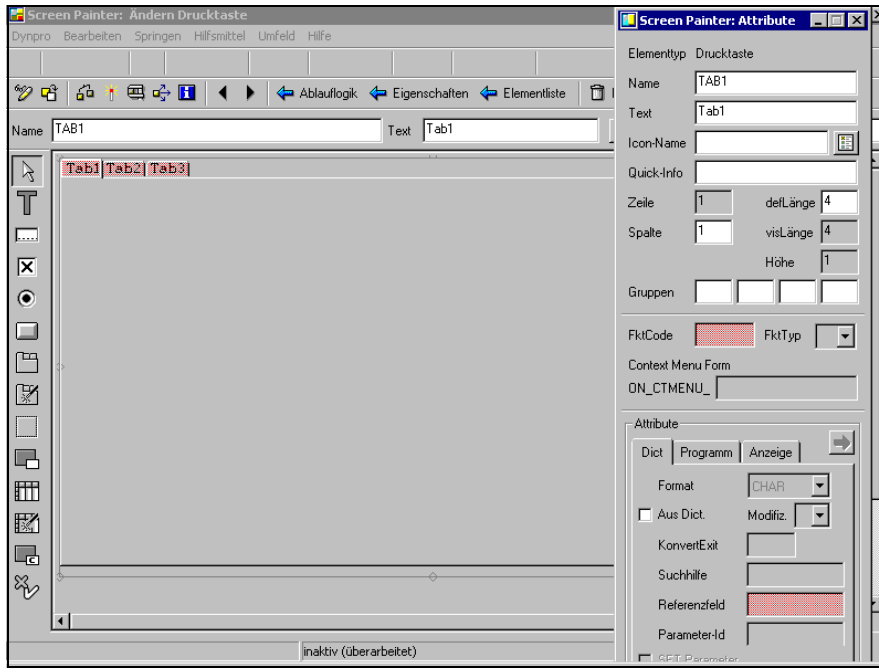

Abbildung 3.6
Träger-Dynpro mit Tab-Strip anlegen (© SAP AG)

Mit einem Doppelclick auf den Tab-Strip-Bereich können die Eigenschaften des Tab-Strips angezeigt werden. Hier muss der Name des Controls definiert werden. Im Beispiel wird der Name `FLIGHTSTAB` verwendet. Weiterhin kann hier die Größe des Tab-Strips auf dem Dynpro sowie die Anzahl der Tab-Seiten festgelegt werden. Das Tab-Strip im Beispiel soll den gesamten nutzbaren Bereich des Dynpros einnehmen (siehe Abbildung 3.6).

Anlegen der Dynpros für die Tab-Seiten

Bei den einzelnen Seiten eines Tab-Strips handelt es sich immer um Subscreens. Diese werden wie normale Subscreens im Dynpro-Editor des Object Navigators angelegt. Wie für herkömmlich verwendete Subscreens gilt auch hier die Einschränkung, dass kein `OKCODE`-Feld in der Elementliste des Dynpros definiert werden darf. Dies muss – unabhängig, ob die Subscreens als Tab-Seiten verwendet werden oder nicht - immer im Träger-Dynpro geschehen. Die Auswertung von Funktionscodes hingegen kann auch im Subscreen erfolgen.

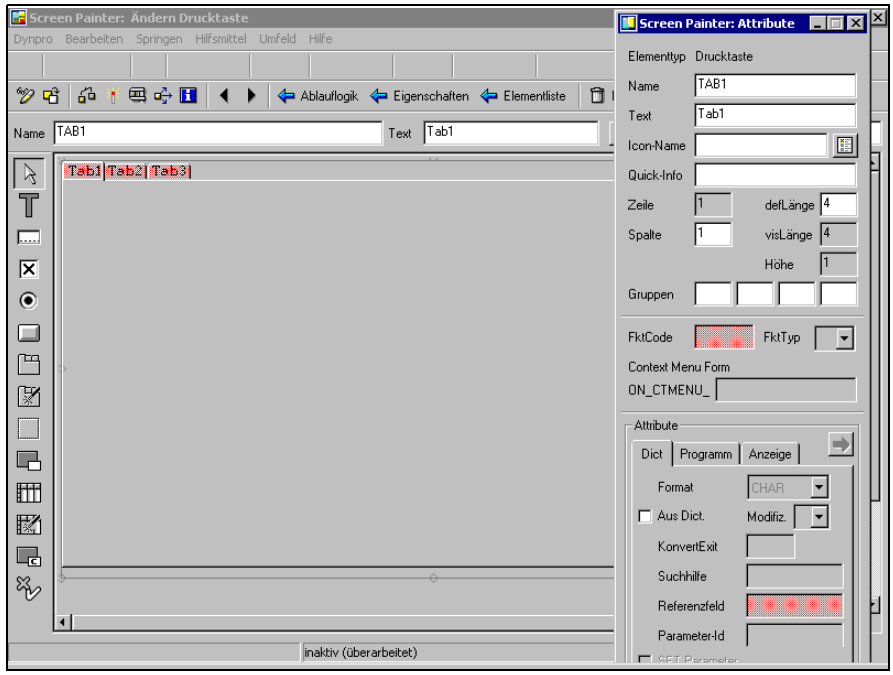

Abbildung 3.7
Anlegen der Subscreens (© SAP AG)

Im Beispiel sollen zunächst drei Subscreens mit den Dynpro-Nummern `9100`, `9200` sowie `9300` für die Aufnahme der jeweiligen Tab-Seiten definiert werden. Da leere Subscreens vom Dynpro-Prozessor nicht angezeigt werden und dem-

entsprechend Tab-Seiten, die einen leeren Subscreen referenzieren, nicht angezeigt werden, wird auf allen drei Subscreens zunächst ein statischer Text, der die jeweilige Seite benennt, dargestellt werden (siehe Abbildung 3.7).

Die Dynpros `9200` sowie `9300` im Modulpool werden analog zum Dynpro `9100` wie in der Abbildung dargestellt angelegt.

Deklaration des Tab-Strips

Außer im Dynpro-Editor muss für den Tab-Strip ein globales Datenfeld angelegt werden, der zur Laufzeit Einfluss auf die Darstellung und das Verhalten des Tab-Stips hat. Dieses Datenfeld wird mit der Anweisung

```
CONTROLS <Tab-Strip-Name> TYPE TABSTRIP.
```

angelegt. Die `CONTROLS`-Anweisung definiert, ähnlich wie die `RANGES`-Anweisung, ein Datenfeld, über das auf die Attribute des Controls zugegriffen werden kann. Im Beispiel des Tab-Strips handelt es sich um ein Datenfeld mit dem in Tabelle 3.7 dargestellten Aufbau. Diese bildet zur Laufzeit die Schnittstelle zwischen dem ABAP-Code auf der einen und dem Dynpro-Prozessor auf der anderen Seite. Im Beispiel dieses Kapitels müsste im Top-Include des Modulpools hierzu die Anweisung

```
CONTROLS FLIGHTSTAB TYPE TABSTRIP.
```

eingefügt werden.

Feldname	Datentyp	Bedeutung
PROGNR		Name des Modulpools, in dem das Dynpro, auf dem das Tab-Strip liegt, definiert ist
DYNNR		Dynpro-Nummer des Dynpros, der auf das Tab-Strip definiert ist
ACTIVETAB		Funktionscode der aktuell dargestellten Tab-Seite

Tabelle 3.7
Aufbau des Datenfeldes welches durch CONTROLS ... TYPE TABSTRIP angelegt wird.

Über diese Struktur kann zur Laufzeit Einfluss auf den Tab-Strip genommen werden. Das wichigste Feld hierbei ist das Feld `ACTIVETAB`, welches den Funktionscode des aktuellen Tab-Reiters enthält. Durch Setzten dieses Feldes wird der Dynpro-Prozessor angewiesen, bei der nächsten Darstellung des Tab-Controlls, die gewünschte Seite zuoberst darzustellen (siehe auch Abschnitt 3.2.3).

3.2.2 Tab-Strips mit Blätterfunktion im SAPGUI

Aktionen im Dynpro-Editor

Nachdem nun sowohl das Träger-Dynpro mit dem Tab-Strip als auch die einzelnen Subscreens des Controls angelegt wurden, kann nun begonnen werden, die Blätterfunktionalität zu implementieren.

Technisch gesehen ist das Blättern im SAPGUI realisiert, indem eine Reihe von Subscreens übereinander auf dem Dynpro angeordnet werden. Für jede Tab-Seite wird auf der jeweiligen Tab-Seite ein eigener Subscreen-Bereich im Dynpro-Editor angelegt und benannt. Bei diesem Verfahren handelt es sich demnach um eine Technik, um Subscreens, die bisher überlappungsfrei auf dem Dynpro angeordnet sein mussten, quasi zu stapeln und übereinander auszugeben, wobei aber immer alle Subscreenbereiche vorhanden sind.

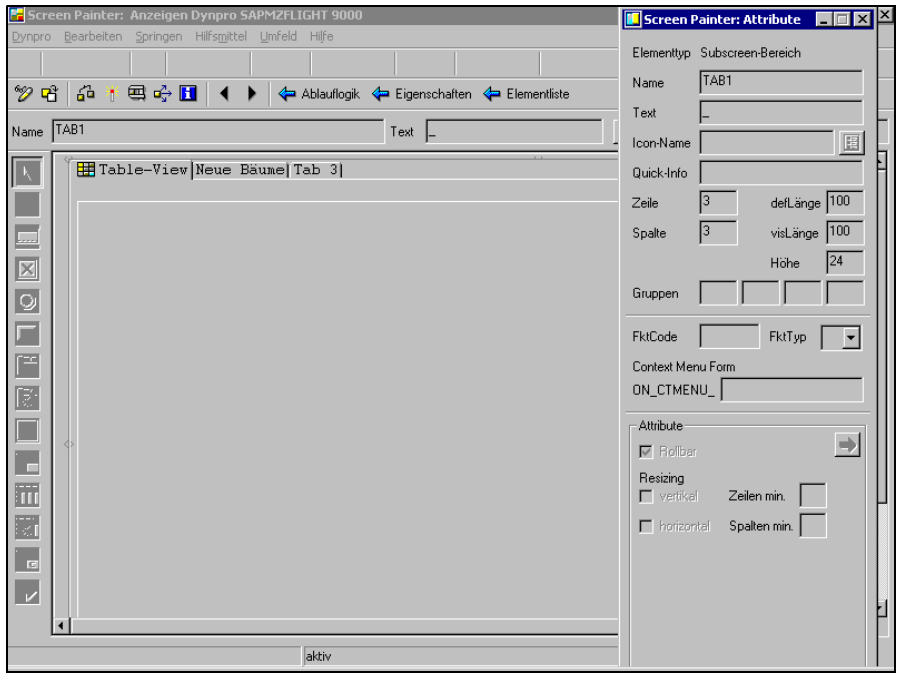

Abbildung 3.8
Anlegen eines Subscreens auf der Tab-Seite (© SAP AG)

Um dies zu bewerkstelligen werden im Dynpro-Editor auf dem Träger-Dynpro (im Beispiel Dynpro `9000`) nacheinander alle Tab-Seiten angewählt und darauf jeweils ein Subscreen angelegt (siehe Abbildung 3.8). Weiterhin muss jedem Tab-Reiter ein Funktionscode zugewiesen werden. Beim Blättern im SAPGUI

des Benutzers müssen diese Funktionscodes den Typ »P« (lokale Funktion des SAPGUI) haben. Hierdurch wird festgelegt, dass beim Betätigen des Tab-Reiters kein PAI-Ereignis ausgelöst wird, sondern die Funktion lokal vom GUI des Benutzers verarbeitet wird.

Damit sind die Aufgaben, die im Dynpro-Editor durchgeführt werden müssen, um die Blätterfunktion im SAPGUI zu realisieren, abgeschlossen. Es handelte sich um

❶ Anlegen eines Subscreen-Bereiches auf jeder Tab-Seite.

❷ Definition der Funktionscodes der Tab-Reiter mit Typ P.

Ablauflogik beim Blättern im SAPGUI

Da beim Blättern im SAPGUI auf jeder Tab-Seite ein eigener Subscreen-Bereich existiert, muss in der Ablauflogik auch jeder der definierten Subscreen-Bereiche gesondert behandelt werden. Da im Beispiel die Zuordnung der Subscreens zu den Subscreen-Bereichen statisch ist, beschränkt sich die Implementierung der Ablauflogik des Dynpros 9000 auf die notwendigen CALL SUBSCREEN-Anweisungen.

```
PROCESS BEFORE OUTPUT.
...
  CALL SUBSCREEN TAB1 INCLUDING 'SAPMZFLIGHT' '9100'.
  CALL SUBSCREEN TAB2 INCLUDING 'SAPMZFLIGHT' '9200'.
  CALL SUBSCREEN TAB3 INCLUDING 'SAPMZFLIGHT' '9300'.
...
PROCESS AFTER INPUT.
...
  CALL SUBSCREEN TAB1.
  CALL SUBSCREEN TAB2.
  CALL SUBSCREEN TAB3.
...
```

Hiermit ist das Tab-Strip vollständig realisiert. Wird jetzt noch ein Transaktionscode erzeugt, der das Träger-Dynpro als Start-Screen hat, kann über diesen das Träger-Dynpro mit den drei Tab-Seiten angezeigt werden. Das die Blätterfunktion beim Wechsel der Tab-Seiten lokal ohne Zuhilfenahme des Applikationsservers erfolgt, kann daran erkannt werden, dass im Fall des Tab-Wechsels keine Kommunikation mit dem Server erfolgt.

3.2.3 Tab-Strips mit Blätterfunktion auf dem Applikationsserver

Anders als beim Blättern im SAPGUI des Benutzers verhält es sich beim Blättern auf dem Applikationsserver. Während im ersten Fall für jede Tab-Seite ein eige-

ner Subscreen-Bereich angelegt wird, erfolgt das Blättern im zweiten Fall durch das Austauschen des Inhalts eines einzigen Subscreen-Bereiches, der über alle Tab-Seiten identisch ist. Über die Tab-Reiter wird demnach nicht die Sichtbarkeit der einzelnen Seiten gesteuert, sondern tatsächlich ein Wechsel des Inhaltes des Subscreen-Bereiches vollzogen.

Realisierung im Dynpro-Editor

Die Aufgabe im Dynpro-Editor ist es, einen einzigen Subscreen-Bereich für alle Tab-Seiten anzulegen. Hierzu werden, die ⇧-Taste gedrückt haltend, nacheinander alle Tab-Reiter angewählt und somit alle Seiten gleichzeitig selektiert (siehe Abbildung 3.9).

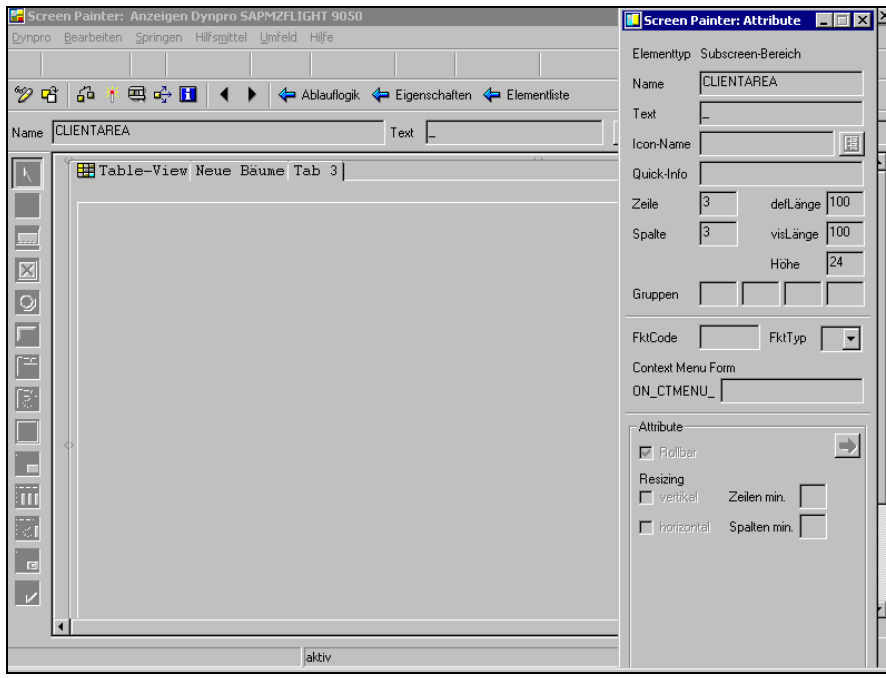

Abbildung 3.9
Selektieren aller Tab-Seiten für die Blätterfunktion auf dem Server (© SAP AG)

Im Anschluss daran wird auf dem Tab-Bereich ein einziger Subscreen angelegt, der sich somit über alle Tab-Seiten erstreckt.

Schließlich muss – wie beim Blättern im SAPGUI – jedem Tab-Reiter ein Funktionscode zugewiesen werden mit dem Unterschied, dass der Funktionstyp in diesem Fall nicht »P« sein darf, sondern in der Regel leer, d.h. eine Standard-Funktion, ist.

Implementierung des Tab-Strips

Da beim Blättern auf dem Applikationsserver nur ein Subscreen-Bereich für alle Tab-Seiten existiert, darf die Ablauflogik dementsprechend auch nur eine CALL SUBSCREEN-Anweisung enthalten und es wird auch nur das jeweils dargestellte Dynpro prozessiert. Da also dynamisch der Inhalt eines einzigen Subscreen-Bereichs ausgetauscht wird, müssen im Programm globale Variablen existieren, die das aktuell sichtbare Dynpro referenzieren. Im Top-Include des Modulpools werden hierzu die folgenden beiden Variablen deklariert:

```
DATA: G_PROGRAM LIKE SY-REPID  VALUE 'SAPMZFLIGHT',
      G_DYNNR   LIKE SY-DYNNR  VALUE '9100'.
```

Die Variablen werden jeweils mit einem Initialwert versehen, damit auch beim ersten Anzeigen des Dynpros ein korrekter Subscreen referenziert wird.

In der Ablauflogik des Träger-Dynpros werden diese beiden Variablen zur Definition des aktuellen Inhalts des Subscreens verwendet.

```
PROCESS BEFORE OUTPUT.
...
  CALL SUBSCREEN TABSEITE INCLUDING G_PROGRAM G_DYNNR.
...
PROCESS AFTER INPUT.
...
  CALL SUBSCREEN TABSEITE.

  MODULE COMMAND_HANDLER.
...
```

Da das Blättern in diesem Fall auf dem Applikations-Server erfolgen soll, muss außerdem in der Ablauflogik ein `MODULE` definiert werden, in dem die Funktionscodes ausgewertet werden. Im dargestellten Quelltext erfolgt dies im `MODULE COMMAND_HANDLER`.

Die Implementierung dieses Moduls muss – was die Funktionscodes des Tab-Strips betrifft – die globalen Variablen korrekt setzen. Weiterhin muss in der Datenstruktur des Tab-Strips die aktive Tab-Seite gesetzt werden, um die Tab-Reiterleiste korrekt anzuzeigen. Im Gegensatz zum Blättern im SAPGUI des Benutzers, wird beim Blättern auf dem Applikations-Server hierdurch der Tab-Reiter definiert, der als aktiv dargestellt wird. Beim Blättern im SAPGUI entfällt diese Kontrollmöglichkeit, da beim Blättern kein ABAP-Code ausgeführt wird. Hier kann lediglich der bei der Anzeige zunächst aktive Tab-Reiter definiert und der zuletzt aktive Tab-Reiter ausgelesen werden.

```
MODULE COMMAND_HANDLER
  CASE OKCODE.
    WHEN 'TAB1'.
      G_PROGRAM = SY-CPROG.
```

```
      G_DYNNR   = 9100.
      FLIGHTSTAB-ACTIVETAB = 'TAB1'.

    WHEN 'TAB2'.
      G_PROGRAM = SY-CPROG.
      G_DYNNR   = 9200.
      FLIGHTSTAB-ACTIVETAB = 'TAB2'.

    WHEN 'TAB3'.
      G_PROGRAM = SY-CPROG.
      G_DYNNR   = 9300.
      FLIGHTSTAB-ACTIVETAB = 'TAB3'.
  ENDCASE.

  CLEAR OKCODE.
ENDMODULE.
```

Wird das Setzen des Feldes `FLIGHTSTAB-ACTIVETAB` im Beispiel nicht durchgeführt, zeigt der Subscreen-Bereich zwar jeweils den richtigen Subscreen an, die Tab-Reiter-Leiste spiegelt jedoch nicht die aktuell dargestellte Seite wieder.

Welche der beiden Methoden zum Wechseln der Tab-Seiten verwendet wird, muss im Einzelfall entschieden werden. Während die Vorteile beim Blättern im SAPGUI darin liegen, dass beim Seitenwechsel keine Kommunikation mit dem Server erforderlich ist, liegen die Vorteile beim Blättern auf dem Server eindeutig darin, dass auch zwischen den Seitenwechseln Plausibilisierungen und Ableitungen erfolgen können.

3.3 Nochmal Bäume: Die »neue« Baumdarstellung

Bereits in Abschnitt 3.1 wurde eine Möglichkeit zur Darstellung von Informationen in Form von Bäumen gezeigt. Die dort beschriebene Methode kann nur in Listen verwendet werden und basiert technologisch auf einer standardisierten Variante des interaktiven Reportings. Dadurch können diese Bäume nur bedingt in Dialogtransaktionen, deren Dynpros im Dynpro-Editor erstellt werden, verwendet werden. Hierfür existiert in R/3 seit Einführung von ABAP OBJECTS eine neue Methode, die auf benutzerdefinierten Bereichen in Dynpros auf der einen Seite und auf Verwendung einer Reihe von ABAP-Klassen, die unter dem Begriff »Controls Technology« zusammengefasst werden, basiert.

Dem Anwender ist diese Art der Baumdarstellung aus dem Easy-Access-Menu von R/3 geläufig. In diesem Abschnitt wird gezeigt, wie derartige Baumdarstellungen programmiert werden können. Zur Demonstation wird erneut das bereits in Abschnitt 3.1 verwendete Beispiel der Flugbewegungen aufgegriffen. Das Beispielprogramm verwendet den in Abschnitt 3.2 erstellten Programmrah-

men. Der hier erstellte Baum wird auf einem der Subscreens `9200` des Tab-Strips angelegt. Abbildung 3.5 zeigt das in diesem Abschnitt erstellte Beispiel.

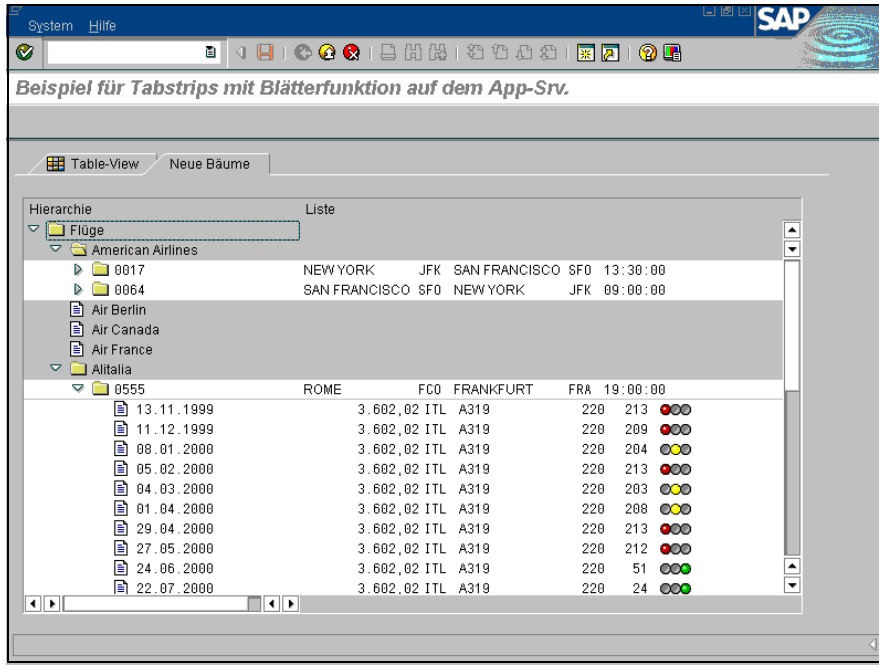

Abbildung 3.10
Baumdarstellung auf einer Seite eines Tab-Strips (© SAP AG)

3.3.1 Allgemeine Architektur

Mit Einführung des R/3 Release 4.6, dem so genannten Enjoy-Release, unterzog SAP nahezu das gesamte System einer grundlegenden Überarbeitung der Benutzerschnittstelle. Einfache Bedienbarkeit, leichte Erlernbarkeit und Personalisierbarkeit standen dabei im Vordergrund. Erreicht wurde dies zum einen durch eine inhaltliche Überarbeitung der Benutzerführung in vielen Transaktionen. Hierbei wurden Informationen umstrukturiert, teilweise wurden Abläufe, die bisher mehrere Bildschirmmasken beanspruchten, zusammengefasst. Auf der anderen Seite wurde das gesamte grafische Frontend neu gestaltet. Die neue Darstellung, die übrigens vollständig im GUI abgewickelt wird und somit auch mit älteren R/3-Releaseständen funktioniert, basiert auf einer streng nach ergonomischen Gesichtspunkten entwickelten Philosophie.

Gleichzeitig verfolgt SAP auch auf der Seite der SAPGUI-Anwendung neue Wege. Lediglich für die Plattform Microsoft Windows existiert eine eigene Implementierung des SAPGUIs. Die übrigen Frontend-Plattformen werden künftig mit dem »SAPGUI for JAVA« bedient. Weiterhin wurden die Transaktionen be-

züglich der Verwendung aus einem Web-Browser heraus optimiert. Mit Release 4.6C sollen ca. 98% der SAP-Transaktionen über Web-Browser und den Internet-Transaction-Server aufrufbar sein. Optisch präsentieren sich die Masken dabei identisch mit denen im SAPGUI.

Aber auch inhaltlich beschritt SAP in diesem Zusammenhang neue Wege. Neu eingeführt wurden Controls, die auf etablierten Komponenten-Modellen im Frontend-Bereich basieren. Zum Zuge kommen hier die ActiveX-Technologie von Microsoft auf den Windows-Plattformen sowie Suns JavaBeans-Technologie auf den übrigen Frontend-Plattformen. Diese werden in die herkömmliche ABAP-Programmierumgebung durch so genannte Custom Controls, die im Dynpro-Editor als Bereiche auf der Maske angelegt werden, integriert. Mit diesen Bereichen werden im ABAP-Code Instanzen des »Control Frameworks« verknüpft.

Die Programmierung erfolgt herbei mehrstufig. Im ersten Schritt wird der Custom Control-Bereich auf dem Dynpro durch eine Instanz einer so genannten Container-Klasse verwaltet. Wie der Name bereits ausdrückt, wird der Bereich dadurch zu einem Behälter, in dem ein oder mehrere Elemente des Control Frameworks eingefügt werden können. Der Container ist dabei für die Anordnung der in ihm enthaltenen Controls verantwortlich. Programmtechnisch handelt ses sich hierbei um eine Unterklasse der ABAP-Klasse `CL_GUI_CONTAINER`. In diesen Container können – je nach Containertyp – wiederum ein oder mehrere Controls, bei denen es sich auch um Container handeln kann, eingefügt werden. In der Regel wird es sich hierbei jedoch um Instanzen der grafischen Controls handeln.

Mit dem Stand von Release 4.6 existieren derartige Komponenten für:

- SAP Trees
- SAP TextEdit
- SAP Picture
- SAP HTML Viewer
- SAP Toolbar sowie das
- AVL Grid Control.

Weiterhin gehören die Klassen des »graphical Frameworks«, welches die Funktionalitäten der bisherigen SAP Grafik abdeckt, in diesen Bereich, wenngleich die Implementierung solcher Controls erheblich aufwendiger ist, als die in der Liste genannten.

Da es sich bei den neuen Controls allesamt um Code handelt, der auf dem Frontend abläuft, müssen die ABAP-Klassen, mit denen im Programm gearbeitet wird, als programmtechnische Repräsentation des Controls (Proxies) verstanden werden. Das Control Framework von R/3 macht hierbei die zugrunde liegende Technologie für den Programmierer weitgehend transparent.

Die Programmierung mit dem Control Frameset von SAP basiert daher immer auf den Schritten:

❶ Anlegen eines Custom Control-Bereiches auf dem Dynpro.

❷ Erzeugen eines Containers zur Verwaltung des Custom Controls.

❸ Füllen des Containers mit grafischen Controls (z.B. Tree).

In diesem Kapitel sollen die grundlegenden Techniken der Programmierung mit dem Control Framework anhand des SAP Trees dargestellt werden. Im Kapitel 4 dieses Buches wird auch vom SAP TextEdit Gebrauch gemacht. Kapitel 5 schließlich verwendet das mit Release 4.6 ebenfalls neue »graphical Framework«, welches ebenfalls die hier vorgestellten Containern verwendet. Allen Controls gemeinsam sind die Schritte 1 und 2. Erst im dritten Schritt wird festgelegt, welches Control im Container verwendet wird.

Die Möglichkeiten des zugrunde liegenden Control Frameworks von SAP sind wesentlich umfangreicher als das, was im Rahmen dieses Buches dargestellt werden kann. Die Beschreibungen beschränken sich daher auf das für die verwendeten Beispiele Nötige. Eine vollständige und detaillierte Darstellung der Möglichkeiten des Control Frameworks im Buch »Control Technology«, welches bei SAP erschienen ist, nachgelesen werden. Dort werden auch die Controls beschrieben, auf die in diesem Buch nicht näher eingegangen werden kann.

3.3.2 Anlegen des Custom Control-Bereiches

Egal, welches Control auf dem Bildschirm dargestellt werden soll, muss zunächst im Dynpro-Editor ein Bereich dafür geschaffen werden. Für das Beispiel dieses Abschnitts wird daher auf dem Dynpro `9200` des Modulpools `SAPMZFLIGHT` ein Custom Control angelegt.

Im grafischen Screen-Editor muss hierzu das Symbol für das Custom Control angewählt werden und im Dynpro-Bereich ein beliebig großer Bereich »aufgezogen« werden (siehe Abbildung 3.11). Die Größe des Bereiches sowie dessen Position auf dem Dynpro kann im grafischen Editor auch nachträglich leicht verändert werden.

Zum Schluss muss der erzeugte Bereich mit einem Namen versehen werden, über den er im ABAP-Code referenziert werden kann. Im Beispiel soll hierfür der Name »`TREEAREA`« verwendet werden.

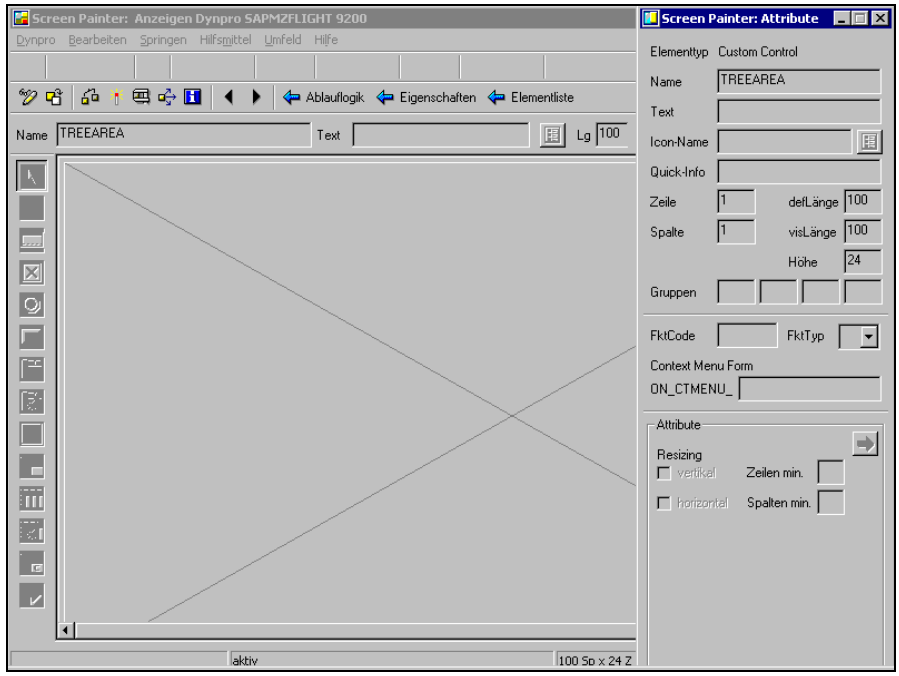

Abbildung 3.11
Grafischer Dynpro-Editor mit Custom-Control-Bereich (© SAP AG)

3.3.3 Container

Im ABAP-Programm wird der im Dynpro-Editor angelegte Bereich auf der Maske von einem so genannten Container verwaltet. Container sind spezielle Klassen im R/3-System, deren Aufgabe ausschließlich darin liegt, derartige Bildschirmbereiche zu verwalten. Sämtliche Controls, die im System verwendet werden können, müssen über Container verwaltet werden. Da Container selbst auch Controls sind, ist es auch möglich, Container in Container einzufügen und somit den Custom-Bereich abermals logisch zu unterteilen.

R/3 unterscheidet hierbei fünf verschiedene Typen von Containern, die jeweils durch eigene ABAP-Klassen repräsentiert werden. Die Klasse `CL_GUI_CUSTOM_CONTAINER`, welche im Folgenden Beispiel verwendet wird, verwaltet einen Custom-Control-Bereich auf einem Dynpro. Beim Container-Typ `CL_GUI_DIALOGBOX_CONTAINER` hingegen entfällt diese Beziehung, da der Container immer als eigenes amodales Dialogfenster angezeigt wird. Ebenfalls ohne Bezug zu einem Custom-Control-Bereich auf einem Dynpro existiert die Klasse `CL_GUI_DOCKING_CONTAINER`, welche entweder an einem der vier Ränder des Dynpro-Fensters oder als frei schwebende Dialogbox über dem Dynpro existieren kann. Dieser Container hat immer Bezug zu einem R/3-Fenster, ohne jedoch Be-

zug zu einem Custom-Control-Bereich auf einem Dynpro zu haben. Alle diese drei Container haben gemeinsam, dass sie lediglich ein Control aufnehmen können. Anders verhält es sich hierbei bei den Containerklasse CL_GUI_SPLITTER_CONTAINER sowie CL_GUI_EASY_SPLITTER_CONTAINER als vereinfachte Variante. Diese beiden Container können jeweils mehr als ein Control beinhalten (ersterer beliebig viele, letzterer genau zwei). Alle diese Container haben eine gemeinsame Oberklasse CL_GUI_CONTAINER und implementieren deren Interface.

Für das Beispiel dieses Kapitels kommt – als einfachste Variante – eine Instanz der Klasse CL_GUI_CUSTOM_CONTAINER zum Einsatz.

Egal, welcher Container-Typ zum Einsatz kommt, muss im ABAP-Programm eine Variable definiert werden, welche die Instanz des Containers referenziert. Dies erfolgt im TOP-Include mit der Anweisung:

```
DATA G_CONTAINER TYPE REF TO CL_GUI_CUSTOM_CONTAINER.
```

Im Anschluss daran muss an geeigneter Stelle im Programm die Instanz erzeugt werden. Hierzu wird die CREATE OBJECT-Anweisung der ABAP-Sprache verwendet. Die Suche nach der geeigneten Stelle ist schnell gefunden. Die Container-Instanz muss vor dem ersten Anzeigen des Dynpros erzeugt werden. Demnach kommt bei Dialogtransaktionen nur das PBO des jeweiligen Dynpros in Frage. Da jedoch nicht vor jedem Anzeigen des Dynpros eine neue Instanz erstellt werden soll, wird dies in einen IF-Block eingeschlossen, der nur einmal durchlaufen wird. Im Beispielprogramm wurde dazu das im Folgenden dargestellte Modul CREATE_CONTAINER definiert.

```
MODULE CREATE_CONTAINER.
  IF G_CONTAINER IS INITIAL.
    CREATE OBJECT G_CONTAINER
       EXPORTING
          CONTAINER_NAME = 'TREEAREA'
       EXCEPTIONS
          CNTL_ERROR = 1
          CNTL_SYSTEM_ERROR = 2
          CREATE_ERROR = 3
          LIFETIME_ERROR = 4
          LIFETIME_DYNPRO_DYNPRO_LINK = 5.
  ENDIF.
ENDMODULE.
```

Beim Erzeugen der Instanz wird diese auch gleichzeitig mit dem Custom Control »TREEAREA« verknüpft. Auf ein Abfragen und Auswerten des Fehlercodes wurde in diesem Beispiel zugunsten der Lesbarkeit des Programms verzichtet.

Der Konstruktor dieser Klasse bietet noch eine Reihe anderer Parameter an, über die das Verhalten beeinflusst werden kann. Diese werden im Beispiel nicht benötigt und sollen hier der Vollständigkeit halber nur kurz aufgelistet werden.

Über EXPORTING-Parameter PARENT kann ein übergeordneter Container, in den die neue Instanz eingefügt werden soll, übergeben werden. Es handelt sich hierbei in der Regel immer um eine Instanz der Klassen CL_GUI_SPLITTER_CONTAINER oder CL_GUI_EASY_SPLITTER_CONTAINER, da nur diese beiden Container-Typen in der Lage sind mehr als ein Control aufzunehmen. Mit den Parametern DYNNR sowie REPID kann das Control fest mit einem Dynpro verknüpft werden. Werden diese beiden Parameter versorgt, muss im Parameter NO_AUTODEF_PROGID_DYNNR der Wert »X« (Defaultwert ist Space) übergeben werden, um diese Zuordnung nicht automatisch durchzuführen. Über den Parameter STYLE kann unter Verwendung der in der Klasse CL_GUI_CONTAINER definierten Konstanten die mit dem Präfix WS_ beginnen, die visuelle Repräsentation des Containers beeinflusst werden.

Zuletzt sei noch der Parameter LIFETIME erwähnt, der im Control Framework eine wichtige Rolle einnimmt. Über ihn wird die Lebensdauer der Control-Instanz festgelegt. Die Klasse CL_GUI_CONTAINER definiert hierfür zwei Konstanten. Der Wert der Konstante CNTL_LIFETIME_IMODE legt fest, dass die Instanz so lange erhalten bleibt, wie der interne Modus, d.h. das Programm. Die Anweisungen LEAVE PROGRAM bzw. LEAVE TO TRANSACTION beenden das aktuelle Programm und zerstören im dem Fall auch die Instanzen des Controls. Demgegenüber steht der Wert der Konstante CNTL_LIFETIME_DYNPRO. Wie der Name bereits vermuten lässt, beschränkt sich hierbei die Lebensdauer der Control-Instanz auf die Lebensdauer des verknüpften Dynpros. Defaultwert dieses Parameters ist CNTL_LIFETIME_IMODE. Dieser sollte nur in begründeten Einzelfällen verändert werden.

Somit steht dem Programm nun eine initialisierte Instanz des Containers zur Verfügung. Diese ist bereits mit dem Custom Control auf dem Dynpro verknüpft. Was noch fehlt ist das grafische Control, welches eigentlich angezeigt werden sollte. Die übrigen Container-Klassen haben entsprechend ihrer Aufgaben unterschiedliche Konstruktor-Schnittstellen.

3.3.4 SAP Tree, die neue Art der Baumdarstellung

Die neue Art der Baumdarstellung mit dem Control Framework bietet dem Programmierer verschiedene Arten von Bäumen für verschiedene Verwendungszwecke. Wie bei den Containern auch, sind diese in unterschiedlichen Klassen realisiert. SAP stellt drei Arten von nutzbaren Bäumen zur Verfügung, deren Klassen in der in Abbildung 3.13 dargestellten Vererbungshierarchie angeordnet sind. Die Klassen mit dickerem Rand sind dabei die programmtechnisch nutzbaren Klassen. Die übrigen Klassen dienen lediglich der Strukturierung und Klassifizierung der Aufgabengebiete der einzelnen Klassen. Instanzen dieser Klassen machen daher für die Programmierung keinen Sinn.

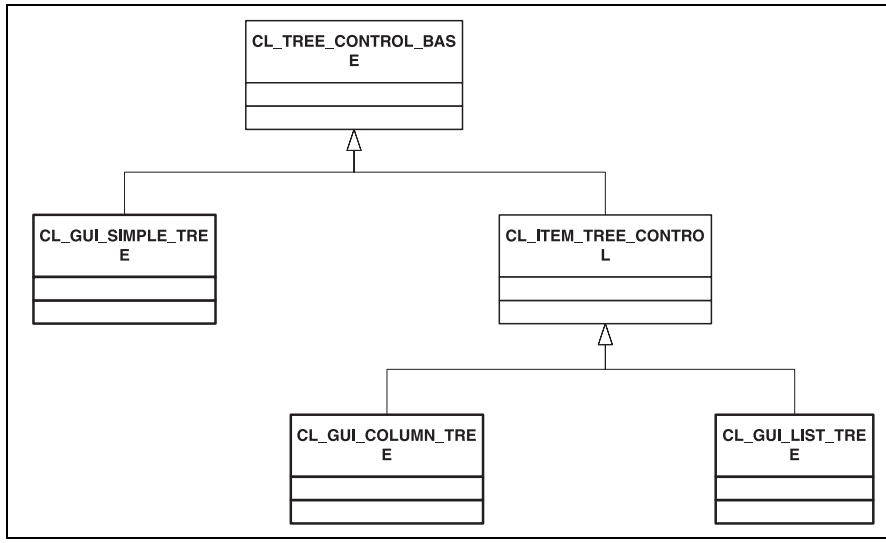

Abbildung 3.12
Vererbungshierarchie der SAP Tree-Klassen

Als Beispiel soll hier der gleiche Baum aufgebaut werden, wie in Abschnitt 3.1. Da diese Art der Baumdarstellung ein Dialogprogramm voraussetzt, wird das Beispielprogramm aus dem vorigen Abschnitt verwendet. Die Baumdarstellung wird hierbei auf der zweiten Seite des Tab-Strip realisiert.

Wie im Abschnitt 3.1 wird der Baum auch in diesem Abschnitt in mehreren Stufen mit wachsender Komplexität aufgebaut. Zunächst wird hierbei eine Instanz der Klasse `CL_GUI_SIMPLE_TREE` verwendet. Bäume, die diesen Typ verwenden, sind dadurch charakterisiert, dass ihre Knoten nur aus einem Icon und einem Textfeld bestehen können. Sollen mehr Informationen zu einem Knoten angezeigt werden, muss ein anderer Baumtyp verwendet werden. Mit dem Hinzufügen der zweiten Darstellungsebene wird daher im Beispiel auf eine Instanz der Klasse `CL_GUI_COLUMN_TREE` gewechselt. Die Klasse `CL_GUI_COLUMN_TREE` ist – genauso wie die Klasse `CL_GUI_LIST_TREE` – von der Klasse `CL_ITEM_TREE_CONTROL` abgeleitet. Sie sind dadurch charakterisiert, dass zu jedem Knoten ein Hierarchieteil, der die Position des Knotens innerhalb des Baums beschreibt, weitere Informationen ausgegeben werden können, um die Knoten ausführlicher darzustellen. Während diese bei Instanzen von `CL_GUI_LIST_TREE` einfach hintereinander ausgegeben werden, sind sie bei Objekten vom Typ `CL_GUI_COLUMN_TREE` als benannte Spalten angelegt. Über diesen Namen kann auf die Spalteninhalte eines Knotens zugegriffen werden. Alle Spalten haben hierbei eine definierte Breite, werden also – unabhängig von der Länge ihres Inhaltes – über alle Knoten hinweg gleich lang ausgegeben. Bei Instanzen der Klasse `CL_GUI_LIST_TREE` werden die Teile eines Knotens einfach hintereinanderweg ausgegeben.

Da bei Bäumen vom Typ CL_GUI_COLUMN_TREE jedoch alle Knoten des Baumes die gleiche Spaltenstruktur verwenden, die zweite und dritte Ebene des Beispiels jedoch verschiedene Datenstrukturen anzeigen soll, wird zur Demonstration der Möglichkeiten der Klasse CL_GUI_LIST_TREE mit der dritten Darstellungsebene auf diesen Baumtyp gewechselt.

Anlegen der Referenzvariable des Baums

SAP Trees beruhen, wie Container auch, auf einer Reihe von ABAP-Klassen. Soll ein SAP Tree verwendet werden, muss daher im ABAP-Programm ebenfalls eine Referenz auf den Baum, über den zur Laufzeit auf den Baum zugegriffen werden kann, angelegt werden. Dies geschieht normalerweise ebenfalls im globalen Datenraum des Programms. Im ersten Schritt soll in diesem Kapitel eine Instanz der Klasse CL_GUI_SIMPLE_TREE verwendet werden. Im TOP-Include des Modulpools wird daher die folgende Deklaration eingefügt.

```
DATA G_TREE TYPE REF TO CL_GUI_SIMPLE_TREE.
```

Diese Variable muss ebenfalls vor der Anzeige des Dynpros, auf dem der Baum erscheinen soll, initialisiert werden.

Erzeugung eines einfachen Baumes

Der erste Schritt bei der Erzeugung einer Baumdarstellung ist die Erzeugung der jeweiligen Tree-Klasse. Hier soll die im vorigen Abschnitt definierte Referenz auf die Klasse CL_GUI_SIMPLE_TREE verwendet werden. Zu dieser Referenz wird nun – vor der ersten Anzeige des Dynpros die entsprechende Instanz der Klasse erzeugt.

Hierzu wird in der Ablaufsteuerung des Dynpros 9200 nach dem Aufruf des Moduls zum Erzeugen des Containers das Modul CREATE_TREE mit unten stehender Implementierung eingefügt.

```
MODULE CREATE_TREE.
* Der Tree soll nur erzeugt werden, wenn das Anlegen des
* Containers erfolgreich war.
  CHECK NOT G_CONTAINER IS INITIAL.

* Anlegen des Baumes nur, wenn noch nicht geschehen.
  CHECK G_TREE IS INITIAL.

  CREATE OBJECT G_TREE
     EXPORTING
        PARENT              = G_CONTAINER
        NODE_SELECTION_MODE = G_TREE->NODE_SEL_MODE_SINGLE
     EXCEPTIONS
        LIFETIME_ERROR              = 1
```

```
        CNTL_SYSTEM_ERROR           = 2
        CREATE_ERROR                = 3
        FAILED                      = 4
        ILLEGAL_NODE_SELECTION_MODE = 5.
ENDMODULE.
```

Die Instanz des Baumes ist somit erzeugt, der Baum enthält allerdings zu diesem Zeitpunkt noch keine Knoten.

Hierzu wird am Ende des Moduls `CREATE_TREE` die neue `FORM`-Routine `CREATE_SIMPLE_TREE_NODES` aufgerufen und die Referenz auf die Instanz des Baumes übergeben.

```
...
PERFORM CREATE_SIMPLE_TREE_NODES USING G_TREE.
...
```

Die Knoten müssen zunächst in Form einer internen Tabelle im Programm erzeugt und anschließend mit einem Aufruf der Methode `ADD_NODES` in den Baum eingefügt werden. Um die Kommunikation des Tree-Controls mit seiner Stellvertretetklasse im ABAP gering zu halten, empfielt SAP nach Möglichkeit, alle Knoten des Baumes mit einem Aufruf von `ADD_NODES` einzufügen. Nur wenn die Anzahl der Knoten größer als 500 im lokalen Netz bzw. 100 bei Verwendung von Weitverkehrs-Netzen (WAN) ist, legt SAP ein mehrfaches Aufrufen der Methode mit einer entsprechend geringeren Anzahl Knoten nahe. Diese Betrachtungen werden in diesem Beispiel vernachlässigt, da die Anzahl der Knoten diese Grenzen nicht erreicht.

Die Knoten des Baumes werden im Programm in einer internen Tabelle aufgebaut und an die Methode `ADD_NODES` als Block übergeben. Die Struktur der Knotentabelle ist hierbei nicht namentlich festgelegt. Es muss sich jedoch um eine im Data Dictionary definierte Struktur handeln. Auch beim Aufbau dieser Struktur gibt SAP wenig Spielraum. Zunächst muss sie die vordefinierte Struktur `TREEV_NODE` enthalten. Im Anschluss an deren Felder muss ein Textfeld mit dem Namen `TEXT` definiert werden. Erst die darauf folgenden Spalten können frei definiert werden, haben aber auf die Darstellung des Knotens keinen Einfluss.

Für das Beispiel soll daher die Struktur `ZFLIGHTSIMPLE` mit folgendem Aufbau angelegt werden:

Feldname	Datentyp	Bedeutung
.INCLUDE TREEV_NODE		Knotenattribute für Simple-Trees
NODE_KEY	CHAR 12	Knotenname (für den gesamten Baum eindeutig)
RELATKEY	CHAR 12	Knotenname des Bezugsknotens

Feldname	Daten-typ	Bedeutung
RELATSHIP	INT4	Beziehung zwischen diesem und Bezugsknoten
HIDDEN	CHAR 1	Kennzeichen, ob Knoten nicht angezeigt werden soll
DISABLED	CHAR 1	Kennzeichen, ob Knoten inaktiv ist
ISFOLDER	CHAR 1	Kennzeichen, ob Knoten eine Unterstruktur besitzt
N_IMAGE	CHAR 6	Icon für Knoten, wenn Teilbaum nicht angezeigt wird
EXP_IMAGE	CHAR 6	Icon für Knoten, wenn Teilbaum ausgeklappt ist
STYLE	INT 4	Anzeigestil des Knotens
LAST_HITEM	CHAR 12	Name der letzten Spalte
NO_BRANCH	CHAR 1	Kennzeichen, ob eine Verbinderlinie zum Konten gezeichnet werden soll
EXPANDER	CHAR 1	Kennzeichen, ob Knoten ausklappbar dargestellt werden soll
DRAGDROPID	INT 2	ID des Knotens für Drag & Drop
TEXT	CHAR	Knotentext

Tabelle 3.8
Aufbau der Struktur ZFLIGHTSIMPLE

Die Aufgabe der Routine CREATE_SIMPLE_TREE_NODES besteht also darin, eine interne Tabelle mit oben dargestellter Struktur zu erzeugen und durch einen Aufruf der Methode ADD_NODES in die Instanz des Baumes einzufügen.

```
FORM CREATE_SIMPLE_TREE_NODES
                  USING P_TREE TYPE REF TO CL_GUI_SIMPLE_TREE.
  DATA: L_NODES TABLE OF TYPE ZFLIGHTSIMPLE.

* Erzeugen des Wurzelknotens
  CLEAR L_NODES.
  L_NODES-NODE_KEY = 'ROOT'.       " eindeutige Knoten-ID
  L_NODES-ISFOLDER = 'X'.          " Knoten ist kein Blatt
  L_NODES-EXPANDER = 'X'.          " Knoten ist ausklappbar
  L_NODES-TEXT     = 'Flüge'(001). " Knotenbeschriftung
  APPEND L_NODES.
```

```
* Hinzufügen der Airline-Knoten aus Tabelle SCARR.
  CLEAR L_NODES.
  L_NODES-RELATKEY  = 'ROOT'.
  L_NODES_RELATSHIP = P_TREE->RELAT_LAST_CHILD.

  SELECT CARRID
         CARRNAM
    INTO L_NODES-NODE_KEY
         L_NODES-TEXT
    FROM SCARR.
    APPEND L_NODES.
  ENDSELECT.

* Einfügen der Knoten in den Baum
  CALL METHOD P_TREE->ADD_NODES
       EXPORTING
          TABLE_STRUCTURE_NAME = 'ZFLIGHTSIMPLE'
          NODE_TABLE           = L_NODES
       EXCEPTIONS
          ERROR_IN_NODE_TABLE           = 1
          FAILED                        = 2
          DP_ERROR                      = 3
          TABLE_STRUCTURE_NAME_NOT_FOUND = 4.
ENDFORM.
```

Zunächst wird der Wurzelknoten des Baums definiert. Diese besteht – analog zum Beispiel in Abschnitt 3.1 lediglich aus dem Text »Flüge«. Anschließend werden aus den Einträgen der Tabelle SCARR die Knoten für die Fluggesellschaften erzeugt. Alle diese Knoten werden mit Bezug zum Wurzelknoten als dessen jeweils letztes Kind angelegt. Eine Umsortierung wie im Abschnitt 3.1 erübrigt sich somit. Als Knoten-ID wird der Primärschlüssel der Tabelle SCARR, die Airline-ID (CARRID) verwendet. Die Anzeige des Knotens hingegen verwendet die Bezeichnung der Fluggesellschaft im Klartext.

Zuletzt werden die erzeugten Knoten mit einem Aufruf von ADD_NODES in die Instanz des Baumes eingefügt.Abbildung 3.15 zeigt den bisher erzeugten Baum.

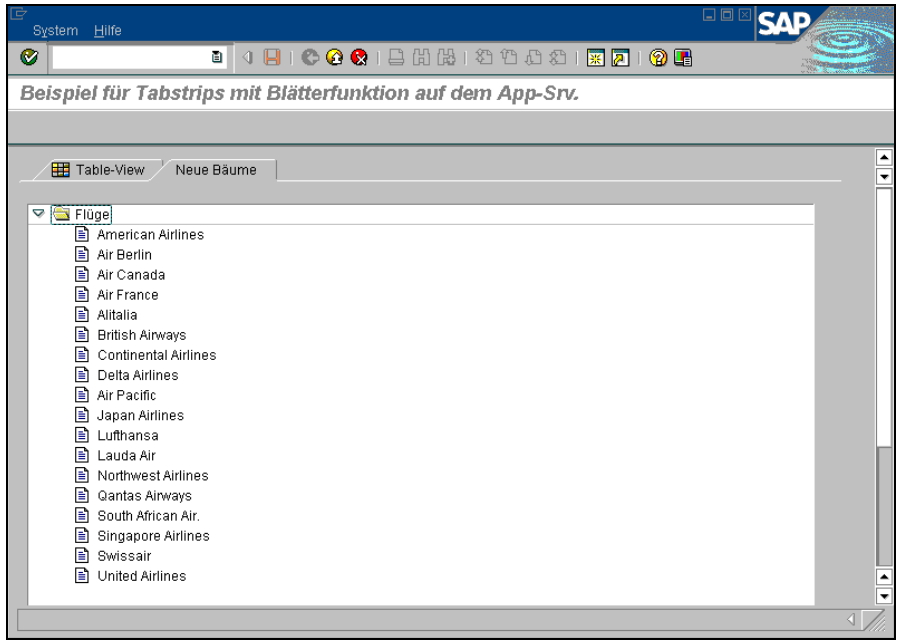

Abbildung 3.13
Baum vom Typ `CL_GUI_SIMPLE_TREE` mit Fluggesellschaften (© SAP AG)

Bäume mit spaltenorienterter Ausgabe

Eine zweite Variante der Baumdarstellung ist der Column-Tree. Hier wird die Darstellung des Baumes in zwei Bereiche unterteilt. Der erste Bereich besteht immer aus einer Spalte, in der die Hierarchie des Baumes abgebildet wird. Im zweiten Bereich können beliebig viele Spalten angelegt werden, in denen Zusatzinformationen zu den einzelnen Knoten angezeigt werden können. Die Spalten werden hierbei für den gesamten Baum angelegt, d.h. sie sind über alle Ebenen des Baumes identisch. Aus diesem Grund eignet sich diese Darstellungsvariante nur für Bäume, in denen entweder auf allen Hierarchie-Ebenen gleichartige Knoten dargestellt werden oder – wie im Beispiel – Bäume, die lediglich auf einer Ebene zusätzliche Informationen enthalten.

Jeder Knoten des Baumes kann in jeder Spalte Informationen anzeigen. Die Breite der Spalten ist unabhängig von der Länge des Inhalts des jeweiligen Feldes immer gleich. Der Benutzer kann die Breite jeder Spalte wie in einer Table-View beliebig verändern. Die Informationen einer Spalte können entweder als normaler Text, oder aber auch in Form einer Checkbox, eine Pushbuttons oder eines Links (wie Text, löst jedoch beim Anklicken mit der Maus einen Event aus) dargestellt werden. Weiterhin kann ín jeder Spalte ein eigenes Icon ausgegeben werden.

In der zweiten Ebene des Beispiels sollen die jeweiligen Flugverbindungen der Fluggesellschaften angzeigt werden. Analog zum Beispiel in Abschnitt 3.1 werden diese aus der Tabelle SPFLI, die in jedem R/3 Basissystem vorhanden ist, ausgelesen und in Form von Spalten innerhalb der Baumdarstellung eingefügt. Dabei bildet in der zweiten Ebene die Flugnummer den Hierarchieteil. Die übrigen Informationen werden im zweiten Bereich des Baumes in Spalten dargestellt. Hierzu wird das Beispiel des vorigen Abschnittes dahingehend verändert, dass der Baum statt mit einer Instanz der Klasse CL_GUI_SIMPLE_TREE durch ein Instanz der Klasse CL_GUI_COLUMN_TREE repräsentiert wird.

Zunächst muss hierzu der Typ der globalen Variable G_TREE geändert werden. Statt die Klasse CL_GUI_SIMPLE_TREE muss die Variable jetzt die Klasse CL_GUI_COLUMN_TREE referenzieren.

```
DATA G_TREE TYPE REF TO CL_GUI_COLUMN_TREE.
```

Weiterhin muss das Module CREATE_TREE, in dem die Instanz dieser Klasse erzeugt wird, angepasst werden, um den Konstruktor dieser Klasse zu bedienen. Da hier eine Struktur für den Aufbau der Überschrift des Hierarchieteils übergeben werden muss, wird der eigentliche Konstruktoraufruf in die FORM-Routine CREATE_TREE_INSTANCE ausgelagert.

```
MODULE CREATE_TREE.
* Der Tree soll nur erzeugt werden, wenn das Anlegen des
* Containers erfolgreich war.
  CHECK NOT G_CONTAINER IS INITIAL.

* Anlegen des Baumes nur, wenn noch nicht geschehen.
  CHECK G_TREE IS INITIAL.

  PERFORM CREATE_TREE_INSTANCE CHANGING G_TREE.
ENDMODULE.

FORM CREATE_TREE_INSTANCE
                CHANGING P_TREE TYPE REF TO CL_GUI_COLUMN_TREE.
DATA L_HEADER TYPE TREEV_HHDR.

* Hierarchie-Überschrift definieren
  L_HEADER-HEADING     = 'Hierarchie'(002).
  L_HEADER-WIDTH       = 50.

* Erzeugen der Instanz
  CREATE OBJECT G_TREE
     EXPORTING
        PARENT                = G_CONTAINER
        NODE_SELECTION_MODE   = G_TREE->NODE_SEL_MODE_SINGLE
        ITEM_SELECTION        = SPACE
        HIERARCHY_COLUMN_NAME = 'HIERARCHY'
```

```
      HIERARCHY_HEADER      = L_HEADER
    EXCEPTIONS
      LIFETIME_ERROR             = 1
      CNTL_SYSTEM_ERROR          = 2
      CREATE_ERROR               = 3
      FAILED                     = 4
      ILLEGAL_NODE_SELECTION_MODE = 5.
ENDFORM.
```

Im Vergleich zum Konstruktor der Klasse CL_GUI_SIMPLE_TREE sind die Parameter HIERACHY_COLUMN_NAME sowie HIERARCHY_HEADER, über die der Name und die Eigenschaften der Überschrift des Hierarchieteils des Baumes definiert werden, neu hinzugekommen. Hierbei wird dem Funktionsbaustein eine Struktur vom Typ TREEV_HHDR mit dem in Tabelle 3.9 dargestellten Aufbau übergeben. Neben der Überschrift der Spalte, kann eine Ikone oder ein Bitmap definiert werden, welches in der Überschrift mit ausgegeben wird. Dazu kann ein Text definiert werden, der als Tooltip angezeigt wird, wenn der Mauszeiger über der Spalte steht. Die Breite der Spalte wird im Feld WIDTH definiert. Abhängig vom Inhalt der Spalte WIDTH_PIX wird dieser Wert als Pixelbreite (»X«) oder als Breite in Zeichen (Space) interpretiert.

Feldname	Datentyp	Bedeutung
T_IMAGE	CHAR 6	Name der Ikone/des Bitmaps
HEADING	CHAR 132	Überschrift
TOOLTIP	CHAR 132	Tooltiptext der Spalte
WIDTH	INT 4	Bereite der Spalte
WIDTH_PIX	CHAR 1	Flag, ob WIDTH in Zeichen (Space) oder Pixeln ('X')

Tabelle 3.9
Aufbau der Struktur TREEV_HHDR

Im Anschluss daran müssen die Spalten, mit denen der Baum angezeigt wird, erzeugt werden. Hierzu wird die Methode ADD_COLUMN der Klasse CL_GUI_COLUMN_TREE verwendet. Der Aufruf dieser Methode wird ebenfalls in einer eigenen FORM-Routine namens CREATE_TREE_COLUMNS implementiert. Der Aufruf dieser Routine erfolgt im oben dargestellten MODULE unmittelbar nach der Erzeugung der Bauminstanz, aber vor dem Erstellen der Knoten.

```
MODULE CREATE_TREE
  ...
  CHECK NOT G_TREE IS INITIAL.
  PERFORM CREATE_TREE_COLUMNS USING G_TREE.
  ...
```

```
ENDMODULE.

FORM CREATE_TREE_COLUMNS USING P_TREE TYPE REF TO CL_GUI_COLUMN_TREE.
* Spalte Abflugstadt
  CALL METHOD P_TREE->ADD_COLUMN
       EXPORTING
              NAME        = 'CITYFROM'
              ALIGNMENT   = P_TREE->ALIGN_LEFT
              WIDTH       = 20
              HEADER_TEXT = 'Von'
       EXCEPTIONS
              COLUMN_EXISTS                = 1
              ILLEGAL_COLUMN_NAME          = 2
              TOO_MANY_COLUMNS             = 3
              ILLEGAL_ALIGNMENT            = 4
              DIFFERENT_COLUMN_TYPES       = 5
              CNTL_SYSTEM_ERROR            = 6
              FAILED                       = 7
              PREDECESSOR_COLUMN_NOT_FOUND = 8.
  ...
ENDFORM.
```

In dieser FORM-Routine wird für jede der Spalten, ein Aufruf der Methode ADD_COLUMN eingefügt. Auf eine Darstellung der vollständigen Routine wurde hier verzichtet.

Zuletzt müssen auch in diesen Baum die Knoten eingefügt werden. Sowohl beim Column-Tree als auch beim anschließend gezeigten List-Tree müssen hierzu zwei interne Tabellen erzeugt werden. In der ersten Tabelle, die auf der Dictionary-Struktur TREEV_NODE basieren muss (siehe Tabelle 3.10), werden die Knoten selbst definiert. In der zweiten Tabelle, deren Struktur die Dictionary-Tabelle TREEV_ITEM (siehe Tabelle 3.12) gefolgt von einem Textfeld mit dem Namen TEXT enthalten muss, werden die Items, in diesem Fall die Spalten, aus denen der Knoten aufgebaut ist, abgelegt. Die Struktur, die der Item-Tabelle zugrunde liegt, muss auf jeden Fall als Dictionary-Struktur angelegt werden. Im Beispiel wird diese unter dem Namen ZFLIGHTITEM erzeugt. Für jeden Knoten existiert demnach in der Node-Tabelle genau ein Satz während die Item-Tabelle für jede gefüllte Spalte jedes Knotens einen Satz enthält.

Feldname	Datentyp	Bedeutung
NODE_KEY	CHAR 12	Knotename (eindeutig im gesamten Baum)
RELATKEY	CHAR 12	Name des Bezugsknotens
RELATSHIP	INT 4	Beziehung zum Bezugsknoten
HIDDEN	CHAR 1	Kennzeichen, ob Knoten sichtbar ist

Feldname	Datentyp	Bedeutung
DISABLED	CHAR 1	Kennzeichen, ob Knoten selektierbar ist
ISFOLDER	CHAR 1	Kennzeichen, ob Knoten Unterknoten besitzt
N_IMAGE	CHAR 6	Icon für Knoten, wenn Teilbaum nicht angezeigt wird
EXP_IMAGE	CHAR 6	Icon für Knoten, wenn Teilbaum ausgeklappt ist
STYLE	INT 4	Ausgabestil des Knotens
LAST_HITEM	CHAR 12	Name des letzten Items, das zum Hierarchieteil des Baumes gehört (nur bei `CL_GUI_LIST_TREE`)
NO_BRANCH	CHAR 1	Kennzeichen, ob eine Verbinderlinie zum Konten gezeichnet werden soll
EXPANDER	CHAR 1	Kennzeichen, ob Knoten ausklappbar dargestellt werden soll
DRAGDROPID	INT 2	ID des Knotens für Drag & Drop

Tabelle 3.10
Aufbau der Struktur `TREEV_NTAB`

Feldname	Datentyp	Bedeutung
NODE_KEY	CHAR 12	Knotename (eindeutig im gesamten Baum)
ITEM_NAME	CHAR 12	Name des Items Bei `CL_GUI_COLUMN_TREE`: Name der Spalte, sonst Nummer der Spalte
CLASS	INT 4	Legt fest, ob es sich bei dem Item um einen Text (`ITEM_CLASS_TEXT`), eine Checkbox (`ITEM_CLASS_CHECKBOX`), einen Button (`ITEM_CLASS_BUTTON`) oder einen Link (`ITEM_CLASS_LINK`) handelt.

FONT	INT 4	Legt die Schriftart des Items Fest. Mögliche Werte:
		`ITEM_FONT_DEFAULT`
		`ITEM_FONT_PROP`
		`ITEM_FONT_FIXED`

DISABLED	CHAR 1	Kennzeichen, ob Item deaktiviert ist
EDITABLE	CHAR 1	Kennzeichen, ob Item geändert werden kann
HIDDEN	CHAR 1	Kennzeichen, ob Item sichtbar ist

Feldname	Datentyp	Bedeutung
ALIGNMENT	INT 4	Ausrichtung des Items (Nur in List-Trees relevant)
T_IMAGE	CHAR 6	Icon, die mit dem Item ausgegeben wird
CHOSEN	CHAR 1	Kennzeichen, ob Checkbox angewählt ist (nur Checkbox)
TOGG_RIGHT	CHAR 1	Kennzeichen bei Checkboxfeldern, ob das Kästchen links (Space) oder rechts ('X') vom Text des Items stehen soll
STYLE	INT 4	Stil des Items
LENGTH	INT 4	Sichtbare Länge des Items (nur in List-Trees)
LENGTH_PIX	CHAR 1	Kennzeichen, ob LENGTH in Pixeln ('X') oder Zeichen (Space). Verwendung nur in List-Trees
IGNOREIMAG	CHAR 1	Kennzeichen, ob LENGTH die Länge des gesamten Items mit Icon und ggf. Checkbox definiert ('X') oder nur die Länge des Textbereichs des Items (Space). Nur bei List-Trees
USEBGCOCOLOR	CHAR 1	Kennzeichen, ob das Item eine etwas andere Hintergrundfarbe bekommen soll ('X') oder die gleiche wie die anderen Items (Space). Nur bei List-Trees

Tabelle 3.11
Aufbau der Struktur TREEV_ITEM

Die Aufgabe der FORM-Routine CREATE_COLUMN_TREE_NODES besteht also darin, diese beiden Tabellen entsprechend der Daten in den Tabellen SCARR sowie SPFLI aufzubauen und an die Methode ADD_NODES_AND_ITEMS der Instanz des Baumes zu übergeben. Der Aufruf dieser Routine ersetzt den Aufruf der bisherigen Routine CREATE_SIMPLE_TREE_NODES.

```
FORM CREATE_COLUMN_TREE_NODES
                  USING P_TREE TYPE REF TO CL_GUI_COLUMN_TREE.
DATA: L_NODE_TAB     TABLE OF TYPE TREEV_NTAB,
      L_ITEM_TAB     TABLE OF TYPE ZFLIGHTITEM.
DATA: L_SPFLI        TABLE OF TYPE SPFLI,
      L_NODE_KEY     LIKE  L_NODE_TAB-NODE_KEY,
      L_HAS_SUBNODES TYPE  C.

* Erzeugen des Wurzelknotens
  L_NODE_TAB-NODE_KEY  = 'ROOT'.
  L_NODE_TAB-ISFOLDER  = 'X'.
```

```
  L_NODE_TAB-EXPANDER  = 'X'.
  L_ITEM_TAB-NODE_KEY  = 'ROOT'.
  L_ITEM_TAB-ITEM_NAME = 'HIERARCHY'.
  L_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  L_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_DEFAULT.
  L_ITEM_TAB-TEXT      = 'Flüge'.
  APPEND: L_NODE_TAB,
          L_ITEM_TAB.

* Aufbauen der Knotenstrukturen
  SELECT *
    FROM SCARR.
*   Einlesen der Flugverbindungen zur Airline
    SELECT *
      INTO TABLE L_SFPLI
      FROM SPFLI
     WHERE CARRID = SCARR-CARRID.

    IF SY-SUBRC EQ 0.
      L_HAS_SUBNODES = 'X'.
    ELSE.
      CLEAR L_HAS_SUBNODES.
    ENDIF.

*   Erzeugen des Airline-Knotens
    PERFORM CREATE_AIRLINE_NODE TABLES   L_NODE_TAB
                                         L_ITEM_TAB
                                USING    P_TREE
                                         'ROOT'
                                         SCARR
                                         L_HAS_SUBNODES
                                CHANGING L_NODE_KEY.

*   Knoten für Flugverbindungen erzeugen
    LOOP AT L_SPFLI.
      PERFORM CREATE_SPFLI_NODE TABLES L_NODE_TAB
                                       L_ITEM_TAB
                                USING  P_TREE
                                       L_NODEKEY
                                       L_SPFLI.
    ENDLOOP.
  ENDSELECT.

* Knoten in Baum einfügen
  CALL METHOD P_TREE->ADD_NODES_AND_ITEMS
       EXPORTING
           NODE_TABLE                = L_NODE_TAB
```

```
      ITEM_TABLE                = L_ITEM_TAB
      ITEM_TABLE_STRUCTURE_NAME = 'ZFLIGHTITEM'
    EXCEPTIONS
      FAILED = 1
      CNTL_SYSTEM_ERROR = 2
      ERROR_IN_TABLES = 3
      DP_ERROR = 4
      TABLE_STRUCTURE_NAME_NOT_FOUND = 5.
ENDFORM.
```

Das tatsächliche Umwandeln der Informationen in die Knotenstruktur wurde hier in die zwei Unterroutinen CREATE_AIRLINE_NODE für die Knoten der ersten Ebene, sowie CREATE_SPFLI_NODE für die Knoten der zweiten Ebene ausgelagert.

Im Anschluss an das Erzeugen der beiden internen Tabellen mit den Knoteninformationen und deren Darstellungsparametern werden die Knoten über einen Aufruf der Methode ADD_NODES_AND_ITEMS der Klasse CL_GUI_COLUMN_TREE in den Baum aufgenommen. Da der Name der Dictionary-Struktur für die Darstellungsattribute nicht fest definiert wurde, wird der Methode neben den beiden genannten Tabellen der Name der, der Item-Tabelle zugrunde liegenden Dictionary-Struktur, übergeben.

Die folgenden Listings zeigen schließlich noch die Implementierungen der beiden Routinen zum Erzeugen der jeweiligen Knoten. Auf der Ebene der Fluggesellschaften besteht die Repräsentation eines Knotens im Baum lediglich aus dem Namen der Fluggesellschaft im Hierarchieteil des Baums. Aus diesem Grund wird in der Node-Tabelle und der Item-Tabelle lediglich je ein Satz für die Airline-Knoten angelegt.

```
FORM CREATE_AIRLINE_NODE TABLES P_NODE_TAB STRUCTURE TREEV_NTAB
                                P_ITEM_TAB STRUCTURE ZFLIGHTITEM
                         USING P_TREE TYPE REF TO CL_GUI_COLUMN_TREE
                               P_PKEY     LIKE TREEV_NTAB-NODE_KEY
                               P_SCARR    LIKE SCARR
                               P_FOLDER   TYPE C
                      CHANGING P_NKEY     LIKE TREEV_NTAB-NODE_KEY.
* Key des neuen Knotens
  P_NKEY = SCARR-CARRID.

* Erzeugen des Knotens
  P_NODE_TAB-NODE_KEY  = P_NKEY.
  P_NODE_TAB-RELATKEY  = P_PKEY.
  P_NODE_TAB-REALTSHIP = P_TREE->RELAT_LAST_CHILD.
  P_NODE_TAB-ISFOLDER  = P_FOLDER.
  P_NODE_TAB-EXPANDER  = P_FOLDER.
  APPEND P_NODE_TAB.

  P_ITEM_TAB-NODE_KEY  = P_NKEY.
```

```
  P_ITEM_TAB-ITEM_NAME = 'HIERARCHY'.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  P_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_DEFAULT.
  P_ITEM_TAB-TEXT      = P_SCARR-CARRNAME.
  APPEND P_ITEM_TAB.
ENDFORM.
```

Anders verhält es sich für die Knoten der Flugverbindungen. Jeder Knoten des Baumes wird auf dieser Ebene durch die Flugnummer im Hierarchieteil sowie die Felder

- Abflugort (CITYFROM)
- Abflughafen (AIRPFROM)
- Zielort (CITYTO)
- Zielflughafen (AIRPTO) sowie die
- Abflugzeit (DEPTIME)

repräsentiert.

Der Key der erzeugten Knoten besteht hier nicht nur aus der ID der Fluggesellschaft (CARRID), wie bei den übergeordneten Knoten, sondern zusätzlich noch über die Flugnummer (CONNID) der Tabelle SPFLI. Um das Bilden des Schlüssels zu vereinfachen, wird im TOP-Include ein Struktur-Typ erzeugt, die die Zuweisung entsprechend der Implementierung der Routine CREATE_SPFLI_NODE vereinfacht.

```
TYPES: BEGIN OF T_NODE_KEY,
         CARRID  LIKE SCARR-CARRID,
         CONNID  LIKE SPFLI-CONNID,
         FLDATE  LIKE SFLIGHT-FLDATE,
       END OF T_NODE_KEY.
```

Die Struktur enthält bereits das für die dritte Baumebene benötigte Flugdatum (FLDATE). Die Zuweisung des Schlüssels an das entsprechende Strukturfeld erfolgt hierbei über eine Hilfsvariable mit dem neu definierten Typ. Der Schlüssel des neuen Knotens wird dabei elegant über die MOVE-CORRESPONDING-Anweisung erstellt. Diese Möglichkeit hätte ebenso bei der Erzeugung der Airline-Knoten verwendet werden können. Dann allerdings unter Verwendung der Feldleiste SCARR.

```
FORM CREATE_SPFLI_NODE TABLES    P_NODE_TAB STRUCTURE TREEV_NODE
                               P_ITEM_TAB STRUCTURE ZFLIGHTITEM
                       USING     P_TREE TYPE REF TO CL_GUI_COLUMN_TREE
                               P_PKEY     LIKE TREEV_NODE-NODE_KEY
                               P_SPFLI    LIKE SPFLI.
  DATA L_NKEY TYPE T_NODE_KEY.
```

```
* Knoten-Key bilden
  MOVE-CORRESPONDING SPFLI TO L_NKEY.

* Erzeugen des Knotens
  P_NODE_TAB-NODE_KEY  = L_NKEY.
  P_NODE_TAB-RELATKEY  = P_PKEY.
  P_NODE_TAB-RELATSHIP = P_TREE->RELAT_LAST_CHILD.
  APPEND P_NODE_TAB.

* Definition der Ausgabe
  P_ITEM_TAB-NODE_KEY  = L_NKEY.
  P_ITEM_TAB-ITEM_NAME = 'HIERARCHY'.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  P_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_DEFAULT.
  WRITE P_SPFLI-CONNID TO P_ITEM_TAB-TEXT.
  APPEND P_ITEM_TAB.

  P_ITEM_TAB-NODE_KEY  = L_NKEY.
  P_ITEM_TAB-ITEM_NAME = 'CITYFROM'.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  P_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_DEFAULT.
  WRITE P_SPFLI-CITYFROM TO P_ITEM_TAB-TEXT.
  APPEND P_ITEM_TAB.
  ...
ENDFORM.
```

Mit diesem Schlüssel wird zunächst der Knotensatz in der Tabelle P_NODE_TAB erzeugt Hierbei wird der neue Knoten immer als letzter Kind-Knoten an den übergeordneten Knoten (Parent), im Beispiel die Fluggesellschaft, angehangen (RELAT_LAST_CHILD). Optional könnte ein neuer Knoten auch als erster Kind-Knoten (RELAT_FIRST_CHILD), sowie als erster (RELAT_FIRST_SIBLING), letzter (RELAT_LAST_SIBLING), nächster (RELAT_NEXT_SIBLNG) oder vorheriger (RELAT_PREVIOUS_SIBLING) Knoten auf der gleichen Ebene wie der Referenzknoten angelegt werden.

Im Anschluss daran wird für jedes Feld, mit dem der Knoten angezeigt werden soll, ein Satz in der Item-Tabelle angelegt. Die erste Spalte, die die Flugnummer enthält, wird im Hierarchieteil des Baumes in der Spalte »HIERARCHY« dargestellt. Die übrigen Spalten, die im Quelltext lediglich angedeutet wurden, werden im hinteren Bereich der Baumdarstellung in den zuvor angelegten Spalten angezeigt. Um den Knoten vollständig zu füllen, müsste ein entsprechender Code-Block für jede der übrigen Spalten in die Routine aufgenommen werden. Die Spalteninhalte werden in der Tabelle P_ITEM_TAB abgelegt.

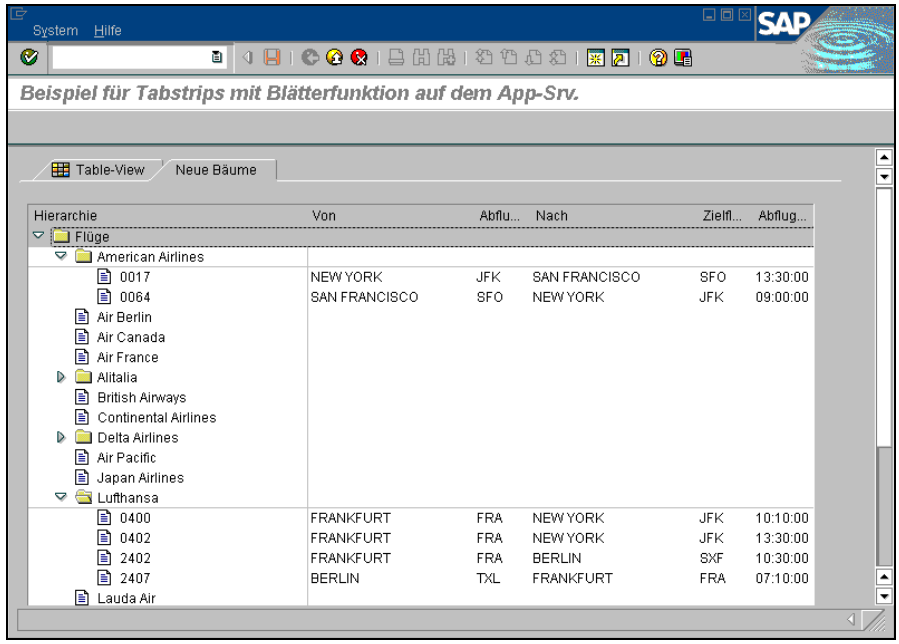

Abbildung 3.14
Darstellung eines Baumes vom Typ `CL_GUI_COLUMN_TREE` (© SAP AG)

Nachdem dies alles ausgeführt wurde, kann die Transaktion gestartet werden. Es spielt hierbei keine Rolle, ob die Transaktionsvariante zum Blättern des Tab-Strip im SAPGUI oder auf dem Applikationsserver verwendet wird. Die Baumdarstellung entspricht immer der in Abbildung 3.17 gezeigten Maske.

Deutlich erkennbar ist hier die Trennung des Hierarchieteils in der Spalte mit dem Titel »Hierarchie«, sowie der übrigen Spalten beginnend mit der Spalte »Von«. Jede der Spalten des Baumes kann vom Benutzer beliebig vergrößert oder verkleinert werden. Weiterhin werden die Standard-Ikonen für die Knoten verwendet, welche bei Knoten mit Unterstrukturen eine – je nach Zustand des Knotens – geöffnete oder geschlossene Akte ist, bei Knoten ohne Unterstrukturen eine beschriftete Seite.

Bäume mit Listenknoten

Die gerade gezeigte Variante, Bäume darzustellen, ist dann sinnvoll, wenn alle Knoten des Baumes mit den gleichen Spalten darstellbar sind oder wenn nur eine – in der Regel die unterste – Hierarchieebene mit mehreren Spalten angezeigt werden soll. Sollen verschiedene Ebenen des Baumes mit unterschiedlichen Spalteninformationen dargestellt werden, ist es schwierig, diese in ein festes Spaltenschema zu pressen. Hier besteht der Bedarf, dass jeder Knoten

für sich gesehen eine eigene visuelle Repräsentation im Baum annehmen kann. Um dieses Manko der spaltenorientierten Bäume zu umgehen, wurde von SAP ein dritter Typ von Bäumen, die List-Trees, definiert.

Bei dieser Baumart wird die mit den Column-Trees eingeführte Teilung der Knoten von deren visuellen Darstellungeigenschaften auf dem Ausgabegerät beibehalten. Auch hier wird, wie bei den Column-Trees auch, der Baum in einen Teil, der die Struktur des Baumes definiert und einen Teil, der die Zusatzinformationen der Knoten enthält, unterteilt. Anders als bei den Column-Trees werden beim List-Tree die Spalten jedoch nicht bei der Erzeugung des Baumes angelegt. Die Spalten eines List-Trees haben keinen Namen, sondern werden entsprechend ihrer Position in einem Knoten durchnummeriert. In dieser Eigenschaft sind die Spalten des List-Trees mit den Textfeldern der in Abschnitt 3.1 dargestellen Bäume vergleichbar, wobei die Anzahl der Felder, die einen Knoten ausmachen, nicht beschränkt ist. Zusätzlich dazu wurde die Beschränkung, dass der Hierarchieteil des Baumes nur aus einer Spalte bestehen kann, aufgehoben.

Vom Programmiermodell unterscheiden sich die List-Trees von den zuvor gezeigten Column-Trees nur in zwei Punkten. Zum einen müssen die Spalten des List-Trees nicht vor deren Verwendung angelegt werden und zum anderen können die einzelnen Spalten nicht mit einem Namen versehen werden, sondern werden – von links beginnend – durchnummeriert. Die Breite jeder Spalte kann dabei für jede Spalte jedes Knotens individuell festgelegt wrden. Dadurch können unterschiedliche Knotentypen leicht mit verschiedenen Spalteninformationen angezeigt werden.

Doch nun zur Praxis. In der dritten Ebene des Baumes der Flugbewegungen sollen zu den Flugverbindungen die Flüge angezeigt werden. Ein Flug ist dadurch gekennzeichnet, dass er eine Flugverbindung an einem bestimmten Datum ist. Dementsprechend wird der Schlüssel der Tabelle `SFLIGHT`, in der die Flüge abgelegt sind, im Vergleich zur Tabelle `SPFLI`, welche die Flugverbindungen enthält, um das Feld `FLDATE`, welches das Flugdatum enthält, erweitert.

Wie im Abschnitt 3.1 dieses Kapitels, soll die Anzeige eines Fluges aus den Feldern

- Flugdatum
- Preis
- Währung
- Flugzeugtyp
- Anzahl Plätze
- Anzahl gebuchter Plätze

bestehen.

Zunächst muss hierzu im Deklarationsteil des Beispielprogramms erneut der Typ der Variablen G_TREE verändert werden. Für den List-Tree kommt hier die Klasse CL_GUI_LIST_TREE zum Einsatz.

```
DATA G_TREE TYPE REF TO CL_GUI_LIST_TREE.
```

Dementsprechend muss die Initialisierung der Variablen in der FORM-Routine CREATE_TREE_INSTANCE an den neuen Konstruktor angepasst werden.

```
FORM CREATE_TREE_INSTANCE
                CHANGING P_TREE TYPE REF TO CL_GUI_COLUMN_TREE.
  DATA: L_HHEADER TYPE TREEV_HHDR,
        L_LHEADER TYPE TREEV_LHDR.

* Hierarchie-Überschrift definieren
  L_HHEADER-HEADING     = 'Hierarchie'.
  L_HHEADER-WIDTH       = 50.

* Listen-Überschrift definieren
  L_LHEADER-HEADING     = 'Liste'.

* Erzeugen der Instanz
  CREATE OBJECT P_TREE
     EXPORTING
        PARENT                = G_CONTAINER
        NODE_SELECTION_MODE   = G_TREE->NODE_SEL_MODE_SINGLE
        ITEM_SELECTION        = SPACE
        WITH_HEADERS          = 'X'
        HIERARCHY_HEADER      = L_HHEADER
        LIST_HEADER           = L_LHEADER
     EXCEPTIONS
        LIFETIME_ERROR             = 1
        CNTL_SYSTEM_ERROR          = 2
        CREATE_ERROR               = 3
        ILLEGAL_NODE_SELECTION_MODE = 4
        FAILED                     = 5.
ENDFORM.                              " CREATE_TREE_INSTANCE
```

Da beim List-Tree die Spalten nicht benannt werden, entfällt hier der Parameter HIERARCHY_COLUMN_NAME der Klasse CL_GUI_COLUMN_HEADER. Gleichwohl kann die Überschrift für den Hierarchieteil in Form einer Struktur vom Typ TREEV_HHDR (siehe Tabelle 3.9) definiert werden. Zusätzlich dazu kann eine Struktur mit der Überschrift des Listenteils des Baumes mit einer Variable vom Typ TREEV_LHDR definiert werden. Da der Listenteil immer den Rest des Raumes im Container einnimmt, entfällt bei dieser Struktur im Vergleich zur Struktur TREEV_HHDR das Feld WIDTH sowie das Feld WIDTH_PIX (siehe Tabelle 3.14).

Feldname	Datentyp	Bedeutung
T_IMAGE	CHAR 6	Name der Ikone/des Bitmaps
HEADING	CHAR 132	Überschrift
TOOLTIP	CHAR 132	Tooltiptext der Spalte

Tabelle 3.12
Aufbau der Struktur TREEV_LHDR

Über den Parameter WITH_HEADERS des Konstruktors kann festgelegt werden, ob der Baum am oberen Rand eine Überschriftenzeile anzeigt (»X«) oder nicht (Space).

Bei der Verwendung von List-Trees werden keine Spalten angelegt. Aus diesem Grund kann der Aufruf der Routine CREATE_TREE_COLUMNS im Modul CREATE_TREE der Ablauflogik entfallen. Deshalb müssen auch in der Routine CREATE_SPFLI_NODE des vorigen Abschnitts die Spaltennamen in den Zuweisungen zum Feld P_ITEM_TAB-ITEM_NAME durch eine Numerierung der Spalten, beginnend mit 1, ersetzt werden.

Beim List-Tree wurde die Beschränkung aufgehoben, dass der Hierarchieteil eines Knotens lediglich aus einer Spalte bestehen kann. Vielmehr kann für jeden Knoten individuell festgelegt werden, wieviele Spalten in der Hierarchie angezeigt werden. Dies geschieht über das Feld LAST_HITEM der Knotentabelle. Die Trennung zwischen Hierarchie und Liste ist somit vollkommen flexibel. Im Beispiel wird festgelegt, dass lediglich die erste Spalte des Knotens im Hierarchieteil angezeigt wird.

```
FORM CREATE_SPFLI_NODE TABLES     P_NODE_TAB STRUCTURE TREEV_NODE
                                  P_ITEM_TAB STRUCTURE ZFLIGHTITEM
                       USING      P_TREE TYPE REF TO CL_GUI_LIST_TREE
                                  P_PKEY     LIKE TREEV_NODE-NODE_KEY
                                  P_SPFLI    LIKE SPFLI
                                  P_FOLDER   TYPE C
                       CHANGING   P_NKEY     LIKE TREEV_NODE-NODE_KEY.
  DATA L_NKEY TYPE T_NODE_KEY.

* Knoten-Key bilden
  MOVE-CORRESPONDING P_SPFLI TO L_NKEY.
  P_NKEY = L_NKEY.

* Erzeugen des Knotens
  P_NODE_TAB-NODE_KEY   = L_NKEY.
  P_NODE_TAB-RELATKEY   = P_PKEY.
  P_NODE_TAB-RELATSHIP  = P_TREE->RELAT_LAST_CHILD.
  P_NODE_TAB-ISFOLDER   = P_FOLDER.
```

```
  P_NODE_TAB-EXPANDER   = P_FOLDER.
  P_NODE_TAB-LAST_HITEM = 1.
  APPEND P_NODE_TAB.

  P_ITEM_TAB-NODE_KEY  = L_NKEY.
  P_ITEM_TAB-ITEM_NAME = 1.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  P_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_DEFAULT.
  P_ITEM_TAB-LENGTH    = 5.
  WRITE P_SPFLI-CONNID TO P_ITEM_TAB-TEXT.
  APPEND P_ITEM_TAB.

  P_ITEM_TAB-NODE_KEY  = L_NKEY.
  P_ITEM_TAB-ITEM_NAME = 2.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.
  P_ITEM_TAB-FONT      = P_TREE->ITEM_FONT_PROP.
  P_ITEM_TAB-LENGTH    = 20.
  WRITE P_SPFLI-CITYFROM TO P_ITEM_TAB-TEXT.
  APPEND P_ITEM_TAB.
  ...
ENDFORM.
```

Das Beispielprogramm demonstriert ebenfalls, dass die Breite jeder Spalte für jeden Knoten unterschiedlich festgelegt werden kann. Hierfür existieren in der Item-Tabelle die Felder LENGTH sowie LENGTH_PIX um festzulegen, ob der Wert in LENGTH als Zeichenbreite oder Pixelbreite interpretiert werden soll.

Um die dritte Ebene des Baumes in die Anzeige zu integrieren wird die Routine CREATE_LIST_TREE_NODES definiert. Die Routine ähnelt in weiten Teilen der Routine CREATE_COLUMN_TREE_NODES aus dem vorigen Abschnitt. Zunächst wird der Wurzelknoten des Baumes erzeugt. Auch hier wird der Spaltenname »HIERARCHY« aus dem vorigen Abschnitt durch den Wert »1« ersetzt. Anschließend werden analog zum Column-Tree die Knoten der Fluggesellschaften aufgebaut. Bei den Knoten der Flüge wird zunächst geprüft, ob der Knoten untergeordnete Knoten besitzt und davon abhängig die Eigenschaften des Knotens angepasst. Hierzu muss auch die Schnittstelle der Routine CREATE_SPFLI_NODE dahingehend angepasst werden, dass ein Kennzeichen, ob der Knoten Unterstrukturen besitzt, in die Routine hineingereicht und der Key des erzeugten Knotens als CHANGING-Parameter zurückgeliefert wird.

Schließlich werden noch die Knoten der dritten Ebene durch einen Aufruf der Routine CREATE_SFLIGHT_NODE erzeugt, bevor die Knotenstrukturen durch den Aufruf der Methode ADD_NODES_AND_ITEMS an die Instanz des Baumes übergeben werden.

Zu beachten sind in diesem Zusammenhang auch die verwendeten Schriftarten bei den Items. Während beim Simple-Tree und beim Column-Tree standardmäßig eine Proportionalschrift verwendet wird, werden die Knoten des List-Trees

per Default mit einer nichtproportionalen Schrift dargestellt. Aus diesem Grund wird im Beispiel nicht der Defaultfont (P_TREE->ITEM_FONT_DEFAULT), sondern explizit die proportionale Schrift (P_TREE->ITEM_FONT_PROP) gewählt.

```
FORM CREATE_LIST_TREE_NODES
                  USING P_TREE TYPE REF TO CL_GUI_LIST_TREE.
DATA: L_NODE_TAB    TYPE STANDARD TABLE OF TREEV_NODE  WITH DEFAULT KEY,
      L_ITEM_TAB    TYPE STANDARD TABLE OF ZFLIGHTITEM WITH DEFAULT KEY.
DATA: L_NODE        LIKE TREEV_NODE,
      L_ITEM        LIKE ZFLIGHTITEM.
DATA: L_SCARR       LIKE SCARR   OCCURS 0 WITH HEADER LINE,
      L_SPFLI       LIKE SPFLI   OCCURS 0 WITH HEADER LINE,
      L_SFLIGHT     LIKE SFLIGHT OCCURS 0 WITH HEADER LINE.
DATA: L_SCARR_KEY   TYPE TREEV_NODE-NODE_KEY,
      L_SPFLI_KEY   TYPE TREEV_NODE-NODE_KEY,
      L_SFLIGHT_KEY TYPE TREEV_NODE-NODE_KEY,
      L_HAS_SUBNODES TYPE C.

* Erzeugen des Wurzelknotens
  L_NODE-NODE_KEY       = 'ROOT'.
  L_NODE-ISFOLDER       = 'X'.
  L_NODE-EXPANDER       = 'X'.
  L_NODE-LAST_HITEM     = 1.
  L_ITEM-NODE_KEY       = 'ROOT'.
  L_ITEM-ITEM_NAME      = 1.
  L_ITEM-CLASS          = P_TREE->ITEM_CLASS_TEXT.
  L_ITEM-FONT           = P_TREE->ITEM_FONT_PROP.
  L_ITEM-LENGTH         = 30.
  L_ITEM-TEXT           = 'Flüge'.
  APPEND: L_NODE TO L_NODE_TAB,
          L_ITEM TO L_ITEM_TAB.

* Aufbauen der Knotenstrukturen
  SELECT *
    INTO TABLE L_SCARR
    FROM SCARR.

  LOOP AT L_SCARR.
*   Einlesen der Flugverbindungen zur Airline
    SELECT  *
      INTO  TABLE L_SPFLI
      FROM  SPFLI
      WHERE CARRID = L_SCARR-CARRID.

    IF SY-SUBRC EQ 0.
      L_HAS_SUBNODES = 'X'.
    ELSE.
```

```
      CLEAR L_HAS_SUBNODES.
    ENDIF.

*   Erzeugen des Airline-Knotens
    PERFORM CREATE_AIRLINE_NODE TABLES   L_NODE_TAB
                                         L_ITEM_TAB
                                USING    P_TREE
                                         'ROOT'
                                         L_SCARR
                                         L_HAS_SUBNODES
                                CHANGING L_SCARR_KEY.

*   Knoten für Flugverbindungen erzeugen
    LOOP AT L_SPFLI.
*     Einlesen der Flugverbindungen zur Airline
      SELECT *
        INTO TABLE L_SFLIGHT
        FROM SFLIGHT
       WHERE CARRID = L_SPFLI-CARRID AND
             CONNID = L_SPFLI-CONNID.

      IF SY-SUBRC EQ 0.
        L_HAS_SUBNODES = 'X'.
      ELSE.
        CLEAR L_HAS_SUBNODES.
      ENDIF.

      PERFORM CREATE_SPFLI_NODE TABLES   L_NODE_TAB
                                         L_ITEM_TAB
                                USING    P_TREE
                                         L_SCARR_KEY
                                         L_SPFLI
                                         L_HAS_SUBNODES
                                CHANGING L_SPFLI_KEY.

      LOOP AT L_SFLIGHT.
        PERFORM CREATE_SFLIGHT_NODE TABLES L_NODE_TAB
                                           L_ITEM_TAB
                                    USING  P_TREE
                                           L_SPFLI_KEY
                                           L_SFLIGHT.
      ENDLOOP.
    ENDLOOP.
  ENDLOOP.

  CALL METHOD P_TREE->ADD_NODES_AND_ITEMS
       EXPORTING
```

```
        NODE_TABLE                = L_NODE_TAB
        ITEM_TABLE                = L_ITEM_TAB
        ITEM_TABLE_STRUCTURE_NAME = 'ZFLIGHTITEM'
      EXCEPTIONS
        FAILED = 1
        CNTL_SYSTEM_ERROR = 2
        ERROR_IN_TABLES = 3
        DP_ERROR = 4
        TABLE_STRUCTURE_NAME_NOT_FOUND = 5.
ENDFORM.                              " CREATE_LIST_TREE_NODES
```

Die Implementierung der Routine CREATE_SFLIGHT_NODES beinhaltet zunächst keine neuen Elemente. Daher wird auf eine Darstellung an dieser Stelle verzichtet.

Verwendung von Ikonen zur Darstellung von Knoten

In den Bäumen des Control Frameworks können sehr einfach auch Ikonen und Bitmaps verwendet werden, um Informationen zu visualisieren. Standardmäßig kann jeder Knoten im Hierarchieteil eine Ikone enthalten. Diese Ikone kann beim Expandieren des Knotens eine andere sein als im komprimierten Zustand. Bei Column-Trees und List-Trees kann zusätzlich hierzu noch jede dargestellte Spalte über eine eigene Ikone, welche links vom Text der Spalte ausgegeben wird, verfügen.

Da die Knoten des Simple-Trees lediglich über eine interne Tabelle aufgebaut wird (exemplarisch sei hier nochmals auf Tabelle 3.8 verwiesen), gestaltet sich die Verwendung von Ikonen hier besonders einfach. Die Knotentabelle enthält zwei Felder N_IMAGE sowie EXP_IMAGE über die die gewünschten Ikonen definiert werden können. Hierbei wird die in N_IMAGE abgelegte Ikone angezeigt, solange der Knoten komprimiert dargestellt wird. Werden die Unterstrukturen des Knotens expandiert, wird automatisch auf die Anzeige des in EXP_IMAGE definierten Bildes gewechselt. Dabei kann die Zuweisung der Ikone über die im TYPE-POOL ICON definierten Konstanten erfolgen. Hierzu müssen im TOP-Include des Modulpools allerdings mit der Anweisung

```
TYPE-POOLS ICON.
```

die Konstanten der Ikonen geladen werden[2].

Bei Column- und List-Trees sind diese beiden Felder mit dem gleichen Verhalten in der Struktur TREEV_NODE (siehe Tabelle 3.10) enthalten. Zusätzlich dazu kann jede Zeile in der Item-Tabelle (siehe Tabelle 3.12) eine weitere Ikone im

2. Bislang wurde hierzu empfohlen, das Include <ICON> ins Programm zu integrieren. In Release 4.6 weist ein Kommentar in diesem Include explizit darauf hin, dass dies in Zukunft nicht empfohlen ist und stattdessen die dargestellte TYPE-POOLS-Anweisung verwendet werden soll.

Feld T_IMAGE definieren. Diese wird im Feld des Hierarchie-Knotens zusätzlich zum in TREEV_NODE definierten Icon ausgegeben.

SAP hat für die Knoten Standard-Ikonen definiert, die verwendet werden, wenn die genannten Felder der Tabellen leer bleiben. Hierarchieknoten mit Unterstruktur haben standardmäßig die vom Windows-Explorer bekannten Ordner-Symbole.

Soll explizit definiert werden, dass ein Knoten kein Icon hat, muss das entsprechende Feld mit dem Wert »BNONE« belegt werden. Soll ein Icon über dessen numerischen Wert und nicht über die Konstante angesprochen werden, muss der Wert in der Form »@<Wert>@« angegeben werden, wobei <Wert> die Nummer des Icons ist.

Der Baum aus dem vorigen Abschnitt soll hier noch um die Ampeldarstellung zur Anzeige der Buchungslage ergänzt werden. Hierzu muss in der FORM-Routine CREATE_SFLIGHT_NODE eine Erweiterung dergestalt vorgenommen werden, dass zunächst über die Felder SEATSMAX und SEATSOCC der Struktur SFLIGHT die Belegungsquote berechnet wird und abhängig von im TOP-Include zu definierender konstanter Schwellwerte das passende Icon ausgewählt wird.

Die Berechnung der Belegungsquote geschieht in der Routine CREATE_SFLIGHT_NODE, wie in Abschnitt 3.1.2 gezeigt, über das Hilfsfeld L_BELEGUNG nach folgendem Muster.

```
DATA: L_BELEGUNG TYPE F.
  ...
  IF P_SFLIGHT-SEATSMAX NE 0.
    L_BELEGUNG = ( ( P_SFLIGHT-SEATSOCC / P_SFLIGHT-SEATSMAX ) * 100 ).
  ELSE.
    L_BELEGUNG = 0.
  ENDIF.
  ...
```

Im TOP-Include wird auch in diesem Beispielprogramm eine Struktur C_BELEGUNG mit den konstanten Schwellwerten für die Auswahl der Ikone eingefügt. Diese Struktur wird identisch wie die gleichnamige Struktur im Beispielprogramm aus Abschnitt 3.1 definiert.

```
CONSTANTS: BEGIN OF C_BELEGUNG,
             WARNING   TYPE I   VALUE 70,
             ALERT     TYPE I   VALUE 95,
           END OF C_BELEGUNG.
```

Schließlich bleibt noch, abhängig von der Belegungsquote ein Feld mit dem gewünschten Icon in die Item-Tabelle einzufügen.

```
  ...
  CLEAR P_ITEM_TAB.
  P_ITEM_TAB-NODE_KEY  = L_NKEY.
```

```
  P_ITEM_TAB-ITEM_NAME = 7.
  P_ITEM_TAB-CLASS     = P_TREE->ITEM_CLASS_TEXT.


* Defaultwert setzen
  P_ITEM_TAB-T_IMAGE = ICON_GREEN_LIGHT.

  IF L_BELEGUNG >= C_BELEGUNG-WARNING.
    P_ITEM_TAB-T_IMAGE = ICON_YELLOW_LIGHT.
  ENDIF.

  IF L_BELEGUNG >= C_BELEGUNG-ALERT.
    P_ITEM_TAB-T_IMAGE = ICON_RED_LIGHT.
  ENDIF.

  APPEND P_ITEM_TAB.
  ...
```

Abbildung 3.15 zeigt die Baumdarstellung nachdem die beschriebenen Änderungen vollzogen wurden. Für die Knoten der dritten Baumebene wird das gewünschte Symbol analog zur Ausgabe des Programms in Abschnitt 3.1 ausgegeben.

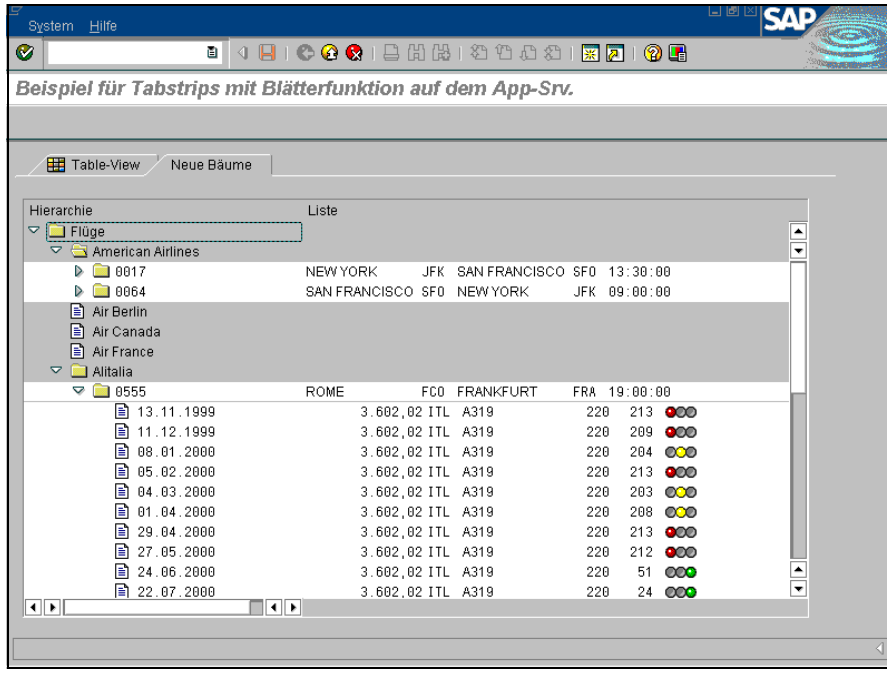

Abbildung 3.15
Baumdarstellung eines List-Trees mit Ikonen (© SAP AG)

Interaktion mit dem Baum

Bisher wurde lediglich gezeigt, wie aus einem Datenmodell die drei Baumvarianten erzeugt werden konnten und wie die Darstellung der Knoten des Baums beeinflusst werden kann. Andersherum besteht auch innerhalb des Control Frameworks, zu dem die Baumdarstellungen gehören, die Möglichkeit, auf Benutzereingaben zu reagieren. Hierzu wurde seitens SAP ein Eventmodell definiert, welches es den Controls ermöglicht, Ereignisse, die im GUI des Benutzers auftreten, an den Applikationsserver und das darauf ablaufende ABAP-Programm weiterzuleiten. Hierzu definiert jedes Custom Control eine Reihe von Ereignissen, die auftreten können und zu denen die Kontrolle wieder an das ABAP-Programm übergeben werden soll.

Da nicht alle Events für ein Programm interessant sind, existiert ein Mechanismus, bei dem sich das ABAP-Programm beim Control mit den Events, für die eine Benachrichtigung erfolgen soll, registrieren muss. Da die Controls des Control Frameworks entweder als Javabean oder als ActiveX auf dem GUI des Benutzers ablaufen, kann hier gefiltert werden und eine Kommunikation mit dem Applikationsserver nur vollzogen werden, wenn sich das ABAP-Programm für den aufgetretenen Event interessiert. Somit wird die Netzwerk- und Serverbelastung gering gehalten und das Programm muss nur auf die Events reagieren, die auch behandelt werden sollen. Diese Art der Rückmeldung war bereits bei der auf Funktionsbausteinen basierenden Baumdarstellung möglich (siehe Abschnitt 3.1.3). Der Unterschied war jedoch, dass bei der alten Baumdarstellung die Kontrolle auf jeden Fall an den Applikationsserver übergeben wurde und dieser lediglich bei Bedarf eine definierte `FORM`-Routine aufgerufen hat. Die Kontrolle liegt bei diesem Verfahren also auf jeden Fall auf dem Server und innerhalb der ABAP-Laufzeitumgebung. Bei den Custom Controls, bei denen Programmcode auf dem Client abläuft, der nicht zur ABAP-Laufzeitumgebung gehört, musste – auch aufgrund der Vielzahl der möglichen Events - ein anderes Verfahren gefunden werden, um effizient auf Ereignisse reagieren zu können.

Um auf einen in der Baumdarstellung auftretenden Event reagieren zu können, müssen die folgenden Schritte durchlaufen werden: Exemplarisch soll hier auf den Event `NODE_DOUBLE_CLICK` reagiert werden, der von der Baumdarstellung ausgelöst wird, wenn der Benutzer einen Doppelclick auf einen Knoten durchführt.

Zunächst muss im ABAP-Programm eine Tabelle der Events, auf die das Programm reagieren soll, angelegt werden. Basierend auf der Struktur `CNTL_SIMPLE_EVENT` (siehe Tabelle 3.13), welche in der Typgruppe `CNTL` definiert ist, steht ein Tabellentyp mit dem Namen `CNTL_SIMPLE_EVENTS` zur Verfügung.

Feldname	Datentyp	Bedeutung
EVENTID	I	ID des Events
APPL_EVENT	C	Kennzeichen, ob es sich bei dem Event um einen Applikationsevent (`'X'`) oder einen Systemevent (`Space`) handelt

Tabelle 3.13
Aufbau der Struktur CNTL_SIMPLE_EVENT

SAP unterscheidet bei Events zwischen so genannten Systemevents und Applikationsevents. Bei ersteren wird vor dem Aufruf der, den Event behandelnden ABAP-Routine, kein `PROCESS AFTER INPUT`-Ereignis ausgelöst. Es findet demnach kein Feldtransport in die Programmvariablen und auch keine Plausibilisierung statt. Im Gegensatz dazu sind Applikationsevents zu sehen, bei denen zunächst das PAI ausgelöst wird und – wenn nicht vorher explizit angefordert – die Auswertung der Events am Ende des PAI stattfindet. Der Programmierer kann die sofortige Auswertung der Applikationsevents durch den Aufruf der Klassenmethode

```
CL_GUI_CFW=>DISPATCH.
```

erzwingen. Diese kann an beliebiger Stelle während der Verarbeitung des PAI stehen. Es handelt sich hierbei nicht um eine Anweisung der Ablauflogik, daher muss dieser Aufruf innerhalb eines `MODUL`s oder einer `FORM`-Routine erfolgen.

Im Beispiel soll lediglich auf den Event `NODE_DOUBLE_CLICK` reagiert werden. Hierzu wird in der `FORM`-Routine `CREATE_TREE_INSTANCE` zunächst die Eventtabelle deklariert und mit dem entsprechenden Satz gefüllt.

```
...
DATA: L_EVENTS TYPE CNTL_SIMPLE_EVENTS,
      L_EVENT  TYPE CNTL_SIMPLE_EVENT.
...
L_EVENT-EVENTID    = CL_ITEM_TREE_CONTROL=>EVENTID_NODE_DOUBLE_CLICK.
L_EVENT-APPL_EVENT = 'X'.
APPEND L_EVENT TO L_EVENTS.
...
```

Im Anschluss daran muss die Eventtabelle an die neu erzeugte Instanz des SAP Trees übergeben werden. Hierfür steht die Methode `SET_REGISTERED_EVENTS` der Klassse `CL_GUI_CONTROL`, von der alle Custom Controls abgeleitet sind, zur Verfügung. Für die neu erzeugte Bauminstanz wird die Methode wie folgt aufgerufen:

```
CALL METHOD P_TREE->SET_REGISTERED_EVENTS
     EXPORTING
```

```
      EVENTS     = L_EVENTS
    EXCEPTIONS
      CNTL_ERROR = 1
      CNTL_SYSTEM_ERROR = 2
      ILLEGAL_EVENT_COMBINATION = 3.
```

Somit sind die Maßnahmen abgeschlossen, die notwendig sind, um dem, auf dem Präsentationsrechner laufenden, Control mitzuteilen, welche Ereignisse in welcher Art zur Kommunikation mit dem Applikationsserver führen sollen. Was noch fehlt, um tatsächlich auf den Event zu reagieren, ist der Code, der aufgerufen wird, wenn das Ereignis im entsprechenden Control ausgelöst wird.

Bei den Routinen, die Events aus dem Control Framework behandeln, muss es sich immer um Methoden einer ABAP-Klasse handeln. Normalerweise wird hierfür innerhalb des ABAP-Programms eine Klasse mit lokaler Gültigkeit deklariert und implementiert, die für jeden registrierten Event eine Methode mit der entsprechenden Schnittstelle zur Verfügung stellt. Bei diesen Methoden spielt es keine Rolle, ob es sich um Klassen-Methoden oder Instanz-Methoden der Klasse handelt. Einleuchtenderweise muss bei Verwendung von Instanzmethoden allerdings eine globale Variable für die Klasse definiert und eine Instanz der Klasse erzeugt werden. Da das grundlegende Verhalten beider Varianten identisch ist, wird an dieser Stelle lediglich auf die Verwendung von Klassen-Methoden zur Ereignisbehandlung eingegangen.

Die Definition der Klasse – ihr Name kann beliebig gewählt werden – enthält hier nur eine Klassenmethode, die aufgerufen werden soll, sobald der Benutzer im Baum einen Knoten doppelt mit der Maus anklickt. Der Event, der dabei ausgelöst wird, hat den Namen NODE_DOUBLE_CLICK.

```
CLASS LCL_EVENT_HANDLER DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS NODE_DBL_CLICK_HANDLER
                  FOR EVENT NODE_DOUBLE_CLICK
                  OF CL_GUI_LIST_CONTROL
                  IMPORTING NODE_KEY
                            SENDER.
ENDCLASS.
```

Mit dieser Deklaration wird die Klassenmethode NODE_DBL_CLICK_HANDLER definiert, die auf den Event NODE_DOUBLE_CLICK der Klasse CL_ITEM_TREE_CONTROL verknüpft werden soll. Neben dem Parameter SENDER, der bei jedem Event übergeben wird und eine Referenz auf das Objekt enthält, welches den Event ausgelöst hat (in unserem Beispiel wird hier immer die in G_TREE referenzierte Instanz des SAP Trees enthalten sein), definiert der Event NODE_DOUBLE_CLICK einen weiteren Parameter, welcher die Knoten ID des betroffenen Knotens enthält.

Durch die Verknüpfung der Methode mit einem bestimmten Event einer festgelegten Klasse ist es möglich, Eventhandler für unterschiedliche Controls in einer

lokalen Klasse zu vereinen. Auf diese Weise kann der Programmcode kompakter gestaltet werden, insbesondere, wenn im Programm viele Custom Controls verwendet werden.

Die Implementierung der Klasse ist an dieser Stelle nicht von Bedeutung. Im Beispiel enthält die Methode `NODE_DBL_CLICK_HANDLER` lediglich einen Aufruf des Funktionsbausteins `POPUP_TO_DISPLAY_TEXT` zur Anzeige der angewählten Knoten-ID. Auf eine Darstellung des Programmcodes wird an dieser Stelle verzichtet.

Zuletzt muss die noch fehlende Verbindung zwischen der Instanz des Baumes und der Klasse, die auf die registrierten Events reagieren soll, hergestellt werden. Hierzu steht die ABAP-Anweisung `SET HANDLER` zur Verfügung. Im konkreten Fall wird der Aufruf

```
SET HANDLER LCL_EVENT_HANDLER DEFINITION=>NODE_DBL_CLICK_HANDLER
        FOR P_TREE.
```

am Ende der Implementierung der Routine `CREATE_TREE_INSTANCE` angefügt. Wird das Programm jetzt gestartet und ein Doppelklick auf einen beliebigen Knoten durchgeführt, erscheint das in Abbildung 3.16 gezeigte Fenster.

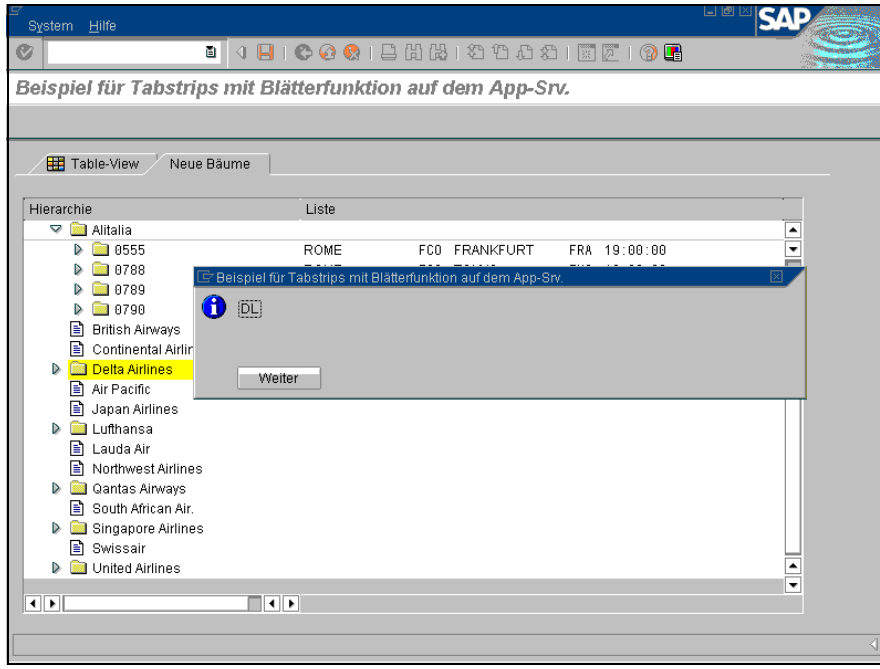

Abbildung 3.16
Darstellung des Dynpros nach Auslösen des Events (© SAP AG)

3.4 DropDown-Listen

In bisherigen R/3-Releaseständen bestand die Möglichkeit, in Eingabefeldern dem Benutzer mögliche Eingabewerte in Form der so genannten Werthilfe oder auch F4-Hilfe anzubieten. Die angebotenen Werte konnten entweder explizit vom Programmierer innerhalb der Verarbeitungsroutinen des PROCESS ON VALUE-REQUEST-Blocks(POV), oder über die Definitionen im Data Dictionary festgelegt werden. Bei letzterer Möglichkeit, ziehen sowohl Prüftabellen als auch Festwerte an der jeweiligen Domäne des Dynpro-Feldes. Der Benutzer musste diese Werthilfe jedoch nicht verwenden, sondern konnte auch einen gültigen Wert direkt in das Eingabefeld schreiben.

Seit Release 4.0 bietet SAP ein weitere Möglichkeit an, dem Benutzer mögliche Eingabewerte in ein Feld anzubieten. Es handelt sich hierbei um die so genannten DropDown-Listen. Optisch erkennt man diese neue Variante der Darstellung daran, dass es nicht möglich ist, direkt in das Eingabefeld auf der Maske zu schreiben. Stattdessen befindet sich rechts neben dem Eingabefeld eine Schaltfläche mit einem Pfeil, über den der Benutzer ein Listenfeld der möglichen Eingabewerte des Feldes angezeigt bekommt und daraus auswählen kann. Ein Beispiel für ein solches DropDown-Listenfeld ist in Abbildung 3.17 dargestellt.

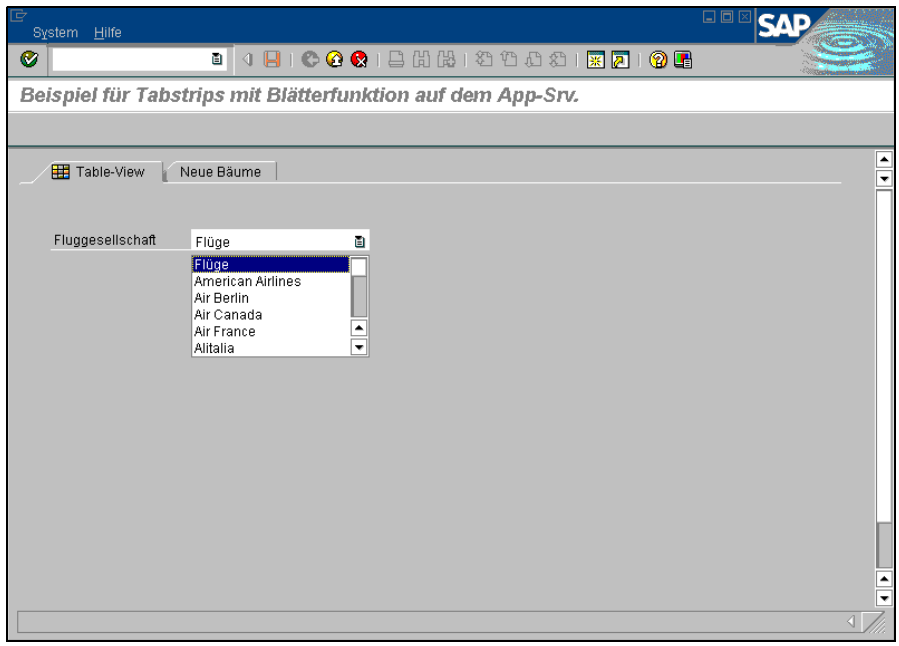

Abbildung 3.17
Maske mit Drop-Down-Listenfeld (© SAP AG)

3.4.1 Anlegen des DropDown-Feldes

Soll ein Eingabefeld als DropDown-Liste dargestellt werden, muss dies bereits im Dynpro-Editor kenntlich gemacht werden. Hierzu muss in den Eigenschaften des Eingabefeldes die Option DropDown aktiviert werden (siehe Abbildung 3.24) wie hier auf dem Dynpro `9100` des Beispielprogramms aus den vorhergehenden Abschnitten. Anders als bei normalen Eingabefeldern besteht bei DropDown-Listen zusätzlich die Möglichkeit, dass ein PAI-Ereignis ausgelöst wird, sobald sich der Wert des Feldes ändert. Hierzu kann dem Feld ein Funktionscode zugewiesen werden, mit dem das PAI ausgelöst wird. Hierdurch kann im Programm unmittelbar auf die Auswahl reagiert und z.B. Listen aktualisiert werden oder ähnliches.

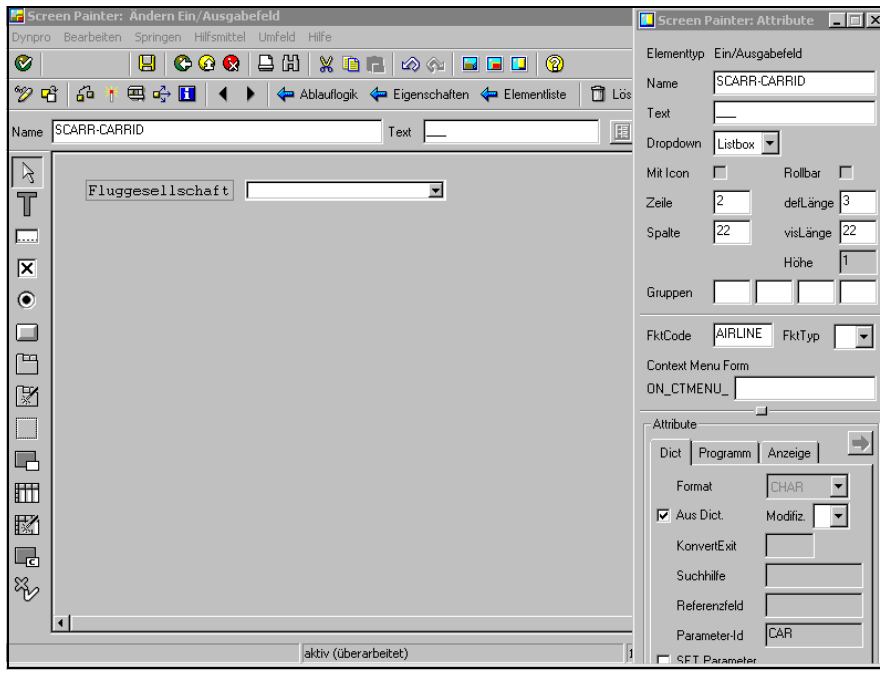

Abbildung 3.18
Dynpro-Editor mit Eigenschaften eines DropDown-Feldes (© SAP AG)

Weiterhin muss die sichtbare Länge des Feldes ggf. angepasst werden, da im DropDown-Feld nicht unbedingt der Feldwert selbst, sondern eine Beschreibung des Schlüsselwerts im Klartext angezeigt werden kann. Im Beispiel dieses Abschnitts ist das DropDown-Feld zwar für das Feld `SCARR-CARRID` definiert, in der Liste und auch im Feld selbst wird jedoch der Name der Fluggesellschaft im Klartext angezeigt. Die sichbare Länge des Feldes auf dem Dynpro muss dem-

entsprechend auf die Länge des Feldes CARRNAME gesetzt werden, obwohl das Dictionary-Feld, welches dem Eingabefeld zugrunde liegt, das Feld CARRID ist.

3.4.2 Beeinflussen des Inhalts der DropDown-Liste

Anders als normale Eingabefelder besteht der Inhalt eines DropDown-Feldes aus einem Schlüsselwert, der den eigentlichen Feldwert repräsentiert und dem Ausgabewert, der im Feld ausgegeben bzw. in der Auswahlliste des Feldes angeboten wird. Die Länge des Schlüsselwertes kann hierbei maximal 40 Stellen betragen, während das Anzeigefeld maximal 80 Stellen lang sein darf. Im vorliegenden Beispiel ist der Schlüsselwert die ID der jeweiligen Fluggesellschaft, wie in der Tabelle SCARR im Feld CARRID definiert. Der Anzeigewert des Feldes ist jedoch der Name der Fluggesellschaft, der in in der gleichen Tabelle im Feld CARRNAME abgelegt ist.

Es gibt hierbei eine Reihe von Möglichkeiten, die Auswahlliste des DropDown-Feldes zu beeinflussen. Zum einen können im PBO des Dynpros die Eingabealternativen durch einen Aufruf des Funktionsbausteins VRM_SET_VALUES explizit gesetzt werden. Als zweite Möglichkeit bietet sich an, im POV des Dynpros auf eine Werthilfe-Anforderung des Benutzers zu reagieren und beispielsweise durch einen Aufruf des Funktionsbausteins F4IF_INT_TABLE_VALUE_REQUEST die Listenausgabe zu beeinflussen. Ist keine dieser beiden Möglichkeiten realisiert, versucht das R/3-System die Eingabealternativen aus den normalen Werthilfe-Möglichkeiten (Prüftabelle, Domänen-Festwerte, Matchcode-Objekt) zu ermitteln. Diese Wege werden in den folgenden Abschnitten jeweils kurz dargestellt. Alle drei genannten Wege liefern im Endeffekt das gleiche Ergebnis für den Anwender. Sie unterscheiden sich daher lediglich in der internen Logik des Programmes.

Definition der Eingabealternativen im PBO

In diesem Abschnitt wird davon ausgegangen, dass für das Dynpro-Feld SCARR-CARRID eine DropDown-Liste definiert werden soll, die bereits im PBO gesetzt wird. Hierzu wird der Funktionsbaustein VRM_SET_VALUES verwendet. Dieser erhält – neben dem Namen des Dynpro-Feldes – eine interne Tabelle als Parameter übergeben, in dem sowohl die Schlüsselwerte als auch die Anzeigewerte für jede Alternative enthalten sind. Diese interne Tabelle muss den Aufbau der, im Type-Pool VRM definierten, Struktur VRM_VALUE haben.

Zunächst muss daher der Type-Pool VRM in das Programm mit der Anweisung

```
TYPE-POOLS VRM.
```

eingebunden werden. Weiterhin wird davon ausgegangen, dass von einem Modul des PBO aus die folgende FORM-Routine aufgerufen wird, welche dem Listenfeld die möglichen Alternativen zuweist.

```
FORM INIT_DropDown_CARRID.
  DATA LISTVALUES TABLE OF TYPE VRM_VALUE.

* Alternativen ermitteln
  SELECT CARRID
         CARRNAME
    INTO LISTVALUES-KEY
         LISTVALUES-TEXT
    FROM SCARR.
    APPEND LISTVALUES.
  ENDSELECT.

* Liste dem Feld zuweisen
  CALL FUNCTION 'VRM_SET_VALUES'
       EXPORTING
          ID     = 'SCARR-CARRID'
          VALUES = LISTVALUES[].
ENDFORM.
```

In der SELECT-Anweisung werden zunächst die Fluggesellschaften aus der Tabelle SCARR ausgelesen und in die Felder KEY und TEXT der internen Tabelle LISTVALUES übertragen. Die Airline-ID (CARRID) wird hierbei als Schlüssel, der Name der Fluggesellschaft als Anzeigetext der Alternativen verwendet.

Diese interne Tabelle wird anschließend zusammen mit dem Feldnamen SCARR-CARRID an den Funktionsbaustein VRM_SET_VALUES übergeben. Weitere Maßnahmen sind bei Verwendung dieser Methode nicht erforderlich.

Ermittlung der DropDown-Liste im POV

Eine andere Möglichkeit, den Inhalt der DropDown-Liste zu bestimmen ist die, im Ereignisblock PROCESS ON VALUE-REQUEST für das Dynpro-Feld ein Modul zu definieren. Dieses Modul wird genauso beim Aufklappen des Listenfeldes aufgerufen wie beim Betätigen der F4-Taste, wenn der Cursor im betreffenden Feld steht. Zum Anzeigen der Liste der Eingabemöglichkeiten wird in diesem Fall der SAP-Funktionsbaustein F4IF_INT_TABLE_VALUE_REQUEST verwendet.

Dieser Funktionsbaustein bekommt im Parameter RETFIELD den Namen des Dynpro-Feldes übergeben, in dem das ausgewählte Item abgelegt werden soll. Weiterhin wird im TABLES-Parameter VALUE_TAB eine interne Tabelle mit den zur Auswahl stehenden Werten übergeben. Diese Tabelle muss jedoch nicht wie beim Funktionsbaustein VRM_SET_VALUES einer festen Struktur gehorchen. Vielmehr wird davon ausgegangen, dass die Tabelle zwei Felder enthält. Beim ersten Feld handelt es sich implizit um den Key und beim zweiten Feld um den Anzeigewert des Eintrags. Über den Parameter VALUE_ORG kann zusätzlich die Sortierung der Werte der DropDown-Liste beeinflusst werden.

Wird auch für diesen Fall eine FORM-Routine zum Füllen der Werteliste definiert, könnte diese folgende Implementierung besitzen.

```
FORM DropDown_CARRID_POV.
  DATA: BEGIN OF LISTVALUES OCCURS 0,
          CARRID   LIKE SCARR-CARRID,
          CARRNAME LIKE SCARR-CARRNAME,
        END OF LISTVALUES.

* Wertetabelle einlesen
  SELECT *
     INTO CORRESPONDING FIELDS OF TABLE LISTVALUES
     FROM SCARR.

* Tabelle anzeigen und auswählen
  CALL FUNCTION 'F4IF_INT_TABLE_VALUE_REQUEST'
       EXPORTING
          RETFIELD    = 'SCARR-CARRID'
          VALUE_ORG   = 'S'
       TABLES
          VALUE_TAB   = LISTVALUES
       EXCEPTIONS
          PARAMETER_ERROR = 1
          NO_VALUES_FOUND = 2
          OTHERS          = 3.
ENDFORM.
```

Nachdem auch hier zunächst die Alternativen aus der Tabelle SCARR ermittelt und in der internen Tabelle LISTVALUES abgelegt wurden, wird hier der Funktionsbaustein F4IF_INT_VALUE_REQUEST aufgerufen, wobei definiert wird, dass die Einträge des Listenfeldes sortiert ausgegeben werden sollen.

Definition der DropDown-Liste über die Standard-Werthilfe

Wird keine der beiden bisher gezeigten Möglichkeiten zum Setzen der DropDown-Liste verwendet, versucht R/3 die Werte der Liste selbständig zu ermitteln. Hierbei kommen die gleichen Mechanismen zum Zuge wie bei der normalen F4-Werthilfe. Dies sind (in der angegebenen Reihenfolge):

- Domänen-Festwerte
- Prüftabellen
- Matchcode-Objekte

Das Feld CARRID der Tabelle SCARR hat keine der genannten Möglichkeiten definiert. In diesem Fall würde die Listbox leer bleiben, wenn nicht eine der zuvor genannten Möglichkeiten zum expliziten Setzen der Alternativen verwendet wird. Das gleichnamige Feld der Tabelle SPFLI jedoch besitzt als Prüftabelle eine

Referenz auf die Tabelle SCARR. Wird auf dem Dynpro demnach statt des Feldes SCARR-CARRID das Feld SPFLI-CARRID verwendet, werden die möglichen Alternativen der DropDown-Liste automatisch ermittelt und die Programmteile der vorigen Abschnitte obsolet. Aus diesem Grund ist im Data Dictionary häufig die Konstellation anzutreffen, dass eine Tabelle sich selbst als Prüftabelle referenziert. Dies mag auf den ersten Blick unnötig und verwirrend aussehen, hilft aber im vorliegenden Beispiel Code zu sparen.

Ebenso könnte zur Eingrenzung der Eingabemöglichkeiten ein Matchcode-Objekt definiert werden. Es wäre auch möglich, die Optionen durch Festwerte an der Domäne des Feldes zu definieren. Im vorliegenden Fall ist dieser Weg allerdings aufgrund der Anwendungssituation nicht möglich.

3.5 Table-Views

Mit Release 3.0 hat SAP das Table-View-Control eingeführt. Es handelt sich hierbei um eine moderne Variante des Step-Loops. Mit Release 4.6 empfiehlt SAP die Table-Views anstatt der bekannten Step-Loops einzusetzen und hat selbst in den meisten Standard-Transaktionen von Step-Loops auf Table-Views umgestellt. Bei den Table-Views handelt es sich um komplexe Dynpro-Elemente, die weitaus mehr Möglichkeiten zulassen, als Step-Loops bisher. So kann z.B. die Breite einzelner Spalten vom Benutzer nach Belieben verändert werden. Bei Bedarf können die aktuellen Darstellungsoptionen eines Table-Views je Benutzer abgespeichert werden und ermöglichen so eine personalisierte Darstellung. Weiterhin kann in Table-Views bei Bedarf horizontal über die Spalten gerollt werden, wobei die Möglichkeit besteht, einzelne Führungsspalten als feste, nicht scrollbare Spalten anzulegen. Horziontal kann dann nur über die nicht-fixen Spalten gerollt werden. Außerdem bieten Table-Views die Möglichkeit, Selektionen sowohl auf Zeilen- als auch auf Spaltenebene durchzuführen.

Die Zeilen einer Table-View können neben statischen Texten und Eingabefeldern auch andere Dialog-Elemente wie z.B. Ankreuzfelder, Auswahlfelder oder auch Schaltflächen enthalten. Einen Nachteil, den Table Views gegenüber Step Loops in der bisherigen Implementierung haben, ist allerdings, dass die Zeilen einer Table-View lediglich eine Bildschirmzeile hoch sein können. Mehrzeilige Listendarstellungen, wie sie mit der Step Loop-Technik möglich sind, können mit Table Views derzeit nicht realisiert werden.

Auch in diesem Abschnitt sollen die Möglichkeiten des neuen Controls anhand der Daten aus dem Beispiel des Abschnitt 3.1 demonstriert werden. Im Table-Control sollen die Flüge einer Fluggesellschaft angezeigt werden. Als Rahmenprogramm soll das in Abschnitt 3.2 begonnene Programm verwendet werden. Das programmierte Table-Control wird auf dem Subscreen 9100 des dort definierten Tab-Strips unterhalb der in Abschnitt 3.4 definierten Drop Down-Liste, über die die gewünschte Fluggesellschaft ausgewählt werden kann, angelegt.

3.5.1 Deklaration einer Table-View

Anders als die bisherigen Dynpro-Elemente in der Dialogprogrammierung, genügt es bei Table Views nicht im Dialogeditor ein entsprechendes Control anzulegen. So wie die Elemente eines Report-Selektionsbildes muss ein Table-View zusätzlich als Datenfeld im Datendeklarationsteil des Modulpools angelegt werden.

Dies geschieht mittels der CONTROLS-Anweisung der ABAP-Sprache in der folgenden Form:

```
CONTROLS <Table-Name> TYPE TABLE-VIEW USING SCREEN <Dynpronr>.
```

Durch diese Anweisung wird – wie z.B. bei der PARAMETERS-Anweisung in Reports – ein Datenbereich deklariert, der den Aufbau der von SAP definierten Struktur CXTAB_CONTROL besitzt (siehe Tabelle 3.14). Die Definition dieser Struktur befindet sich in der Typgruppe CXTAB. Sie beschreibt programmintern die Struktur und die Eigenschaften des Table-Views. Wie in späteren Abschnitten gezeigt wird, können diese Eigenschaften durch das Programm zur Laufzeit verändert werden.

Feldname	Typ	Inhalt
FIXED_COLS	I	Anzahl der fixen (nicht horizontal scrollbaren) Spalten am linken Rand der Tabelle
LINES	I	Anzahl der (dargestellten) Zeilen im Control
TOP_LINE	I	Erste aktuell sichtbare Zeile der Tabelle auf dem Dynpro
CURRENT_LINE	I	Aktuell bearbeitete Listenzeile in der LOOP-Schleife des PAI
LEFT_COL	I	Nummer der ersten dargestellten Seite
LINE_SEL_MODE	I	Zeilenauswahl-Modus des Controls Mögliche Werte: 0 = Zeilen können nicht ausgewählt werden 1 = eine Zeile kann selektiert werden 2 = mehrere Zeilen können selektiert werden
COL_SEL_MODE	I	Spaltenauswahl-Modus des Controls Mögliche Werte: 0 = keine Spalten können markiert werden 1 = eine Spalte kann markiert werden 2 = mehrere Spalten können markiert werden
LINE_SELECTOR	C(1)	Kennzeichen, ob eine Markierungsspalte am linken Rand des Controls dargestellt werden soll

Feldname	Typ	Inhalt
V_SCROLL	C(1)	
H_GRID	C(1)	Kennzeichen, ob die Zeilen optisch durch Linien werden sollen
V_GRID	C(1)	Kennzeichen, ob die Spalten durch Linien optisch getrennt werden sollen
COLS	CXTAB_COLUMN	Tabelle mit Spaltenbeschreibungen (ohne Kopfzeile!)

Tabelle 3.14
Struktur CXTAB_CONTROL für allgemeine Attribute einer Table-View

Die Struktur enthält im Wesentlichen die Felder für die Eigenschaften, die im unteren Bereich des Attributefensters für das Control im Screen-Editor abgelegt sind (siehe Abbildung 3.26). Weiterhin enthält die Struktur eine Tabelle mit den Spalteninformationen des Table-Views. Diese Tabelle, deren Sätze den in Tabelle 3.15 dargestellten Aufbau haben, enthält für jede im Screen-Editor angelegte Spalte des Table-Controls einen Eintrag. Die Struktur der Tabelle 3.15 ist unter dem Namen CXTAB_COLUMN ebenfalls in der Typgruppe CXTAB definiert.

Feldname	Typ	Inhalt
SCREEN	SCREEN	Enthält die Felder der Systemstruktur SCREEN, welche die Attribute der Spalten zur Laufzeit darstellen.
INDEX	I	Spaltennummer innerhalb der Table-View.
SELECTED	ICON-OLENG	Kennzeichen, ob die Zeile markiert ist.
VISLENGTH	ICON-OLENG	Sichtbare Breite der Spalte im Table-View.
INVISIBLE	C(1)	Kennzeichen, ob Spalte nicht angezeigt wird.

Tabelle 3.15
Struktur CXTAB_COLUMN für Spaltenattribute einer Table-View

Zu beachten ist, dass die in der Struktur CXTAB_CONTROL enthaltene Tabelle COLS mit den beschriebenen Spalteninformationen ohne Kopfzeile deklariert wurde. Ein Zugriff auf diese Tabelle muss demnach immer über eine separat definierte Feldleiste erfolgen. Dies wird nochmals im Abschnitt 3.5.5 dargestellt.

3.5.2 Table-Views auf dem Dynpro anlegen

Ein Table-View wird zunächst im Dynpro-Editor angelegt. Hier werden neben seiner Größe auch die darin enthaltenen Spalten und deren Attribute definiert.

Um das Table-View für das Beispiel anzulegen, wird das Dynpro `9100` des Programms `SAPMZFLIGHT` im Dynpro-Editor bearbeitet. Dort befindet sich bereits das in Abschnitt 3.3 definierte Listenfeld mit den Fluggesellschaften. Unterhalb dieses Listenfeldes soll ein Table-Control angelegt werden.

Im grafischen Screen-Editor wird hierzu aus der Werkzeugspalte am linken Fensterrand das »Table-View«-Tool ausgewählt (4. von unten). Im Anschluss daran wird im Dynpro-Bereich des Editors der Table-View mit der Maus in der gewünschten Größe aufgezogen. Die Veränderung der Größe des Controls ist auch im Nachhinein jederzeit über die Markierungen am Rand des Table-Views möglich. Mit einem doppelten Mausklick auf das Element können die Eigenschaften des Controls angezeigt und geändert werden (siehe Abbildung 3.19). Im Beispiel erhält der Table-View den Namen `FLIGHTSVIEW`.

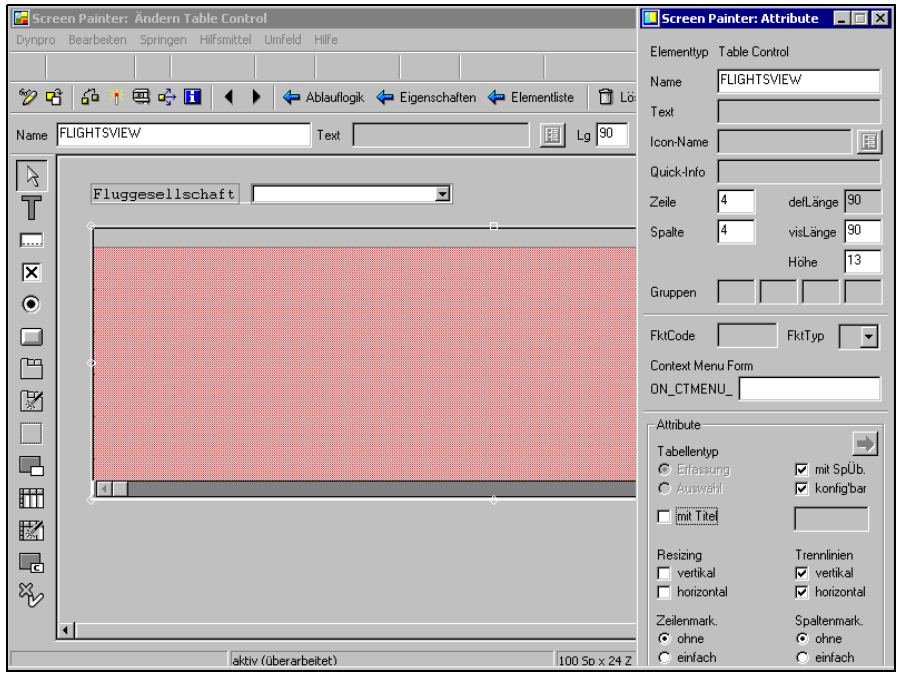

Abbildung 3.19
Anlegen eines Table-Views (© SAP AG)

Weiterhin wird im Beispiel aus der Abbildung festgelegt, dass im Table-View weder Spalten noch Zeilen markierbar sind und sowohl die Zeilen als auch die Spalten durch Linien voneinander getrennt dargestellt werden sollen.

Im nächsten Schritt müssen in das nun bestehende Table-View die Spalten eingefügt werden. Hier sollen einige Spalten der Kopfstruktur der Tabelle `SPLFI` eingefügt werden. Analog zum Vorgehen, wenn für die Tabellenfelder Ein- oder Ausgabefelder auf dem Dynpro erzeugt werden sollen, wird hier über den Menüweg »Springen->Sekundärfenster->Dict./Programmfelder« oder kürzer mit F6 in das in Abbildung 3.20 bereits mit den ausgewählten Feldern dargestellte Fenster verzweigt. Wird das Fenster geschlossen, verwandelt sich der Mauszeiger in eine schemenhafte Darstellung der ausgewählten Felder. Erfolgt der folgende Mausklick auf den Dynpro-Bereich des Editors, werden auf dem Dynpro jeweils das Eingabefeld mit dem dazugehörigen Beschriftungsfeld (wenn ausgewählt) angelegt.

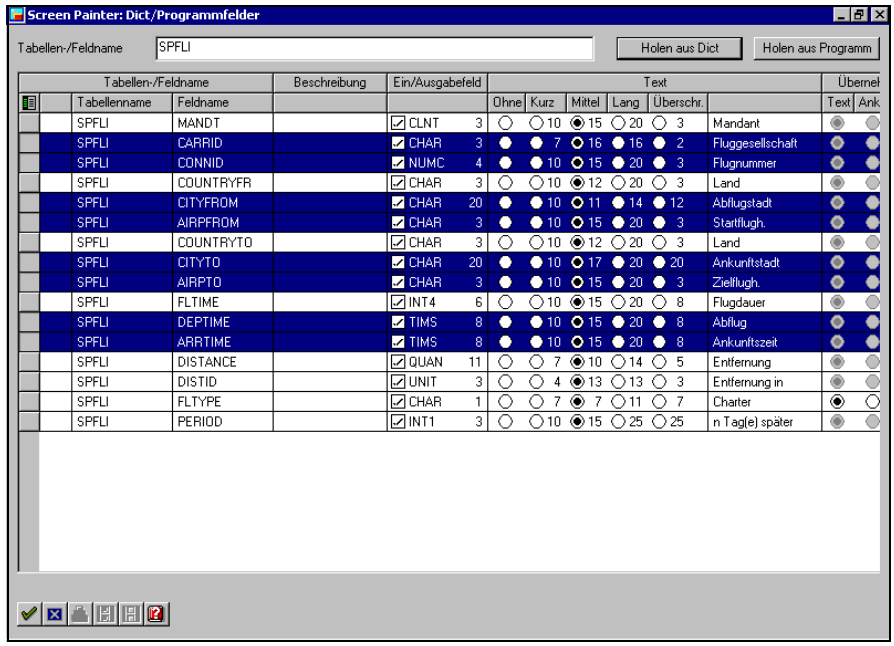

Abbildung 3.20
Auswahl der Dictionary-Felder für den Table-View (© SAP AG)

Erfolgt der Click jedoch auf dem Table-View, wird für jedes ausgewählte Feld eine Spalte im Table-View angelegt. Außerdem wird am Anfang der Spalte eine Spaltenüberschrift, analog zum Beschriftungsfeld bei normalen Ein-/Ausgabefeldern, angelegt (siehe Abbildung 3.21). Jetzt sind alle Spalten als Eingabefelder angelegt. Die Spaltenüberschriften entsprechen den Beschriftungen der Felder, wenn kein Table-View verwendet würde. Die standardmässige Breite der Spal-

ten im Table-View entspricht der definierten Breite der Spalten im Data Dictionary.

Über die Feldattribute der Spalten werden nacheinander alle Spalten als nicht eingabebereit definiert und weiterhin die erste Bildschirm-Gruppe auf den Wert »TAB« gesetzt. Letzteres wird später in diesem Kapitel von Bedeutung sein.

Somit ist das Table-View im Dynpro-Editor vollständig angelegt. Der grafische Editor kann nun verlassen werden. Als nächster Schritt folgt, den Table-View in der Ablauflogik des Dynpros zu verankern und dafür zu sorgen, dass die entsprechenden Informationen zur Laufzeit im Dynpro erscheinen.

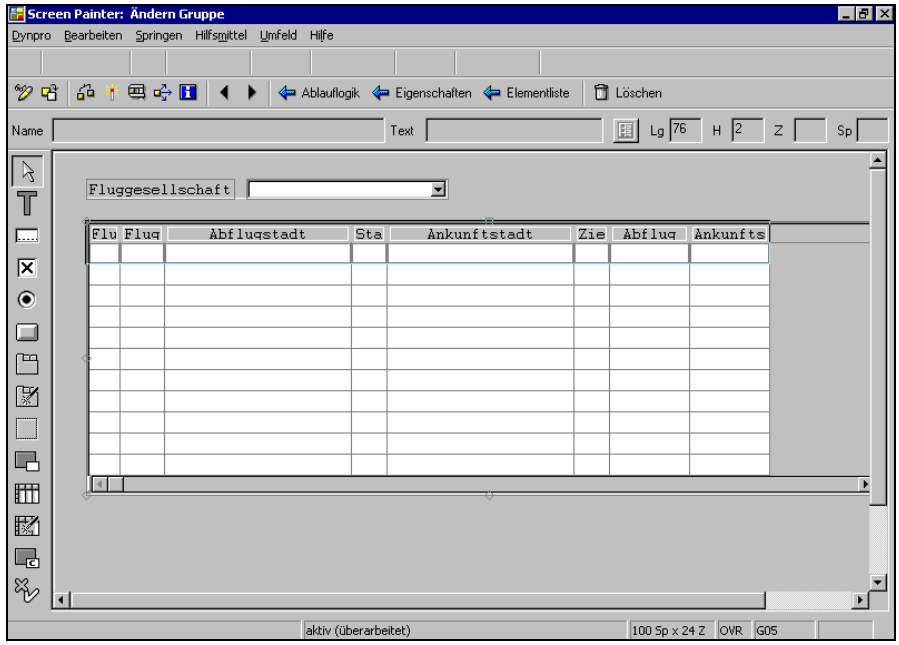

Abbildung 3.21
Table-View mit Spalten im grafischen Screen-Editor (© SAP AG)

3.5.3 Table-Views in der Ablauflogik

In der Ablauflogik verhalten sich Table-Views ähnlich wie Step Loops. Auch für Table-Views muss sowohl im PBO als auch im PAI eine Loop-Schleife angelegt werden. Anders als bei Step-Loops muss bei den Table-Views jedoch das Control, für welches die LOOP-Schleife codiert werden, explizit angegeben werden. Um das vertikale Blättern in den Sätzen des Table-Views zu ermöglichen, muss weiterhin – ebenfalls analog zum Step-Loop – ein Feld definiert, werden, welches während der LOOP-Schleife die Nummer des aktuell bearbeiteten Satzes und

außerhalb der LOOP-Schleife die Zeilennummer des ersten dargestellten Satzes enthält.

```
PROCESS-BEFORE-OUTPUT.
...
  LOOP AT SPFLI_TAB WITH CONTROL FLIGHTSVIEW CURSOR SPFLI_CURSOR.
    MODULE SHOW_FLIGHT_9100.
  ENDLOOP.
...
PROCESS-AFTER-INPUT.
...
  LOOP AT SPFLI_TAB.
  ENDLOOP.
...
```

Im Module SHOW_FLIGHT_9100 werden lediglich die aktuellen Daten aus der Feldleiste der Tabelle SPFLI_TAB in die Feldleiste der Tabelle SPFLI, deren Felder als Spalten im Table-View definiert sind, übertragen.

```
MODULE SHOW_FLIGHT_9100 OUTPUT.
  MOVE-CORRESPONDING SPFLI_TAB TO SPFLI.
ENDMODULE.                              " SHOW_FLIGHT_9100  OUTPUT
```

Dies wurde so realisiert, da nur bei direkter Verwendung von Dictionary-Strukturen als Spalten der Table-View die Werthilfen und Matchcode-Hilfen im Dynpro zur Verfügung stehen. Bei Verwendung der Spalten der Tabelle SPFLI_TAB direkt müssten diese explizit realisiert werden.

Da der Table-View zunächst lediglich dazu genutzt wird, Informationen anzuzeigen, also keine Daten im Table-View verändert werden können, kann die LOOP-Schleife im PAI leer bleiben.

Nachdem an dieser Stelle die Schritte

- Deklaration des Table-Controls,
- anlegen des Table-Views im Dynpro-Editor und
- implementieren der Ablauflogik

durchgeführt wurden, stellt sich sich das Table-View zur Laufzeit wie in Abbildung 3.32 gezeigt dar. Beim Betreten des Dynpros ist noch keine Fluggesellschaft ausgewählt. Dementsprechend ist das Table-Control unterhalb des Listenfeldes initial leer. Wird eine Fluggesellschaft ausgewählt, zu der Flüge gespeichert sind (im Beispiel »ALITALIA«), werden die entsprechenden Flüge im Table-Control angezeigt. Dies geschieht, da das Listenfeld einen Funktionscode definiert hat und somit unmittelbar nach Verändern der ausgewählten Fluggesellschaft ein PAI-Ereignis ausgelöst wird, in welchem die interne Tabelle SPFLI_TAB mit den Flügen der gewählten Airline gefüllt wird. Im darauf folgen-

den PBO des Dynpros wird der aktuelle Inhalt der Tabelle SPFLI_TAB in die Zeilen des Table-Views übertragen.

Wie gewünscht sind die einzelnen Spalten des Table-Views nicht eingabebereit. Allerdings enthalten die Spalten Abflugzeit sowie Ankunftszeit auch in den Zeilen einen Wert, in denen kein Flug angezeigt wird. Wie dies zu verhindern ist, wird in Abschnitt 3.5.5 beschrieben.

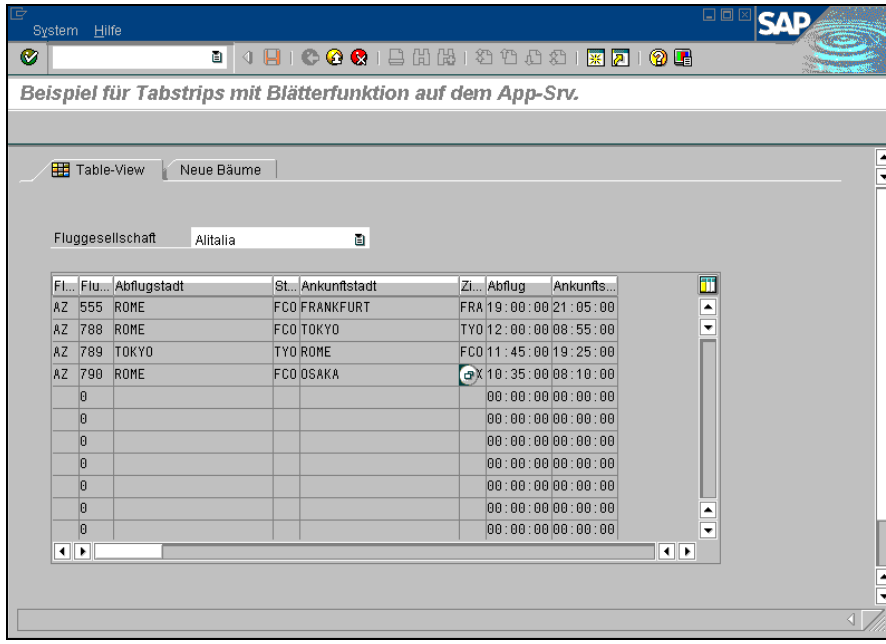

Abbildung 3.22
Darstellung der Table-View ohne dynamische Veränderungen (© SAP AG)

3.5.4 Table-Controls benutzerspezifisch individualisieren

Zu Laufzeit können die Spaltenbreiten der einzelnen Spalten sowie die Reihenfolge der Spalten eines Table-Controls vom Benutzer verändert werden. Sobald das Programm erneut betreten wird, sind die individuellen Einstellungen des Benutzers jedoch wieder verloren und das Table-Control wird in der, im Dynpro-Editior bzw. Programm festgelegten Form angezeigt.

Über eine Option in den Eigenschaften von Table-Views kann jedoch festgelegt werden, dass jeder Benutzer individuell Anzeigevarianten zu dem Table-View anlegen kann. Wenn der Benutzer im Anschluss daran das Dynpro erneut betritt, wird das Table-View in der, zuvor vom Benutzer selbst vorgegebenen Form angezeigt. Der Benutzer kann auch mehrere Anzeigevarianten zu einem Table-View anlegen und im Anschluss daran schnell in eine andere Anzeige-

variante wechseln. Um diese Funktionalität zu unterstützen, ist keinerlei Programmierung notwendig. Die Steuerung erfolgt lediglich über ein Optionsfeld in den Eigenschaften des Table-Views.

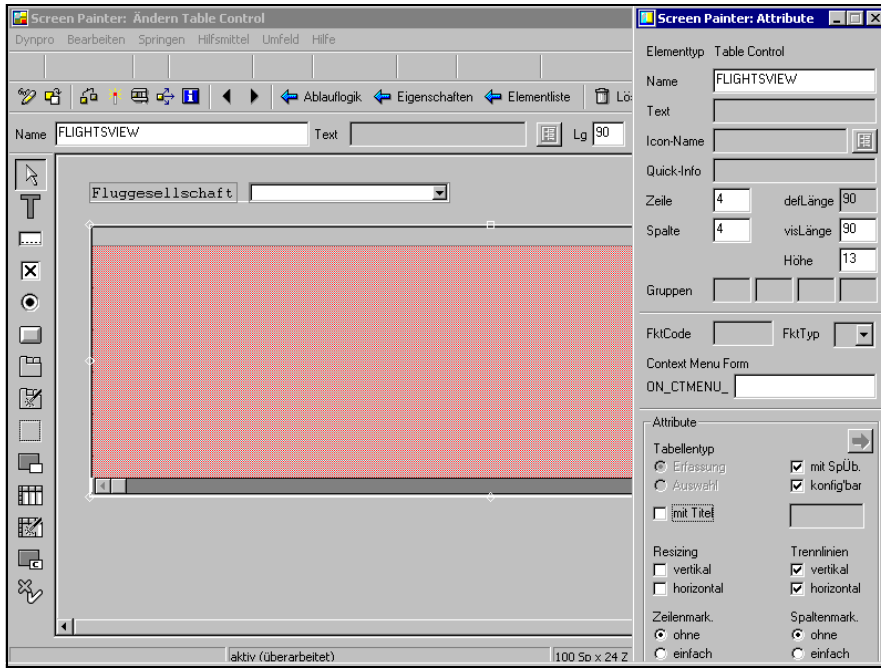

Abbildung 3.23 Eigenschaften eines Table-Views (© SAP AG)

Diese Funktionalität wird aktiviert, wenn in den Eigenschaften des Table-Views die Option Konfigurierbar eingeschaltet wird. Dieser Schalter ist im unteren Bereich der Eigenschaften im Rahmen »Attribute« abgelegt. Wird die Option angewählt, erscheint zur Laufzeit in der rechten oberen Ecke des Table-Views ein Symbol (siehe Abbildung 3.24), über die der Konfigurationsdialog, wie in Abbildung 3.35 dargestellt, aufgerufen wird.

Die Einstellungen, die hier gespeichert werden können, beziehen sich immer auf den aktuellen Zustand des Controls. Mit einem Namen versehen können die aktuellen Einstellungen permanent gespeichert werden. Diese Einstellungen sind benutzerspezifisch, d.h. jeder Benutzer kann eigene Varianten des Table-Controls anlegen. Gespeichert werden die Breite sowie die Reihenfolge der Spalten.

Benutzer, die über die Berechtigung `S_ADMI_FCD` verfügen, können von diesem Dynpro aus über die Schaltfläche »Administrator« in ein weiteres Fenster verzweigen, in dem wesentlich weitreichendere Möglichkeiten zur Beeinflussung der Table-View-Darstellung möglich sind (siehe Abbildung 3.23).

Abbildung 3.24
Tabelleneinstellungen für Table-Views (© SAP AG)

In diesem Dynpro können einzelne Spalten ausgeblendet und die Einstellungen für horizontale sowie vertikale Linien zum Trennen von Zeilen bzw. Spalten geändert werden. Weiterhin kann die Anzahl der festen Führungsspalten, d.h. die Anzahl der Spalten, die beim horizontalen Rollen nicht mitgerollt werden, verändert werden. Mit der Schaltfläche »Aktivieren« wird aus den aktuellen Einstellungen des Table-Views, eine für alle Benutzer des aktuellen Mandanten sichtbare Systemvariante erzeugt. Diese Variante definiert neben den in diesem Dynpro festgelegten Werten auch die Breite und Reihenfolge der im Table-View enthaltenden Spalten.

Für jedes Table-View im System kann maximal eine solche Systemvariante erzeugt werden. Aktivieren verschiedene Benutzer eine Systemvariante, wird die zuvor gesicherte Variante mit den aktuellen Einstellungen überschrieben. Ob bereits eine Systemvariante für das aktuelle Table-View exisitert, kann am Ankreuzfeld in der linken unteren Ecke des in Abbildung 3.25 dargestellten Dynpros ersehen werden.

Eine solche Systemvariante kann gelöscht werden, indem in diesem Fenster die Schaltfläche »Löschen« betätigt wird.

Da die Einstellungen, die in diesem Fenster getätigt werden, Einfluss auf alle Benutzer des Mandanten haben, sollte mit der Vergabe der Berechtigung `S_ADMI_FCD` sparsam umgegangen werden. Für die meisten Benutzer genügt es ohnehin, die Einstellungen bezüglich der Breite und Reihenfolge der Tabellenfelder zu verändern.

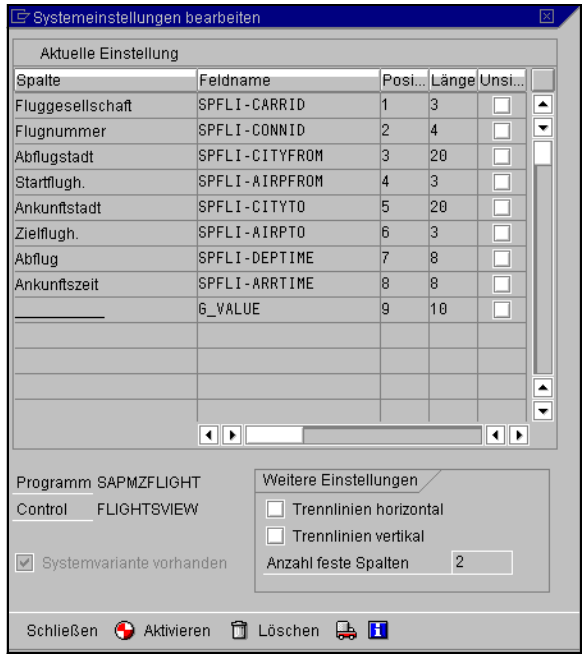

Abbildung 3.25
Administrationseinstellungen zu einem Table-View (© SAP AG)

3.5.5 Dynamische Veränderungen am Table-View

Die bisherigen Ausführungen zeigen, wie die statischen Eigenschaften von Table-Views die Darstellung zur Laufzeit beeinflussen. In diesem Abschnitt sollen verschiedene Möglichkeiten gezeigt werden, zur Laufzeit auf die Attribute des Table-Views zuzugreifen und somit dynamische Veränderungen am Table-View vorzunehmen.

Zellenattribute ändern

Zunächst soll auf den am Ende von Abschnitt 3.5.3 festgestellten Missstand, dass an sich leere Zeilen mit den Initialwerten der verwendeten Datentypen dargestellt werden, eingegangen werden. Während sich dies bei Zeichenfeldern nicht auswirkt, da der Initialwert für diese leer ist, hat dieser Umstand bei numerischen, Datums- oder – wie im Beispiel – Zeitfeldern sehr störende Auswirkungen. Die einfachste Möglichkeit, diesem Umstand zu begegnen ist, die Zellen, die keine Werte enthalten, auszublenden.

Die Elementeliste des Dynpros enthält sowohl für den Table-View an sich, als auch sämtliche darin enthaltene Spalten Einträge. Wie bei Step-Loops können

die Attribute der Spalten aber nur innerhalb der LOOP-Schleife in der Ablauflogik des Dynpros verändert werden, wobei die getätigten Einstellungen nur für die jeweilige Zelle, adressiert durch die aktuelle LOOP-Zeile und den eindeutigen Spaltennamen, verändert werden. Im Beispiel bedeutet dies, dass die Ausgabe sämtlicher Spalten einer leeren LOOP-Zeile unterdrückt werden soll.

Hierzu wird in der Ablauflogik des Dynpros 9100 innerhalb der LOOP-Schleife des PBO das neue Module CLOSE_FIELDS aufgerufen.

```
...
  LOOP AT SPFLI_TAB WITH CONTROL FLIGHTSVIEW CURSOR SPFLI_CURSOR.
    MODULE SHOW_FLIGHT_9100.
    MODULE CLOSE_FIELDS.
  ENDLOOP.
...
```

Die Reihenfolge der Modul-Aufrufe spielt hierbei keine Rolle. Die Implementierung des Moduls besteht aus einer LOOP-Schleife über die Systemtabelle SCREEN. Hierbei kommt die Tatsache, dass die Spalten der Table-View allesamt in der Bildgruppe 1 den Wert »TAB« enthalten, sehr gelegen. Dadurch kann auf die Auswertung von Spaltennamen verzichtet werden. Ein Anfügen von Spalten in der Table-View ist weniger fehleranfällig, da kein Programmcode angepasst werden muss, um das Verhalten der neuen Spalten dem der bisherigen Spalten anzupassen, sondern lediglich eine Eigenschaft der neuen Spalte gesetzt werden muss.

```
MODULE CLOSE_FIELDS OUTPUT.
  LOOP AT SCREEN.
    IF SCREEN-GROUP1 = 'TAB'.
      IF SPFLI_TAB IS INITIAL.
        SCREEN-INPUT = 0.
        SCREEN-ACTIVE = 0.
        SCREEN-INVISIBLE = 1.
      ELSE.
        SCREEN-INPUT = 0.
      ENDIF.
      MODIFY SCREEN.
    ENDIF.
  ENDLOOP.
ENDMODULE.                              " close_fields  OUTPUT
```

In der Schleife wird geprüft, ob der aktuelle Tabellensatz initial ist, und – falls ja – werden alle Felder, die den entsprechenden Wert im Attributfeld »SCREEN-GROUP1« enthalten, unsichtbar gesetzt (SCREEN-INVISIBLE).

Durch diese Veränderung entspricht die Anzeige des Table-Views (siehe Abbildung 3.26) wesentlich besser den Erwartungen, die man an dieses Control stellt.

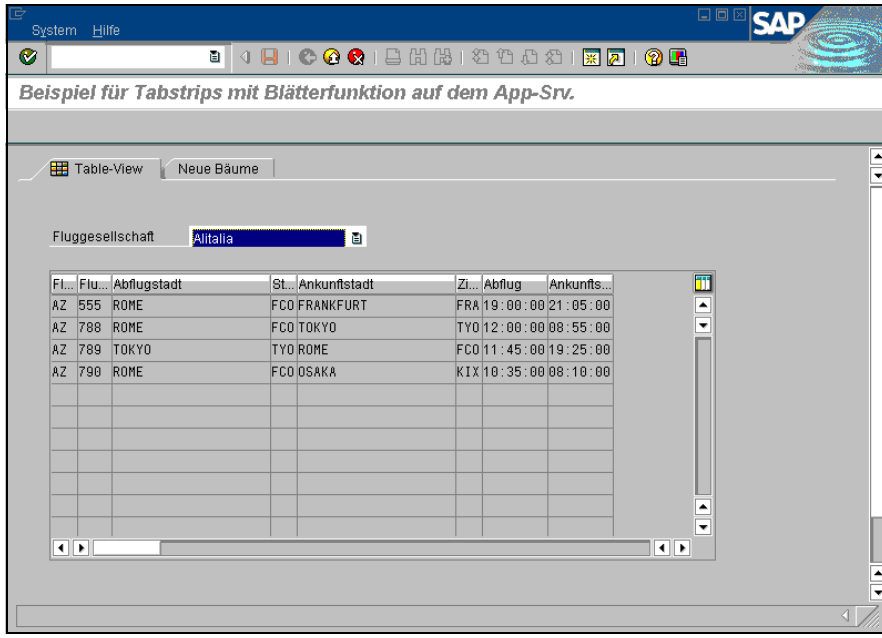

Abbildung 3.26 Table-View ohne Anzeige leerer Zeilen (© SAP AG)

Es ist wichtig, nochmals darauf hinzuweisen, dass durch dieses Verfahren lediglich Zellen der Table-View ausgeblendet werden. Es können hiermit keine ganzen Spalten versteckt werden.

Spalten zur Laufzeit ausblenden

Anders als die Sichtbarkeit von Zellen wird die Sichtbarkeit ganzer Spalten nicht über die Elementliste des Dynpros gesteuert. Hier kommt die über die `CONTROLS`-Anweisung definierte Datenstruktur des Table-Views zum tragen.

Wie in Tabelle 3.18 gezeigt, enthält die Struktur `CXTAB_CONTROL` in Feld `COLS` eine Tabelle, die für jede Spalte der Table-View einen Eintrag mit der in Tabelle 3.20 dargestellten Struktur enthält. Die Struktur dieser Spaltentabelle enthält unter anderem die Felder der System-Struktur `SCREEN`. Weiterhin definiert sie ein Feld mit dem Namen `INVISIBLE`, mit dem die gesamte Spalte ausgeblendet werden kann.

Da es sich bei diesem Datenmodell um ein statisches Modell handelt, erfolgen Manipulationen nicht innerhalb der `LOOP`-Schleife der Ablauflogik, sondern an beliebiger anderer Stelle außerhalb der `LOOP`-Schleife. Bei Veränderungen der Strukturen ist zu beachten, dass zur Laufzeit weder Spalten zum Table-View hinzugefügt, noch Spalten aus dem Table-View gelöscht werden können. Lediglich die Attribute der im Screen-Editor angelegten Spalten können dynamisch

verändert werden. Steht die Anzahl der anzuzeigenden Spalten zum Zeitpunkt der Programmierung nicht endgültig fest, bleibt demnach keine andere Wahl, als die maximale Anzahl Spalten im Control anzulegen und dynamisch die nicht benötigten Spalten unsichtbar zu machen.

Zur Demonstration dieser Möglichkeit soll im Beispielprogramm die Spalte CARRID in der Ablauflogik des Dynpros auf unsichtbar gesetzt werden. Hierzu wird im PBO an beliebiger Stelle außerhalb der LOOP-Schleife der Aufruf für das Module HIDE_CARRID eingefügt.

```
...
MODULE HIDE_CARRID.
...
```

Da die Spaltentabelle COLS in der Struktur CXTAB_CONTROL ohne Kopfzeile deklariert wurde, muss zur Bearbeitung der Tabelle eine lokale Variable verwendet werden. Da in Modulen keine lokalen Variablen angelegt werden können, wird die eigentliche Implementierung der Funktionalität in eine gleichnamige FORM-Routine HIDE_CARRID, welche die Dynpro-Struktur des Table-Views als Parameter erhält, ausgelagert.

```
MODULE HIDE_CARRID.
  PERFORM HIDE_CARRID CHANGING FLIGHTSVIEW.
ENDMODULE.
```

Die Implementierung dieser FORM-Routine besteht lediglich darin, aus der Tabelle COLS der Struktur FLIGHTSVIEW den Eintrag für die Spalte SPFLI-CARRID zu suchen und das INVISIBLE-Attribut dieses Eintrags zu setzen. Dementsprechend einfach ist der Code dieser Routine aufgebaut.

```
FORM HIDE_CARRID CHANGING P_TABLE TYPE CXTAB_CONTROL.
  DATA: L_COL TYPE CXTAB_COLUMN.

  READ TABLE P_TABLE-COLS
     INTO L_COL
     WITH KEY SCREEN-NAME = 'SPFLI-CARRID'.
  IF SY-SUBRC EQ 0.
    L_COL-INVISIBLE = 'X'.
    MODIFY P_TABLE-COLS
      FROM L_COL
     INDEX SY-TABIX.
  ENDIF.
ENDFORM.
```

Manipulationen, die den Table-View an sich oder die darin enthaltenen Spalten insgesamt betreffen, müssen immer über diese Struktur erfolgen, während dynamische Veränderungen an einzelnen Zellen über die Systemtabelle SCREEN innerhalb der LOOP-Schleife in der Ablauflogik erfolgen müssen.

Spaltenüberschriften ändern

In Fällen, in denen die Anzahl der Spalten des Table-Views nicht von vornherein bestimmt werden können, steht häufig zum Zeitpunkt der Programmerstellung auch noch nicht endgültig fest, welche Daten in den variablen Spalten angezeigt werden sollen und welche Überschriften die jeweiligen Spalten haben.

Wenn die Spalten des Table-Views mit der in Abschnitt 3.5.2 beschriebenen Methode aus Data Dictionary-Feldern angelegt werden, sind die Spaltenüberschriften als statische Texte angelegt. Der Text dieser Elemente kann zur Laufzeit nicht verändert werden. Besteht die Anforderung, zur Laufzeit die Spaltenüberschrift zu ändern, muss die standardmäßig angelegte Spaltenüberschrift durch ein vom Programmierer selbst definiertes Feld ersetzt werden. Hierzu muss ein globales Datenfeld angelegt werden, welches die gewünschte Überschrift enthält.

```
DATA: G_TITLE(20) TYPE C.
```

Im Anschluss daran wird im Screen-Editor das betreffende Titelfeld gelöscht und über den Dialog »Dict./Programmfelder« ein Feld für die Variable `G_TITLE` angelegt. Um das gewünschte Feld als Titelfeld anzulegen, muss nach Beendigung des Dialogs an die leere Stelle, an der zuvor der standardmäßig angelegte Titel stand, geklickt werden.

Über eine Zuweisung an das Feld `G_TITLE` im PBO des Dynpros kann somit der Titel der betreffenden Spalte dynamisch geändert werden. Häufig tritt dies in Zusammenhang mit Spalten auf, die ebenfalls nicht aus dem Data Dictionary entnommen werden, sondern als globale Programmvariable im Modulpool angelegt wurden. Abbildung 3.41 zeigt schließlich den Table-View mit einer zusätzlichen Spalte am Ende der Liste mit der Überschrift »Titel«. Hierzu wurde im TOP-Include zusätzlich das Feld `G_VALUE` als Zeichenfeld mit der Länge 20 deklariert. Innerhalb der `LOOP`-Schleife im PBO des Dynpros wird ein neues Modul `SET_VALUE_9100` aufgerufen, in dem lediglich eine Zuweisung des Feldes `SPFLI_CURSOR` an die Variable `G_VALUE` erfolgt. Weiterhin wurde die Deklaration des Feldes `G_TITLE` dahingehend ergänzt, dass der Initialwert »Titel« bereits bei der Deklaration zugewiesen wird.

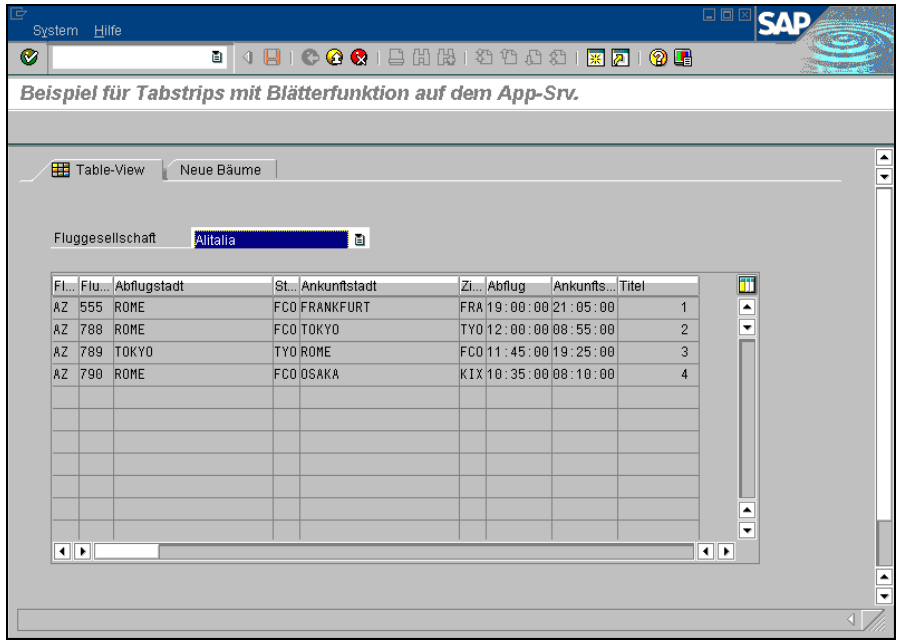

Abbildung 3.27
Table-View mit zusätzlicher Spalte mit Referenz auf eine Programmvariable (© SAP AG)

4 Arbeiten mit Texten

Texte spielen innerhalb von R/3 eine große Rolle. Neben den reinen betriebswirtschaftlichen Informationen können an den verschiedensten Anwendungsobjekten zusätzliche Texte für den Benutzer oder den Ausdruck abgelegt werden. Diese Informationen werden in einem gesonderten Bereich der SAP-Datenbank in Form so genannter Textobjekte abgelegt. Zur Verwaltung dieser Textobjekte sind im Data Dictionary mehrere Strukturen definiert, über die der Zugriff auf die Textobjekte innerhalb des Systems möglich sind.

Diese Textobjekte bieten die Möglichkeit, formatierten Text zu speichern. Man spricht hierbei auch von Textobjekten aus dem Bereich von SAPScript, der Textverarbeitungskomponente des R/3-Systems. SAPScript-Textobjekte, die nicht mit Bezug zu einem Anwendungsobjekt abgelegt werden, werden Standard-Texte genannt. SAP bietet für diese Texte eine ganze Reihe Funktionen, um Textobjekte zu speichern, lesen oder zu bearbeiten. Diese und noch mehr Funktionen zur Manipulation von Textobjekten sind in der Funktionsgruppe `STXD` unter dem Namen »SAPScript Program Interface 1« zusammengefasst. Hier finden sich weitere Funktionsbausteine, um Textobjekte als ganzes bearbeiten (z.B. löschen, umbenennen usw.). Die Funktionsgruppe `STXE` hat den Namen »SAPScript Program Interface 2« und beinhaltet Funktionen, um Symbole innerhalb des Textes des Textobjekts durch Werte zu ersetzten. Eine ausführliche Beschreibung aller Funktionen von SAPScript würde den Rahmen dieses Buches jedoch sprengen. Lediglich die Grundfunktionen, die notwendig sind, um Texte zu lesen, zu speichern und zu bearbeiten, werden hier vorgestellt.

Auf der anderen Seite existieren Texte, die keine Formatierungen beinhalten. Es handelt sich hierbei um eine Aneinanderreihung von einfachen Textzeilen ohne jede Formatierungsmöglichkeit. SAP bietet für diese Art Texte keine standardisierte Ablagemöglichkeit, wie für Standard-Texte oder Textobjekte, gleichwohl existieren Werkzeuge, um diese Texte zu bearbeiten. ABAP-Programme selbst

sind ein Beispiel für diese Texte. Formatierungen gleich welcher Art sind in Programmtexten unüblich und werden auch nicht vorgenommen.

Der Schwerpunkt dieses Kapitels liegt auf der Arbeit mit Textobjekten. Es wird gezeigt, wie vom Benutzer Textobjekte im Dialog bearbeitet werden können. Weiterhin wird dargestellt, wie Textobjekte in der Datenbank gespeichert oder aus der Datenbank gelesen werden können. Eine Zusammenfassung der Funktionsbausteine, die hierfür verwendet werden, findet sich am Ende dieses Kapitels. Für die Speicherung nicht-formatierter Texte stehen in R/3 keine derartigen Strukturen bereit, wohl aber zum Bearbeiten der Texte in Dialogtransaktionen. Der Programmierer ist in der Regel selbst für die Datenhaltung und –speicherung verantwortlich.

4.1 Textobjekte mit Formatierungen

4.1.1 Aufbau von Textobjekten

Textobjekte setzen sich aus zwei Teilen zusammen. Die Textzeilen selber werden in einer internen Tabelle abgelegt, deren Struktur zwei Felder umfasst. Diese Struktur ist im Data Dictionary unter dem Namen `TLINE` abgelegt (siehe Tabelle 4.1).

Feldname	Datentyp	Bedeutung
TDFORMAT	CHAR 2	Formatierungsspalte der Textzeile
TDLINE	CHAR 132	Datenbereich (Text) der Zeile

Tabelle 4.1
Aufbau der Struktur `TLINE`

In der Spalte `TDLINE` der Struktur ist die Textinformation der Zeile abgelegt. Hierfür stehen maximal 132 Zeichen zur Verfügung. Die Textzeile kann – neben dem eigentlichen Text – auch Zeichenformatierungen, wie im – dem Textobjekt zugeordneten - Textstil definiert, enthalten. Absatzformatierungen hingegen werden nicht im Feld `TDLINE` abgelegt sondern in der eigens hierfür vorhandenen Spalte `TDFORMAT`. Auch die in dieser Spalte verwendeten Formatierungen müssen zuvor im verwendeten Textstil definiert werden. Der Textstil in SAP-Textobjekten ist vergleichbar mit den Formatierungen in der Formatvorlage von Office-Anwendungen. Während diese jedoch in der Regel auch die Seitenformatierungen beinhalten, werden diese innerhalb von SAPScript in den Formularen definiert. Einem Textobjekt liegt neben dem Stil also immer auch ein Formular zugrunde.

Neben den eigentlichen Textzeilen gehört zu jedem Textobjekt eine Kopfstruktur, in der die Verwaltungsinformationen zum Textobjekt abgelegt sind. Diese Struktur ist ebenfalls im Data Dictionary definiert und unter dem Namen THEAD abgelegt (siehe Tabelle 4.2).

Feldname	Datentyp	Bedeutung
TDOBJECT	CHAR 10	Objektname des Textobjektes
TDNAME	CHAR 70	Name des Textobjektes
TDID	CHAR 4	Text-ID des Textes
TDSPRAS	LANG	Sprachkennzeiche
TDTITLE	CHAR 50	Titel des Textes
TDFORM	CHAR 16	Name des SAPScript-Formulars, das dem Text zugrunde liegt
TDSTYLE	CHAR 8	Name des SAPScript-Stils, der dem Textobjekt zugrunde liegt
TDVERSION	NUMC 5	Versionsnummer des Textobjektes
TDFUSER	CHAR 12	Benutzer, der den Text erstellt hat
TDFRELES	CHAR 4	Release der Texterstellung
TDFDATE	DATS 8	Datum der Texterstellung
TDFTIME	TIMS 6	Uhrzeit der Texterstellung
TDLUSER	CHAR 12	Benutzer, der den Text zuletzt geändert hat
TDLRELES	CHAR 4	Release der letzten Textänderung
TDLDATE	DATS 8	Datum der letzten Änderung
TDLTIME	TIMS 6	Uhrzeit der letzten Änderung
TDLINESIZE	NUMC 3	Zeilenbreite der Textzeilen
TDTXTLINES	NUMC 5	Aktuelle Anzahl der Textzeilen in der Zeilentabelle
TDHYPHENAT	CHAR 1	Kennzeichen, ob Silbentrennung aktiv ist
TDOSPRAS	LANG 2	Originalsprache des Textobjekts
TDTRANSTAT	NUMC 1	Status der Übersetzung
TDMACODE1	CHAR 16	Kurztitel 1 zum Textobjekt
TDMACODE2	CHAR 16	Kurztitel 2 zum Textobjekt

Feldname	Datentyp	Bedeutung
TDREFOBJ	CHAR 10	Objekt des referenzierten Textobjektes
TDREFNAME	CHAR 70	Name des referenzierten Textobjektes
TDREFID	CHAR 4	Text-ID des referenzierten Textobjektes
TDTEXTTYPE	CHAR 6	Format des Textes (SAPScript)
TDCOMPRESS	CHAR 1	Kennzeichen, ob Text komprimiert ist (SAPScript)
MANDT	CLNT 3	Mandant
TDOCLASS	CHAR 4	Objektklasse des Textobjektes (SAPScript)

Tabelle 4.2
Aufbau der Struktur THEAD

Über die ersten vier Felder dieser Struktur werden Textobjekte in R/3 eindeutig identifiziert. Im Feld TDOBJECT wird das Anwendungsobjekt zu dem das Textobjekt gehört abgelegt. Standardtexte enthalten hier beispielsweise den Wert »TEXT«, Langtexte zu Aufträgen den Wert »AUFK«. Die möglichen Werte für das Feld TDOBJECT sind in der Tabelle TTXOB im R/3-System hinterlegt.

Das Feld TDNAME enthält den Namen des Textobjektes. Bei Standardtexten kann dieser vom Benutzer selbst definiert werden. Bei Texten, die zu anderen Anwendungsobjekten gehören, wird dieser Name vom System generiert und enthält in der Regel den Primärschlüssel des Anwendungsobjektes.

Das dritte Feld, über das Textobjekte unterschieden werden, ist das Feld TDID. Es ermöglicht eine weitere Kategorisierung. Die möglichen Werte für dieses Feld sind – abhängig vom Inhalt des Feldes TDOBJECT – in der Tabelle TTXID definiert.

Textobjekte werden im System sprachabhängig definiert. Hierzu wird das vierte Schlüsselfeld der Header-Struktur verwendet. Im Feld TDSPRAS wird hierzu der Sprachschlüssel des Textobjektes abgelegt.

Die weiteren Felder der Struktur sind die Verwaltungsfelder zum Textobjekt. Dem Feld TDTITLE, in dem der Titel des Textes abgelegt wird, folgen die Felder TDFORM und TDSTYLE, die den Namen des SAPScript-Formulars und des SAPScript-Textstils enthalten. Im Textstil werden sowohl die Absatz- als auch die Zeichenformate, die im Textobjekt verwendet werden können, definiert. Im Formular wird weiterhin der Seitenaufbau, mit dem das Textobjekt dargestellt werden soll, festgelegt. Genauere Informationen zu den Konzepten von SAPScript kann der Dokumentation zum System R/3 entnommen werden.

Im Feld TDVERSION wird die Versionsnummer des Textobjektes abgelegt. Nach ihrer Erzeugung ist in diesem Feld der Wert 1 enthalten. Mit jeder Änderung des Textobjektes wird der Inhalt dieses Feldes um eins erhöht. Daran schließen sich

vier Felder an, um den Benutzer sowie das Datum und die Uhrzeit der Texterstellung festzuhalten (TDFUSER, TDFDATE, TDFTIME). Weiterhin wird der R/3-Releasestand der Texterstellung im Feld TDFRELES abgelegt (die R/3-Version 4.6C würde durch die Zeichenkette »46C« dargestellt). Die gleichen vier Felder, mit dem einzigen Unterschied, dass das »F« im Feldnamen durch ein »L« ersetzt wurde, folgen im Anschluss, um die entsprechenden Informationen für die letzte Änderung des Textobjektes festzuhalten.

Zuletzt von Bedeutung für den Programmierer sind die Felder TDLINESIZE sowie TDTXTLINES. Im ersteren ist die Zeilenbreite im Editor für jede Zeile des Textes abgelegt. Das Feld TDLINE der Struktur TLINE ist immer 132 Stellen breit. In vielen Fällen erlaubt die Anwendung allerdings nicht so viele Stellen. Aus diesem Grund kann die Zeilenbreite über das Feld TDLINESIZE auf die – aus Anwendungssicht gewünschte – Breite gesetzt werden. Beim Speichern des Textobjektes wird vom System im Feld TDTXTLINES die aktuelle Anzahl Zeilen im Textobjekt abgelegt.

Im Feld TDHYPHENAT wird ein Kennzeichen für den Editor geführt, ob die Silbentrennung im Textobjekt aktiv sein soll (»X«) oder nicht (Space).

Das Feld TDOSPRAS enthält das Sprachkürzel der Sprache, in der das Textobjekt ursprünglich angelegt wurde. Falls es sich bei dem Textobjekt um eine Übersetzung eines bestehenden Textes handelt, enthält dieses Feld das Sprachkennzeichen der Originalsprache und das Feld TDSPRAS das Sprachkürzel für die aktuelle Sprache des Textobjekts. Der Status der Übersetzung ist im Feld TDTRANSTAT abgelegt.

Die Felder TDMACODE1 sowie TDMACODE2 enthalten weitere Kurztitel, die zum jeweiligen Textobjekt abgelegt bzw. angezeigt werden sollen.

Ein Textobjekt in R/3 kann ein anderes Textobjekt referenzieren. Diese Referenz wird in den Feldern TDREFOBJ, TDREFNAME, TDREFID der Kopfstruktur abgelegt. Die Sprache des referenzierten Textobjektes ist immer die Sprache des referierenden Objektes, d.h. der Inhalt des Feldes TDSPRAS.

Schließlich befinden sich in der Struktur THEAD noch der Mandant, in dem das Textobjekt existiert sowie drei weitere Felder, die für die Verarbeitung des Textobjektes in SAPScript verwendet werden.

Im SAPScript-Editor können die Kopfinformationen zum jeweiligen Textobjekt angezeigt werden (siehe Abbildung 4.1). Das dargestellte Dynpro enthält die meisten Felder der Struktur THEAD. Dem Dynpro kann auch entnommen werden, dass nicht alle Felder zwangsweise Eingaben enthalten müssen. Im dargestellten Beispiel wird lediglich ein Formular zum Textobjekt definiert, nicht jedoch ein expliziter Textstil.

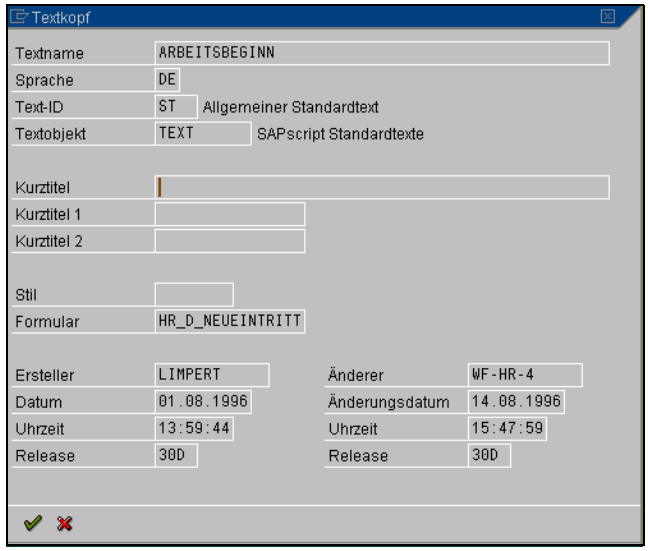

Abbildung 4.1
Textkopf eines SAP-Standardtextes (© SAP AG)

4.1.2 Lesen von Textobjekten

Zum Lesen von Textobjekten aus der Datenbank stehen die Funktionsbausteine `READ_TEXT` sowie `READ_STDTEXT` zur Verfügung. Während bei `READ_TEXT` alle Parameter, die das Textobjekt beschreiben, übergeben werden müssen, nimmt der Baustein `READ_STDTEXT` einige Werte an und bietet zusätzlich Funktionalitäten an, um im Falle eines Fehlers automatisch eine alternative Version des Textobjektes zu lesen.

READ_TEXT

Der Funktionsbaustein `READ_TEXT` ist der allgemeine Baustein, um Textobjekte aus der Datenbank zu lesen. Die Schnittstelle des Funktionsbausteins enthält alle Felder, um einen beliebigen Textbaustein im System eindeutig zu identifizieren. Falls der spezifizierte Textbaustein nicht vorhanden ist, bricht der Funktionsbaustein mit einer Exception ab.

```
FUNCTION READ_TEXT
   IMPORT
      CLIENT
      ID
      LANGUAGE
      NAME
      OBJECT
```

```
    ARCHIVE_HANDLE
    LOCAL_CAT
  EXPORT
    HEADER
  TABLES
    LINES
  EXCEPTIONS
    ID
    LANGUAGE
    NAME
    NOT_FOUND
    OBJECT
    REFERENCE_CHECK
    WRONG_ACCESS_TO_ARCHIVE.
```

Im Parameter CLIENT wird dem Funktionsbaustein der Mandant, aus dem der Text gelesen werden soll, übergeben. Mit dem Funktionsbaustein können Textobjekte also auch mandantenübergreifend gelesen werden. Wird dieser Parameter nicht versorgt, wird als Defaultwert der aktuelle Mandant angenommen. Die folgenden vier Parameter ID, LANGUAGE, NAME sowie OBJECT sind die Felder, die das Textobjekt innerhalb des Mandanten eindeutig beschreiben. Hier werden keine Standard-Werte vom Funktionsbaustein angenommen, so dass alle vier Felder explizit beim Aufruf des Funktionsbausteins übergeben werden müssen.

Im Rahmen der Archivierung können Textobjekte bereits aus dem R/3-System gelöscht und lediglich im Archiv-System vorhanden sein. Der Funktionsbaustein bietet auch die Möglichkeit, Textobjekte aus dem Archiv zu lesen, indem im Parameter ARCHIVE_HANDLE eine aktuelle Zugriffsnummer für das Archiv übergeben wird. Diese Zugriffsnummer liefert der Funktionsbaustein ARCHIVE_OPEN_FOR_READ zurück. Er öffnet eine bestimmte Archiv-Datei und liefert eine Zugriffsnummer, die bei jedem Lesevorgang aus dem Archiv mitgegeben werden muss, zurück. Wird an READ_TEXT eine Zugriffsnummer übergeben, die nicht zu einer aktuell geöffneten Archiv-Datei gehört, wird die Ausnahme WRONG_ACCESS_TO_ARCHIVE ausgelöst.

Neu hinzugekommen bei fast allen Funktionsbausteinen der Funktionsgruppe STXD ist der Parameter LOCAL_CAT. Wird hier ein »X« an den jeweiligen Funktionsbaustein übergeben, werden lokale Kataloge für den Zugriff auf die Textobjekte verwendet. Ein lokaler Katalog wird innerhalb der Grenzen des aktuellen internen Modus angelegt. In einem internen Modus kann jedoch nur einen lokalen Katalog existieren. Wird versucht, einen weiteren internen Katalog anzulegen, lehnt der Funktionsbaustein die Verarbeitung ab. Weiterhin gilt, dass alle Zugriffe auf Funktionen der Funktionsgruppe STXD den gleichen Wert im Parameter LOCAL_CAT verwenden müssen. Wird ein Funktionsbaustein mit dem Wert »X« in LOCAL_CAT aufgerufen, obwohl zuvor ein Funktionsbaustein ohne diesen Wert aufgerufen wurde, bricht die Transaktion mit der Abbruchmeldung A239 ab.

Die Dokumentation dieses Konzeptes in R/3 ist allerdings sehr verwirrend, von daher sollte vorher genau überlegt werden, ob in eigenen Programmen auf diese Funktionalität zurückgriffen werden soll.

Die Rückgabewerte des Funktionsbausteins sind die Kopfstruktur des Textobjekts im Parameter HEADER sowie die Zeilen des Textobjekts in Form einer internen Tabelle im Parameter LINES.

READ_STDTEXT

Der Funktionsbaustein READ_STDTEXT bietet ein vereinfachtes Interface, um Standardtextobjekte zu lesen. Standardtexte sind dadurch gekennzeichnet, dass sie keinem Anwendungsobjekt des R/3-Systems zugeordnet sind, und sie enthalten im Feld TDOBJECT der Kopfstruktur immer den Wert »TEXT«. Da der Name Standard-Text bereits nahe legt, dass es sich hierbei um in der Regel unveränderliche Texte handelt, bietet der Funktionsbaustein READ_STDTEXT einen Durchgriff auf den Mandanten 000 des Systems, in dem die allgemeine Version des Textobjekts definiert werden kann.

```
FUNCTION READ_STDTEXT
   IMPORT
      ID
      LANGUAGE
      NAME
      USE_AUX_LANGUAGE
      USE_THRUCLIENT
      LOCAL_CAT
   EXPORT
      HEADER
   TABLES
      LINES
   EXCEPTIONS
      ID
      LANGUAGE
      NAME
      NOT_FOUND
      REFERENCE_CHECK.
```

Zunächst erhält der Funktionsbaustein, wie beim Funktionsbaustein READ_TEXT auch, die ID sowie die Sprache (LANGUAGE) und den Namen (NAME) des gewünschten Textobjekts übergeben. Die Übergabe des Mandanten erübrigt sich in diesem Fall, da der Funktionsbaustein immer zuerst im aktuellen Mandanten nach dem Textobjekt sucht. Wenn der Leseversuch im aktuellen Mandanten scheitert und im Feld USE_TRUCLIENT der Wert »X« übergeben wird, erfolgt ein weiterer Leseversuch im Mandanten 000. Höhere Priorität als der Mandantendurchgriff hat der Zugriff auf das Textobjekt in einer Alternativsprache. Dieser erfolgt nur, wenn im Parameter USE_AUX_LANGUAGE der Wert »X« übergeben wird. Die Alter-

nativsprachen, die für diesen Zugriff in Frage kommen, werden in der Tabelle T002C definiert.

Für den Parameter LOCAL_CAT gelten die gleichen Aussagen, wie beim Funktionsbaustein READ_TEXT.

Auch der Funktionsbaustein READ_STDTEXT liefert im Erfolgsfall die Kopfstruktur des gelesenen Textobjektes im Parameter HEADER und die Zeilen in Form einer internen Tabelle im Parameter LINES.

4.1.3 Speichern von Textobjekten

Anders als beim Lesen, wird beim Speichern von Textobjekten nicht zwischen Standardtexten und anderen Textobjekten unterschieden. Einheitlich für alle Textobjekt-Typen steht hierfür der Funktionsbaustein SAVE_TEXT zur Verfügung.

```
FUNCTION SAVE_TEXT
   IMPORT
      CLIENT
      HEADER
      INSERT
      SAVEMODE_DIRECT
      OWNER_SPECIFIED
      LOCAL_CAT
   EXPORT
      FUNCTION
      NEWHEADER
   TABLES
      LINES
   EXCEPTIONS
      ID
      LANGUAGE
      NAME
      OBJECT.
```

Im Parameter HEADER wird dem Funktionsbaustein die Kopfstruktur des zu speichernden Textobjekts übergeben. Der Parameter LINES enthält – wie beim Lesen – eine interne Tabelle mit den Zeilen des Textobjekts. Beim Speichern spielt der Mandant, der in der Kopfstruktur des Textbausteins übergeben wird, keine Rolle. Das Textobjekt wird in jedem Fall in dem Mandanten gespeichert, der im Parameter CLIENT übergeben wird. Als Default wird hier der aktuelle Mandant angenommen, falls der Parameter nicht spezifiziert wird.

Genauso kann über den Parameter OWNER_SPECIFIED definiert werden, ob der Besitzer des Textobjekts (Ersteller bei neuen Textobjekten, oder letzter Änderer bei geänderten Texten) aus der Kopfstruktur im Paramter HEADER übernommen, oder ob der aktuelle Benutzer in das entsprechende Feld des Textobjekts eingetragen werden soll.

Über den Parameter INSERT kann dem Funktionsbaustein angezeigt werden, dass es sich bei dem Textobjekt um einen neuen Text handelt. Normalerweise überprüft der Funktionsbaustein automatisch, ob das Textobjekt bereits existiert oder nicht. Durch diesen Parameter kann dieser Schritt übersprungen werden, was einen leichten Performancegewinn bewirkt.

Textbausteine können entweder sofort oder im Rahmen der Verbuchung in der Datenbank gespeichert werden. Normalerweise wird der Sicherungsmodus je Objekttyp (TDOBJECT) in der Tabelle TTXOB im Feld TDSAVEMODE festgelegt. Wird in dieser Tabelle der Wert »V« als Sicherungsmodus festgelegt, erfolgt die Sicherung der Textobjekte im Rahmen der Verbuchung. Beim Wert »D« entsprechend im Dialog. Soll sichergestellt werden, dass ein Textobjekt sofort in der Datenbank gespeichert wird, ohne Rücksicht auf den allgemeinen Wert der Tabelle TTXOB, kann im Parameter SAVEMODE_DIRECT der Wert »X« übergeben werden.

Der Funktionsbaustein liefert im Erfolgsfall im Parameter NEWHEADER die aktualisierte Kopfstruktur zurück. Weiterhin enthält der Rückgabeparameter FUNCTION den Wert »I«, falls der Textbaustein neu in der Datenbank angelegt wurde und den Wert »U«, falls ein bereits bestehendes Textobjekt aktualisiert wurde. Falls im Parameter FUNCTION nichts zurückgeliefert wird, ist im Funktionsbaustein keine Verarbeitung erfolgt.

4.1.4 Texte editieren mit dem Fullscreen-Editor

Das SAPScript Program Interface 1 bietet auch die Möglichkeit, ein Textobjekt über einen Standard-Editor, dem Fullscreen-Editor, wie er an vielen Stellen im R/3 verwendet wird, zu bearbeiten.

Beim Aufruf des Editors, welcher als Funktionsbaustein realisiert ist, wechselt die Ansicht im GUI in den Vollbild-Editor. Das Verhalten und die angebotenen Editiermöglichkeiten sind hierbei in weiten Grenzen vom Programmierer beeinflussbar. Der Fullscreen-Editor steht in zwei Versionen zur Verfügung. Zum einen bietet SAP den - aus älteren Release-Ständen bekannten - Editor weiterhin unter der Bezeichnung Zeileneditor an. Neu hinzugekommen ist der in Abbildung 4.2 gezeigte Fullscreen-Editor in Form des so genannten PC-Editors. Das Beispiel zeigt diesen Editor während der Bearbeitung eines Standard-Textbausteins im R/3-Release 4.6C.

SAP bietet hiermit einen komfortablen WYSIWYG Editor an und möchte damit den umständlichen Zeileneditor aus den früheren Release-Ständen ablösen. Im PC-Editor entfällt die Formatierungsspalte, am linken Bildrand, über die bislang das Absatzformat festgelegt werden konnte. Diese Aufgabe übernimmt ein Listenfeld oberhalb des Editierbereichs des Textes. Auch die Zeichenformate können jetzt über ein Listenfeld eingestellt werden. In der Abbildung wurde auf die Zeichenfolge »Fullscreen-Editor« das Zeichenformat »Zeichenfolge hervorgehoben« angewandt.

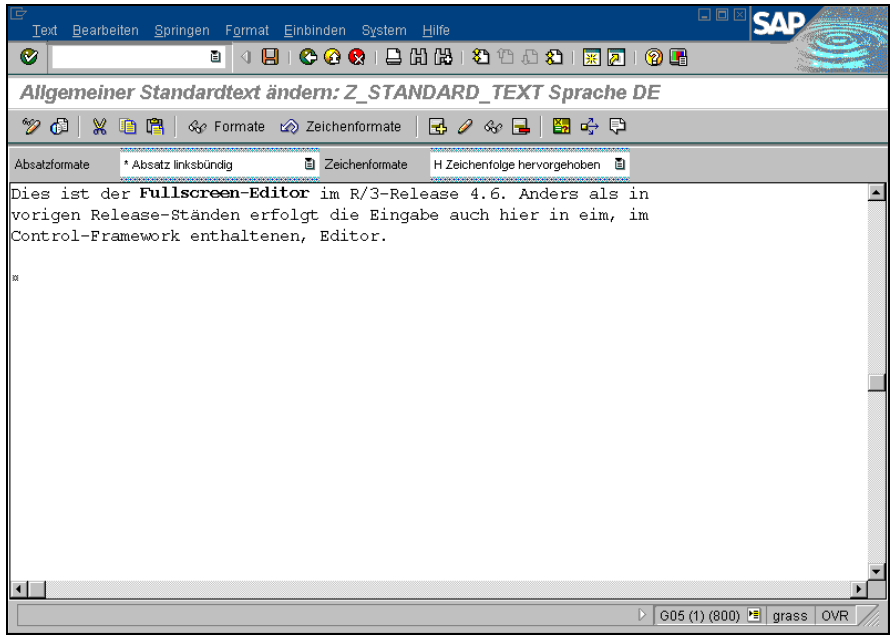

Abbildung 4.2 Screen-Shot des Fullscreen-Editors im Release 4.6 (© SAP AG)

In den beiden Listenfeldern werden die Absatz- bzw. Zeichenformate, die im verwendeten Stil bzw. Formular definiert wurden, angeboten. Sollen neue Formatierungsmöglichkeiten hinzugefügt werden, muss dies über die Pflege des Stils bzw. Formulars erfolgen.

Anders als beim Zeileneditor erfolgt das Editieren beim PC-Editor lokal im GUI des Benutzers. Demnach wird nicht bei jedem Return oder bei Betätigen einer Drucktaste sofort eine Kommunikation mit dem Applikationsserver durchgeführt, sondern nur zu bestimmten Zeitpunkten, wie zum Beispiel beim Wechsel zwischen Anzeige und Editiermodus. Hierdurch wird die Netzwerk- und Serverbelastung signifikant verringert und der Bedienkomfort auf Seiten des Benutzers deutlich erhöht.

Die Aufrufschnittstelle für den Editor – egal ob PC-Editor oder Zeileneditor – ist in einem Funktionsbaustein mit dem Namen `EDIT_TEXT` realisiert, der ebenfalls zur Funktionsgruppe `STXD` gehört.

```
FUNCTION EDIT_TEXT
  IMPORT
     DISPLAY
     EDITOR_TITLE
     HEADER
     PAGE
```

```
    WINDOW
    SAVE
    LINE_EDITOR
    CONTROL
    PROGRAM
    LOCAL_CAT
  EXPORT
    FUNCTION
    NEWHEADER
    RESULT
  TABLES
    LINES
  EXCEPTIONS
    ID
    LANGUAGE
    LINESIZE
    NAME
    OBJECT
    TEXTFORMAT
    COMMUNICATION.
```

In den Parametern HEADER und LINES wird dem Funktionsbaustein das zu bearbeitende Textobjekt übergeben. Somit wird auch festgelegt, welches SAPScript-Formular und welcher Textstil zur Bearbeitung verwendet werden soll (siehe Kopfstruktur).

Über den Parameter DISPLAY kann gesteuert werden, ob der Text im Editor editierbar ist oder nur zur angezeigt werden soll. Wird hier ein »X« übergeben, kann der Text vom Benutzer nicht geändert werden. Im Editor sind dann die Funktionscodes zum Sichern des Textes deaktiviert.

Über den Parameter LINE_EDITOR kann festgelegt werden, ob der Editor im PC-Modus (Space) oder im Zeilenmodus (»X«) gestartet werden soll.

Für die Aufbereitung des Textobjektes im Editor ist es relevant, für welche Seite bzw. welches Seitenfenster innerhalb des verwendeten Formulars der Text gedacht ist. Diese werden in den Parametern PAGE bzw. WINDOW an den Editor übergeben.

Innerhalb des Fullscreen-Editors kann der Benutzer, je nach Aufrufmodus, den veränderten Text über die »Sichern«-Schaltfläche abspeichern. Über den Parameter SAVE des Funktionsbausteins EDIT_TEXT kann dabei gesteuert werden, ob in diesem Fall der Funktionsbaustein das Sichern in der Datenbank durchführen soll (»X«), oder ob dies dem rufenden Programm obliegt (Space). Beim Betätigen der »Sichern«-Schaltfläche wird der Editor auf jeden Fall verlassen und der veränderte Text dem rufenden Programm in den entsprechenden Parametern NEWHEADER und LINES zurückgeliefert. Während jedoch im ersten Fall das Textobjekt auch innerhalb des Funktionsbausteins in der Datenbank gespeichert wird,

muss dies im letzteren Fall immer vom rufenden Programm durchgeführt werden. Dies ist insbesondere beim Anlegen neuer Anwendungsobjekte sinnvoll, deren Erzeugung auch nach Pflege des dazugehörigen Textobjektes vom Anwender ohne zu sichern abgebrochen werden kann.

Die Steuerung des Editors erfolgt über eine Struktur, die im Parameter CONTROL an den Funktionsbaustein EDIT_TEXT übergeben wird. Sie hat den Aufbau der Data Dictionary-Struktur ITCED, die in Tabelle 4.3 dargestellt ist.

Feldname	Datentyp	Bedeutung
SHOWTPFM	CHAR 1	Anzeige des Template-Formates
NOENDLINES	CHAR 1	Kennzeichen, ob am Ende des Textes automatisch leere Zeilen angefügt werden sollen (Space) oder keine Leerzeilen am Ende angefügt werden (X)
USERTITLE	CHAR 1	Kennzeichen, ob der im Parameter EDITOR_TITLE übergebene Titel als Editortitel verwendet werden soll, oder generisch ein Standard-Titel erzeugt werden soll
SCROLLEND	CHAR 1	Kennzeichen, ob der Cursor beim Betreten des Editors am Ende des Textes (X) oder am Anfang des Textes stehen soll (Space)
APP_PREV	CHAR 1	Kennzeichen, ob im GUI-Status die Option »Voriger Text« aktiv sein soll
APP_NEXT	CHAR 1	Kennzeichen, ob im GUI-Status die Option »Nächster Text« aktiv sein soll
APP_SUBID	CHAR 2	Oberfläche für verschiedene Applikationen
CHANGEMODE	CHAR 1	Kennzeichen, ob Wechsel von Anzeige zu Änderung erlaubt
EXIT_SYMB	CHAR 30	Name des Funktionsbausteins für Exit-Routine
EXIT_INCL	CHAR 30	Names des Funktionsbausteins für Exit-Routine

Tabelle 4.3
Aufbau der Struktur ITCED

Textobjekte in SAPScript können so genannte Template-Zeilen enthalten. Diese sind, soweit sichtbar, durch das Zeichen »>« in der Formatspalte gekennzeichnet. Das eigentliche Absatzformat wird in den ersten beiden Zeichen der Textzeile definiert. Über das Feld SHOWTPFM der Struktur ITCED kann festgelegt werden, ob bei den Templatezeilen das zugehörige Template-Format in der For-

matspalte (TDFORMAT der Zeilenstruktur) angezeigt (»X«) oder ob deren Ausgabe unterdrückt werden soll.

Standardmäßig zeigt der Editor am Ende des Textes eine Reihe von Leerzeilen an, um dem Benutzer zu ermöglichen, neue Zeilen zu erfassen. Über das Feld NOENDLINES kann unterdrückt werden, dass beim Betreten des Editors am Ende des Textes automatisch leere Zeilen angefügt werden. Bei Verwendung des PC-Editors spielte diese Option keine große Rolle mehr, denn das Editieren des Textes erfolgt lokal im GUI des Benutzers. Daher wird beim Betätigen der Enter-Taste, um neue Zeilen anzufügen, keine Kommunikation mit dem Applikationsserver durchgeführt. Bei Verwendung des Zeileneditors würde in diesem Fall eine Verbindung mit dem Applikationsserver hergestellt, das PAI des Editorfensters durchgeführt, eine Zeile angefügt und nach durchlaufen des PBO des Editors wieder die Kontrolle an den Benutzer zurückgegeben. In diesem Szenario ist es wesentlich sinnvoller, bereits beim Betreten des Editors leere Zeilen am Ende des Textes zur Verfügung zu haben.

Das Feld USERTITLE der Struktur ITCED steuert, ob der Titel des Editors aus dem Parameter EDITOR_TITLE des Funktionsbausteins entnommen wird (»X«), oder ob der Editor-Titel vom Funktionsbaustein generiert werden soll (Space). Wird der Editor-Titel generiert, setzt er sich aus den folgenden Feldern zusammen:

```
<TTXIT> <Statustext>: <Textname> <EDITOR_TITLE> Sprache <Sprache>
```

Wobei die Felder folgende Bedeutung haben:

Feldname	Bedeutung
<TTXIT>	Beschreibung der Text-ID (TDID) wie in Tabelle TTXIT hinterlegt.
<Statustext>	Statustext des Editors (z.B. »ändern« oder »anzeigen«).
<Textname>	Name des Textobjektes aus der Kopfstruktur (TDNAME)
<EDITOR_TITLE>	Inhalt des Parameters EDITOR_TITLE des Funktionsbausteins.
<Sprache>	Sprache des Textes aus der Kopfstruktur (TDSPRAS).

Tabelle 4.4
Bedeutung der Felder im zusammengesetzten Editor-Titel

Wird im Feld SCROLLEND der Struktur der Wert »X« übergeben, wird der Eingabecursor beim Betreten des Editors an das Ende des Textes gesetzt. Der Editor scrollt in diesem Moment so, dass das Textende im Fenster des Editors sichtbar ist.

Der SAPScript-Editor bietet für die Bearbeitung von Textobjekten eine Reihe von GUI-Oberflächen mit unterschiedlichen Verarbeitungsmöglichkeiten an. Welche der TextEditor-Oberflächen im Einzelfall verwendet wird, hängt vom

Objekttyp (TDOBJECT) des jeweiligen Textobjektes ab. In der Transaktion SE75 kann je Textobjekt das zu verwendende GUI festgelegt werden. Weiterhin kann hier auch der Verbuchungsmodus sowie das zu verwendende SAPScript-Formular und –stil definiert werden. Wird für den Objekttyp des Textobjekts der GUI-Status TA festgelegt, können bei Bedarf Buttons in der Schaltflächenleiste des Editors angezeigt werden, um zum nächsten bzw. vorigen zu editierenden Text zu verzweigen. Die Sichtbarkeit dieser Schaltflächen wird über die beiden Felder APP_NEXT bzw. APP_PREV der Struktur ITCED an den Editor übergeben. Ein »X« im jeweiligen Feld besagt, dass die dazugehörige Schaltfläche im GUI des Editors angezeigt wird.

Über das Feld APP_SUBID kann der in der Transaktion SE75 definierte GUI-Status explizit vom Programmierer übersteuert werden. Ansonsten gilt dabei die gleiche Logik, wie bei der Ermittlung des konkreten GUI-Status bei Verwendung des Feldes der Tabelle TTXOB (SE75).

Das Feld CHANGEMODE der Struktur ITCED legt fest, ob der Benutzer im Editor von der Anzeige in den Editormodus wechseln kann oder umgekehrt. Dieser Wechsel ist möglich, wenn in diesem Feld der Wert »X« an den Funktionsbaustein übergeben wird.

Innerhalb von Textobjekten können Symbole verwendet werden, die zur Laufzeit durch die Inhalte von Variablen aus dem Programmkontext ersetzt werden. Um für die Druckausgabe bzw. die Druckvorschau des Textobjekts im Editor die Ersetzung durchführen zu können, benötigt der Editor den Namen des Programms, aus dessen globalem Datenbereich die Variablen für die Symbolersetzung entnommen werden. Der Name dieses Programmes kann dem Editor im Parameter PROGRAM übergeben werden. Wird dieser Wert nicht gesetzt, nimmt das System hierfür den Inhalt der Systemvariable SY-CPROG, welches das laufende ABAP-Programm auf höchster Ebene enthält.

Nach Beendigung des Editors enthält der Rückgabeparameter FUNCTION die im Editor durchgeführte Aktion. Hierbei können die in Tabelle 4.5 dargestellten Werte vorkommen.

Wert	Bedeutung
D	Das Textobjekt wurde gelöscht
U	Der Inhalt des bestehenden Textobjekts wurde geändert
I	Das Textobjekt wurde neu angelegt

Tabelle 4.5
Mögliche Rückgabewerte des SAPScript-Editors im Feld FUNCTION

Neben der Kopfstruktur des Textbausteins nach der Bearbeitung (NEWHEADER) liefert der Funktionsbaustein weiterhin eine Struktur zurück, die dem Programm

anzeigt, auf welche Weise der Editor vom Benutzer verlassen wurde (RESULT). Der Rückgabeparamter RESULT hat hierbei den Aufbau der Dictionary-Struktur ITCER, wie er in Tabelle 4.6 dargestellt ist.

Feldname	Datentyp	Bedeutung
USEREXIT	CHAR 1	Zuletzt ausgeführte Benutzeraktion
FUNCTION	CHAR 1	Ausgeführte Text-Änderung
PCEDITOR	CHAR 1	Zuletzt benutzter Editor (Zeileneditor oder PC-Editor)

Tabelle 4.6
Aufbau der Struktur ITCER

Im Feld USEREXIT der Struktur wird zurückgegeben, auf welche Weise der Editor verlassen wurde. Mögliche Werte für dieses Feld sind in Tabelle 4.7 angegeben.

Wert	Bedeutung
E	Editor wurde über EXIT (gelber Pfeil) verlassen
B	Editor wurde über Zurück (grüner Pfeil) verlassen
C	Editor wurde über Abbrechen (rotes Kreuz) verlassen
P	Editor wurde über Schaltfläche »voriger Text« verlassen
N	Editor wurde über Schaltfläche »nächster Text« verlassen

Tabelle 4.7
Mögliche Werte des Feldes USEREXIT der Struktur ITCER

Im Feld FUNCTION der Struktur wird der Inhalt des Rückgabeparamters FUNCTION des Funktionsbausteins wiederholt. Schließlich enthält das Feld PCEDITOR der Struktur nach Beendigung des Editors den Wert »X«, wenn beim Verlassen des Editors der PC-Editor aktiv war. Hat der Benutzer zuletzt mit dem Zeileneditor gearbeitet, enthält dieses Feld keinen Wert.

4.1.5 Transport von Textobjekten

Wenn Textobjekte im Entwicklungssystem gepflegt wurden, muss im Anschluss daran ein Transport ins Produktivsystem erfolgen, um die Texte auch dort nutzen zu können. Dies erfolgt in Customizing-Aufträgen über das Transportwesen in R/3.

Normalerweise werden die Einträge in den Transportaufträgen für Textobjekte – so wie bei Entwicklungsobjekten auch – automatisch vom System generiert. In seltenen Fällen ist es jedoch notwendig, diese Einträge manuell zu erstellen.

Im Folgenden soll exemplarisch gezeigt werden, wie ein Transportauftrag zum Transport von Textobjekten erstellt werden kann.

Im ersten Schritt wird hierzu ein Customizing-Auftrag erstellt. Dies erfolgt in der Transaktion SE01 (Transport Organizer). Die Nummernvergabe für Transportaufträge erfolgt stets intern. Im Einstiegsbild der Transaktion wird ein Transportauftrag mit der ANLEGEN-Schaltfläche erstellt und im darauf folgenden Fenster die Option »Customizing-Auftrag« ausgewählt (siehe Abbildung 4.3).

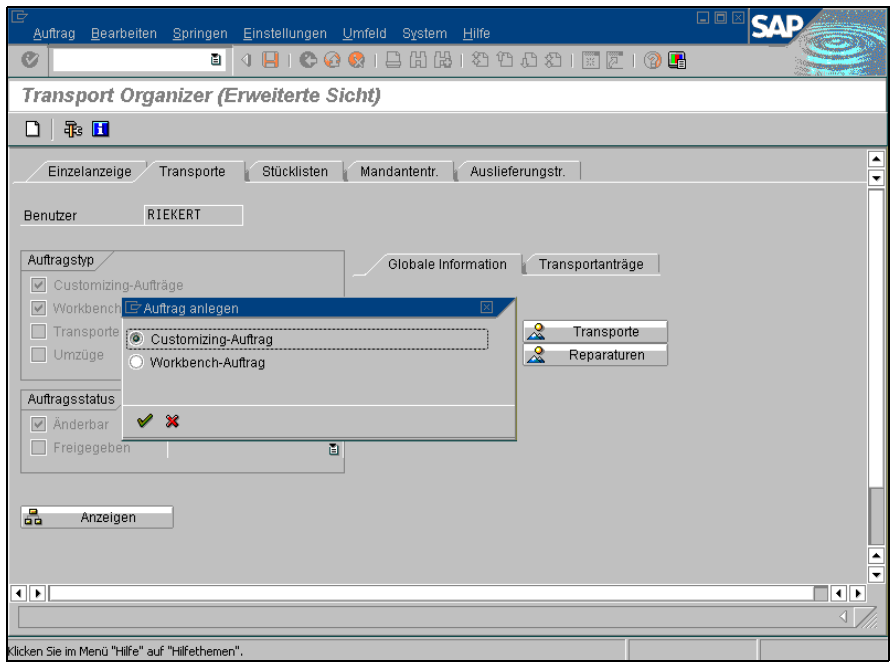

Abbildung 4.3
Transport Organizer (SE01) beim Anlegen eines neuen Auftrags (© SAP AG)

Im Folgenden erscheint eine Maske, in der eine Beschreibung für den Transportauftrag erfasst werden muss, bevor der Benutzer in die Übersichtsmaske gelangt. Um Textobjekte zu transportieren, müssen manuell die entsprechenden Einträge in der Objektliste des Auftrags vorgenommen werden. In der Objektliste des Auftrags muss ein Eintrag, wie in Abbildung 4.4 beispielhaft dargestellt, getätigt werden. Die manuell getätigten Einträge müssen gesichert werden und anschließend kann der Transportauftrag freigegeben und exportiert werden.

Einträge zum Transport von Textobjekten müssen zur Programm-ID R3TR und dem Objekttyp TEXT erfolgen. Der Objektname setzt sich für Textobjekte aus einer durch Komma getrennten Liste der vier Schlüsselfelder für Textobjekte zusammen.

Insgesamt haben Einträge in Transportaufträgen für Textobjekte den Aufbau:

```
R3TR TEXT <OBJECT>,<NAME>,<ID>,<LANGUAGE>
```

Hierbei muss beachtet werden, dass das Sprachkennzeichen nicht bei der Anmeldung mit zwei Zeichen angegeben wird, sondern lediglich mit der bisher verwendeten einstelligen Darstellung.

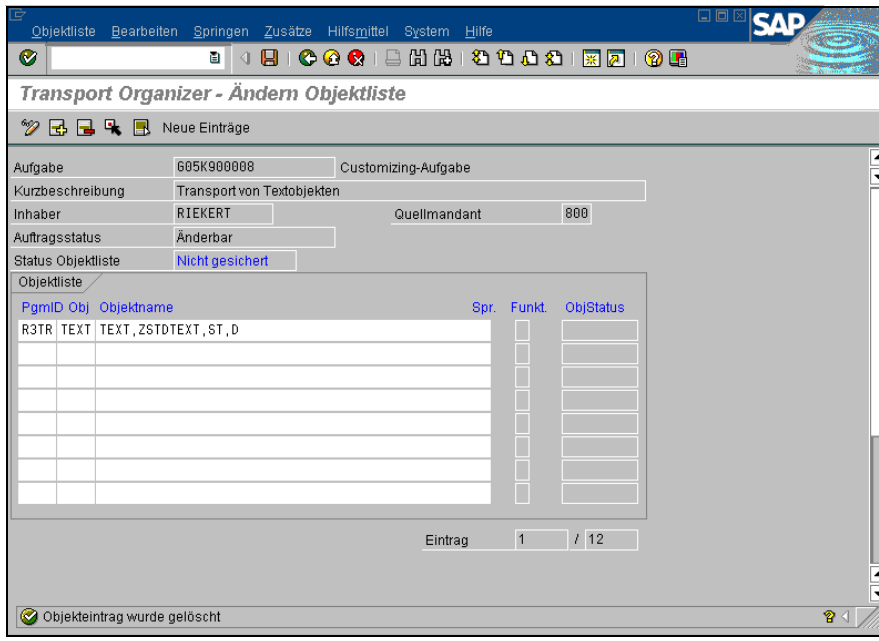

Abbildung 4.4
Eintrag in der Objektliste zum Transport von Textobjekten (© SAP AG)

4.2 Verwendung von Textobjekten

Textobjekte werden in verschiedenen Kontexten im R/3-System verwendet. Wichtigster Bereich hierbei sind Textobjekte, die an die betriebswirtschaftlichen Anwendungsobjekten des Systems gekoppelt sind. Nahezu jedes Anwendungsobjekt bietet an verschiedenen Stellen die Möglichkeit solche Textobjekte zu hinterlegen.

Zum anderen können Textobjekte als Standardtexte zur Verwendung in den verschiedensten Umfeldern verwendet werden. Standardtexte können in belie-

bige Textobjekte integriert werden und bieten als grundlegende Eigenschaft, dass beim Lesen ggf. ein Mandantendurchgriff in den Mandanten 000 des Systems erfolgen kann.

4.2.1 Textobjekte an betriebswirtschaftlichen Objekten

An fast allen Anwendungsobjekten des R/3-Systems existiert die Möglichkeit, Textinformationen zu hinterlegen. Da diese Texte in der Regel mit dem Anwendungsobjekt ausgedruckt werden können, handelt es sich meist um Textobjekte, wie im Abschnitt 4.1 beschrieben. Das Anlegen und Bearbeiten dieser Texte aus selbsterstellten Programmen heraus kann demzufolge mit den in Abschnitt 4.1.4 beschriebenen Fullscreen-Editor des R/3-Systems erfolgen. Der Programmierer muss jedoch immer beachten, dass mit Verlassen des TextEditors zwar das Textobjekt in der Datenbank abgelegt werden kann, dass aber die Kennzeichnung am Anwendungsobjekt, dass zusätzliche Texte existieren, niemals von den Standard-Editoren gesetzt werden, sondern immer im Verantwortungsbereich des Programms liegen, welches das Anwendungsobjekt als solches bearbeitet.

4.2.2 Standardtexte

Standardtexte sind Textbausteine, die im R/3-System hinterlegt werden können, um die Erstellung von Dokumenten zu erleichtern und zu standardisieren.

Standardtexte zeichnen sich dadurch aus, dass Sie zum Objekttyp »TEXT« angelegt sind. Ansonsten handelt es sich hierbei um Textobjekte, wie zuvor beschrieben. Lediglich im Bereich der Funktionen bieten Standardtexte dem Programmierer erweiterte Möglichkeiten.

Textobjekte sind – wie in Abschnitt 4.1.1 gezeigt – mandantenabhängig. Viele der Textbausteine, die im System verwendet werden, können jedoch mandantenübergreifend in mehreren Einheiten eines Konzerns gültig sein. Um den Aufwand für die Textpflege – die Texte sind schließlich das, was die Geschäftspartner eines Unternehmens in Form der Korrespondenz erhalten – möglichst gering zu halten, bieten die Standardtexte die Möglichkeit, eine allgemein gültige Version des Textes im Mandanten 000 des jeweiligen Systems anzulegen und lediglich von der Standardformulierung abweichende Versionen in den einzelnen aktiven Mandanten des Systems vorzuhalten.

Wie in Abschnitt 4.1.2 gezeigt, steht für das Lesen von Standardtexten ein leistungsfähiger Funktionsbaustein READ_STDTEXT zur Verfügung, der bei Bedarf versucht ein Textobjekt in einer alternativen Sprache oder eben – wie beschrieben – aus dem Mandaten 000 zu lesen.

Hierdurch können im Bereich der Textpflege hohe Kosten gespart werden und eine wichtige Fehlerquelle bei der Erstellung der Geschäftskorrespondenz ausgeschlossen werden.

4.3 Texte ohne Formatierungen

Während Textobjekte wie zuvor beschrieben, eine von SAP vordefinierte Struktur besitzen, haben Texte ohne Formatierungen dies nicht. Es handelt sich hierbei vereinfacht gesagt lediglich um unstrukturierten Text, abgelegt in Form von Tabellen der Zeilen oder in Form einer Variablen, die alle Zeilen des Textes enthält. Sie werden verwendet, wenn kein semantischer Bedarf danach besteht, einen Text für einen Ausdruck aufzubereiten. Klassisches Beispiel hierfür sind sicherlich ABAP-Programme selbst.

SAP stellt für den Aufbau dieser Texte keine Strukturen zur Verfügung. Gleichwohl existiert jedoch die Möglichkeit, auch diese Texte vom Benutzer bearbeiten zu lassen. Eine standardisierte Speicherung – wie bei den Textobjekten – existiert hierbei allerdings nicht.

Die Bearbeitung solcher Texte findet seit Release 4.6 in einem neuen Control, dem so genannten SAP TextEdit statt, welches Teil des bereits in Kapitel 3 verwendeten, ebenfalls neuen Control Frameworks ist. In den Standardtransaktionen von R/3 wird dieses Control auch häufig verwendet, um Textobjekte von Anwendungsobjekten ohne Maskenwechsel in den Standard-Transaktionen zu bearbeiten. Der Benutzer hat dabei keine Möglichkeit, Formatierungen und Hervorhebungen in den Text einzubringen, wie dies bei Verwendung des Vollbild-Editors möglich wäre.

Da – wie gesagt – die hier beschriebenen Texte keinen definierten Aufbau und keine standardisierte Speicherung haben, beschränken sich die Ausführungen dieses Abschnitts auf eine Beschreibung der Programmierung des SAP TextEdits. Mit diesem können beliebige Texte innerhalb eines Dynpros angezeigt oder bearbeitet werden.

Die Ausführungen zum SAP TextEdit können im Rahmen dieses Buches nicht vollständig sein. Es werden lediglich einige Kernfunktionalitäten vorgestellt, die den schnellen Einstieg in die Verwendung dieses Controls ermöglichen soll. Programmiertechniken zu den Ereignissen, die im Zusammenhang mit dem TextEdit definiert sind, können aus den Ausführungen des Kapitels 3 entnommen werden. Weitere Informationen bzgl. der zur Verfügung stehenden Methoden kann leicht aus dem System selbst gewonnen werden.

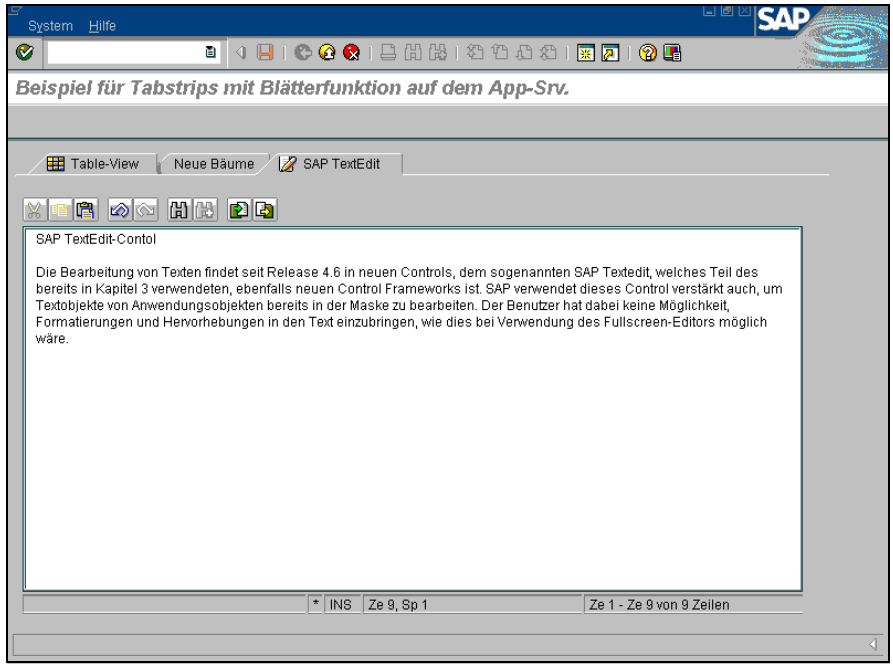

Abbildung 4.5
SAP TextEdit auf Tab-Strip-Seite (© SAP AG)

Um diese Beschreibung auf das Wesentliche konzentrieren zu können, wird als Programmrahmen das in Kapitel 3 entwickelte Beispielprogramm erweitert. Die Erweiterung sieht vor, dass für das TextEdit eine weitere Tab-Seite im Tab-Strip hinzugefügt wird. Auf dieser soll ein TextEdit angelegt werden, um einen zu Beginn leeren Text zu editieren. Die Maske des Programms mit der neuen Tab-Seite wird sich wie in Abbildung 4.5 zu sehen darstellen.

4.3.1 Eigenschaften des SAP TextEdit-Controls

Wie bereits erwähnt handelt es sich beim SAP TextEdit um ein weiteres Control aus dem mit Release 4.6 eingeführten Control Frameworks. Analog zur neuen Baumdarstellung (SAPTree) handelt es sich bei der Instanz der Klasse `CL_GUI_TextEdit`, welche in diesem Zusammenhang zum Zuge kommt, lediglich um eine programminterne Repräsentation für das Control selbst, welches Remote auf dem Präsentationsrechner des Benutzers entweder im SAPGUI oder in der Browserdarstellung abläuft. Die Implementierung des Controls selbst, ist entweder als ActiveX auf den Microsoft-Plattformen, oder in Form von Java-Beans auf den übrigen Plattformen realisiert. Die Kommunikation des Controls

selbst mit seiner Repräsentation im ABAP erfolgt über verschiedene Klassen des Control Frameworks.

Für den Programmierer ist der Unterschied zwischen einem TreeView und einem TextEdit lediglich im ABAP-Programm selbst, nicht jedoch im Dynpro-Editor relevant. In beiden Fällen läuft das Control in einem Container, der im Dynpro-Editor in Form einer Customer-Area realisiert wird.

Das TextEdit ist für den Benutzer erheblich komfortabler zu benutzen, als der bisher bekannte Zeileneditor. Wird das TextEdit im »Bearbeiten«-Modus (auch eine reine Textanzeige ist möglich) dargestellt, stehen die aus den bekannten Desktop-Anwendungen gewohnten Funktionalitäten wie

- Einfüge-/Überschreibmodus,
- Textauswahl mit Maus und Tastatur,
- Ausschneiden/Kopieren/Einfügen,
- Suchen/Ersetzen,
- Widerrufen sowie auf Wunsch ein
- Automatischer Zeilenumbruch

zur Verfügung.

Weiterhin kann die maximale Textlänge in Zeichen gesetzt sowie das Laden und Speichern in lokalen Dateien optional vom Control angeboten werden.

In der Darstellung besitzt das TextEdit in der Regel eine Toolbar mit Schaltflächen für die oben genannten Funktionen oberhalb des Eingabebereiches sowie eine Statuszeile unterhalb des Eingabebereichs. In der Statuszeile werden Informationen über den aktuellen Text sowie der Status des Editors selbst dargestellt (siehe Abbildung 4.5). Weiterhin können beliebige Nachrichten für den Benutzer im Bereich der Statuszeile ausgegeben werden.

Sowohl die Ausgabe der Toolbar als auch die der Statuszeile ist optional und können bei Bedarf vom Programmierer unterdrückt werden. Das TextEdit besteht in diesem Fall lediglich aus dem Eingabebereich. Diese Form der Darstellung wird auch häufig in den Standard-Transaktionen des R/3-Systems verwendet. SAP verzichtet oft zugunsten einer geringeren Anzahl von Bildwechseln auf die Möglichkeit, innerhalb der Langtexte zu den Anwendungsobjekten Formatierungen zuzulassen.

4.3.2 Erzeugung der Tab-Seite für das TextEdit

Träger für das TextEdit wird im Beispiel dieses Kapitels das, in Kapitel 3 entwickelte, Bespielprogramm `SAPMZFLIGHT` sein. Im Träger-Dynpro, auf welchem das Tab-Strip abgelegt ist, wird eine weitere Tab-Seite durch Änderung der Eigenschaften des Tab-Strips hinzugefügt.

In den Eigenschaften der neuen Seite wird die Beschriftung des Tab-Reiters, sowie der Funktionscode, der beim Betätigen des Reiters ausgeführt wird, definiert. Je nach verwendetem Blättertyp des Tab-Strips handelt es sich hierbei um einen Funktionscode, der lokal im SAPGUI des Benutzers verarbeitet werden soll (Funktionstyp »P«) oder um einen Funktionscode, der ein PAI/PBO-Ereignispaar auslöst und somit das Blättern auf dem Applikationsserver realisieren lässt (Funktionstyp » «). Detaillierte Informationen zu den Funktionstypen und den Seitenwechseln in Tab-Strips kann Kapitel 3.2 entnommen werden.

Im Anschluss daran muss die neue Tab-Seite in die Ablauflogik des Dynpros aufgenommen werden. Es wird davon ausgegangen, dass die neue Seite als Subscreen mit der Dynpro-Nummer `9400` im Modulpool angelegt wird.

Beim Blättern im SAPGUI geschieht dies durch Hinzufügen der jeweiligen `CALL SUBSCREEN`-Anweisung sowohl im PBO als auch PAI des Dynpros. Beim Blättern auf dem Applikationsserver entsprechend durch Reagieren auf den Funktionscode für die neue Tab-Seite und umsetzten der aktiven Seite in den globalen Datenfeldern des Programms.

Auf eine Darstellung des Codes wird hier verzichtet, da dieser analog zu dem bereits in Kapitel 3 gezeigten Anweisungen ist.

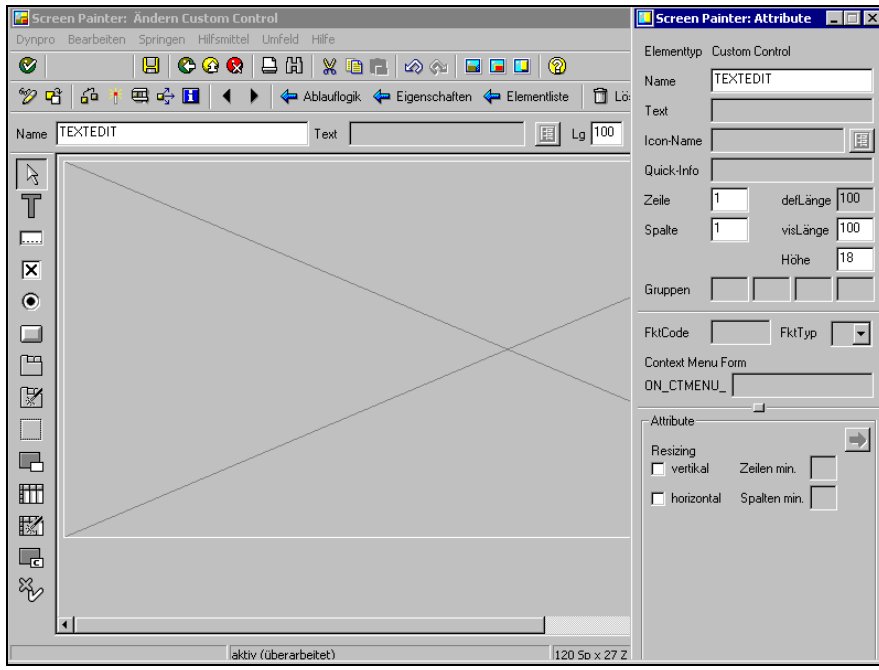

Abbildung 4.6
Eigenschaften des Containers für das TextEdit-Control (© SAP AG)

Nun muss noch das neue Dynpro mit der Dynpro-Nummer 9400 im Modulpool angelegt werden, um das Programm wieder in einen funktionsfähigen Zustand zu versetzten.

Als letzte Aktion – um die Arbeiten im Dynpro-Editor zu vervollständigen – muss schließlich noch die Custom Control-Area auf dem gerade erstellten Subscreen angelegt werden.

In den Eigenschaften des Custom Controls ist neben der Größe des Bereichs auf dem Dynpro, lediglich der Name des Controls, welcher im ABAP-Programm verwendet wird, um auf den Bereich zuzugreifen, zu pflegen. Im obigen Beispiel (Abbildung 4.6) wurde hier der Name TextEdit verwendet.

Nachdem die geänderten Entwicklungsobjekte aktiviert wurden, kann die Transaktion fehlerfrei ausgeführt werden. Der neue Tab-Reiter ist - wie definiert – vorhanden und die neue Seite, zeigt einen dunklen Bereich an der Stelle in der Maske, an der später das TextEdit-Control zu sehen sein wird.

4.3.3 Erzeugung des SAP TextEdits

Nachdem die notwendigen Vorarbeiten im Bereich der Dynpros durchgeführt wurden, erfolgt die weitere Implementierung der Funktionalität im ABAP-Code.

Zunächst müssen für das Custom Control sowie für das SAP TextEdit-Control globale Variable zur programminternen Repräsentation der Objekte erzeugt werden. Für das Custom Control selbst wird hier - analog zum Tree-View - eine Instanz der Container-Klasse CL_GUI_CUSTOM_CONTAINER verwendet.

Das TextEdit selbst wird in ABAP durch eine Instanz der Klasse CL_GUI_TEXTEDIT repräsentiert. Im TOP-Include des Programms werden daher die folgenden Zeilen eingefügt:

```
DATA:  G_TEXTCONTAINER   TYPE REF TO CL_GUI_CUSTOM_CONTAINER,
       G_TEXTEDIT        TYPE REF TO CL_GUI_TEXTEDIT.
```

In der Ablauflogik des neuen Dynpros werden die beiden neu deklarierten Variablen initialisiert. Zunächst wird der Container, in dem das TextEdit dargestellt werden soll, erzeugt. Dies erfolgt einmalig beim ersten Durchlaufen des PBO im Module 9400_CREATE_CONTAINER.

```
MODULE 9400_CREATE_CONTAINER OUTPUT.
  IF G_TEXTCONTAINER IS INITIAL.
*   Erzeugen des Containers
    CREATE OBJECT G_TEXTCONTAINER
        EXPORTING
          CONTAINER_NAME    = 'TEXTEDIT'
        EXCEPTIONS
          CNTL_ERROR        = 1
```

```
          CNTL_SYSTEM_ERROR = 2
          CREATE_ERROR      = 3
          LIFETIME_ERROR    = 4.
  ENDIF.
ENDMODULE.                              " 9400_create_container  OUTPUT
```

Der erste Aufruf des PBO wird daran erkannt, dass die Variable G_TEXTCONTAINER noch nicht initialisiert ist. Bei allen weiteren PBO-Aufrufen enthält diese Variable eine Referenz auf die hier erzeugte Container-Instanz. Der Name »TEXTEDIT« ist der im Dynpro-Editor vergebene Name für den Custom Control-Bereich.

Im Anschluss an den Container muss noch das TextEdit-Control selbst erzeugt werden. Dies geschieht in einem eigenen Module 9400_CREATE_TEXTEDIT unter der Voraussetzung, dass die vorherige Erzeugung des Containers fehlerfrei geglückt ist.

```
MODULE 9400_CREATE_TEXTEDIT OUTPUT.
* Das Textedit soll nur erzeugt werden, wenn das Anlegen des
* Containers erfolgreich war.
  CHECK NOT G_TEXTCONTAINER IS INITIAL.

* Anlegen des Baumes nur, wenn noch nicht geschehen
  CHECK G_TEXTEDIT IS INITIAL.

  CREATE OBJECT G_TEXTEDIT
      EXPORTING
        PARENT                     = G_TEXTCONTAINER
*       MAX_NUMBER_CHARS           =
*       STYLE                      = 0
*       WORDWRAP_MODE              = WORDWRAP_AT_WINDOWBORDER
*       WORDWRAP_POSITION          = -1
*       WORDWRAP_TO_LINEBREAK_MODE = FALSE
*       FILEDROP_MODE              = DROPFILE_EVENT_OFF
*       LIFETIME                   =
*       NAME                       =
*     EXCEPTIONS
*       ERROR_CNTL_CREATE          = 1
*       ERROR_CNTL_INIT            = 2
*       ERROR_CNTL_LINK           = 3
*       ERROR_DP_CREATE           =  4
*       GUI_TYPE_NOT_SUPPORTED    = 5
*       others                    = 6
        .
ENDMODULE.                              " 9400_CREATE_TEXTEDIT  OUTPUT
```

Das abgebildete Programmfragment zeigt den Konstruktor-Aufruf der Klasse CL_GUI_TEXTEDIT in minimaler Ausprägung. Lediglich der Parameter PARENT, in dem die Referenz auf den Container, in dem das Control angezeigt werden soll,

übergeben wird, ist als Pflichtparameter definiert. Die übrigen Parameter sind optional und werden mit entsprechenden Defaultwerten übergeben.

Im Parameter MAX_NUMBER_CHARS wird die maximale Länge des Textes in Zeichen übergeben. Hierüber kann verhindert werden, dass der Benutzer im Control mehr Text erfasst, als tatsächlich in den dahinter liegenden Datenfeldern abgespeichert werden kann. Wird – wie hier gezeigt – der Paramter nicht übergeben bzw. der Wert »0« übergeben, besteht keine Längenbeschränkung im Text-Control und der Benutzer kann einen beliebig langen Text erzeugen.

Über den Parameter Style können Eigenschaften des Controls gesetzt werden. Hierbei können die in der Klasse CL_GUI_CONTROL definierten Konstanten, deren Name mit WS_ beginnt, verwendet werden. Mehrere dieser Werte können durch einfache Addition der jeweiligen Konstanten kombiniert werden. Bei diesen Konstanten handelt es sich um Variable auf einer sehr technischen Ebene, wie z.B. WS_CLIPCHILDREN, mit dem das Verhalten eines Fensters beim Überlappen durch andere Fenster definiert werden kann. Das Setzten dieser Variable durch den Programmierer wird in der Regel nicht notwendig sein, da R/3 standardmäßig eine, für das jeweilige Control passende, Einstellung annimmt.

Das SAP TextEdit kann auf Texteingaben, die die sichtbare Breite des Controls überschreiten, auf unterschiedliche Weise reagieren. Das gewünschte Verhalten wird dem Control über die Parameter WORDWRAP_MODE, WORDWRAP_POSITION sowie WORDWRAP_TO_LINEBREAK_MODE beim Konstruktor-Aufruf übergeben. Der Parameter WORDWRAP_MODE bestimmt hierbei das generelle Zeilenumbruch-Verhalten des Controls. In der Klasse CL_GUI_TEXTEDIT sind hierfür drei Konstanten definiert, die das unterschiedliche Verhalten festlegen. Diese sind in Tabelle 4.8 dargestellt.

Name der Konstante	Bedeutung
WORDWRAP_OFF	Kein automatischer Zeilenumbruch
WORDWRAP_AT_WINDOWBORDER	Automatischer Zeilenumbruch am jeweils sichtbaren Rand des Controls
WORDWRAP_AT_FIXED_POSITION	Automatischer Zeilenumbruch an einer definierten Spalte

Tabelle 4.8
Konstanten für den Parameter WORDWRAP_MODE des SAP TextEdit

Die Konstante WORDWRAP_OFF bewirkt, dass kein Zeilenumbruch stattfindet. Stattdessen rollt der Textbereich des Controls nach links aus dem sichtbaren Bereich des Fensters heraus und ermöglicht dem Benutzer, weiteren Text in der Zeile einzugeben. Die Breite der Zeile ist dabei nicht beschränkt.

Beim durch WORDWRAP_AT_WINDOWBORDER definierten Zeilenumbruchverhalten, wird der Text beim Erreichen des rechten Randes des Text-Controls automatisch umgebrochen. Je nachdem, wie groß das Textcontrol auf dem Dynpro dargestellt wird, erfolgt der Zeilenumbruch an anderer Stelle. Diese Option eignet sich besonders für Texte, die auch in anderen Zusammenhängen, wie zum Beispiel beim Ausdruck verwendet werden. Da der Text auf dem Papier in der Regel ohnehin anders umgebrochen werden muss, als auf dem Bildschirm, ist es sogar sinnvoll, dieses vom jeweiligen Kontext abhängige Verhalten dem Benutzer zu visualisieren.

Soll der Text jedoch unabhängig vom verwendeten Zeichensatz und von der jeweiligen Fensterbreite an einer festen Position umgebrochen werden, kann dies durch Verwendung der Konstante WORDWRAP_AT_FIXED_POSITION erreicht werden. In diesem – und nur in diesem Fall (!) – definiert der Parameter WORDWRAP_POSITION die maximale Zeilenbreite in Zeichen. Bei den anderen Zeilenumbruchmodi hat dieser Parameter keine Funktion.

Schließlich hat noch der Parameter WORDWRAP_TO_LINEBREAK_MODE des Konstruktors die Funktion festzulegen, ob beim Auslesen des Textes aus dem Zwischenspeicher des Controls Zeilenumbrüche in tatsächliche Zeilenumbrüche umgewandelt (auch »harte« Zeilenumbrüche genannt) werden, wie sie auch das Betätigen der ↵-Taste während der Eingabe erzeugen, oder ob die auomatisch erzeugten Zeilenumbrüche beim Auslesen nicht berücksichtigt werden (»weicher«-Zeilenumbruch). Im letzteren Fall werden alle Zeilen bis zum nächsten »harten« Zeilenumbruch logisch als eine Einheit (Zeile) behandelt. Bei erneuter Ausgabe in einem schmaleren oder breiteren Eingabefenster würden die Umbrüche in diesem Fall neu ermittelt, während sie im ersten Fall durch den Text festgelegt wären, da diese Zeilenumbrüche nicht mehr von denen, die explizit durch die ↵-Taste gesetzt wurden, unterschieden werden können.

Die durch diese drei Parameter gesetzten Werte für das Zeilenumbruchverhalten können nachträglich durch einen Aufruf der Methode SET_WORDWRAP_BEHAVIOR der Klasse CL_GUI_TEXTEDIT jederzeit verändert werden.

Über den Parameter FILEDROP_MODE kann das Drag&Drop-Verhalten des Controls im GUI des Benutzers beeinflusst werden. Die Darstellung der Drag&Drop-Funktionalitäten des Control Frameworks würden den Rahmen dieses Buches sprengen. An dieser Stelle sei daher auf das Buch »Controls Technology« von SAP verwiesen.

Der Parameter LIFETIME entspricht exakt dem gleichnamigen Parameter der Klassen des SAP TreeView. Das Verhalten kann daher in Abschnitt 3.3 dieses Buches nachgelesen werden. Über den Parameter NAME schließlich kann der Instanz des Controls ein Name mitgegeben werden. Was dieser bewirkt, ist im System jedoch nicht dokumentiert.

4.3.4 Setzen und Auslesen des editierten Textes

Eine der wichtigsten Funktionen eines Editors ist das Setzten des Textes vor dem Editieren bzw. das Auslesen des Textes nach der Bearbeitung durch den Benutzer. Das SAP Textedit-Control bietet hierfür zwei Verfahren an, die verwendet werden können. Entscheidend für die Wahl der Methode sind die Einstellungen der Zeilenumbruchparameter und damit letztlich die Art des Textes.

Setzen bzw. Auslesen des Textes als interne Tabellen

Wie beim bisher bekannten Zeileneditor können dabei die Zeilen eines Textes als eine aus Zeichenketten bestehende Tabelle gesehen werden. Eine Möglichkeit, auf die Texte des Editors zuzugreifen besteht darin, den Text in Form einer internen Tabelle, die aus den Zeilen des Textes besteht, zu übergeben bzw. zu übernehmen. Für dieses Verfahren stehen die beiden Methoden `SET_TEXT_AS_R3TABLE` sowie `GET_TEXT_AS_R3TABLE` zur Verfügung.

```
METHOD SET_TEXT_AS_R3TABLE
  IMPORTING
    TABLE
  EXCEPTIONS
    ERROR_DP        = 1
    ERROR_DP_CREATE = 2.
```

Im Parameter TABLE werden die Zeilen der internen Tabelle übergeben. Hierbei spielt es keine Rolle, wie lange die Zeilen sind. Vor der Anzeige des Textes im Editor wird dieser entsprechend den aktuell gesetzten Zeilenumbruchsparametern formatiert.

```
METHOD GET_TEXT_AS_R3TABLE
  IMPORTING
    ONLY_WHEN_MODIFIED
  EXPORTING
    TABLE
    IS_MODIFIED
  EXCEPTIONS
    ERROR_DP                = 1
    ERROR_CNTL_CALL_METHOD  = 2
    ERROR_DP_CREATE         = 3
    POTENTIAL_DATA_LOSS     = 4.
```

Auch bei der Methode `GET_TEXT_AS_R3TABLE` erfolgt der Transfer des Textes in Form einer internen Tabelle. Hierbei ist zu beachten, dass die Zeilen des Editors abgeschnitten werden, wenn ihre Länge die einer Zeile der internen Tabelle überschreitet. In diesem Zusammenhang sei erwähnt, dass im Sinne des Text-Edit-Controls eine Zeile immer durch einen »harten« Zeilenumbruch beendet wird. Dieser wird gesetzt, wenn der Benutzer eine Zeile mit der ↵-Taste ab-

schließt, oder wenn der Parameter WORDWRAP_TO_LINEBREAK_MODE des Controls auf CL_GUI_TEXTEDIT=>TRUE gesetzt ist. Alle anderen Zeilenumbrüche, die der Benutzer im Editor sieht, sind so genannte »weiche« Zeilenumbrüche und werden von den hier genannten beiden Methoden ignoriert. Für das Edit-Control kann eine Zeile daher wesentlich länger sein, als es für den Benutzer im Editor erscheint.

SAP empfiehlt, diese beiden Methoden nur im Zusammenhang mit der Zeilenumbruch-Methode WORDWRAP_AT_FIXED_POSITION zu verwenden. Bei den anderen beiden Verfahren besteht nicht die Möglichkeit, eine maximale Zeilenlänge zu definieren. Nur wenn die maximale Zeilenlänge bekannt ist, kann garantiert werden, dass beim Auslesen des Textes keine Textinformationen verloren gehen.

Die Methode GET_TEXT_AS_R3TABLE bietet weiterhin die Möglichkeit, den Text nur dann vom Front-End des Benutzers auf den Server zu transferieren, wenn der Benutzer Änderungen im Text vorgenommen hat. Hierfür steht der Parameter ONLY_WHEN_MODIFIED zur Verfügung. Wird hier der Wert CL_GUI_TEXTEDIT=>TRUE übergeben und der Text hat sich nicht geändert, wird der Text nicht übertragen und im Parameter TABLE eine leere Tabelle zurückgeliefert. Um die Datenmenge, die zwischen dem Applikationsserver und dem GUI übetragen werden müssen, möglichst gering zu halten, ist die Verwendung dieses Parameters sehr anzuraten.

Aus dem Rückgabeparameter IS_MODIFIED kann zusätzlich erkannt werden, ob sich der Text im Edit-Control geändert hat. Als Werte werden auch hier die, in der Klasse CL_GUI_TEXTEDIT definierten, boolschen Konstanten verwendet.

Texttransfer als Stream

Die andere Methode, den Text in einen SAP Textedit zu setzten bzw. aus ihm auszulesen besteht darin, die Textinformationen nicht in Form einer Tabelle, die für jede Textzeile eine Zeile enthält, zu übergeben, sondern als Datenstrom zum bzw. vom Editor zu betrachten. Auch hierbei wird die Textinformation in Form einer internen Tabelle übergeben, wobei allerdings nicht unbedingt eine Tabellenzeile einer Textzeile entsprechen muss. Die Zeilen des Textes werden hierbei – getrennt durch »harte« und »weiche« Zeilenumbrüche – direkt hintereinander in den Tabellenzeilen übertragen. Auf diese Weise wird der Text sehr kompakt, da keine Tabellenzeilen mit Leerzeichen aufgefüllt werden und anders herum keine Textzeilen abgeschnitten werden, da Tabellenzeilen zu kurz sind, um sie aufzunehmen.

Bei der Textübertragung als Datenströme werden die »weichen«, d.h. aufgrund der konkreten Eigenschaften des Textedit-Controls automatisch erzeugten Zeilenumbrüche, übertragen und stehen somit dem ABAP-Programmierer zur Verfügung. Bei der im vorigen Abschnitt beschriebenen Methode besteht diese Möglichkeit nicht.

```
METHOD SET_TEXT_AS_STREAM
  IMPORTING
    TEXT
  EXCEPTIONS
    ERROR_DP        = 1
    ERROR_DP_CREATE = 2.
```

Die Aufrufschnittstellen der beiden Methoden SET_TEXT_AS_STREAM und GET_TEXT_AS_STREAM gleichen denen der oben dargestellten Methoden. Lediglich der Parameter TABLE der beiden zuvor genannten Methoden wurde hier in TEXT umbenannt. Ansonsten gelten alle Aussagen analog.

```
METHOD GET_TEXT_AS_STREAM
  IMPORTING
    ONLY_WHEN_MODIFIED
  EXPORTING
    TEXT
    IS_MODIFIED
  EXCEPTIONS
    ERROR_DP                = 1
    ERROR_CNTL_CALL_METHOD  = 2.
```

4.3.5 Umschalten zwischen Eingabe-/Anzeige-Mode

Das SAP TextEdit kann nicht nur verwendet werden, um Texte vom Benutzer bearbeiten zu lassen, sondern auch, um lediglich eine Anzeigefunktionalität für Texte zu realisieren. Hierfür kann am TextEdit-Control ein Flag gesetzt werden, welches den Modus des Editors definiert. Zwischen den beiden Modi kann umgeschaltet werden, indem die Methode SET_READONLY_MODE der Klasse CL_GUI_TEXTEDIT aufgerufen wird.

```
METHOD SET_READONLY_MODE
  IMPORTING
    READ_ONLY
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1
    INVALID_PARAMETER      = 2.
```

Die Methode erhält lediglich einen Parameter, der den Editiermodus beschreibt. Dieser kann den Wert CL_GUI_TEXTEDIT=>TRUE enthalten, wenn das Text-Control den Text nicht veränderbar anzeigen soll. Der andere gültige Wert ist der in der Klassenvariable CL_GUI_TEXTEDIT=>FALSE definierte Wert, der den Texteditor anweist, den Text vom Benutzer bearbeitbar darzustellen.

Wird der Parameter nicht übergeben, wird der Wert TRUE angenommen und der Text entsprechend lediglich angezeigt.

4.3.6 Ein-/Ausblenden der Toolbar und Statuszeile

Im einführenden Abschnitt dieses Kapitels wurde bereits erwähnt, dass beim SAP Textedit-Control die Möglichkeit besteht, sowohl die Toolbar oberhalb, als auch die Statuszeile unterhalb des Editierbereiches des Controls bei Bedarf auszublenden.

Hierfür stehen in der Klasse CL_GUI_TEXTEDIT die Methoden SET_TOOLBAR_MODE bzw. SET_STATUSBAR_MODE zur Verfügung.

```
METHOD SET_TOOLBAR_MODE
  IMPORTING
    TOOLBAR_MODE
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1
    INVALID_PARAMETER      = 2.

METHOD SET_STATUSBAR_MODE
  IMPORTING
    STATUSBAR_MODE
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1
    INVALID_PARAMETER      = 2.
```

Wie gezeigt, verfügen diese beiden Methoden über eine sehr ähnliche Schnittstelle. Im einzigen Parameter der Methoden kann – wie bei der Methode SET_READONLY_MODE – entweder der Wert CL_GUI_TEXTEDIT=>TRUE zum Anzeigen bzw. CL_GUI_TEXTEDIT=>FALSE zum Verbergen der Toolbar bzw. der Statuszeile übergeben werden.

Wenn die Statuszeile im Control sichtbar ist, besteht die Möglichkeit, dem Benutzer hier Informationen anzuzeigen. Hierfür existiert die Methode SET_STATUS_TEXT, die als einzigen Parameter den in der Statuszeile darzustellenden Text erhält.

```
METHOD SET_STATUS_TEXT
  IMPORTING
    STATUS_TEXT
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

Der so gesetzte Text bleibt erhalten, bis er explizit durch einen weiteren Aufruf dieser Methode oder implizit durch eine Systemmeldung ersetzt wird.

4.3.7 Hervorheben von Kommentarzeilen

Nicht nur für Kommentarzeilen geeignet, aber ursprünglich von SAP hierfür gedacht, existiert die Möglichkeit, im Edit-Control Zeilen, die mit einer bestimm-

ten Zeichenfolge beginnen, hervorzuheben. Im ABAP-Editor handelt es sich hierbei um die Zeilen, die mit dem Zeichen »*« beginnen und als Kommentarzeilen im ABAP definiert sind.

In diesem Punkt ist die Implementierung des TextEdits sehr stark auf die Bedürfnisse als ABAP-Editor ausgerichtet. Neben der Möglichkeit, Kommentarzeilen farblich hervorzuheben, besteht die Möglichkeit, beliebige Zeilen zu Kommentarzeilen zu machen, indem die definierte Zeichenfolge vorangestellt wird, bzw. umgekehrt wieder zu entfernen. Auf diese Möglichkeiten soll im Folgenden kurz eingegangen werden, da auch denkbar ist, dass diese für andere Zwecke sinnvoll genutzt werden könnten.

Definieren der Zeichenfolge für Kommentarzeilen

In der Programmiersprache ABAP sind Zeilen, die mit dem Zeichen »*« beginnen bekannterweise Kommentarzeilen. Im Programmeditor werden diese Zeilen optisch durch eine andere Farbgebung hervorgehoben. Diese Eigenschaft wurde auch im Release 4.6, bei dem der ABAP-Editor auf das TextEdit-Control umgestellt wurde, beibehalten. Allgemeiner ausgedrückt bietet das TextEdit-Control die Möglichkeit, Zeilen, die mit einer bestimmten Zeichenfolge beginnen, anders zu behandeln als die übrigen Zeilen.

Um die Zeichenfolge, für die dieses Verhalten gelten soll, festzulegen, bietet die Klasse `CL_GUI_TEXTEDIT` die Methode `SET_COMMENTS_STRING`.

```
METHOD SET_COMMENTS_STRING
  IMPORTING
    COMMENTS_STRING
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

Im Parameter `COMMENTS_STRING` wird hierbei die Zeichenfolge, die Kommentarzeilen kennzeichnen soll, übergeben. Prinzipiell kann hier jede Zeichenfolge definiert werden, um Kommentarzeilen zu kennzeichnen.

Festlegen, ob Kommentarzeilen hervorgehoben werden sollen

Standardmäßig werden Kommentarzeilen im ABAP-Editor farblich hervorgehoben. Diese Hervorhebung kann jedoch durch einen Aufruf der Methode `SET_HIGHLIGHT_COMMENTS_MODE` ein- bzw. ausgeschaltet werden.

Die Methode erhält – analog zu den Methoden `SET_STATUSBAR_MODE` oder `SET_TOOLBAR_MODE` – entweder den Wert `CL_GUI_TEXTEDIT=>TRUE` oder entsprechend `CL_GUI_TEXTEDIT=>FALSE`, um die Hervorhebungsfunktion entsprechend ein- bzw. auszuschalten.

```
METHOD SET_HIGHLIGHT_COMMENTS_MODE
  IMPORTING
    HIGHLIGHT_COMMENTS_MODE
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1
    INVALID_PARAMTER       = 2.
```

Textzeilen in Kommentarzeilen verwandeln

Die Klasse CL_GUI_TEXTEDIT stellt zwei Methoden zur Verfügung, um Textzeilen in Kommentarzeilen zu verwandeln, indem der Zeile die - mit der Methode SET_COMMENTS_STRING definierte – Zeichenfolge vorangestellt wird.

Mit der Methode COMMENT_LINES können bestimmte Zeilen des Textes in Kommentarzeilen verwandelt werden. Dem Funktionsbaustein wird die Zeilennummer der ersten gewünschten Zeile sowie die Zeilennummer der letzten Zeile, die auskommentiert werden soll, übergeben.

```
METHOD COMMENT_LINES
  IMPORTING
    FROM_LINE
    TO_LINE
    ENABLE_EDITING_PROTECTED_TEXT
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

Wird im Parameter FROM_LINE ein Wert kleiner als eins übergeben, wird mit der ersten Zeile des Textes begonnen. Ebenso wird beim Parameter TO_LINE die letzte Zeile angenommen, wenn der übergebene Index größer ist als die Anzahl Zeilen im Text.

Das TextEdit-Control bietet weiterhin die Möglichkeit, Zeilen gegen Eingaben zu schützen. Auf dieses Feature wird hier jedoch nicht näher eingegangen. Mit dem Parameter ENABLE_EDITING_PROTECTED_TEXT wird festgelegt, ob auch diese Zeilen von der Auskommentierung betroffen sein sollen (CL_GUI_TEXTEDIT=>TRUE) oder ob diese Zeilen bei der Verarbeitung ausgelassen werden sollen (CL_GUI_TEXTEDIT=>FALSE). Wird dieser Parameter nicht übergeben, wird als Defaultwert der Wert FALSE angenommen, d.h. geschützte Zeilen werden nicht mit dem Kommentarzeichen versehen.

Diese Methode wird verwendet, wenn Zeilen über ihre Zeilen-Indizes angesprochen werden können. Mit der Methode COMMENT_SELECTION können die aktuell ausgewählten Zeilen im Editor in Kommentarzeilen verwandelt werden. Die Aufrufschnittstelle dieser Methode ist einfacher als die der Methode COMMENT_LINES, da keine Zeilen-Indizes übergeben werden müssen.

```
METHOD COMMENT_SELECTION
  IMPORTING
    ENABLE_EDITING_PROTECTED_TEXT
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

Für den Parameter ENABLE_EDITING_PROTECTED_TEXT gilt das gleiche wie bei der Methode COMMENT_LINES. Entsprechend werden geschützte Zeilen bei der Aktion berücksichtigt, oder nicht.

Kommentarzeilen in normale Zeilen umwandeln

Analog zu den Methoden zum Einfügen der Kommentarzeichen am Anfang von Textzeilen existieren zwei Methoden, um diese wieder zu entfernen. Entsprechend der Methode COMMENT_LINES gibt es die Methode UNCOMMENT_LINES.

```
METHOD UNCOMMENT_LINES
  IMPORTING
    FROM_LINE
    TO_LINE
    ENABLE_EDITING_PROTECTED_TEXT
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

Die Aufrufschnittstelle der Methode ist identisch mit der des Methode COMMENT_LINES. Ebenso verhält es sich bei der Methode UNCOMMENT_SELECTION als Gegenstück zur Methode COMMENT_SELECTION.

```
METHOD UNCOMMENT_SELECTION
  IMPORTING
    ENABLE_EDITING_PROTECTED_TEXT
  EXCEPTIONS
    ERROR_CNTL_CALL_METHOD = 1.
```

4.3.8 Weitere Funktionen des SAP Textedit

Das TextEdit-Control von SAP bietet dem Programmierer weitere leistungsfähige Optionen zur Textbeeinflussung. Allen voran sei die Möglichkeit, sich auf Ereignisse, die vom Editor ausgelöst werden, zu registrieren und so auch während des Editierens Einfluss auf das Control nehmen zu können.

Die hierfür verwendeten Techniken sind die gleichen, wie beim SAP TreeView. Lediglich die Ereignisse (Events) unterscheiden sich naturgemäß erheblich. Auf eine nähere Betrachtung der hierfür notwendigen Programmiertechniken soll daher an dieser Stelle auf die entsprechenden Abschnitte in Kapitel 3 verwiesen werden. Informationen bezüglich der zur Verfügung stehenden Ereignisse sowie deren Parameter, kann aus dem R/3-System übernommen werden.

Weiterhin stellt das Control umfangreiche Methoden zur Verfügung, um während des Editiervorgangs aus ABAP heraus Einfluss auf das Control zu nehmen. Bei diesen handelt es sich beispielsweise um Methoden, um die aktuelle Selektion zu ermitteln bzw. zu setzten, Textzeilen einzurücken oder um eine Methode zum Suchen und Ersetzen. Auch diese Methoden sollen hier jedoch nicht näher betrachtet werden, da die benötigten Informationen, um diese Optionen zu nutzen, ebenfalls leicht aus dem R/3-System entnommen werden können.

4.4 Verwendete und wichtige Funktionsbausteine

In diesem Abschnitt sollen die wesentlichen Funktionsbausteine der Funktionsgruppe STXD im Überblick dargestellt werden. Die Auflistung enthält – neben den im Verlauf des Kapitels ausführlich vorgestellten Funktionen – weitere Funktionsbausteine, mit denen Textobjekte manipuliert werden können. Diese Zusammenstellung soll jedoch keine vollständige Liste der zur Verfügung stehenden Funktionen darstellen.

4.4.1 COPY_TEXTS

Mit dem Funktionsbaustein COPY_TEXTS können ein oder mehrere Textobjekte innerhalb des R/3-Systems kopiert werden. Die Paramter SAVEMODE_DIRECT, INSERT sowie LOCAL_CAT entsprechen denen des Funktionsbausteins SAVE_TEXT.

Zu beachten ist hierbei, dass Textobjekte, die zu betriebswirtschaftlichen Anwendungsobjekten gehören ohne weiteres kopiert werden können. Allerdings muss die Verknüpfung mit dem jeweiligen Anwendungsobjekt separat hergestellt werden. Der Funktionsbaustein COPY_TEXTS beinhaltet lediglich das Kopieren selbst.

Import-Parameter	
SAVEMODE_DIRECT	Kennzeichen, ob das Textobjekt sofort, d.h. synchron gespeichert werden soll (»X«) oder im Rahmen der Verbuchung der LUW (Space)
INSERT	Kennzeichen, ob Textobjekt neu ist (»X«) oder bereits vorhanden ist und geändert werden soll (Space). Hierdurch kann bei neuen Textobjekten die ansonsten durchgeführte Prüfung übersprungen werden
LOCAL_CAT	Verwendung von lokalen Katalogen
Export-Parameter	
ERROR	Kennzeichen, ob beim Kopieren ein Fehler aufgetreten ist

Tabellen-Parameter	
TEXTS	Tabelle, die die zu kopierenden Texte definiert (ITCTC)
Exceptions	
Keine	

Da der Funktionsbaustein im Verlauf dieses Kapitels nicht näher beschrieben wurde, zeigt Tabelle 4.9 den Aufbau der Kopierstruktur ITCTC. Jede Zeile des Tabellenparameters TEXTS definiert ein zu kopierendes Textobjekt mit dem jeweiligen Zielobjekt. Ein Quelltext kann mehrfach in der Tabelle vorkommen, während ein Zielobjekt nur einmal vorkommen sollte. Im Feld SUBRC der Struktur wird für jeden Kopiervorgang der Returncode entsprechend der Exceptions des Funktionsbausteins SAVE_TEXT abgelegt. Tritt bei mindestens einem Kopiervorgang ein Fehler auf (SUBRC <> 0), liefert der Funktionsbaustein im Parameter ERROR den Wert »X« zurück.

Feldname	Datentyp	Bedeutung
DESTOBJECT	CHAR 10	Objekttyp des Originaltextes
DESTNAME	CHAR 70	Textname des Originaltextes
DESTID	CHAR 4	Text-ID des Originaltextes
DESTLANG	LANG	Sprachschlüssel des Originaltextes
SRCOBJECT	CHAR 10	Objekttyp für Zieltext
SRCNAME	CHAR 70	Textname für Zieltext
SRCID	CHAR 4	Text-ID für Zieltext
SRCLANG	LANG	Sprachschlüssel für Zieltext
SUBRC	INT	Rückgabewert beim Kopieren

Tabelle 4.9
Aufbau der Kopierstruktur ITCTC

4.4.2 DELETE_TEXT

Der Funktionsbaustein DELETE_TEXT löscht ein Textobjekt aus der Datenbank oder dem lokalen Katalog. Das gewünschte Textobjekt wird über die Parameter des Funktionsbausteins eindeutig identifiziert. Wird im Parameter SAVEMODE_DIRECT der Wert«X« übergeben, kann im Parameter TEXTMODE_ONLY festgelegt werden, ob das Textobjekt in der Datenbank oder lediglich im lokalen Katalog gelöscht werden soll.

Import-Parameter	
CLIENT	Mandant
ID	Text-ID des Textobjektes
LANGUAGE	Sprachschlüssel des Textes
NAME	Name des zu löschenden Textes
OBJECT	Objekttyp des zu löschenden Textes
SAVEMODE_DIRECT	Kennzeichen, ob das Textobjekt sofort, d.h. synchron gespeichert werden soll (»X«) oder im Rahmen der Verbuchung der LUW (Space)
TEXTMEMORY_ONLY	Kennzeichen, ob das Textobjekt aus der Datenbank gelöscht werden soll (Space) oder nur aus dem lokalen Katalog
LOCAL_CAT	Verwendung von lokalen Katalogen
Exceptions	
NOT_FOUND	Textobjekt wurde nicht gefunden

4.4.3 EDIT_TEXT

Der Funktionsbaustein `EDIT_TEXT` ist die Aufrufschnittstelle für den R/3-Fullscreen-Editor. Dieser kann sich dem Benutzer entweder als Zeileneditor wie aus älteren R/3-Releaseständen bekannt oder als moderner PC-Editor darstellen. Beim PC-Editor erfolgt die Bearbeitung des Textes vollständig im GUI des Benutzers. Eine Kommunikation mit dem Applikationsserver wird – bei den normalen Editierfunktionen – nicht durchgeführt. Erst bei Beendigung des Editors oder bei Veränderung einer Editor-Einstellung erfolgt das Auslösen der PAI/PBO-Ereignisse.

Import-Parameter	
DISPLAY	Kennzeichen, ob Text editiert (»X«) oder lediglich angezeigt (Space) werden soll
EDITOR_TITLE	Titelzeile des Editors. Ggf. wird dieser um weitere Komponenten erweitert.
HEADER	Kopfstruktur (`THEAD`) des zu bearbeitenden Textobjekts
PAGE	Seite im Formular, auf der der Text dargestellt werden soll
WINDOW	Seitenfenster des Formulars für die Textdarstellung

SAVE	Kennzeichen, ob Textobjekt vom Editor in der Datenbank gespeichert werden soll (»X«) oder ob dies vom aufrufenden Programm durchgeführt wird (Space)
LINE_EDITOR	Kennzeichen, ob der Editor als Zeileneditor gestartet werden soll (»X«) oder als PC-Editor (Space)
CONTROL	Steuerstruktur zur Beeinflussung des Editors (ITCED)
PROGRAM	ABAP-Programm, welches zur Auflösung von Textsymbolen bei der Druckvorschau verwendet werden soll
LOCAL_CAT	Verwendung von lokalen Katalogen
Export-Parameter	
FUNCTION	Aktion, die mit dem Textobjekt im Editor durchgeführt wurde »I« Textobjekt wurde neu erstellt »U« Textobjekt wurde verändert »D« Text wurde gelöscht
NEWHEADER	Kopfstruktur des Textobjekts nach Beendigung des Editors
RESULT	Ergebnisstruktur enthält Informationen über Aktion des Benutzers beim Verlassen des Editors (ITCER)
Tabellen-Parameter	
LINES	Interne Tabelle mit Zeileninformationen des Textobjekts (TLINE)
Exceptions	
ID	Ungültige Text-ID in Kopfstruktur
LANGUAGE	Ungültige Sprache in Kopfstruktur
LINESIZE	Ungültige Zeilenbreite in Kopfstruktur
NAME	Ungültige Textname in Kopfstruktur
OBJECT	Ungültiges Textobjekt in Kopfstruktur
TEXTFORMAT	Ungültiges Textformat für Editor
COMMUNICATION	Kommunikationsfehler

4.4.4 INIT_TEXT

Der Funktionsbaustein INIT_TEXT wird dazu verwendet, um die Arbeitsbereiche für ein Textobjekt zu initialisieren. Die interne Tabelle mit den Textzeilen wird gelöscht sowie die Kopfstruktur mit den bekannten Daten aus der Aufrufschnittstelle vorbelegt.

Import-Parameter	
ID	Text-ID
LANGUAGE	Sprachkennzeichen
OBJECT	Objekttyp des Textobjekts
NAME	Name des Textobjekts
Export-Parameter	
HEADER	Neu initialisierte Kopfstruktur (THEAD)
Tabellen-Parameter	
LINES	Initialisierte Zeilentabelle (TLINE)
Exceptions	
ID	Ungültige Text-ID
LANGUAGE	Ungültiges Sprachkennzeichen
NAME	Ungültiger Textname
OBJECT	Ungültiger Objekttyp

4.4.5 READ_STDTEXT

Mit dem Funktionsbaustein READ_STDTEXT werden Textobjekte mit dem Objekttyp »TEXT« gelesen. Falls der gewünschte Textbaustein in der angegebenen Sprache nicht vorhanden ist, wird bei Bedarf versucht, das Textobjekt in einer Alternativsprache zu lesen. Scheitert auch dies, kann ein Mandantendurchgriff auf den Mandanten 000 des Systems erfolgen, und dort ggf. auch auf das Textobjekt in der Alternativsprache zugegriffen werden. Der Zugriff auf den Standardtext erfolgt somit maximal 4-stufig:

❶ Aktueller Mandant, gewünschte Sprache
❷ Aktueller Mandant, Alternativsprache
❸ Mandant 000, gewünschte Sprache
❹ Mandant 000, Alternativsprache

Import-Parameter	
ID	Text-ID (TDID)
LANGUAGE	Sprachkennzeichen des Textobjekts (TDSPRAS)
NAME	Name des Textobjekts (TDNAME)
USE_AUX_LANGUAGE	Kennzeichen, ob bei Bedarf auf Alternativsprache lt. Tabelle T002C ausgewichen werden soll
USE_THRUCLIENT	Kennzeichen, ob bei Bedarf ein Mandantendurchgriff auf Mandant 000 erfolgen soll
LOCAL_CAT	Verwendung von lokalen Katalogen
Export-Parameter	
HEADER	Kopfstruktur des gelesenen Textobjekts (THEAD)
Tabellen-Parameter	
LINES	Zeilentabelle des Textobjekts (TLINE)-
Exceptions	
ID	Ungültige Text-ID
LANGUAGE	Ungültige Sprache
NAME	Ungültiger Name
NOT_FOUND	Textobjekt zum Schlüssel nicht vorhanden
REFERENCE_CHECK	Verweiskette zwichen Textobjekten ist unterbrochen

4.4.6 READ_TEXT

Mit dem Funktionsbaustein READ_TEXT werden existierende Textobjekte aus der Datenbank gelesen. Die Schnittstelle des Funktionsbausteins enthält alle Felder, die ein Textobjekt in einem R/3-System eindeutig identifizieren.

Wird das Textobjekt nicht gefunden, bricht die Funktion mit einer Exception ab.

Import-Parameter	
CLIENT	Mandant
ID	Text-ID (TDID)
LANGUAGE	Sprachkennzeichen des Textobjekts (TDSPRAS)

NAME	Name des Textobjekts (TDNAME)
OBJECT	Objekttyp des Textobjekts (TDOBJECT)
ARCHIVE_HANDLE	Aktuelle Zugriffsnummer zu einer geöffneten Archivdatei
LOCAL_CAT	Verwendung von lokalen Katalogen
Export-Parameter	
HEADER	Kopfstruktur des gelesenen Textobjekts (THEAD)
Tabellen-Parameter	
LINES	Zeilentabelle des Textobjekts (TLINE)
Exceptions	
ID	Ungültige Text-ID
LANGUAGE	Ungültige Sprache
NAME	Ungültiger Name
NOT_FOUND	Textobjekt zum Schlüssel nicht vorhanden
OBJECT	Objekttyp ungültig
REFERENCE_CHECK	Verweiskette zwichen Textobjekten ist unterbrochen
WRONG_ACCESS_TO_ARCHIVE	Zugriffsnummer zu Archivdatei ist nicht gültig oder Archivdatei ist nicht geöffnet

4.4.7 SAVE_TEXT

Mit der Funktion SAVE_TEXT werden Textobjekte in der Datenbank des R/3-Systems abgelegt. Es kann sich hierbei sowohl um neue Textobjekte als auch bereits bestehende Texte handeln, die mit diesem Funktionsbaustein aktualisiert werden.

Wenn nicht durch andere Mechanismen übersteuert, werden sämtliche Verwaltungsinformationen des zu speichernden Textobjekts aus der übergebenen Kopfstruktur der Aufrufschnittstelle (HEADER) übernommen.

Import-Parameter	
CLIENT	Mandant
HEADER	Kopfstruktur des zu speichernden Textobjekts (THEAD)
INSERT	Kennzeichen, ob Textobjekt neu ist (»X«) oder bereits vorhanden ist und geändert werden soll (Space). Hierdurch kann bei neuen Textobjekten die ansonsten durchgeführte Prüfung übersprungen werden
SAVEMODE_DIRECT	Kennzeichen, ob das Textobjekt sofort, d.h. synchron gespeichert werden soll (»X«) oder im Rahmen der Verbuchung der LUW (Space)
OWNER_SPECIFIED	Kennzeichen, ob der Benutzername für die Kopfstruktur (THEAD) aus der übergebenen Kopfstruktur im Parameter HEADER übernommen werden soll (»X«) oder aus dem Feld SY-UNAME
LOCAL_CAT	Verwendung von lokalen Katalogen
Export-Parameter	
FUNCTION	Aktion, die mit dem Textobjekt durchgeführt wurde »I« Textobjekt wurde neu erstellt »U« Textobjekt wurde verändert »D« Text wurde gelöscht
NEWHEADER	Kopfstruktur des Textobjekts nach dem Sichern
Tabellen-Parameter	
LINES	Zeilentabelle des Textobjekts (TLINE)
Exceptions	
ID	Ungültige Text-ID in der Kopfstruktur
LANGUAGE	Ungültige Sprache in der Kopfstruktur
NAME	Ungültiger Textname in der Kopfstruktur
OBJECT	Ungültiger Objekttyp in der Kopfstruktur

Programmierung SAP-Grafik

5

5.1 Allgemeines zur Grafik

Schon seit längerem besteht in R/3 die Möglichkeit, sowohl Geschäftsgrafiken als auch Strukturgrafiken in die Anwendungen einzubeziehen. Bis zur Einführung des Control Frameworks (siehe Kapitel 3) bestand jedoch lediglich die Möglichkeit, Grafiken durch einen externen Viewer, der optisch an das bekannte SAPGUI angelehnt ist, anzuzeigen. Zur Erstellung dieser Grafiken standen und stehen eine Reihe von Funktionsbausteinen zur Verfügung, deren Verwendung hier auch gezeigt werden soll.

Der Schwerpunkt dieses Kapitels liegt jedoch in der neuen, auf dem Control Framework basierenden Technik, Geschäfts- sowie Strukturgrafiken anzuzeigen. Da die grundlegenden Techniken der Programmierung von Geschäfts- und Strukturgrafiken identisch sind, wird auf eine Darstellung von Strukturgrafiken verzichtet. Der Leser kann sich das hierfür notwendige Wissen aus der Online-Dokumentation zum Release 4.6 sowie dem R/3-System selbst erarbeiten. Wichtiger sind hier die grundlegenden Techniken, deren Dokumentation im System nur teilweise detailliert genug ist.

Die Beispielprogramme dieses Kapitels greifen wiederum auf die Tabellen aus dem Flugbuchungssystem zurück. Für einen – im Beispiel fest vorgegebenen – Flug sollen die Verteilung der Buchungen auf die einzelnen Klassen in Form einer Grafik dargestellt werden. Dies erfolgt zum einen als Balkengrafik der absoluten Werte, aber auch als Tortengrafik, um die Verteilung innerhalb des Fluges darzustellen.

5.2 Grafiken über die Funktionsbausteinschnittstelle

Bisher konnten Geschäftsgrafiken lediglich über die Anzeige in einem externen Viewer auf dem Präsentationsrechner angezeigt werden. Die Programmierung dieser Darstellung erfolgte durch den Aufruf eines Funktionsbausteins. SAP stellt hierfür eine ganze Reihe von Funktionsbausteinen zur Verfügung, die allesamt in der Funktionsgruppe BUSG zusammengefasst sind.

- GRAPH_2D
- GRAPH_3D
- GRAPH_MATRIX_2D
- GRAPH_MATRIX_3D
- GRAPH_MATRIX_4D
- GRAPH_MATRIX

Diese Funktionsbausteine dienen allesamt dazu, Geschäftsgrafiken anzuzeigen. Unterschiedlich ist nur die Aufrufschnittstelle, die bei den ersten beiden Funktionen einfacher gehalten ist als bei den übrigen, die mit dem Präfix GRAPH_MATRIX_ beginnen. Dadurch sind sie jedoch auch erheblich unflexibler in Bezug auf die Einsatzmöglichkeiten. Ansonsten sprechen die Namen der Funktionsbausteine für sich.

In diesem Abschnitt sollen exemplarisch für alle Bausteine der Funktionsgruppe die Bauteine GRAPH_2D als einfache Variante und GRAPH_MATRIX_2D vorgestellt werden. Innerhalb dieser Bausteine wird wiederum nur ein kleiner Ausschnitt der Möglichkeiten dargestellt. Auf die Darstellung der interaktiven Manipulationsmöglichkeiten wurde beispielsweise vollständig verzichtet. Auch soll nicht jeder Grafiktyp dargestellt werden. Aus der Vielzahl der Variationen soll lediglich die Tortengrafik verwendet werden. Die übrigen Typen werden jedoch sehr ähnlich angesprochen.

5.3 Geschäftsgrafiken mit GRAPH_2D

Die Bausteine GRAPH_2D, der hier gezeigt werden und GRAPH_3D, bieten dem Programmierer eine vereinfachte Möglichkeit zur Anzeige von Geschäftsgrafiken in dem Fall, dass der Aufbau der Datentabelle einer bestimmten Struktur entspricht, sowie keine wesentliche Einflussnahme auf die Gestaltung der Grafik genommen werden soll.

Wird die Aufrufschnittstelle um die internen Parameter sowie die Parameter zur Realisierung interaktiver Wertveränderungen reduziert, bietet sich dem Programmierer hier ein kompakter Funktionsbaustein.

```
FUNCTION GRAPH_2D
   IMPORT
      DISPLAY_TYPE
      MAIL_ALLOW
      SMFONT
      SO_CONTENTS
      SO_RECEIVER
      SO_SEND
      SO_TITLE
      TITL
      VALT
      WINPOS
      WINSZX
      WINSZY
   TABLES
      DATA
   EXCEPTIONS
      GUI_REFUSE_GRAPHIC.
```

Mit den hier gezeigten Parametern kann eine Business-Grafik angezeigt werden. Weiterhin kann die Grafik entweder automatisch oder vom Benutzer initiiert als Mail über SAPOffice, die Mailkomponente des Systems R/3 (siehe Kapitel 10), versandt werden. Hierfür steht der Parameter `MAIL_ALLOW` sowie die Parameter, die mit dem Präfix `SO_` beginnen zur Verfügung.

Eine vollständige Darstellung der Aufrufschnitttstelle des Funktionsbausteins `GRAPH_2D` ist am Ende des Kapitels in Abschnitt 5.8.1 zu finden.

5.3.1 Anzeige einer Grafik

Die Anzeige einer Grafik soll in diesem Abschnitt an einem Beispiel verdeutlicht werden. Verwendet werden die, bereits aus Kapitel 3 bekannten, Flugbuchungen. Die Buchungen zu einem Flug sind hierbei in der Tabelle `SBOOK` abgelegt. Um das Programm einfach zu halten, wird der Flug, dessen Buchungen ausgewertet werden sollen, am Anfang des Programms konstant definiert.

```
REPORT  ZFLIGHTGRAPH.

* Tabellendeklarationen
TABLES SBOOK.      "Flugbuchung

* Konstantendefinition des auszuwertenden Fluges
CONSTANTS: C_CARRID   LIKE SBOOK-CARRID  VALUE 'AZ',
           C_CONNID   LIKE SBOOK-CONNID  VALUE '0555',
           C_FLDATE   LIKE SBOOK-FLDATE  VALUE '19991113'.
```

Der Funktionsbaustein GRAPH_2D erhält die anzuzeigenden Daten in Form einer internen Tabelle. Der Aufbau dieser Tabelle ist nicht exakt vorgegeben. Festgelegt ist jedoch, dass die erste Spalte der Tabelle ein Zeichenfeld mit der Bezeichnung des jeweiligen Datenfeldes sein muss. Die zweite Spalte der Tabelle muss ein Zahlentyp (I, P oder F) sein. Sie enthält den jeweils darzustellenden Wert selbst. Die Namen der Spalten spielen hierbei keine Rolle.

```
* Datendefinitionen
DATA: BEGIN OF G_DATA OCCURS 0,
        CLASS(20)     TYPE C,
        BOOKINGS      TYPE I,
      END OF G_DATA.
```

Der eigentliche Verarbeitungsteil des Programms besteht aus zwei Teilen. Zunächst müssen die darzustellenden Informationen aus der Datenbank gelesen werden und in der zuvor deklarierten internen Tabelle abgelegt werden.

Dies erfolgt mit einer SELECT-Anweisung auf die Tabelle SBOOK, wobei die Anzahl der Buchungssätze zum gewünschten Flug, gruppiert nach der gebuchten Klasse, ermittelt werden.

```
START-OF-SELECTION.
* Datenbasis laden
  SELECT CLASS COUNT(*)
         INTO  (G_DATA-CLASS, G_DATA-BOOKINGS)
         FROM  SBOOK
         WHERE CARRID = C_CARRID AND
               CONNID = C_CONNID AND
               FLDATE = C_FLDATE
         GROUP BY CLASS.
    APPEND G_DATA.
  ENDSELECT.
```

Zuletzt erfolgt die Anzeige der gelesenen Daten über den Funktionsbaustein GRAPH_2D.

```
* Grafik anzeigen
  CALL FUNCTION 'GRAPH_2D'
    EXPORTING
*     DISPLAY_TYPE            = '  '
*     SMFONT                  = ' '
      TITL                    = 'Buchungen Flug AZ555 am 13.11.1999'
      VALT                    = 'Gebuchte Plätze'
    TABLES
      DATA                    = G_DATA
    EXCEPTIONS
      GUI_REFUSE_GRAPHIC      = 1
      OTHERS                  = 2.
```

Im ersten Parameter des Bausteins wird der Typ der Geschäftsgrafik festgelegt, der zur Anzeige verwendet werden soll. Die Aufrufschnittstelle von GRAPH_2D definiert einen weiteren Parameter namens TYPE, mit dem dies auch möglich ist, der aber laut SAP nicht für diesen Zweck verwendet werden soll. Tabelle 5.1 zeigt die für 2-dimensionale Geschäftsgrafiken zulässigen Werte des Parameters DISPLAY_TYPE.

DISPLAY_TYPE	**Grafik**
VB	Vertikale Balken
VS	Vertikale gestapelte Balken
HB	Horizontale Balken
HS	Horizontale gestapelte Balken
TD	Perspektivische Balken
VT	Vertikale Dreiecke
ST	Stufenlinien
MS	Stufenflächen
LN	Linien
SA	Gestapelte Flächen
MA	Maskierte Flächen
PI	Tortendiagramm
TP	Perspektivisches Tortendiagramm
PO	Polardiagramm
PP	Relatives Polardiagramm

Tabelle 5.1
Zulässige Grafiktypen für 2D-Geschäftsgrafiken

Über den Parameter SMFONT kann festgelegt werden, dass bei Bedarf eine kleinere Schrift innerhalb der Darstellung der Grafik verwendet wird. In Versuchen mit dem Funktionsbaustein konnte jedoch kein Unterschied bei der Verwendung dieses Parameters festgestellt werden.

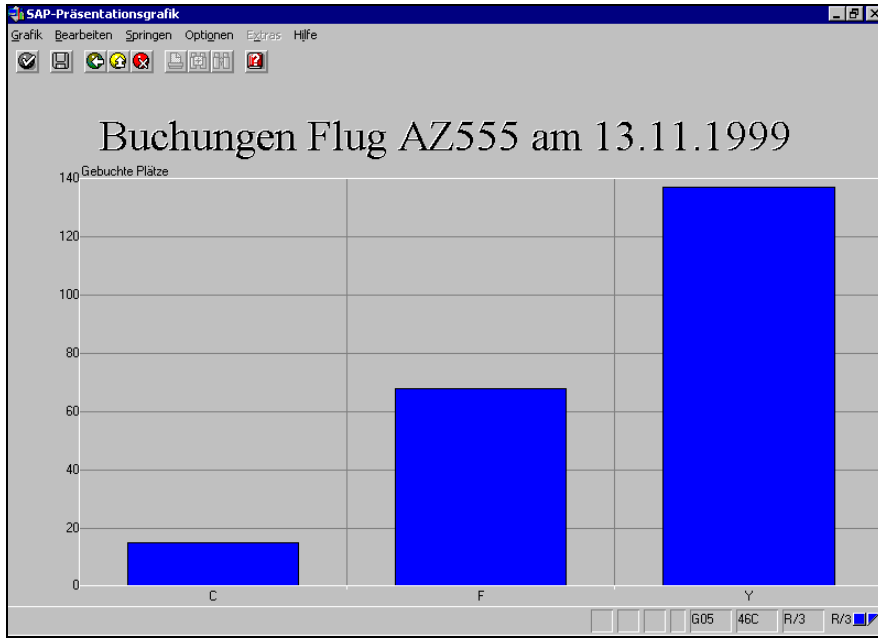

Abbildung 5.1
2D-Balkengrafik im externen Viewer (© SAP AG)

Der Grafik-Viewer verwendet die im Aufrufparameter `TITL` übergebene Zeichenkette als Überschrift der Grafik. Diese erscheint immer zentriert über der eigentlichen Grafikdarstellung im Viewer. Der Parameter `VALT` definiert die Bezeichnung der Skala der Grafik, d.h. der Y-Achse bei den 2-dimensionalen Darstellungen.

Wird das Programm gestartet, erscheint über dem normalen SAPGUI-Fenster das Fenster des Grafik-Viewers mit der in Abbildung 5.1 dargestellten Grafik.

Da kein Grafiktyp für die Anzeige der Grafik definiert wurde, verwendet das System den Defaulttyp, der in diesem Fall vertikale Balken sind. Wird im Feld `DISPLAY_TYPE` beispielsweise der Wert »PI« übergeben, verändert sich die Anzeige wie in Abbildung 5.2 dargestellt.

Das System rechnet hierbei die anzuzeigenden Werte automatisch um, da eine Tortengrafik - im Gegensatz zu Balkengrafik - keine absoluten Werte sondern lediglich Anteile darstellen kann.

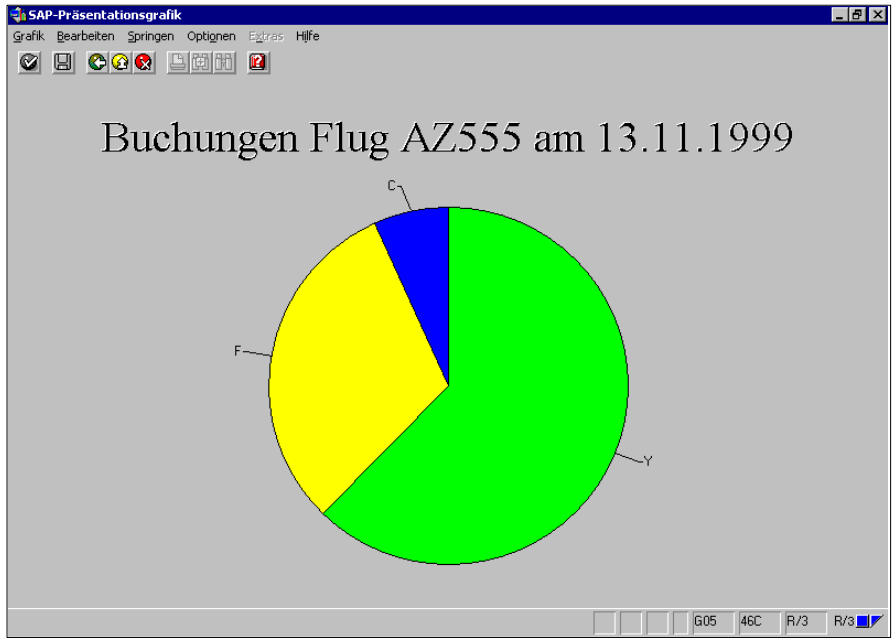

Abbildung 5.2
2D-Tortengrafik im externen Viewer (© SAP AG)

Neben der Überschrift der Grafik sowie der Bezeichnung der Y-Skala hat der Programmierer bei Verwendung des Bausteins `GRAPH_2D` ausschließlich die Möglichkeit, den verwendeten Grafiktyp zu bestimmen. Weitere Einflussmöglichkeit auf die Darstellung der Grafik stehen hier nicht zur Verfügung. Es besteht lediglich noch die Möglichkeit, zu steuern, wie die Grafik in Form einer Mail versendet werden kann, sowie die Größe und Position des Viewer-Fensters auf dem Bildschirm des Betrachters.

5.3.2 Größe und Position des Viewer-Fensters

Die Größe und die Position des Grafik-Viewers auf dem Bildschirm des Benutzers wird über die drei Parameter `WINPOS`, `WINSZX` sowie `WINSZY` der Aufrufschnittstelle von `GRAPH_2D` definiert.

Der Parameter `WINPOS` enthält hierbei einen Wert zwischen »1« und »9«, um die gewünschte Position des Fenster auf dem Bildschirm zu definieren. Welchen Effekt die jeweilgen Werte bewirken sollen, ist in Tabelle 5.2 dargestellt.

WINPOS	horizontale Ausrichtung	vertikale Ausrichtung
1	links	oben
2	zentriert	oben
3	rechts	oben
4	links	zentriert
5	zentriert	zentriert
6	rechts	zentriert
7	links	unten
8	zentriert	unten
9	rechts	unten

Tabelle 5.2
Bedeutung des Parameters WINPOS

Wird keiner der dargestellten Werte übergeben, wird der Viewer jeweils 1/3 der Bildschirmgröße vom unteren bzw. linken Bildschirmrand entfernt angezeigt.

Die Werte in den Parametern WINSZX bzw. WINSZY definieren die prozentuale Ausdehnung des Viewerfensters zur Auflösung des Bildschirm des Benutzers.

Beispiel:

Es wird von einer Bildschirmgröße von 800x600 Bildpunkten ausgegangen. Werden in den Parametern WINPOS, WINSZX und WINSZY die Werte »5«, »50« und »50« an den Funktionsbaustein übergeben, wird der Viewer in der Größe 400x300 Bildpunkten zentriert auf dem Bildschirm des Benutzers ausgegeben.

5.3.3 Grafik über SAPOffice versenden

Die erstellen Grafiken können über das Mailsystem SAPOffice des R/3-Systems, welches im Kapitel 10 näher beschrieben ist, an andere Benutzer versendet werden. Hierfür stellt der Funktionsbaustein die Parameter MAIL_ALLOW, SO_CONTENTS, SO_RECEIVER, SO_SEND sowie SO_TITLE in der Aufrufschnittstelle zur Verfügung.

Wird im Parameter MAIL_ALLOW der Wert »X« übergeben, werden im Viewer die entsprechenden Funktionscodes zum Versenden der Grafik als Mail aktiviert. Wird dies vom Benutzer verwendet, hat er die Möglichkeit, den Titel und Inhalt sowie die Empfänger der Mail im Dialog zu bestimmen.

Der Mailversand kann auch ohne vorherige Anzeige der Grafik auf dem Bildschirm durchgeführt werden. Hierfür muss im Parameter SO_SEND der Wert »X«

übergeben werden. Das System generiert daraufhin eine Mail mit dem im Parameter SO_TITLE definierten Titel. Die im Parameter SO_CONTENTS übergebene Zeichenkette wird zusätzlich zur Grafik in die Mail aufgenommen. Schließlich kann über den Parameter SO_RECEIVER der Benutzername des R/3-Benutzers, der die Mail erhalten soll, festgelegt werden.

5.4 Geschäftsgrafiken mit GRAPH_MATRIX_2D

Wie bereits erwähnt hat der Benutzer bei der Verwendung des Funktionsbausteins GRAPH_2D lediglich sehr eingeschränkte Möglichkeiten, auf die Gestaltung der Grafik im Viewer-Fenster Einfluss zu nehmen. Reichen die im Abschnitt 5.3 dargestellten Möglichkeiten nicht aus, oder können die grafisch darzustellenden Informationen nicht in der gewünschten Form übergeben werden, muss zu einem der weitaus flexibleren Funktionsbauseine mit dem Namenspräfix GRAPH_MATRIX_ gewechselt werden. Im Beispiel handelt es sich um den Baustein GRAPH_MATRIX_2D.

Die, abermals um nicht benötigte Parameter bereinigte, Aufrufschnittstelle hat im Gegensatz zum im vorigen Abschnitt gezeigten Funktionsbaustein keine Felder, um den Typ der Geschäftsgrafik festzulegen. Dafür sind die beiden Parameter NCOL sowie NROW neu hinzugekommen.

Ebenfalls neu sind die Tabellenparameter OPTS sowie TCOL.

```
FUNCTION GRAPH_MATRIX_2D
   IMPORT
      MAIL_ALLOW
      NCOL
      NROW
      SMFONT
      SO_CONTENTS
      SO_RECEIVER
      SO_SEND
      SO_TITLE
      TITL
      VALT
      WINPOS
      WINSZX
      WINSZY
   TABLES
      DATA
      OPTS
      TCOL
   EXCEPTIONS
      COL_INVALID
      OPT_INVALID.
```

Anders als beim Funktionsbaustein GRAPH_2D kann die Datentabelle für diesen Funktionsbaustein mehrere Datenspalten enthalten. Die erste Spalte der Tabelle muss auch hier eine Zeichenkette sein und bezeichnet den Namen der Spalte. Daran schließen sich ein oder mehrere numerische Spalten an. Die Datentabelle bildet somit ein zweidimensionales Feld numerischer Werte. Bei der Anzeige kann der Funktionsbaustein angewiesen werden als Basis entweder alle Werte einer Zeile oder alle Werte einer Spalte zur Anzeige zu bringen. Dies wird über die Parameter NCOL bzw. NROW der Aufrufschnittstelle gesteuert. Die Spalten bzw. Zeilen werden hierbei mit 1 beginnend durchnummeriert.

Werden beide Parameter versorgt, so hat der Parameter NCOL Vorrang, und es werden die Werte einer Spalte über alle Zeilen der Tabelle grafisch aufbereitet. Um dies zu demonstrieren, wird die interne Tabelle um eine weitere Spalte, in der das durchschnittliche Gepäckgewicht je Klasse abgelegt wird, ermittelt. Tatsächlich macht es in der Regel keinen Sinn, die Anzahl der Flugbuchungen auf der gleichen Skala aufzutragen, wie das durchschnittliche Gepäckgewicht je Buchung, aber um das Prinzip zu verdeutlichen soll das Beispiel genügen.

Um die notwendige Datenbasis zu schaffen, wurde die Tabelle G_DATA aus dem vorigen Abschnitt um das Feld AVG_LUGGWEIGHT erweitert. Ebenfalls erweitert wurde die SELECT-Anweisung, um die entsprechenden Daten aus der Datenbank zu ermitteln.

```
...
* Datentabelle
DATA: BEGIN OF G_DATA OCCURS 0,
        CLASS(20)      TYPE C,
        BOOKINGS       TYPE I,
        AVG_LUGGWEIGHT TYPE F,
      END OF G_DATA.

START-OF-SELECTION.
* Datenbasis laden
  SELECT        CLASS
                COUNT( * )
                AVG( LUGGWEIGHT )
         INTO  (G_DATA-CLASS,
                G_DATA-BOOKINGS,
                G_DATA-AVG_LUGGWEIGHT)
         FROM  SBOOK
         WHERE CARRID = C_CARRID AND
               CONNID = C_CONNID AND
               FLDATE = C_FLDATE
         GROUP BY CLASS.
    APPEND G_DATA.
  ENDSELECT.
```

Aus der so ermittelten Datenbasis soll die erste Zeile als Grafik dargestellt werden. Hierfür wird beim Aufruf des Funktionsbausteins GRAPH_MATRIX_2D der Parameter NROW mit dem Wert »1« belegt und der Parameter NCOL nicht übergeben.

```
  ...
* Grafik anzeigen
  CALL FUNCTION 'GRAPH_MATRIX_2D'
    EXPORTING
*     NCOL              = ' '
      NROW              = 1
      TITL              = 'Flug AZ555 - Avg. Gepäckgewicht'
      VALT              = '[Bookings|kg]'
    TABLES
      DATA              = G_DATA
      OPTS              = G_OPT
      TCOL              = G_TCOL
    EXCEPTIONS
      COL_INVALID       = 1
      OPT_INVALID       = 2
      OTHERS            = 3.
```

Das Zeichenfeld am Anfang der Zeile wird hierbei ignoriert, da sie nicht die Spaltenüberschriften der jeweiligen Spalten der Zeile enthalten kann. Aus diesem Grund akzeptiert der Funktionsbaustein den neuen Tabellenparameter TCOL, in dem die Überschriften und die Anzahl der Spalten definiert werden, die für die Datenzeile angezeigt werden sollen.

Vor dem Aufruf des Funktionsbausteins müssen jedoch im Deklarationsteil des Programms die interen Tabellen G_OPT und G_TCOL definiert werden.

```
...
* Optionen
DATA: BEGIN OF G_OPT OCCURS 0,
        OPTION(20)     TYPE C,
      END OF G_OPT.

* Überschriften
DATA: BEGIN OF G_TCOL OCCURS 0,
        HEADER(20)     TYPE C,
      END OF G_TCOL.
...
```

Der Aufbau dieser Tabellen ist nicht weiter vorgegeben. Zeichenketten bieten sich aufgrund der abzulegenden Informationen allerdings an.

Die Überschriften der beiden Spalten, die in der Grafik angezeigt werden sollen, werden vor dem Aufruf des Funktionsbausteins in der internen Tabelle G_TCOL abgelegt.

```
   ...
* Spaltenüberschriften setzen
  G_TCOL-HEADER = 'Buchungen'.
  APPEND G_TCOL.
  G_TCOL-HEADER = 'Gepäck Avg.'.
  APPEND G_TCOL.
  ...
```

Somit ist der Funktionsaufruf funktionsfähig. Da kein Grafiktyp definiert wurde, verwendet das System – wie beim Baustein GRAPH_2D auch – die 2-dimensionale Balkengrafik.

An diesem Punkt kommt der noch nicht erwähnte Tabellenparameter OPT ins Spiel. Hier können die Anzeigeattribute der Grafik in weiterem Maße beeinflusst werden, als es beim Baustein GRAPH_2D möglich ist.

Um die in Abbildung 5.3 dargestellte Ausgabe zu erhalten, muss durch Setzen von einigen Parametern Einfluss auf die Grafikausgabe genommen werden.

```
...
* Optionen setzen
  G_OPT-OPTION  = 'P2TYPE = TD'.
  APPEND G_OPT.
  G_OPT-OPTION  = 'P2WIDT = 1'.
  APPEND G_OPT.
  G_OPT-OPTION  = 'FIFRST = 2D'.
  APPEND G_OPT.
  ...
```

Die Optionen werden hierbei immer in der Form

```
<Option> = <Wert>
```

gesetzt.

Zunächst wird über die Option P2TYPE der Typ der Grafik festgelegt. Auf die einzelnen Grafiktypen wird auch hier über die bereits in Tabelle 5.1 gezeigten Kürzel zugegriffen. Anschließend wird die Breite der einzelnen Balken festgelegt. Der hier gezeigte Wert »1« entspricht einer geringen Balkenbreite. Gültige Werte für diese Option sind die Zahle 1 bis 5. Zuletzt wird über die Option FIFRST die Erstdarstellung der Grafik festgelegt. Hier sind die in Tabelle 5.3 dargestellten Werte definiert. Allerdings sind nicht alle Werte in jedem Zusammenhang gültig. Die dreidimensionale Darstellung ist beispielsweise bei Verwendung des Funktionsbausteins GRAPH_MATRIX_2D nicht zugelassen.

FIFRST	Bedeutung
2D	2D-Darstellung
3D	3D-Darstellung
GP	Gruppendarstellung
PU	Übersichtsdarstellung

Tabelle 5.3
Mögliche Werte für die Option FIFRST

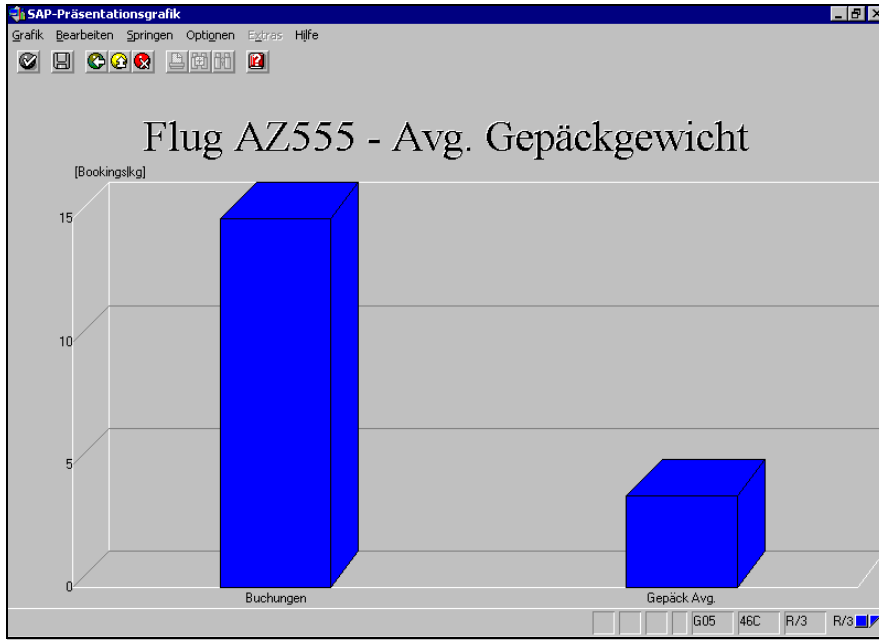

Abbildung 5.3
Perspektivische Balken mit GRAPH_MATRIX_2D (© SAP AG)

SAP definiert eine ganze Reihe dieser Schalter. Tabelle 5.4 zeigt alle zulässigen Optionen im Überblick. Hierbei sind alle Optionen, auch die, die beispielsweise nur für 3D-Grafiken sinnvoll sind, aufgeführt.

Option	Bedeutung
FIFRST	Erstdarstellung der Grafik. Mögliche Werte siehe Tabelle 5.3.
FISTK2	2D-Erstdarstellung ist gestapelt
FISTK3	3D-Erstdarstellung ist gestapelt
P2TYPE	Grafiktyp der 2D-Grafik. Mögliche Werte siehe Tabelle 5.1.
P2PCNT	2D-Grafik zeigt Prozentwerte
P2SCAL	Skalenlinien anzeigen
P2WIDT	Breite der Balken (1-5)
P2ATEX	Verwendung alternativer Texte
P2ALCR	Verwendung alternativer Farben
P2BACK	Hintergrundfarbe bei 2D-Grafiken
P3TYPE	3D-Grafiktyp TO Türme PY Pyramiden ED Wände WE Keile NET Flächen LI Linien
P3TORD	Zeilen und Spalten vertauscht anzeigen
P3FZER	Negative Werte invertiert anzeigen
P3SIDE	Anzahl der Seiten der grafischen Elemente (3-8)
P3SIZE	Größe der Grundfläche der grafischen Elemente (1-5)
P3SCAL	Skalenlinien an den 3D-Objekten anzeigen
P3LINS	Skalenlinien am 3D-Bezugsrahmen anzeigen
P3CTYP	Art der Färbung der grafischen Objekte PL einfarbig SI nach Seitenflächen RO nach Zeile CO nach Spalte
DTDORD	Reihenfolge der Dimensionen

DTPEAK	Daten zoomen
DTRSTK	gestapelte Anzeigen umdrehen
DTSUPZ	Null-Werte unterdrücken
DTINTR	Grafikpunkte interpolieren
D3REFL	Zeilen/Spalten vertauschen
D3RDM1	Zeilen umdrehen
D3RDM2	Spalten umdrehen
SCDECC	Dezimaltrennzeichen
SCENGR	technische Schreibweise der Skalenbeschriftung
SCUNIT	Skalierungseinheiten WD Wort EE Zehnerpotenz EU Technisch
TIFULL	Volle Überschrift
TISIZE	Größe der Überschrift (1-3)
CLPALT	Farbscheme (A-F)
CLBACK	Hintergrundfarbe
CLPAPR	Hintergrund schwarz (»B«) oder Weiß (»W«)

Tabelle 5.4
Optionen für Funktionsbausteine GRAPH_MATRIX_

Nicht jede Kombination der Optionen ist sinnvoll und zulässig. Welche Optionen kombinierbar sind, muss im Einzelfall durch Probieren herausgefunden werden.

In den übrigen Punkten unterscheidet sich die Funktionalität der GRAPH_MATRIX-Bausteine nicht von denen der GRAPH-Bausteine. Auch hier kann die Position und die Größe des Viewer-Fensters über die gleichen Parameter der Aufrufschnittstelle festgelegt werden, wie bei GRAPH_2D. Ebenso sind die Möglichkeiten, die generierte Grafik in Form einer Mail zu versenden mit den im vorigen Abschnitt beschriebenen Optionen identisch.

Der Hauptunterschied zwischen den beiden hier exemplarisch vorgestellten Baustein-Typen liegt demnach in der Struktur der Datentabelle sowie in den Möglichkeiten, auf die Darstellungsparameter der erzeugten Grafik Einfluss zu nehmen.

5.5 Grafiken mit der Controls Technology

5.5.1 Überblick

Mit Einführung des Control Frameworks wurde seitens SAP auch eine neue Möglichkeit der Darstellung von Geschäfts- und Strukturdaten etabliert. Während diese bisher nur in externen Viewern angezeigt werden konnten, die sich optisch zwar am SAPGUI orientierten, jedoch eigenständige Applikationen waren, werden bei der neuen Möglichkeit zur Anzeige von Grafiken die Techniken des Control Frameworks verwendet.

Im Einzelnen bedeutet dies, dass die Grafiken in Containern, welche wiederum in Customer-Control-Areas innerhalb der normalen Dynpros angelegt werden, angezeigt werden. Diese Technik wurde in diesem Buch bereits in Kapitel 3 zur Anzeige von Baumdarstellungen eingeführt und in Kapitel 4 bei der Bearbeitung von Texten aufgegriffen.

Grafiken stellen daher ein weiteres wichtiges Anwendungsgebiet der Container-Technik dar, nutzen diese aber auf eine etwas andere Art, als dies bisher dargestellt wurde.

Wie bei den anderen Controls auch, werden hier Customer-Areas auf dem Dynpro angelegt, welche zur Laufzeit von so genannten Container-Objekten verwaltet werden. Diese Container stellen lediglich den Ausführungsrahmen für die jeweiligen Controls dar, egal, ob es sich hierbei um einen TreeView, ein Text-Edit oder eine Geschäftsgrafik handelt.

Bei den aus der ABAP-Welt heraus sichtbaren Klassen handelt es sich jedoch nicht um die tatsächlichen grafischen Komponenten. Diese laufen – entweder als JavaBeans oder als ActiveX-Controls – auf dem Frontend-Rechner des Benutzers ab (siehe Abbildung 5.4). Die ABAP-Klassen, die in diesem Umfeld beteiligt sind, stellen lediglich Platzhalter (Proxies) innerhalb der ABAP-Welt zur Kommunikation mit diesen entfernt ablaufenden Komponenten dar.

Die Aufgaben dieser Proxy-Klassen beschränken sich hierbei darauf, auf der einen Seite die Kommunikation mit den entfernt ablaufenden visuellen Komponenten durchzuführen und auf der anderen Seite eine Schnittstelle für die anzuzeigenden Informationen sowie die Parametriesierung der Grafik durch den Programmierer zur Verfügung zu stellen. Aus dem Gesagten geht bereits hervor, dass zum Aufgabenbereich der Proxy-Klassen nicht die Verwaltung der anzuzeigenden Informationen selbst gehört. Diese obliegt den so genannten Daten-Containern, welchen neben der Verwaltung der Daten auch die Publizierung von Datenänderungen obliegt.

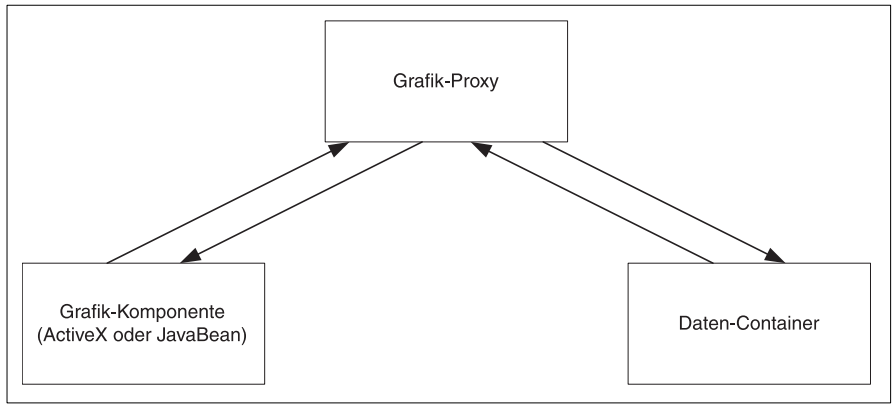

Abbildung 5.4
Grundlegende Architektur der komponentenbasierten Grafik

Gründe für diese Aufgabenteilung sind zum einen die semantischen Unterschiede in den Aufgaben, zum anderen auch eine höhere Flexibilität bezüglich des Aufbaus der Anwendungsdaten selbst. Das Proxy-Objekt bekommt lediglich Kenntnis darüber, auf welche Art und Weise die anzuzeigenden Daten aus dem Container extrahiert werden können, nicht jedoch, wie die Informationen innerhalb des Containers verwaltet werden.

Ein anderer wichtiger Grund für die vollzogene Trennung zwischen dem Datenmodell und der Anzeige liegt darin begründet, dass ein Datenbestand unter Umständen gleichzeitig in mehreren Grafiken angezeigt wird. Durch die Trennung von Daten und Anzeige können beliebig viele Anzeige-Objekte gleichzeitig auf denselben Datenbestand zugreifen, die Anzeigeobjekte selbst laufen jedoch auf einem anderen Rechner ab.

In der Objektorientierung ist diese Architektur, die eine Teilung zwischen dem tatsächlichen Anzeigeobjekt und einer Repräsentation dessen, im lokalen Kontext unter dem Namen Proxy-Pattern bekannt. Nähere Informationen zu diesem Muster und dessen Anwendung kann aus dem Buch »Design Patterns« von Erich Gamma und anderen[1] nachgelesen werden.

Aus dieser Architektur werden die im Folgenden beschriebenen Schritte, die zur Anzeige von Grafiken über das Control Framework durchlaufen werden müssen, deutlich.

Zunächst muss für die anzuzeigenden Informationen ein Daten-Container erstellt werden, der die Daten selbst enthält und den Zugriff auf diese definiert. Weiterhin muss auf einem Dynpro eine Custom Control-Area angelegt werden und im ABAP-Programm ein Container-Objekt zu dessen Verwaltung etabliert

1. Gamma, Erich, et al.: Design Patterns. Elements of Reusable Object-Oriented Software, 1. Aufl., Reading (Massachusetts): Addison-Wesley Publishing Company, 1995.

werden. Zuletzt wird eine Instanz eines Grafik-Proxies zur Verknüpfung dieser beiden Elemente erstellt.

In dieser Reihenfolge soll daher auch bei der Entwicklung des Beispielprogramms dieses Kapitels vorgegangen werden.

SAP unterscheidet in diesem Zusammenhang zwischen Geschäfts- und Strukturgrafiken. Diese spiegelt sich lediglich in unterschiedlichen Implementierungen der gleichen Konzepte wieder. Im Folgenden wird daher auf Strukturgrafiken nicht weiter eingegangen.

5.5.2 Daten-Container

Die Aufgabe der Daten-Container ist es, die Anwendungsdaten zu verwalten, die in der oder den Grafiken angezeigt werden können. Aus diesem Grund müssen Daten-Container-Klassen das ABAP-Objects-Interface `IF_DC_MANAGEMENT`, in welchem der Zugriff der ABAP-Anwendung auf den Daten-Container definiert ist, implementieren. Diese Schnittstelle beinhaltet Methoden, um den Container selbst zu verwalten

Sowohl die Applikation, als Lieferant der Daten, als auch die Grafik-Proxies, die die Informationen aus den Daten-Containern auslesen müssen und zur Ausgabe an die Viewer-Klassen weiterleiten, benötigen Zugriff auf die im Container enthaltenen Informationen. Um diesen beiden Parteien gleichermaßen Zugriff auf die Daten des Containers zu ermöglichen, müssen Container das ABAP-Interface `IF_DC_ACCESS` implementieren, welches diese Zugriffe regelt.

Wenn sich die Daten im Daten-Container verändern, wird diese Umstand automatisch an alle beim Container registrierten Proxy-Objekte gemeldet. Die Daten des Containers können hierbei sowohl von der Anwendung, als auch aus einem Viewer heraus modifiziert werden. Um den Vorgang der Registrierung und Deregistrierung zu standardisieren, müssen Container daher das Interface `IF_DC_SUBSCRIPTION` implementieren.

Alle ABAP-Klassen, die diese drei Interfaces implementieren, können als Daten-Container fungieren. Da die Implementierung der Schnittstellen jedoch unter Umständen sehr aufwendig ist, bietet SAP zum einen die Möglichkeit, einen generischen Daten-Container zu verwenden, und zum anderen die Möglichkeit, basierend auf einer im Data Dictionary definierten Struktur, einen eigenen Container generieren zu lassen. Hierzu steht im System der »Data Container Wizard« in Form des Reports `GFW_DCWIZARD` zur Verfügung. Daten-Container kapseln also die Dictionary-Strukturen und definieren eine funktionale Schale um die ansonsten funktionslosen Strukturen. Das Modell sieht vor, dass diese funktionale Schale aus den oben genannten drei Elementen besteht.

Im Folgenden sollen kurz die Kernfunktionalitäten der drei genannten Interfaces dargestellt werden.

Zugriff auf die Daten im Daten-Container (IF_DC_ACCESS)

Das Interface IF_DC_ACCESS definiert Methoden, um auf die im Daten Container abgelegten Datenobjekte zuzugreifen. Ein Datenobjekt ist in diesem Zusammenhang eine Ausprägung der, dem Container zugrunde liegenden Dictionary-Struktur.

Das Interface definiert sowohl Methoden, um schreibend (SET_VALUE) als auch lesend auf den Container zuzugreifen (GET_VALUE). Weiterhin können einzelne oder alle Objekte des Containers gelöscht werden (DEL_OBJECT, CLEAR).

Das Auslesen der Objekte des Containers kann auch sequentiell über die Methoden GET_PREDECESSOR bzw. GET_SUCCESSOR erfolgen. Änderungen an den Daten im Container können über die Methode GET_CHANGES ermittelt werden.

Innerhalb eines Containers muss jedes gespeicherte Objekt eine ID haben, über die, innerhalb der im Container gespeicherten Objekte, eindeutig auf das Objekt zugegriffen werden kann.

Die Proxy-Klassen bieten die Möglichkeit, die Werte innerhalb des Containers zu gruppieren und somit bei Bedarf mehrere Wertreihen innerhalb eines Diagramms anzuzeigen. Weiterhin kann ein Feld innerhalb der Daten-Container-Struktur als Filter definiert werden, um bei Bedarf nur auf einen Teil der Informationen des Daten-Containers zuzugreifen.

Verwaltung des Containers (IF_DC_MANAGEMENT)

Das Interface IF_DC_MANAGEMENT definiert zum einen die Erzeugung (INIT) und Zerstörung (FREE) von Containern, zum anderen auch die Methoden zum Anstoßen der Verteilung von Änderungen an die registrierten Grafik-Proxies (DISTRIBUTE_CHANGES) und zur persistenten Verwaltung von Daten Containern.

Registrierung auf Datenänderungen (IF_DC_SUBSCRIPTION)

Dieses Interface beinhaltet lediglich zwei Methoden, die ausschließlich von Grafik-Proxies aufgerufen werden. Mit der Methode SUBSCRIBE registriert sich ein Proxy-Objekt bei einem Daten-Container, um in Zukunft von Datenänderungen innerhalb des Containers in Kenntnis gesetzt zu werden.

Die Methode, um diese Registrierung aufzuheben, ist die zweite Methode des Interfaces und hat den Namen UNSUBSCRIBE.

Generierung von Daten-Containern

In den meisten Anwendungsfällen kann mit dem im System vordefinierten generischen Daten-Container für Präsentationsgrafiken gearbeitet werden. Ist dies nicht möglich oder gewünscht, kann jederzeit ein eigener Daten-Container erstellt werden, der die oben genannten drei Interfaces implementiert.

Da dies einen erheblichen Aufwand bedeuten würde, stellt SAP einen Report zur Verfügung, mit dem der Quelltext für einen Daten-Container generiert und als Datei im lokalen Dateisystem des Programmierers abgelegt werden kann. Der Report mit dem dies geschieht, hat den Namen GFW_DCWIZARD.

Um die Funktionsweise dieses Reports zu verstehen, muss an dieser Stelle ein wenig näher auf den Zugriff der Proxies auf den Container eingegangen werden.

Bei der Initialisierung der Proxy-Klasse muss dem Proxy mitgeteilt werden, wie es auf den Daten-Container zuzugreifen hat. Die Proxies setzten hierbei folgende Struktur voraus:

- Jedes Objekt innerhalb eines Daten-Containers ist eindeutig über ein Feld der zugrunde liegenden Dictionary-Struktur identifizierbar.
- Jedes der Objekte des Daten-Containers definiert bis zu drei Werte, die für die Darstellung des Objekts in den Präsentationsgrafiken verwendet werden können.
- Die Objekte eines Daten-Containers können gruppiert werden, um mehrere Datenreihen innerhalb eines Containers unterscheiden zu können. Diese Gruppierung muss als Begriff in einem Attribut des Datenobjekts abgelegt sein.
- Zur Einschränkung der Informationen innerhalb der oben definierten Gruppen besteht die Möglichkeit, ein Feld der Struktur als Filter-Feld zu definieren, über die die Datenbasis während der Anzeige eingeschränkt werden kann.

Über den Filter kann daher die Menge der, der Grafik zugrunde liegenden Objekte eingeschränkt werden, während über das Gruppenfeld parallele Datenreihen aufgebaut werden. Diese beiden Konzepte können beliebig miteinander verschachtelt werden.

Mit diesem Wissen, ist der Aufbau des Selektionsbildschirms des Reports GFW_DCWIZARD einfach zu verstehen (Abbildung 5.5).

Zunächst muss die Data Dictionary-Struktur, welche die Basis des neu zu erstellenden Containers bildet, definiert werden. Daran schließt sich die Definition des Feldes der Struktur, welches die eindeutige Objekt-ID enthält, an.

Die Gruppenbildung innerhalb der Daten-Container erfolgt über dynamische Verfahren. Somit kann zur Laufzeit durch Wahl eines anderen Gruppenfeldes eine andere Sicht auf die Daten des Containers erzeugt werden.

Anders verhält es sich beim Filter, der ebenfalls als Feld in der Container-Struktur enthalten sein muss. Schließlich müssen noch eine Reihe von Verwaltungsinformationen, wie z.B. der Klassenname des Daten-Containers, definiert werden, bevor die Generierung gestartet werden kann.

Der Data-Container-Wizard erstellt hierbei nicht direkt Code im R/3-System, sondern generiert eine Datei im lokalen Dateisystem des Benutzers, welche manuell vom Programmierer in das System geladen werden muss.

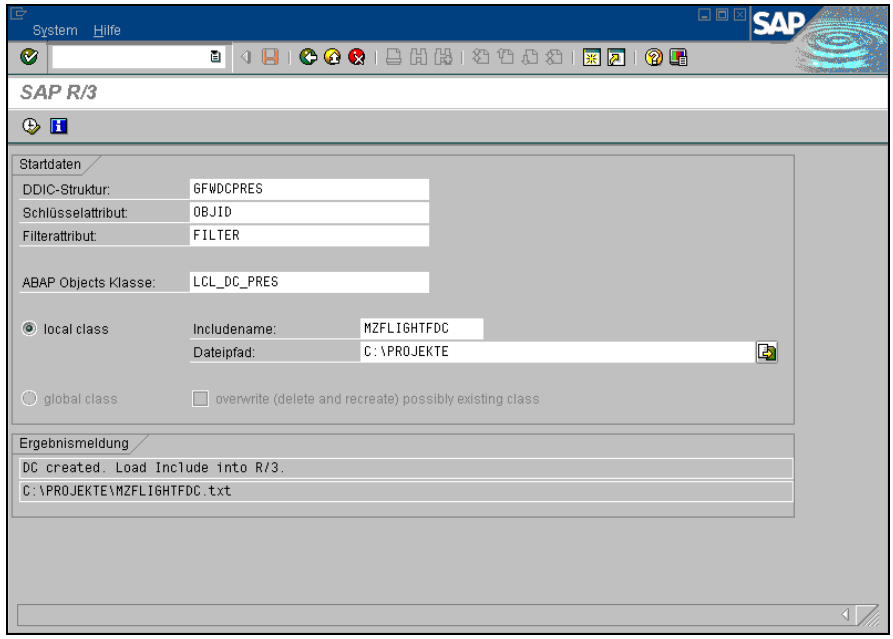

Abbildung 5.5
Selektionsbildschirm des Data-Container Wizards (Report GFW_DCWIZARD) (© SAP AG)

Das dargestellte Beispiel zeigt die Parameter, die nötig wären, um den im System bereits vorhandenen generischen Daten-Container zu generieren. Im Beispiel würde allerdings lediglich der Code für eine lokale Klasse erzeugt werden.

5.5.3 Grafik-Proxy

Verbindungsglied zwischen dem Daten-Container auf der einen und dem Controls-Framework mit den auf dem Präsentationsrechner ablaufenden Controls, sind die so genannten Grafik-Proxies. Diese bilden in der ABAP-Umgebung die programmtechnische Repräsentation der remote ablaufenden Controls.

Über die Grafik-Proxies können nicht die dargestellten Daten manipuliert werden, wohl aber die Art der Darstellung. Hierfür hat SAP eine Architektur geschaffen, die es ermöglicht, optische Eigenschaften eines Controls über vordefinierte parametrisierbare Klassen an bestimmte Stellen an das Proxy anzuschließen. Dieser Vorgang wird in diesem Zusammenhang Customization, die dazugehörigen Klassen Customization-Objekte genannt.

Je nach Grafik-Proxy existieren unterschiedliche Anschlussmöglichkeiten für Customization-Objekte an das Proxy. Diese Anschlüsse werden hier Ports genannt. Jeder Port definiert hierbei einen in der Darstellung anpassbaren Bereich der Grafik.

Im Beispiel dieses Kapitels soll ausschließlich auf die Proxy-Klasse für Präsentationsgrafiken eingegangen werden. Diese Klasse ist im System unter dem Namen `CL_GUI_GP_PRES` definiert.

Diese Klasse definiert 8 Ports, über die die Darstellung der Grafik beeinflusst werden kann. Diese sind im Einzelnen:

- Diagramm
- Chart
- Primäre Achse (X-, Y- und Z-Dimension)
- Sekundäre Achse (X-, Y- sowie Z-Dimension)

Mit diesen Ports können Instanzen der so genannten Customizing-Bundles verknüpft werden. Diese Klassen bündeln eine Reihe visueller Eigenschaften in einer Klasse und können en Block einem Diagramm zugewiesen werden. Die Namen dieser Klassen beginnen mit dem Präfix `CL_CU_`. Auf eine detaillierte Darstellung der einzelnen Klassen soll hier verzichtet werden. Im Beispiel dieses Kapitels werden die Klassen `CL_CU_DISPLAY_CONTEXT` sowie `CL_CU_VALUES` exemplarisch verwendet werden. Für Informationen zu den übrigen Möglichkeiten sei an dieser Stelle auf die Dokumentation sowie die Klassen im System verwiesen.

5.6 Erstellen einer Grafik mit einer Datenreihe

In den folgenden Abschnitten soll Schritt für Schritt ein Report aufgebaut werden, der die Möglichkeiten der SAP Grafik demonstriert. Eine vollständige Darstellung der Möglichkeiten kann hier nicht gegeben werden, jedoch bietet das Beispielprogramm einen schnellen Einstieg in die Technologie und eine einfache Basis für eigene Versuche.

Es soll das Beispiel aus dem vorigen Abschnitt aufgegriffen werden. Für den vorgegebenen Flug werden zunächst die gleichen Daten ermittelt und in einer Instanz des generischen Daten-Containers für Präsentationsgrafiken (`LCL_DC_PRES`) abgelegt werden.

Im Anschluss daran wird ein Dynpro erstellt, auf welchem – analog zu den Ausführungen des Kapitels 3 und 4 – ein Custom Control-Bereich mit dazugehörigem GUI-Container angelegt wird.

Schließlich werden diese beiden Komponenten durch ein Grafik-Proxy verbunden, welches schließlich über Customizing-Klassen angepasst wird.

Das Beispielprogramm wird in Form eines Reports realisiert. Hierdurch kann eine deutlichere Trennung zwischen der Applikation und der Darstellung an sich erfolgen.

5.6.1 Erstellen und Füllen des Daten-Containers

Das Beispiel dieses Kapitels verwendet eine Instanz des generischen Daten-Containers LCL_GP_PRES. Hierdurch können einfach Parallelen zwischen diesem Buch und der SAP-Dokumentation hergestellt werden.

Zunächst muss im Report das Include, in dem die Implementierung des Daten-Containers enthalten ist, eingebunden werden. Im vorliegenden Fall beginnt der Report daher mit der Sequenz

```
REPORT  ZBUCH_GRAPH.

* Einbinden des generischen Daten-Containers.
INCLUDE GFW_DC_PRES.
```

woran sich – analog zum Beispiel aus Abschnitten 5.3 und 5.4 – die Definition der Konstanten, die den darzustellenden Flug definieren anschließen.

```
* Konstantendefinition des auszuwertenden Fluges
CONSTANTS: C_CARRID   LIKE SBOOK-CARRID  VALUE 'AZ',
           C_CONNID   LIKE SBOOK-CONNID  VALUE '0555',
           C_FLDATE   LIKE SBOOK-FLDATE  VALUE '19991113'.
```

Schließlich müssen im Deklarationsteil des Reports noch Variablen für den Daten-Container, die Zugriffsnummer des Programms auf den Container sowie ein globales Feld zur Aufnahme der Return-Codes der aufgerufenen Methoden angelegt werden.

```
* Variables for the Data-Container-Handling
DATA:      G_DC          TYPE REF TO LCL_DC_PRES,
           G_ID_AT_DC    TYPE I,
           G_RETVAL      TYPE SYMSGNO.
```

Das Feld G_ID_AT_DC definiert diese Zugriffsnummer für den Daten-Container. Diese muss bei jedem Aufruf zum Ablegen von Informationen in dem Container übergeben werden. Daher erfolgt die Speicherung in einer globalen Variable.

```
START-OF-SELECTION.

* Daten-Container erzeugen
  PERFORM CREATE_DC  CHANGING  G_DC
                               G_ID_AT_DC.

* Daten-Container mit Werten füllen
  PERFORM FILL_DC    USING     G_DC
                               G_ID_AT_DC
                     CHANGING  G_RETVAL.
```

Im START-OF-SELECTION-Ereignisblock des Reports wird zunächst der Daten-Container angelegt und anschließend in einem zweiten Schritt mit den gewünschten Daten versorgt. Dies erfolgt durch den Aufruf von zwei FORM-Routinen.

```
FORM CREATE_DC CHANGING P_DC       TYPE REF TO LCL_DC_PRES
                        P_ID_AT_DC TYPE I.
  DATA: L_RETVAL TYPE SYMSGNO.

  CREATE OBJECT P_DC.

  CALL METHOD P_DC->IF_DC_MANAGEMENT~INIT
    IMPORTING
      ID     = P_ID_AT_DC
      RETVAL = L_RETVAL.
ENDFORM.                    " CREATE_DC
```

Die Erstellung des Daten-Containers erfolgt hierbei wiederum in zwei Schritten. Zunächst muss die Instanz an sich erstellt werden, bevor diese durch einen Aufruf der INIT-Methode, die Teil des Interface IF_DC_MANAGEMENT ist, initialisiert wird. Diese erfolgt nicht im Rahmen des Konstruktors der Klasse, da Konstruktoren per se keine Rückgabewerte außer der erzeugten Instanz definieren können. Der Rückgabeparameter ID wird jedoch benötigt, um Daten in den Container einzufügen bzw. zu manipulieren.

Sobald der Container dergestalt erzeugt wurde, können die Daten eingefügt werden. Dies erfolgt im Beispiel in einer eigenen FORM-Routine.

```
FORM FILL_DC USING    P_DC       TYPE REF TO LCL_DC_PRES
                      P_ID_AT_DC TYPE I
             CHANGING P_RETVAL   TYPE SYMSGNO.
  DATA: L_GFWDCPRES   TYPE GFWDCPRES.
        L_CLASS       TYPE SBOOK-CLASS,
        L_COUNT       TYPE I.

  SELECT        CLASS
                COUNT( * )
         INTO  (L_CLASS,
                L_COUNT)
         FROM  SBOOK
         WHERE CARRID = C_ CARRID AND
               CONNID = C_ CONNID AND
               FLDATE = C_ FLDATE
         GROUP BY CLASS.
    L_GFWDCPRES-OBJID = L_CLASS.
    L_GFWDCPRES-X_VAL = L_CLASS.
    L_GFWDCPRES-Y_VAL = L_COUNT.
```

```
    CALL METHOD P_DC->SET_OBJ_VALUES
      EXPORTING
         ID     = P_ID_AT_DC
         OBJ    = L_GFWDCPRES
      IMPORTING
         RETVAL = P_RETVAL.
    CHECK P_RETVAL = CL_GFW=>OK.
  ENDSELECT.
ENDFORM.                    " FILL_DC
```

Die Datenermittlung ist in diesem Fall recht einfach und kann mit einer einfachen SELECT-Anweisung realisiert werden. Zur eindeutigen Identifizierung der Objekte genügt in diesem Schritt die gebuchte Klasse. Im Abschnitt 5.7 muss dieser Begriff erweitert werden, um Buchungen mehrerer Flüge unterscheiden zu können.

Um zu zeigen, dass das Erstellen und Füllen des Daten-Containers im Bereich der Anwendung liegt, wurden diese Schritte direkt aus dem Ereignisblock START-OF-SELECTION heraus ausgeführt. Die Erstellung des Dynpro-Containers sowie des Grafik-Proxies erfolgt auf einer anderen Ebene und wurde daher in die Implementierung des Dynpros verlagert, auf welchem die Darstellung der Grafik erfolgen soll.

5.6.2 Erstellung des Dynpros mit dem GUI-Container

Analog zum dritten und vierten Kapitel muss im Report ein Dynpro definiert werden, welches einen Custom Control-Bereich mit dem Namen »GRAPHIC« enthält. Im Beispiel wurde hierfür das Dynpro mit der Nummer 0100 angelegt. Zum Dynpro wurde ein gleichnamiger GUI-Status definiert, der lediglich das Cancel-Kommando (CANC) beinhaltet, um das Programm zu beenden.

Die Implementierung der Routine zur Auswertung der Funktionscodes beendet das Programm beim zuvor definierten Funktionscode CANC. Weitere Aktionen sind hier zum jetzigen Zeitpunkt noch nicht notwendig.

In der Ablauflogik des Dynpros wird das MODULE CREATE_CONTAINER_0100 zum Erstellen des Containerobjekts aufgerufen.

```
MODULE CREATE_CONTAINER_0100 OUTPUT.
* Nur ausführen, wenn der Container noch nicht erzeugt wurde
  CHECK G_CONTAINER IS INITIAL.

  CREATE OBJECT G_CONTAINER
    EXPORTING
      CONTAINER_NAME = 'GRAPHIC'.
ENDMODULE.                 " CREATE_CONTAINER_0100  OUTPUT
```

Die Erzeugung der Instanz erfolgt hierbei nur einmal. Ist die globale Variable, welche die Instanz der Klasse CL_GUI_CUSTOM_CONTAINER enthält, nicht initial, hat dieser Schritt bereits stattgefunden und wird im Folgenden übersprungen.

Die Variable G_CONTAINER wird hierbei global im Programm deklariert.

```
DATA: G_CONTAINER  TYPE REF TO CL_GUI_CUSTOM_CONTAINER.
```

5.6.3 Erzeugung des Grafik-Proxies

Somit ist sowohl der Daten-Container, als auch der Container auf dem Dynpro für die Darstellung der Grafik vorbereitet. Was noch fehlt ist das Grafik-Proxy, welches die beiden Welten verbindet und die Darstellung der Grafik definiert.

In der Ablauflogik des Dynpros wird zu diesem Zweck ein weiterer Modul-Aufruf integriert. Insgesamt hat die Ablauflogik dann das im Folgenden dargestellte Aussehen.

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
  MODULE CREATE_CONTAINER_0100.
  MODULE CREATE_GRAPHIC_0100.
  MODULE DISTRIBUTE_0100.

PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0100.
```

Die Implementierung des Moduls CREATE_GRAPHIC_0100 prüft zunächst, ob die Initialisierung des GUI-Containers erfolgreich stattgefunden hat und dass noch keine Grafik darin erstellt wurde.

```
MODULE CREATE_GRAPHIC_0100 OUTPUT.
* Nur, wenn der Container erzeugt werden konnte...
  CHECK NOT G_CONTAINER IS INITIAL.
* .. und die Grafik noch nicht exisitiert.
  CHECK G_GRAPHIC IS INITIAL.

* Instanz des Grafik-Proxies erzeugen
  CREATE OBJECT G_GRAPHIC.

* Proxy initialisieren, d.h. Verknüpfung zwischen Data-Container und
* GUI-Container herstellen
  CALL METHOD G_GRAPHIC->IF_GRAPHIC_PROXY~INIT
         EXPORTING
            PARENT     = G_CONTAINER
            DC         = G_DC
            PROD_ID    = CL_GUI_GP_PRES=>CO_PROD_CHART
            FORCE_PROD = GFW_TRUE
```

```
        IMPORTING
           RETVAL     = G_RETVAL.

  CHECK G_RETVAL = CL_GFW=>OK.

* Tell the Proxy how to retrieve the values from the DC
  CALL METHOD G_GRAPHIC->SET_DC_NAMES
        EXPORTING
           OBJ_ID     = 'OBJID'
           DIM1       = 'X_VAL'
           DIM2       = 'Y_VAL'
           GRP_ID     = 'GRPID'
        IMPORTING
           RETVAL     = G_RETVAL.

  CHECK G_RETVAL = CL_GFW=>OK.

* Show the Graphic
  CALL METHOD G_GRAPHIC->IF_GRAPHIC_PROXY~ACTIVATE
        IMPORTING
           RETVAL     = G_RETVAL.
ENDMODULE.                 " CREATE_GRAPHIC_0100  OUTPUT
```

Im Anschluss daran wird die Instanz des Grafik-Proxies, welche als globale Variable mit dem Typ `CL_GUI_GP_PRES` definiert ist, erzeugt. Die neue Instanz der Klasse wird über einen Aufruf der Methode `INIT` initialisiert. Dabei wird dem Proxy sowohl die Instanz des Daten-Containers, als auch die Instanz des GUI-Containers übergeben.

Über die beiden Parameter `PROD_ID` und `FORCE_PROD` wird festgelegt, welches Grafikprodukt zur Anzeige der Grafik verwendet werden soll. Bei den Grafik-Produkten handelt es sich um eine weitere Modularisierung zwischen dem Proxy und dem tatsächlichen Viewer. Eine Darstellung dieser Zusammenhänge kann aus der SAP-Dokumentation entnommen werden.

In »Generierung von Daten-Containern« in Abschnitt 5.5.2 wurde bereits erwähnt, dass bei der Initialisierung des Grafik-Proxies diesem mitgeteilt werden muss, auf welche Art die zur Anzeige benötigten Informationen aus dem Daten-Container ermittelt werden können.

Zu diesem Zweck definiert die Klasse `CL_GUI_GP_PRES` die Methode `SET_DC_NAMES`, welche aufgerufen wird, um dem Proxy die Namen der Felder des Daten-Containers mitzuteilen, die die eindeutige ID (`OBJ_ID`), die Gruppe (`GRP_ID`) sowie die Feldnamen der bis zu drei Wertfelder des Datenobjekts enthalten.

Nachdem dies erfolgt ist, ist das Grafik-Proxy vollständig initialisiert. Über einen Aufruf der Methode `ACTIVATE` kann die Anzeige der Grafik aktiviert werden.

Die Grafik erscheint somit auf dem Dynpro. Analog kann mit der Methode DEACTIVATE die Anzeige auf dem Dynpro jederzeit unterdrückt werden.

Auf die Darstellung der Grafik wurde in diesem Fall kein Einfluss genommen. Das System nimmt in diesem Fall eine Reihe von Standardeinstellungen vor, so dass die Ausgabe des Programmes dem in Abbildung 5.6 gezeigten Bildschirm entspricht.

```
MODULE DISTRIBUTE_0100 OUTPUT.
  CALL METHOD G_DC->IF_DC_MANAGEMENT~DISTRIBUTE_CHANGES
    IMPORTING
      RETVAL      = G_RETVAL.
ENDMODULE.                    " DISTRIBUTE_0100  OUTPUT
```

Im Modul DISTRIBUTE_0100 am Ende des PBO wird lediglich die Methode DISTRIBUTE_CHANGES des Daten-Containers aufgerufen, um die Änderungen an alle registrierten Proxies zu übertragen. In diesem Beispiel könnte dieser Aufruf auch entfallen, da lediglich ein Proxy die Daten darstellt und die Daten an sich konstant sind.

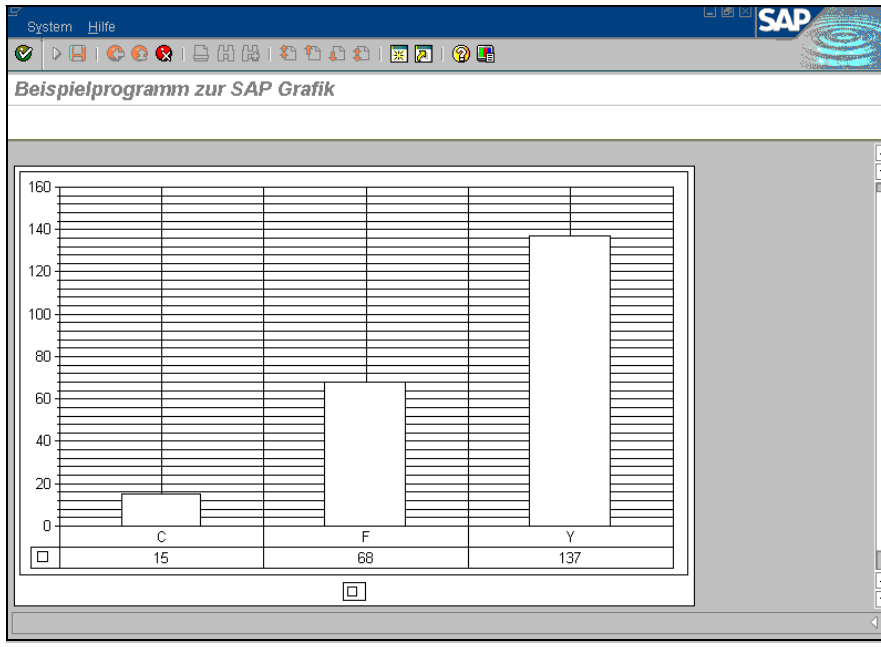

Abbildung 5.6
SAP Grafik ohne Beeinflussung der Darstellung (© SAP AG)

5.6.4 Anpassen der Darstellung über Customizing

Im Abschnitt 5.5.3 wurde erklärt, dass die Darstellung der Grafiken auf dem Bildschirm in weiten Grenzen vom Programmierer beeinflusst werden kann. Hierfür bieten die Proxy-Klassen Anschlussmöglichkeiten für andere Objekte, die die Darstellungsattribute für einen Bereich definieren. Jedes Proxy-Objekt kann dabei andere Anschlüsse (Ports) definieren. Der Zugriff auf die Ports erfolgt jedoch standardisiert über die Methoden, die im Interface `IF_GRAPHIC_PROXY` definiert sind.

Das hier verwendete Proxy `CL_GUI_GP_PRES` definiert acht solcher Anschlussmöglichkeiten.

- Diagramm
- Chart
- Primäre Achse (X-, Y- und Z-Dimension)
- Sekundäre Achse (X-, Y- sowie Z-Dimension)

Jeder dieser Ports kann mit Customizing-Objekten versehen werden. SAP defininiert eine ganze Reihe Customizing-Objekte, deren Namen mit dem Präfix `CL_CU_` beginnen.

Klasse	Aufgabe
CL_CU_DISPLAY_CONTEXT	Die Klasse enthält alle Darstellungsoptionen einer Grafik. Sie kann verwendet werden, um nicht explizit anders definierte Werte zu ererben.
CL_CU_DIAGRAM	Die Klasse enthält alle Attribute, die die Grafik als ganzes Betreffen. Dies beinhaltet auch Informationen über Ausgabebereiche, falls mehrere Grafiken innerhalb eines Diagramms ausgegeben werden.
CL_CU_DRAWING_AREA	Enthält alle Darstellungseigenschaften eines Ausgabebereiches. Eine Diagramm kann mehrere solcher Ausgabebereiche mit jeweils einer Grafik beinhalten.
CL_CU_AXIS	Definiert den Wertebereich einer Achse einer Grafik
CL_CU_SCALE	Definiert die Darstellung einer Achse basierend auf einer Instanz von CL_CU_AXIS
CL_CU_VALUES	Definiert eine Wertereihe, die in einer Grafik angezeigt wird. Wertereihen haben in Strukturgrafiken keine Bedeutung.
CL_CU_DATA_SHEET	Definiert die Darstellung einer Tabelle als Ausgabebereich

Klasse	Aufgabe
CL_CU_POINT	Definiert die Darstellung eines Punktes in einer Präsentationsgrafik bzw. eines Knotens in einer Strukturgrafik
CL_CU_ITEM	Definiert eine Darstellungseinheit

Tabelle 5.5
Customizing-Klassen

Ein Customizing-Objekt bündelt eine Reihe von grafischen Einstellungen, die zusammengenommen die Darstellung der Grafik definieren.

Im Beispiel sollen die Customizing-Klassen verwendet werden, um den Grafiktyp der Grafik festzulegen. Hierfür wird im Modul `CREATE_GRAPHIC_0100` vor dem Aktivieren der Grafikanzeige die Routine `CUSTOMIZE_GRAPHIC` aufgerufen.

```
...
* Customize the graphic
  PERFORM CUSTOMIZE_GRAPHIC USING G_GRAPHIC.
...
```

In dieser Routine werden zwei Customizing-Bundles deklariert. Die Instanz der Klasse `CL_CU_DISPLAY_CONTEXT` wird verwendet, um die Standardeinstellungen zu erben, da lediglich die Ausgabefarbe sowie der Typ der Grafik selbst festgelegt werden sollen.

```
FORM CUSTOMIZE_GRAPHIC USING    P_PROXY  TYPE REF TO CL_GUI_GP_PRES.
  DATA: L_RETVAL          TYPE SYMSGNO,
        L_BUNDLE_DISPLAY  TYPE REF TO CL_CU_DISPLAY_CONTEXT,
        L_BUNDLE_VALUES   TYPE REF TO CL_CU_VALUES.

* Customizing-Objekt erzeugen
  CREATE OBJECT L_BUNDLE_VALUES
    EXPORTING
      INSTANCE_ID = 'CLASSES'.

* Display Context erzeugen
  CREATE OBJECT L_BUNDLE_DISPLAY
    EXPORTING
      INSTANCE_ID = 'dummy'.

* Set Color of Data-Series
  CALL METHOD L_BUNDLE_DISPLAY->IF_CUSTOMIZING~SET
    EXPORTING
      ATTR_ID     = CL_CU_DISPLAY_CONTEXT=>CO_BG_CLR_PLT_ID
      VALUE       = 0.
```

```
* Copy Display-Context into Values
  CALL METHOD L_BUNDLE_VALUES->IF_CUSTOMIZING~SET
    EXPORTING
      ATTR_ID     = CL_CU_VALUES=>CO_CURVE_CONTEXT
      VALUE       = L_BUNDLE_DISPLAY.

* Customizing vornehmen
  CALL METHOD L_BUNDLE_VALUES->IF_CUSTOMIZING~SET
    EXPORTING
      ATTR_ID     = CL_CU_VALUES=>CO_STYLE
      VALUE       = L_CHARTYPE.

* Activate Customizing in Proxy
  CALL METHOD P_PROXY->IF_GRAPHIC_PROXY~ADD_CU_BUNDLE
    EXPORTING
      PORT        = IF_GRAPHIC_PROXY=>CO_PORT_CHART
      KEY         = ' '
      BUNDLE      = L_BUNDLE_VALUES
    IMPORTING
      RETVAL      = L_RETVAL.
ENDFORM.                    " customize_graphic
```

Der abgebildete Quelltext zeigt, dass beim Instanziieren einer Customizing-Klasse ein Name für die Instanz definiert werden muss. Dieser Name ermöglicht es, die Customizing-Einstellungen ggf. benutzerspezifisch zu speichern und zu einem späteren Zeitpunkt wieder zu aktivieren. Auf diese Möglichkeit soll hier jedoch nicht näher eingegangen werden, sondern stattdessen auf die SAP Dokumentation zur SAP Grafik verwiesen werden.

Interessanter sind die konkreten Definitionen von Attributen. Die Customizing-Klassen definieren hierfür jeweils eine Methode `SET`, die einen Schlüssel und einen Wert als Parameter erhalten. Welche Schlüssel jeweils zulässig sind, definieren die Klassen selbst. Die Hintergrundfarbe eines Ausgabebereiches wird über die Konstante `CL_CU_DISPLAY_CONTEXT=>CO_BG_CLR_PLT_ID` am Ausgabebereich definiert, während der Typ der Grafik ein Attribut einer Wertereihe (`CL_CU_VALUES`) ist.

Die Klasse `CL_CU_VALUES` lässt eine ganze Reihe unterschiedlicher Grafiktypen zu. Eine Liste der wichtigsten Typen findet sich in Tabelle 5.6 wieder. Da der Typ der Grafik an der jeweiligen Datenreihe definiert wird, können Grafiktypen teilweise innerhalb einer Grafik gemischt werden.

CO_STYLE	Aufgabe
1	Vertikale 2D-Balken, jede Wertereihe hat einen eigenen Balken
2	Vertikale 2D-Balken, Balken werden gestapelt, Werte addiert
3	Vertikale 2D-Balken, Balken werden gestapelt, Prozentualer Anteil der Wertebereiche wird ausgegeben
4-6	Wie 1-3 jedoch 3D
7	Vertikale 3D-Balken, Wertereihen werden in Z-Achse aufgetragen
8-10	Wie 1-3 jedoch horizontal
11-13	Wie 4-6 jedoch horizontal
14-16	Vertikales 2D-Liniendiagramm, Darstellung der Wertereihen wie bei 1-3
17-19	Wie 14-16, Punkte werden jedoch optisch hervorgehoben
20	3D-Liniendiagramm, Wertereihen wie 17
21-23	Horizontales 2D-Liniendiagramm, Wertereihen wie 14-16
24-26	Horizontales 2D-Liniendiagramm, Wertereihen wie 17-19
27, 28	2D-Tortendiagramm
29, 30	2D-Tortendiagramm mit gezoomten Werten
31, 32	3D-Tortendiagramm
38-40	2D Flächendiagramm, Wertereihen wie 1-3
41-43	3D-Flächendiagramm, Wertereihen wie 4-7
44, 45	2D-Tortendiagramm mit mehreren Wertereihen

Tabelle 5.6
Wichtige Grafiktypen bei der SAP Grafik

Sobald ein Customizing-Objekt vollständig initialisiert ist, kann es über die Methode `CU_ADD_BUNLDE` einem Port an einem Proxy-Objekt zugewiesen werden. Über den Parameter `KEY` kann ggf. noch gesteuert werden, für welche Objektgruppe des Daten-Containers die jeweilige Einstellung vorgenommen werden soll.

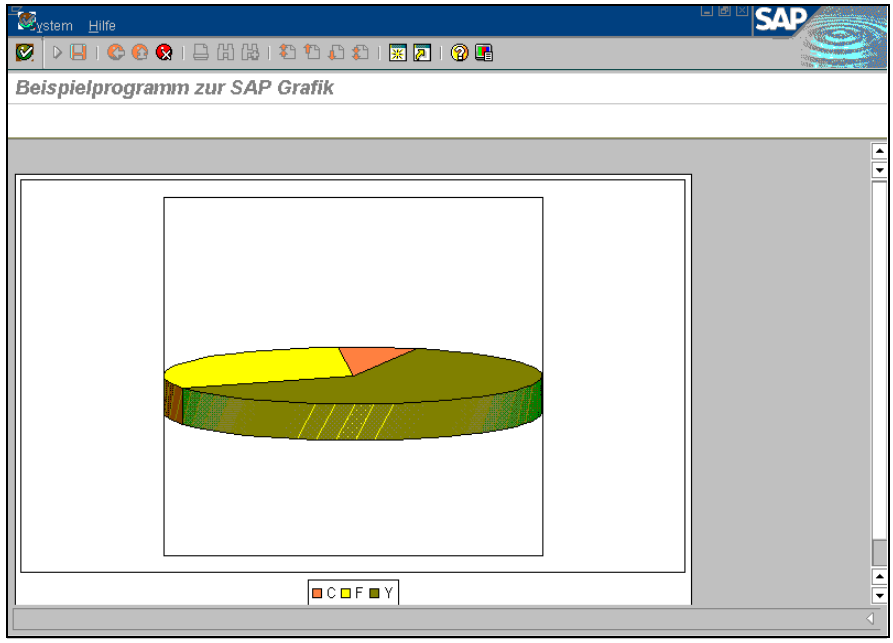

Abbildung 5.7
Darstellung der Buchungsanteile als 3D-Tortendiagramm (© SAP AG)

Abbildung 5.7 zeigt die Darstellung der Grafik, nachdem die Customizing-Einstellungen aktiviert wurden. Der Unterschied zu Abbildung 5.6 zeigt, in welchem Umfang die Darstellung durch das Customizing beeinflusst werden kann. Außer dem Customizing wurden keinerlei Veränderungen am zugrunde liegenden Programm durchgeführt.

5.7 Darstellung mehrerer Datenreihen

Das bisherige Beispiel beschränkte sich darauf, eine Wertereihe im Diagramm darzustellen. Gleichwohl ist es möglich, innerhalb eines Daten-Containers mehrere, voneinander unabhängige Wertereihen zu definieren und zur Anzeige zu bringen.

Der Code des Programms muss hierfür im Wesentlichen an zwei Punkten modifiziert werden. Zum einen müssen in der `FORM`-Routine `FILL_DC` die Daten der übrigen Wertereihen ermittelt und im Daten-Container abgelegt werden, zum anderen muss das Customizing der Grafik dahingehend angepasst werden, dass die zweite Wertereihe auch zur Ausgabe gelangt.

Um dies zu demonstrieren, sollen im Beispielprogramm die Werte für einen weiteren Flug aufgenommen werden. Neben dem 13.11.1999 sollen jetzt auch

die Buchungen vom 11.12.1999 des gleichen Fluges dargestellt werden. Das zweite Datum wird in Form der Kontante C_FLDATE2 am Anfang des Programms definiert.

```
FORM FILL_DC USING    P_DC       TYPE REF TO LCL_DC_PRES
                      P_ID_AT_DC TYPE I
             CHANGING P_RETVAL   TYPE SYMSGNO.
  DATA: L_GFWDCPRES   TYPE GFWDCPRES.
  DATA: L_CLASS       TYPE SBOOK-CLASS,
        L_COUNT       TYPE I.

* Lesen des ersten Fluges
  SELECT        CLASS
                COUNT( * )
         INTO  (L_CLASS,
                L_COUNT)
         FROM  SBOOK
         WHERE CARRID = C_CARRID AND
               CONNID = C_CONNID AND
               FLDATE = C_FLDATE
         GROUP BY CLASS.
    CONCATENATE L_CLASS
                'A'
           INTO L_GFWDCPRES-OBJID.
    L_GFWDCPRES-GRPID = 'A'.
    L_GFWDCPRES-X_VAL = L_CLASS.
    L_GFWDCPRES-Y_VAL = L_COUNT.

    CALL METHOD P_DC->SET_OBJ_VALUES
      EXPORTING
        ID     = P_ID_AT_DC
        OBJ    = L_GFWDCPRES
      IMPORTING
        RETVAL = P_RETVAL.

    CHECK P_RETVAL = CL_GFW=>OK.
  ENDSELECT.

* Lesen des zweiten Fluges
  SELECT        CLASS
                COUNT( * )
         INTO  (L_CLASS,
                L_COUNT)
         FROM  SBOOK
         WHERE CARRID = C_CARRID  AND
               CONNID = C_CONNID  AND
               FLDATE = C_FLDATE2
```

```
        GROUP BY CLASS.
    CONCATENATE L_CLASS
                'B'
           INTO L_GFWDCPRES-OBJID.
    L_GFWDCPRES-GRPID = 'B'.
    L_GFWDCPRES-X_VAL = L_CLASS.
    L_GFWDCPRES-Y_VAL = L_COUNT.

    CALL METHOD P_DC->SET_OBJ_VALUES
      EXPORTING
        ID     = P_ID_AT_DC
        OBJ    = L_GFWDCPRES
      IMPORTING
        RETVAL = P_RETVAL.

    CHECK P_RETVAL = CL_GFW=>OK.
  ENDSELECT.
ENDFORM.                    " FILL_DC
```

Die beiden Datenreihen werden nacheinander in den beiden SELECT-Anweisungen ermittelt. Da hier die gebuchte Klasse nicht mehr eindeutig ist, genügt es nicht mehr, diese als Objekt-ID zu verwenden. Statt dessen wurde an das Objekt des ersten SELECTs das Literal »A«, an das des zweiten SELECTs das Literal »B« angehangen. Um die Wertereihen in der Darstellung unterscheiden zu können, wurden außerdem die obigen Literale im Feld GRPID der Container-Struktur abgelegt.

Nach der Routine enthält der Container somit die sechs in Tabelle 5.7 dargestellten Einträge. Die Einträge im Feld OBJ_ID sind über die gesamte Tabelle eindeutig, während die Einträge der Spalte GRPID die Sätze in zwei Gruppen aufteilen. Innerhalb der Gruppen definieren die Spalten XVAL und YVAL jeweils Wertepaare, die in der Grafik angezeigt werden.

OBJ_ID	GRPID	XVAL	YVAL
CA	A	C	15
FA	A	F	68
YA	A	Y	137
CB	B	C	14
FB	B	F	67
YB	B	Y	134

Tabelle 5.7
Objekte im Daten-Container bei zwei Wertereihen

Um die zweite Wertereihe in der Grafik anzuzeigen, müssen Veränderungen am Customizing der Grafik vorgenommen werden. Die 3D-Tortengrafik, die bisher verwendet wurde, ist nicht für die Darstellung von mehr als einer Wertereihe geeignet. Daher wurde auf den Diagrammtyp »7«, welcher 3-dimensionale Balken in übersichtlicher Anordnung definiert, gewechselt.

Ansonsten wurden die Customizingeinstellungen für die erste Wertereihen lediglich kopiert und dahingehend verändert, dass die zweite Wertereihe mit einer anderen Farbe dargestellt wird. Auch hier gilt, dass der Typ der Grafik für jede Wertereihe gesondert definiert wird. Auch die Kopie der Customizing-Einstellungen muss daher den Grafiktyp »7« definieren, um schließlich die in Abbildung 5.8 gezeigte Darstellung zu erhalten.

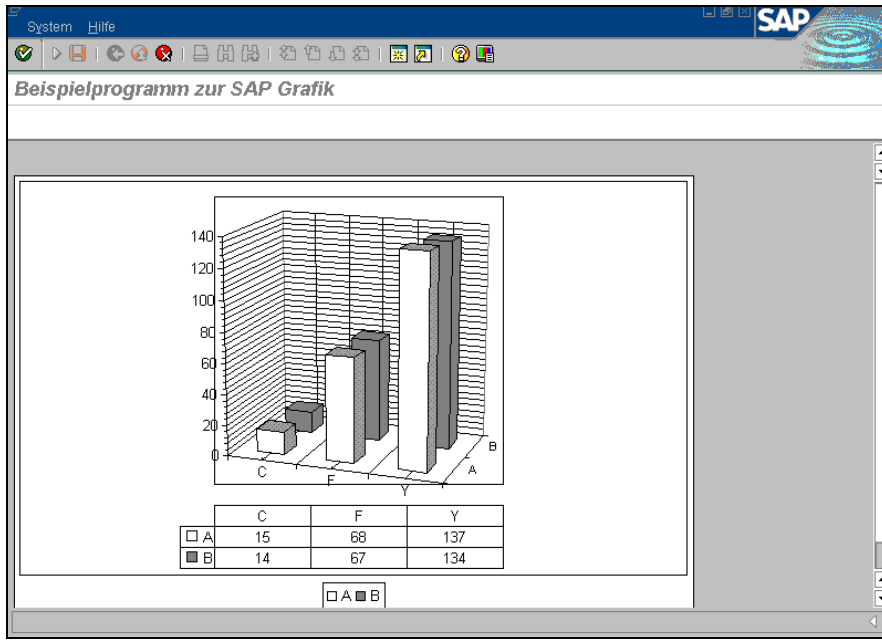

Abbildung 5.8
3D-Balkengrafik mit zwei Wertereihen (© SAP AG)

Die Darstellung mehrerer Wertereihen funktioniert nur dann, wenn dem Grafik-Proxy bei dessen Initialisierung das Strukturfeld, welches die Gruppen definiert, mitgeteilt wurde. Wird beim Aufruf der Methode `SET_DC_NAMES` (siehe 5.6.3) der optionale Parameter `GRP_ID` nicht entsprechend übergeben, werden in keinem Fall mehr als eine Wertereihe ausgeben.

Zuletzt sei noch angemerkt, dass vor Beendigung des Programms die Methode `FREE` des Grafik-Proxies aufgerufen werden sollte, um eine saubere Deinitialisierung der verwendeten Speicherbereiche durchzuführen. Auf diese Weise wer-

den unter Umständen noch belegte Ressourcen für die Kommunikation zwischen dem Proxy und der Grafik selbst freigeben.

5.8 Verwendete Funktionsbausteine

Exemplarisch für alle Funktionsbausteine der Funktionsgruppe BUSG wurden hier die Funktionsbausteine GRAPH_2D sowie GRAPH_MATRIX_2D vorgestellt. Die Funktionsweise der übrigen Funktionsbausteine ist sehr ähnlich zu den dargestellten.

5.8.1 GRAPH_2D

Der Funktionsbaustein GRAPH_2D bietet eine einfache Möglichkeit, 2-dimensionale Grafiken anzuzeigen. Die Aufrufschnittstelle des Funktionsbausteins ist möglichst einfach gehalten, bietet daher aber auch nicht so viele Möglichkeiten, wie der Funktionsbaustein GRAPH_MATRIX_2D, dessen Aufrufschnittstelle größere Einflussmöglichkeiten beinhaltet.

Import-Parameter	
AUTO_CMD_1	intern
AUTO_CMD_2	intern
TYPE	Grafiktyp (bitte statt dessen DISPLAY-TYPE verwenden)
DISPLAY_TYPE	Grafiktyp
INBUF	intern
INFORM	Erlaubt Diakogfunktionalität
MAIL_ALLOW	Kann Grafik als Mail versandt werden
PWDID	interner Dialogparameter
SET_FOCUS	Fokus setzen beim Nachladen
SMFONT	Kennzeichen, ob kleinere Schriftarten verwendet werden sollen
SO_CONTENTS	Untertitel der Grafik im erzeugten SAPOffice-Dokument
SO_RECEIVER	Empfänger des SAPOffice-Dokuments
SO_SEND	Versenden der Grafik über SAPOffice. Keine Anzeige der Grafik auf dem Bildschirm.
SO_TITLE	Titel des SAPOffice-Dokuments

STAT	interner Dialogparameter
SUPER	interner Dialogparameter
TIMER	intern
TITL	Titel der Grafik
VALT	Bezeichung der Skala (Y)
WDID	interner Dialogparameter
WINID	interner Dialogparameter
WINPOS	Position des Grafikfensters auf dem Bildschirm
WINSZX	Horizontale Größe des Fensters in %
WINSZY	Vertikale Größe des Fensters in %
X_OPT	intern
NOTIFY	Aktivierung von »Einstellungen Sichern«
Export-Parameter	
B_KEY	Gedrückte Taste
B_TYP	Typ der gedrückten Taste
CUA_ID	Beschreibung des Events der gedrückten Taste
MOD_COL	Veränderte Zeile der Datentabelle
MOD_ROW	Veränderte Spalte der Datentabelle
MOD_VAL	Neuer Wert
M_TYP	interer Dialogparameter
RBUFF	intern
RWNID	interner Dialogparameter
Tabellen-Parameter	
DATA	Tabelle mit Daten für die Grafik
Export-Parameter	
GUI_REFUSE_GRAPHIC	Frontend kann Grafik nicht anzeigen

5.8.2 GRAPH_MATRIX_2D

Mit dem Funktionsbaustein GRAPH_MATRIX_2D, der stellvertretend für alle Funktionsbausteine, deren Name mit GRAPH_MATRIX_ beginnt, gezeigt wird, können Businessgrafiken mit mehr Gestaltungsoptionen des Programmierers angezeigt werden.

Sollen lediglich einfache Datenstrukturen angezeigt werden, kann unter Umständen die Verwendung von GRAPH_2D einfacher sein.

Import-Parameter	
AUTO_CMD_1	intern
AUTO_CMD_2	intern
INBUF	intern
INFORM	Erlaubt Diakogfunktionalität
MAIL_ALLOW	Kann Grafik als Mail versandt werden
NCOL	Spaltennummer in Datentabelle, in der die Werte für die Grafik enthalten sind
NROW	Zeilennumer der Datentabelle, ab die Daten in der Grafik angezeigt werden sollen
PWDID	interner Dialogparameter
SET_FOCUS	Fokus setzen beim Nachladen
SMFONT	Kennzeichen, ob kleinere Schriftarten verwendet werden sollen
SO_CONTENTS	Untertitel der Grafik im erzeugten SAPOffice-Dokument
SO_RECEIVER	Empfänger des SAPOffice-Dokuments
SO_SEND	Versenden der Grafik über SAPOffice. Keine Anzeige der Grafik auf dem Bildschirm.
SO_TITLE	Titel des SAPOffice-Dokuments
STAT	interner Dialogparameter
SUPER	interner Dialogparameter
TIMER	intern
TITL	Titel der Grafik
VALT	Bezeichung der Skala (Y)

WDID	interner Dialogparameter
WINID	interner Dialogparameter
WINPOS	Position des Grafikfensters auf dem Bildschirm
WINSZX	Horizontale Größe des Fensters in %
WINSZY	Vertikale Größe des Fensters in %
X_OPT	intern
NOTIFY	Aktivierung von »Einstellungen Sichern«
Export-Parameter	
B_KEY	Gedrückte Taste
B_TYP	Typ der gedrückten Taste
CUA_ID	Beschreibung des Events der gedrückten Taste
MOD_COL	Veränderte Zeile der Datentabelle
MOD_ROW	Veränderte Spalte der Datentabelle
MOD_VAL	Neuer Wert
M_TYP	interer Dialogparameter
RBUFF	intern
RWNID	interner Dialogparameter
Tabellen-Parameter	
DATA	Tabelle mit Daten für die Grafik
OPTS	Gestaltungsoptionen für die Grafik
TCOL	Spaltenüberschriften
Export-Parameter	
COL_INVALID	Ungültige Spalte
OPT_INVALID	Ungültige Gestaltungsoptionen definiert

6 Batch-Input-Techniken

6.1 Was ist Batch-Input

Hinter dem Begriff Batch-Input (BTC) verbirgt sich eine Technik in R/3 Dialogtransaktionen unsichtbar abspielen zu lassen. Es handelt sich hierbei um ein automatisiertes Abspielen von Benutzereingaben, d.h. sämtliche Plausibilisierungen und Ableitungen, die auch bei manueller Bedienung der Transaktion ablaufen würden, werden auch im BTC durchgeführt. Der Weg durch die Dynpros wird dem System hierbei in Form einer internen Tabelle mitgeteilt. Diese Tabelle aufzubauen ist der wichtigste und zugleich schwierigste Schritt beim Batch-Input. In der Tabelle werden sämtliche Benutzereingaben, d.h. sowohl in Felder als auch Funktionstasten u.ä., sowie alle Bildschirmwechsel definiert.

Man kann sich die Technik so vorstellen, als müsste einer Person der Weg durch eine ihr unbekannte Stadt erklärt werden. Der Person müssen Ortsbeschreibungen und Anweisungen mit auf den Weg gegeben werden, um einen erreichten Punkt zu erkennen und um zu wissen, wie an diesem Punkt weiter zu verfahren ist. Der Person muss hierbei jeder Punkt (z.B. Kreuzung), der ihr auf dem Weg durch die Stadt begegnet, beschrieben werden und an jedem dieser Punkte eine oder mehrere Aktionen, die die Person dort auszuführen hat (z.B. links abbiegen). Wichtig hierbei ist, dass die Beschreibung lückenlos und vollständig ist. Dies heißt, dass weder Punkte auf dem Weg fehlen dürfen, noch zuviele Orte beschrieben sein dürfen. Sobald ein Ort beschrieben wird, der nicht stimmt, oder eine Aktion, die nicht durchgeführt werden kann (z.B. wenn kein Zebrastreifen da ist, wo einer überquert werden soll), weiß die Person nicht mehr, wo sie ist und bricht die gesamte Prozedur erfolglos ab.

Übertragen auf den Batch-Input-Ablauf bedeutet dies, dass die Ablauf-Tabelle sämtliche Dynpros, die betreten werden, nennen muss und sämtliche Aktionen, die der Benutzer auf diesen Dynpros durchführt, benannt werden müssen. Dies umfasst Feldeingaben aber auch das Auslösen von Funktionscodes, um Aktio-

nen auszulösen oder Bildwechsel vorzunehmen. Wichtig hierbei ist, dass der gesamte Ablauf vor Aufruf der Transaktion definiert werden muss, der Programmierer hat also während des Abspielens des Batch-Input keinerei Möglicheiten mehr, auf den Programmlauf Einfluss zu nehmen. Hier wird das Problem beim Batch-Input deutlich. Wenn der Person aus obigen Beispiel der Name einer einzigen Kreuzung falsch genannt wird, oder nur ein einziges Mal falsch abgebogen wird, ist sie hoffnungslos verloren und das gewünschte Ziel unerreichbar. Bei der Programmierung von Batch-Input-Abläufen müssen die auszuführenden Transaktionen daher genau untersucht werden und etwaige Verzweigungen aufgrund von Anwendungsdaten oder Systemeinstellungen vorausgesehen und berücksichtigt werden.

Erschwerend kommt hierbei noch hinzu, dass sich Dialogtransaktionen in R/3 häufig im »echten« Dialog, d.h. bei manueller Bedienung durch einen Benutzer, anders verhalten, als im »Batch«-Mode. Dies muss bei der Erstellung der Batch-Input-Abläufe ebenfalls berücksichtigt werden. In Abschnitt 6.9 wird auf dieses Phänomen nochmals eingegangen.

Bei der Erstellung von Batch-Input-Abläufen bietet das System dem Programmierer jedoch Unterstützung an. Über den so genannten »Batch-Input-Recorder« (Transaktion `SHDB`) können Batch-Input-Abläufe während der Bedienung einer Dialogtransaktion aufgenommen und später entweder als Batch-Input-Mappe (siehe Abschntt 6.6) oder als generiertes Programm für die Verwendung der `CALL TRANSACTION`-Anweisung abgelegt werden. Der aufgezeichnete Ablauf entspricht jedoch nur dem Ablauf unter der Datenkonstellation, die während der Aufzeichnung verwendet wurde. Näheres zum Batch-Input-Recorder ist in Abschnitt 6.7 dargestellt.

Wie bereits erwähnt, können Batch-Input-Abläufe auch als so genannte Mappen im System abgelegt werden. Es handelt sich hierbei um die Speicherung der Batch-Input-Abläufe im R/3-System zur asynchronen Verarbeitung. Diese Funktionalität wird über eine Reihe von Funktionsbausteinen realisiert. Wie diese genutzt werden und was sich sonst hinter dieser Technik verbirgt, wird im Abschnitt 6.6 dieses Kapitels erläutert.

Das ABAP-Sprachkonstrukt, welches verwendet wird, um Batch-Input-Abläufe abzuspielen, ist die Anweisung `CALL TRANSACTION`. Die Funktionalität dieser Anweisung beschränkt sich jedoch nicht nur auf das Abspielen vollständiger Transaktionen. Die Optionen und Nutzungsmöglichkeiten dieser Anweisung werden ausführlich im Abschnitt 6.5 gezeigt.

6.2 Warum Batch-Input?

Das Programmiermodell in R/3 beinhaltet die Möglichkeit bei der Programmierung von Dynpros Plausibilisierungen mit sehr feiner Granularität durchzuführen. Hierbei können beliebige Kombinationen von Feldern und Feldgruppen

zusammen ausgewertet werden und die Eingabebereitschaft nur bestimmter Felder im Falle eines Fehlers erreicht werden. Bedingt durch dieses Modell befindet sich jedoch sehr viel Programmintelligenz in den Ablaufsteuerungen der Dynpros. Gepaart mit der Tatsache, dass auf dieser Ebene keine lokalen Programmvariablen und kein Mechanismus zur Übergabe von Parametern vorhanden sind, ergibt dies ein stark an die Benutzeroberfläche gekoppeltes Ablaufmodell.

Eine strikte Trennung zwischen der Darstellungsebene und der Ebene der Business-Logik wurde lange Zeit nicht konsequent durchgeführt. Bei der Komplexität des Systems R/3 ist es daher so gut wie unmöglich, bei eigenen Programmen sämtliche Plausibilisierungen und Ableitungen nachzubilden, die das System in seinen Dialogtransaktionen durchführt. Auch das Speichern von Informationen in den Anwendungstabellen des Systems R/3 sollte allenfalls mit größter Vorsicht vorgenommen werden, da an einer Transaktion häufig eine Vielzahl von Tabellen mit zum Teil nicht offensichtlichen Verbindungen beteiligt sind. Unbedachtes direktes Schreiben in der Datenbank führt demnach mit größter Wahrscheinlichkeit zu inkonsistenten Datenbeständen im System R/3. Seit einiger Zeit versucht SAP diesem Umstand durch Bereitstellung spezieller Funktionsbausteine, den Business APIs (BAPIs) entgegenzuwirken. Im Release 4.6 ist die Anzahl dieser BAPIs im Vergleich zu den vorigen Releaseständen erheblich gestiegen. Leider sind allerdings viele der BAPIs auch im Release 4.6C noch fehlerhaft und inperformant. Weiterhin ist der Programmcode der BAPIs oftmals so komplex, dass selbst einfache Modifikationen nur von sehr erfahrenen Programmierern vorgenommen werden können. Der klare Vorteil von Batch-Input ist hierbei, dass sämtliche Plausibilisierungen, die während der Dialogverarbeitung einer Transaktion durchgeführt werden, im Batch-Input ebenfalls aktiv sind. Dies führt zwar in der Anfangsphase zu vermehrten Transaktionsabbrüchen aufgrund fehlerhafter Batch-Input-Informationen, in der Folge jedoch zu sehr stabilem Code, der aus sich heraus in keiner Situation einen Datenschiefstand erzeugt.

6.3 Funktionsweise von Batch-Input

Beim Batch-Input werden normale Dialogtransaktionen ohne Benutzereingaben automatisch durchgeführt. Sämtliche Benutzeraktionen werden in Form einer Tabelle, der Batch-Input-Tabelle, vor Aufruf der eigentlichen Funktion beschrieben. Beim Ablauf werden lediglich die Aktionen durchgeführt, die in dieser Tabelle beschrieben sind. Zusätzlich dazu enthält diese Tabelle einen Eintrag für jedes ausgelöste `PROCESS-BEFORE_OUTPUT`-Ereignis. Für jedes PBO muss ein Eintrag in der Tabelle enthalten sein, der das prozessierte Dynpro beschreibt.

Wenn die Tabelle vollständig ist, kann die beschriebene Dialogtransaktion entweder direkt über die Anweisung `CALL TRANSACTION` ausgeführt werden, oder aber die Tabelle wird – ggf. mit mehreren gleichartigen Tabellen zusammen – in

einer so genannten Batch-Input-Mappe zur späteren Abarbeitung abgelegt. Diese Mappen wiederum können entweder direkt oder als Job eingeplant werden. Zu bemerken ist jedoch, dass – egal wie ein BTC ausgeführt wird – immer ein Dialogprozess mit der Verarbeitung beauftragt wird. Dies ist der Fall, da jedem BTC eine Dialogtransaktion zugrunde liegt. Ein Aufruf eines Batch-Inputs aus einem Verbucherbaustein heraus ist demnach nicht möglich und führt unweigerlich zu einem Verbucherabbruch. Ein Weg, um dennoch aus einem Verbucher heraus einen Batch-Input durchführen zu können ist, den Batch-Input oder `CALL TRANSACTION` in einem Report zu implementieren und diesen als Job einzuplanen, der beispielsweise nach Auslösen eines Events aktiviert wird. Näheres hierzu ist in den Kapiteln 7 und 8 zu finden.

Wie bereits erwähnt, enthält die BTC-Tabelle für jeden PBO und für jede Feldeingabe, die der Benutzer tätigen möchte, einen Eintrag. Die Verarbeitung der Tabelle erfolgt immer sequentiell, d.h. die Reihenfolge der Einträge der Tabelle entspricht der Abarbeitungsreihenfolge während der Verarbeitung. Das System prüft hierbei bei jedem PBO, ob die tatsächliche Abarbeitung der Transaktion dem beschriebenen Vorgehen entspricht. Wird ein nicht in der BTC-Tabelle beschriebenes Dynpro prozessiert oder wird ein beschriebenes Dynpro nicht an der entsprechenden Stelle von der Dialogtransaktion prozessiert, bricht die Verarbeitung des Batch-Input mit einem Fehler ab[1]. Gleichermaßen bricht die Verarbeitung ab, wenn ein Feldinhalt nicht wie in der BTC-Tabelle gesetzt werden konnte (z.B. weil das Feld nicht auf dem Dynpro vorhanden ist oder nicht eingabebereit ist) oder Pflichtfelder auf dem Dynpro nicht versorgt werden. Schließlich bricht die Transaktion auch ab, wenn eine E-Message von der Dialogtransaktion ausgegeben wird oder ein `COMMIT WORK` oder `ROLLBACK WORK`, der das Ende der LUW kennzeichnet, durchgeführt wird (siehe Kapitel 8).

Nochmals hervorzuheben ist, dass die BTC-Tabelle immer sequentiell abgearbeitet wird. Verzweigungen, Bedingungen oder andere Sprünge sind bei diesem Verfahren nicht möglich. Die Tabelle wird – beginnend mit dem ersten Satz, der ein PBO-Ereignis beschreibt – sequentiell Satz für Satz abgearbeitet. Für den Programmierer bedeutet dies, dass speziell die Dynprofolge der aufzurufenden Transaktion genau untersucht werden muss. Alle Fälle, in denen R/3 unterschiedliche Dynpros zur Anzeige bringt, müssen vorhergesehen werden oder der Batch-Input bricht fehlerhaft ab. Erschwerend hinzu kommt hierbei noch, dass einige Transaktionen im R/3-System bei der Bedienung durch einen realen Benutzer anders reagieren, als im `CALL TRANSACTION`-Modus oder bei Verwendung von Batch-Input. Nähere Informationen hierzu werden im Kapitel 6.9 dargestellt.

1. Ausgenommen von dieser Aussage sind Aufrufe von CALL TRANSACTION im Modus E, die in diesem Fall sichtbar in die Dialogtransaktion verzweigen.

6.4 Erzeugen einer Batch-Input-Tabelle

6.4.1 Grundsätzlicher Aufbau von BTC-Tabellen

Egal ob mit CALL TRANSACTION oder Batch-Input gearbeitet wird, liegt vor dem eigentlichen Aufruf der Aufbau der Batch-Input-Tabelle. Diese wird als interne Tabelle mit dem Aufbau der Data Dictionary-Structur BDCDATA (siehe Tabelle 6.1) definiert.

Feldname	Datentyp	Bedeutung
PROGRAMM	CHAR 40	Name des Modulpools
DYNPRO	NUMC 4	Dynponummer
DYNBEGIN	CHAR 1	Kennzeichen für Start eines Dynpros
FNAM	CHAR 132	Feldname
FVAL	CHAR 132	Feld-Inhalt

Tabelle 6.1
Aufbau der Struktur BDCDATA

In diese Tabelle müssen wie erwähnt für jedes PBO-Ereignis sowie alle Feldzuweisungen und Funktionscodes Einträge vorgenommen werden. Eine besondere Rolle kommt hierbei dem Feld DYNBEGIN zu. Wird in diesem Feld ein »X« abgelegt, bedeutet dies, dass der Satz sich auf ein neues Dynpro, sprich ein PBO-Ereignis bezieht. In diesem Fall ist der Inhalt der Felder FNAM sowie FVAL unerheblich. Ausgewertet werden lediglich die Inhalte der Felder PROGRAM und DYNPRO, welche eindeutig ein Dynpo im R/3-System beschreiben. PROGRAM enthält hierbei den Namen des Modulpools oder des Reports, in dem das jeweilige Dynpro mit der in DYNPRO definierten Dynpro-Nummer abgelegt ist.

Ist das Feld DYNBEGIN leer, werden die beiden genannten Felder igonriert und lediglich der Inhalt der Felder FNAM und FVAL vom System ausgewertet. FNAM enthält hierbei den Namen eines Dynpro-Feldes, wie er aus der technischen Hilfe zu jedem Dynpro-Element ermittelt werden kann (siehe Abbildung 6.1). Der Batch-Input-Name des Feldes ist hierbei im unteren Bereich des Fensters in der Gruppe »Feldbezeichnungen für Batch-Input« abgelegt. Der Inhalt der Variablen FVAL wird beim Abspielen der Tabelle in das jeweilige Dynpro-Feld geschrieben. Funktionstasten oder – allgemeiner ausgedrückt – Funktionscodes, die der Benutzer eingeben würde, werden über den besonderen Feldnamen BDC_OKCODE in der Tabelle abgelegt. Das Feld FVAL enthält in dieser Situation den auszuführenden Funktionscode. Soll der Cursor zur Laufzeit auf ein bestimmtes Dynpro-Element gesetzt werden, (z.B. um auf einem bestimmten Feld eine Aktion

durchzuführen), kann dies über den ebenfalls reservierten Feldnamen BDC_CURSOR bewerkstelligt werden. Das Feld FVAL enthält dann den BTC-Namen des gewünschten Dynpro-Feldes.

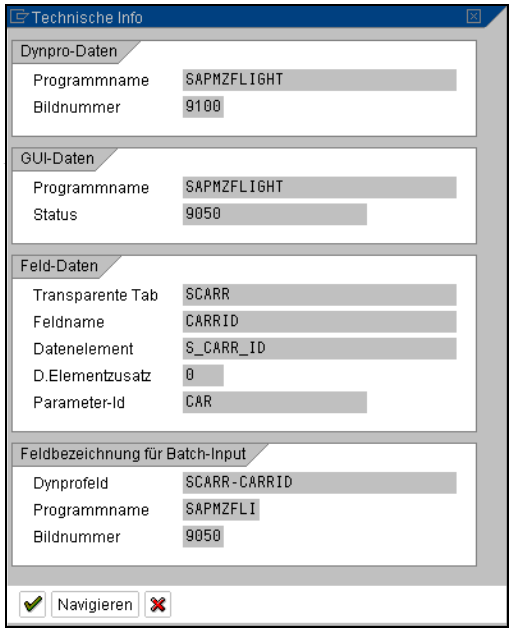

Abbildung 6.1
BTC-Name eines Dynpro-Feldes in der technischen Hilfe (© SAP AG)

6.4.2 Routinen zum Aufbau von BTC-Tabellen

Für jeden Dynpro-Wechsel und für jedes Setzen von Feldinhalten müssen immer wieder die gleichen Anweisungen ausgeführt werden, um die Kopfzeile der BTC-Tabelle zu löschen und die im jeweiligen Kontext benötigten Felder zu belegen. Die Programmzeilen, um beispielsweise den Start eines neuen Dynpros einzufügen, haben folgenden Aufbau.

```
CLEAR BDCTAB.
BDCTAB-DYNBEGIN = 'X'.
BDCTAB-PROGRAM  = 'SAPLCOIH'.
BDCTAB-DYNNR    = '0120'.
APPEND BDCTAB.
```

Dieser ist für jeden Dynprostart bis auf den Namen des Programms und der Dynpro-Nummer identisch, es ändern sich also lediglich zwei der dargestellten fünf Zeilen Code. Die Zeilen, um einen Feldinhalt zu setzen, füllen entsprechend die Felder FVAL und FNAM.

Wird für jeden Dynpro-Wechsel bzw. jedes Setzen von Feldinhalten diese Sequenz direkt in den Code eingefügt, wird zum einen das Programm sehr aufgebläht, zum anderen leidet die Lesbarkeit des Programmes erheblich. Aus diesem Grund verwenden Programmierer, die häufiger BTC-Abläufe realisieren müssen, für diese Aufgaben in der Regel Routinen, um den eigentlichen Programmcode zu entlasten.

Je nach persönlicher Vorliebe des einzelnen Programmierers können hierzu ein oder mehrere Routinen realisiert werden. Ebenfalls eine Entscheidung des Programmierers ist es, ob die Routinen als FORM-Routinen in einem – oft mehrfach verwendeten - Include, als Funktionsbausteine einer Funktionsgruppe oder als ABAP-Objects-Klassen realisiert werden. Bei Verwendung von Funktionsbausteinen wird der Vorteil des kompakten Codes allerdings zugunsten von Modularisierungsbemühungen aufgegeben, da der Aufruf eines Funktionsbausteins in der Regel länger ist als die fünf dargestellten Zeilen Code, die benötigt werden, wenn die Tabelle direkt versorgt wird. Der Ansatz mit ABAP-Objects verspricht große Vorteile bezüglich der Wiederverwendbarkeit. Auch die Kapselung der BTC-Tabelle spricht für die Verwendung dieses Ansatzes.

Aus Gründen der Einfachheit der Darstellung wird hier eine Lösung mit FORM-Routinen dargestellt. Die Integration der Funktionalitäten in eine der anderen Methoden ist jedoch sehr einfach.

Der Code könnte im System als Include abgelegt werden und an den verschiedensten Stellen im System verwendet werden. Zu beachten ist jedoch, dass in dem Programm, wo das Include eingebunden wird, keine Variable mit dem Namen BTC_TABLE existieren darf, da diese als globale Variable – einem der sehr seltenen Fällen wo dies sinnvoll erscheint – angelegt wurde.

```
DATA: BTC_TABLE LIKE BDCDATA OCCURS 0 WITH HEADER LINE.

*---------------------------------------------------
* Routine zum Löschen der aktuellen BTC-Tabelle
*---------------------------------------------------
FORM BTC_REFRESH.
  CLEAR BTC_TABLE.
  REFRESH BTC_TABLE.
ENDFORM.

*---------------------------------------------------
* Begin eines neuen Dynpros (bzw. PBO)
*---------------------------------------------------
FORM BTC_DYNPRO USING P_PROGRAM  LIKE  BDCDATA-PROGRAM
                      P_DYNPRO   LIKE  BDCDATA-DYNNR.
  CLEAR BTC_TABLE.

  BTC_TABLE-PROGRAM  = P_PROGRAM.
  BTC_TABLE-DYNPRO   = P_DYNPRO.
```

```
  BTC_TABLE-DYNBEGIN = 'X'.
  APPEND BTC_TABLE.
ENDFORM.

*---------------------------------------------------
* Feldwert setzen
*---------------------------------------------------
FORM BTC_DATA   USING P_FNAM
                      P_FVAL.
  CLEAR BTC_TABLE.

  BTC_TABLE-FNAM     = P_FNAM.
  BTC_TABLE FVAL     = P_FVAL.
  APPEND BTC_TABLE.
ENDFORM.

*---------------------------------------------------
* Aktulle BTC-Tabelle ermitteln
*---------------------------------------------------
FORM BTC_TABLE_GET TABLES P_BTC_TABLE STRUCTURE BDCDATA.
  P_BTC_TABLE[] = BTC_TABLE[].
  P_BTC_TABLE   = BTC_TABLE.
ENDFORM.

*---------------------------------------------------
* Aktuele bTC-Tabelle setzen
*---------------------------------------------------
FORM BTC_TABLE_SET TABLES P_BTC_TABLE STRUCTURE BDCDATA.
  BTC_TABLE[] = P_BTC_TABLE[].
ENDFORM.
```

Neben den beiden eigentlichen Routinen zum Anfügen eines Dynpro-Ereignisses und zum Setzen eines Feldwertes enthält der Code noch weitere Funktionen, um die BTC-Tabelle zu löschen, die aktuelle BTC-Tabelle zu ermitteln und zu setzen. Diese Funktionen bewirken, dass der Anwender der Funktionen niemals direkt auf die globale Variable `BTC_TABLE` zugreifen muss. Bei Verwendung von ABAP-Objects wäre die Tabelle ein Attribut der Klasse und von außen nicht zugreifbar. Hierdurch soll der Programmierer vom Wissen um Implementierungsdetails fremden Programmcodes weitestgehend befreit werden.

Wie erwähnt handelt es sich beim gezeigten Code lediglich um einen Vorschlag, wie solche Funktionen realisiert werden können. Andere Lösungsmöglichkeiten sind möglich und im Einzelfall sicherlich auch sinnvoller als die dargestellte Implementierung.

6.4.3 Verwendung numerischer Felder

Besondere Aufmerksamkeit beim Aufbau der Batch-Input-Tabelle verdienen numerische Felder. Anders als in Charakter-Feldern, findet die Eingabe in numerischen Feldern in der Regel rechtsbündig statt. Da das Feld FVAL der BDCDATA-Struktur 132 Stellen breit ist, würde eine einfache WRITE-Zuweisung eines numerischen Feldes an dieses Feld bewirken, dass der Feldinhalt rechtsbündig, d.h. mit der 132. Stelle endend im Feld FVAL platziert würde. Beim Batch-Input werden die Feldinhalte jedoch von links beginnend in die Dynpro-Felder der aufgerufenen Transaktion übertragen. Wenn der benutzte Inhalt des Feldes FVAL breiter ist als das jeweilige Dynpro-Feld, bricht der Batch-Input mit einer Fehlermeldung ab, da der gewünschte Feldinhalt nicht korrekt gesetzt werden kann.

Um dies zu verhindern, muss sichergestellt sein, dass die verwendete Breite der Variable FVAL nicht größer ist als die tatsächliche Breite des Dynpro-Feldes, in das der Inhalt übertragen werden soll. Dies kann durch Verwendung einer Hilfsvariable, in die der numerische Wert in der gewünschten Länge übertragen wird, realisiert werden.

```
...
  DATA: L_FVAL   LIKE BDCDATA-FVAL.

  CLEAR L_FVAL.
  WRITE <num-Variable> TO L_FVAL(<Breite>).
  PERFORM BTC_DATA USING <Feldname>
                         L_FVAL.
...
```

In der WRITE-Anweisung wird der numerische Wert rechtsbündig in den durch <Breite> definierten Bereich der Variable L_FVAL übertragen. Mit der vorangestellten CLEAR-Anweisung wird sichergestellt, dass nicht noch die Reste einer vorherigen Zuweisung außerhalb des hier verwendeten Bereichs enthalten ist und den Batch-Input zum Abbruch bringt.

Obwohl die erweiterte Syntaxprüfung von R/3 an dieser Stelle die Verwendung von MOVE anstatt WRITE vorschlägt, ist dies nicht zu empfehlen, da die MOVE-Anweisung – im Gegensatz zur WRITE-Anweisung – Konvertierungsexits nicht berücksichtigt und daher gegebenenfalls nicht interpretierbare Werte in die Dynpro-Felder übertragen werden können.

Das hier gezeigte Vorgehen gilt nicht nur für numerische Felder, sondern kann auch bei Problemen mit Datums- sowie Uhrzeitfeldern verwendet werden.

6.4.4 Step-Loops und Table-Views

Eine besondere Herausforderung bei der Erstellung von Batch-Input-Abläufen bilden Step-Loops und Table-Views. In beiden wird eine Tabelle von gleichartig-

gen Daten angezeigt. Während im Step-Loop-Verfahren die Informationen eines Eintrags auch mehrzeilig dargestellt werden können, ist dies bei Verwendung von Table-Views nicht möglich. Hier besteht jedoch die Möglichkeit des horizontalen Rollens, falls mehr Felder dargestellt werden sollen, als Platz auf dem Dynpro ist. Im Folgenden wird nur noch die Rede von Table-Views sein, da dies die von SAP für die Zukunft empfohlene Methode der Darstellung ist. Die Angaben gelten jedoch in gleicher Weise auch für Step-Loops.

Die zwei großen Themenbereiche, die hier angesprochen werden sollen, sind die Frage, wie im Batch-Input auf eine bestimmte Zeile des Table-Views zugegriffen werden kann und die Problematik, dass zum Zeitpunkt der Programmierung speziell bei Step-Loops unter Umständen nicht bekannt ist, wieviele Zeilen auf dem Bildschirm des Benutzers sichtbar sind.

Wie kann auf die Zeilen eines Step-Loops zugegriffen werden?

Im Batch-Input kann auf die einzelnen Zeilen eines Table-Views über einen Index zugegriffen werden. Dieser Index nummeriert die sichtbaren Zeilen des Table-Views, beginnend mit 1 durch. Auf die Felder einer Zeile kann zugegriffen werden, indem dem normalen BTC-Namen des Feldes der Index der jeweiligen Zeile in Klammern angefügt wird.

Das Beispiel in Abbildung 6.2 zeigt die Darstellung eines Arbeitsplans aus dem Bereich Instandhaltung. Die einzelnen Vorgänge der Anleitung werden hierbei in einem Table-View dargestellt. Die Vorgangsnummer ist im Feld `AFVGD-VORNR` dargestellt. Soll auf die Vorgangsnummer des vierten dargestellten Vorgangs im Batch-Input zugegriffen werden, muss der Feldname `AFVGD-VORNR(4)` in die Batch-Input-Tabelle übernommen werden.

Hierbei darf die gesamte Zeichenkette keine Leerzeichen enthalten. Andernfalls würde der Feldname im Batch-Input nicht erkannt werden.

```
...
DATA: L_INDEX(20)       TYPE C,
      L_FNAME_WORK(100) TYPE C,
      L_FNAM            LIKE BDCDATA-FNAM.

WRITE SY-INDEX TO L_INDEX.
CONCATENATE 'AFVGD-VORNR('
            L_INDEX
            ')'
       INTO L_FNAME_WORK.
CONDENSE L_FNAME_WORK NO-GAPS.
L_FNAM = L_FNAME_WORK.
...
```

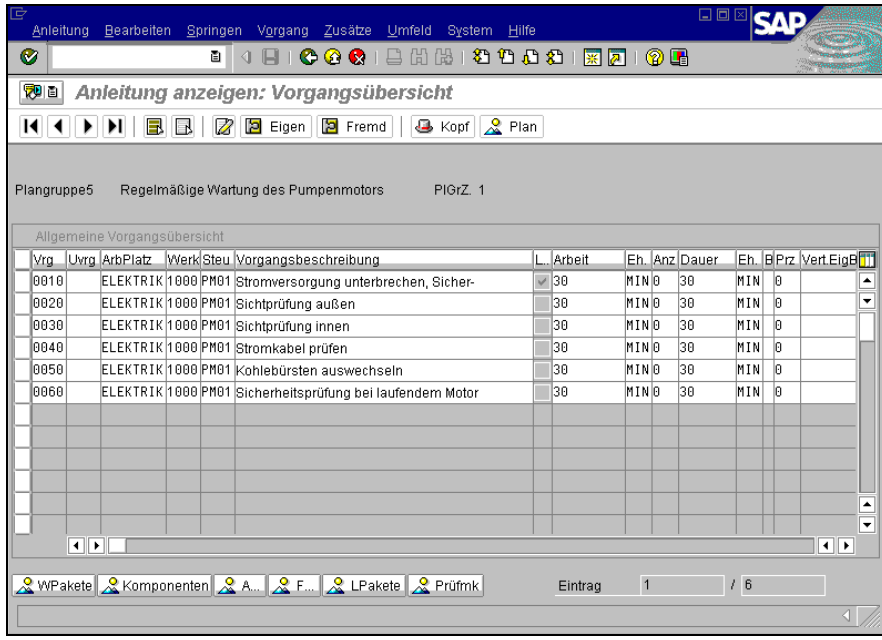

Abbildung 6.2
Darstellung einer Anleitung aus dem Bereich PM mit einem Table-View (© SAP AG)

Im gezeigten Programmfragment wird zunächst der in einem numerischen Feld – hier `SY-INDEX` – vorhandene Index in ein Zeichenfeld übertragen. Im Anschluss daran wird der Feldname zusammen mit dem in Klammern stehenden Index konkateniert. Der hier entstehende String enthält innerhalb der Klammern mehrere Leerzeichen, welche – vor der endgültigen Zuweisung zum Feld `FNAM` mit der Anweisung `CONDENSE` entfernt werden.

Auch dieses Beispiel ist nur exemplarisch zu sehen. Zur Lösung dieses Problems sind auch eine Vielzahl anderer Ansätze denkbar.

Wieviele Zeilen werden auf dem Bildschirm angezeigt?

Im vorigen Abschnitt wurde erwähnt, dass der Zugriff auf die Zeilen eines Table-Views über den Index der dargestellten Zeile erfolgt. Wenn ein Index verwendet wird, der größer als die Anzahl der dargestellten Zeilen ist, bricht der Batch-Input ab, da das gewünschte Feld nicht auf dem Dynpro vorhanden ist. Dadurch stellt sich die Frage, woher der Entwickler weiß, wieviele Zeilen auf dem Dynpro angezeigt werden. Bei Step-Loops mit fester Länge sowie Table-Views ist die Aussage einfach. Es sind so viele Zeilen auf dem Dynpro vorhanden, wie die fixe Länge des Step-Loops ist. Ggf. ist das Gesamte GUI-Fenster zwar scrollbar, aber für den Batch-Input sind alle Felder vorhanden. Ungünsti-

ger verhält es sich bei Step-Loops variabler Länge. Auch hier ist die Antwort auf die Frage einfach, allerdings auch frustrierend: Der Programmierer weiß es nicht. Sicher ist lediglich, dass mindestens eine Zeile angezeigt wird.

Am besten und sichersten ist es, wenn der Batch-Input so gestaltet werden kann, dass die gewünschte Zeile am Anfang der Liste, also in der ersten Zeile des Table-Views, dargestellt wird. Verschiedene Transaktionen (z.B. im Bereich der Pflege von Arbeitsplänen) lässt dies über Funktionscodes und Dialogfenster zu. Bevor BTC-Abläufe erstellt werden, sollte demnach genau untersucht werden, ob die aufzurufende Transaktion derartige Funktionalitäten zur Verfügung stellt.

Ist dies nicht möglich, und die Bearbeitung der in der Liste dargestellten Sätze kann nicht auf andere Art gelöst werden, muss eine Dynpro-Größe – z.B. die Standard-Größe – angenommen und das Programm hierauf abgestimmt werden. Der Programmierer muss sich jedoch jederzeit im Klaren sein, dass der produzierte Code sehr fehleranfällig ist, falls die Benutzer häufig unter anderen Bildschirmauflösungen oder Fenstergrößen arbeiten. Dieses Problem kann jedoch unter Umständen vernachlässigt werden, wenn die Dynpros der Transaktion, die den Batch-Input aufruft, durch organisatorische oder gestalterische Maßnahmen auf den Vollbildmodus der Benutzer ausgelegt werden. Um sicher zu gehen, sollte jedoch der Fall des Transaktionsabbruchs in solchen Fällen besonders berücksichtigt werden und die Batch-Input-Daten beispielsweise in Form einer BTC-Mappe zur späteren Nachverbuchung abgelegt werden.

6.5 Verwendung von CALL TRANSACTION

Nachdem die Batch-Input-Tabelle aufgebaut wurde, kann diese dem System zur Abarbeitung übergeben werden. Hierfür existiert in der Programmiersprache ABAP die Anweisung `CALL TRANSACTION` mit der unten dargestellten Syntax.

```
CALL TRANSACTION <Transaktionscode>
                 [ AND SKIP FIRST SCREEN ]
                 [ USING <BTC-Tabelle>
                    [ UPDATE { 'S' | 'A' | 'L' }   ]
                    [ MODE { 'A' | 'N' | 'E' } ]
                    [ MESSAGES INTO <Message-Tab> ] ].
```

Der Anweisung muss auf jeden Fall der aufzurufende Transaktionscode übergeben werden. Wird keiner der weiteren Parameter spezifiziert, gelangt der Benutzer so direkt in die gewünschte Transaktion. Dies ist ein häufig verwendetes Mittel, um Verzweigungen über Transaktionsgrenzen hinweg zu realisieren. Sobald die gerufene Transaktkion beendet wird, fährt ABAP mit der Ausführung der im Porgramm auf die, dem `CALL TRANSACTION` folgenden Anweisung fort.

Werden zumindest für alle Mussfelder des Eingangsdynpro der gerufenen Transaktion Werte auf programmtechnischem Weg an die aufgerufene Transaktion übergeben, kann der erste Screen der Transaktion über den Zusatz AND SKIP FIRST SCREEN übersprungen werden. Als Möglichkeit für die Wertübergabe kommt derzeit ausschließlich das Konstrukt von Parameter-Ids (SPA/GPA-Parameter) in Frage.

Der Anweisung CALL TRANSACTION kann mit der USING-Klausel eine Batch-Input-Tabelle (auch BTC-Tabelle genannt) übergeben werden, die explizite Anweisungen für den Ablauf der Transaktion enthält. Diese Tabelle muss den in Abschnitt 6.4 beschriebenen Aufbau der Data Dictionary-Struktur BDCDATA haben. Die übergebene Tabelle muss hierbei nicht den vollständigen Transaktionsablauf enthalten. Ähnlich wie bei der Option AND SKIP FIRST SCREEN kann durch diesen Mechanismus lediglich eine Vorbelegung der gerufenen Transaktion erfolgen. Eine Beschränkung auf Felder der ersten Maske, wie sie bei der zuerst genannten Option der Fall ist, besteht hierbei nicht. Es können demnach bereits komplexe Abläufe in der Transaktion durchgeführt werden und dem Benutzer lediglich ein Kontrollblick auf die erfassten Daten gewährt werden. Dies funktioniert allerdings nur bei Verwendung des Aufrufmodus »E« (siehe unten).

Die übrigen Parameter der Anweisung CALL TRANSACTION machen im Wesentlichen bei Verwendung des letztgenannten Mechanismus Sinn. Über CALL TRANSACTION wird eine andere Transaktion in einem eigenen Logical Unit of Work aufgerufen. Neben der Tatsache, dass dies bedeutet, dass dem mindestens eine eigene Transaktion auf Datenbank-Ebene zugeordnet ist, heisst dies auch, dass beim Abschluss der LUW Aktionen im Rahmen des Verbucherkonzepts von R/3 stattfinden können. Diese laufen normalerweise asynchron zur Verarbeitung des Dialogprozesses in so genannten Verbucherprozessen ab (genaueres hierzu ist in Kapitel 8 dargestellt). Häufig ist es jedoch nicht wünschenswert, dass die Verbuchung der mit CALL TRANSACTION gerufenen Transaktionen asynchron zum übrigen Programmlauf erfolgt. Dies gilt insbesondere dann, wenn mehrere aufeinander aufbauende Transaktionen in Folge aus einem Programm heraus aufgerufen werden sollen. Wenn ein realer Benutzer diese Folge ausführen würde, bildet dies meist kein Problem, da in der Regel die Verbuchung im System schneller ist, als der Benutzer die Aktionen initiiert. Ist der Benutzer doch schneller, erhält er eine Meldung und wird die Aktion später erneut durchführen. Anders verhält es sich, wenn die Transaktionsfolge von einem Programm ausgelöst wird. Die erwähnte Verzögerung beim Transaktionswechsel wird in der Regel vernachlässigbar kurz sein. Auch kann das Programm nur bedingt auf die Fehlermeldung reagieren. Das Programmieren von Verzögerungsschleifen würde das System insgesamt unnötig belasten und führt zu einem insgesamt schwer beherrschbaren Zustand. Ein Abbruch nach mehreren erfolglosen Versuchen ist hier auch nicht das beste Mittel, tauchen diese Probleme doch häufig auf und würden umfangreiche manuelle Nacharbeiten nach sich ziehen. Um diese Problematik zu umgehen, bietet die Anweisung CALL TRANSACTION die Option UPDATE an. Wird hier der Wert »S« übergeben, wird die

Verbuchung der gerufenen Transaktion synchron durchgeführt, d.h. der Aufruf von CALL TRANSACTION blockiert, bis alle Verbuchungen der gerufenen Transaktion abgeschlossen sind. Entsprechend das Gegenteil bewirkt der Wert »A«, der, wenn die Option nicht angegeben ist, der Defaultwert für diese Einstellung ist. Soll die Verbuchung im Prozess des Aufrufers erfolgen, kann auch der Wert »L« übergeben werden.

Als weiterer Parameter kann der Modus, in dem die gerufene Transaktion ablaufen soll, festgelegt werden. Hierüber kann die Sichtbarkeit des Transaktionsablaufs während des Call-Modus definiert werden. Es stehen drei Modi zur Verfügung. Wird der Modus auf »N« gesetzt, wird die Transaktion vollständig im Hintergrund, d.h. dunkel, ausgeführt. Kein Dynpro der Transaktion wird sichtbar. Tritt ein Fehler während der Verarbeitung auf, bricht die Transaktion ab und CALL TRANSACTION liefert einen Fehlercode zurück. Im Gegensatz hierzu steht der Modus »A«, bei dem die Transaktion vollständig sichtbar abgespielt wird. Der Benutzer sieht hierbei jedes auftretende Dynpro. Über den Batch-Input gesetzte Feldwerte werden hierbei farblich gekennzeichnet dargestellt (standardmäßig in rot). Dieser Modus eignet sich sehr gut, um semantische (d.h. Anwendungs-) Fehler im Ablauf eines Batch-Input zu lokalisieren. Als Kompromiss zwischen diesen beiden Modi existiert ncoh der Modus »E«, bei dem der Batch-Input ebenfalls vollständig dunkel prozessiert wird. Tritt jedoch ein Fehler während der Verarbeitung auf (E-Message), schaltet der Batch-Input in den sichtbaren Modus um, der Benutzer bekommt die Fehlersituation angezeigt und kann den Fehler unter Umständen korrigieren, um die Transaktion erfolgreich zu Ende zu bringen. In produktiven Umgebungen wird der Wert »A« nur in seltenen Fällen angewandt, konfrontiert er den Benutzer doch mit Masken, die eigentlich von alleine ablaufen sollten. Hier wird dieser Modus allenfalls zur Fehlersuche verwandt werden. Im Normalfall werden Transaktionen daher im Modus ,N' oder ,E' ablaufen.

Während der Verarbeitung einer Transaktion können die unterschiedlichsten Meldungen auftreten, die zum Teil Fehlersituationen beschreiben, zum Teil aber lediglich Informationen über den Transaktionsablauf geben. Wird eine Transaktion von einem Benutzer durchgeführt, werden diese Meldungen normalerweise am unteren Rand des SAPGUI oder in Pop-Up-Fenstern angezeigt. Beim Batch-Input besteht jedoch nicht die Möglichkeit, auf diese Anzeigen zuzugreifen. Daher bietet die Anweisung CALL TRANSACTION eine Möglichkeit, alle Meldungen, die während der Verarbeitung des Batch-Input aufgetreten sind, in einer Tabelle zu sammeln und an den rufenden Programmcode zu übergeben. Hierzu muss an CALL TRANSACTION über den Parameter MESSAGES INTO <Msg-Tab> eine interne Tabelle übergeben werden, die den Aufbau der Data Dictionary-Struktur BDCMSGCOLL hat (siehe Tabelle 6.2).

Feldname	Datentyp	Bedeutung
TCODE	CHAR 20	Transaktionscode
DYNAME	CHAR 40	Programmname
DYNUMB	CHAR 4	Dynpro-Nummer
MSGTYP	CHAR 1	Message Typ
MSGSPRA	CHAR 1	Sprachkennzeichen der Meldung
MSGID	CHAR 20	Message ID
MSGNR	CHAR 3	Message Nummer
MSGV1	CHAR 100	Variabler Nachrichtenteil 1
MSGV2	CHAR 100	Variabler Nachrichtenteil 2
MSGV3	CHAR 100	Variabler Nachrichtenteil 3
MSGV4	CHAR 100	Variabler Nachrichtenteil 4
ENV	CHAR 4	Monitoring Aktivitäten
FLDNAME	CHAR 132	BDC-Feldnamen

Tabelle 6.2
Aufbau der Struktur `BDCMSGCOLL`

Jede Info- (`I`-), Warnungs- (`W`-), Fehler- (`E`-) oder Abbruch- (`A`-) Meldung, die während der Verarbeitung aufgetreten ist, wird in dieser Tabelle in der Reihenfolge ihres Auftretens abgelegt.

In der Struktur wird nicht der Klartext der jeweiligen Meldung abgelegt sondern die interne Repräsentation der Meldungen, wie sie auch in der `SY`-Struktur zur Laufzeit zu finden ist. Mit den hier verfügbaren Informationen kann allerdings leicht der Klartext der Meldung über einen Aufruf eines passenden Funktionsbausteins (Beispielsweise `BAPI_MESSAGE_GET_DETAIL`) ermittelt werden.

Der Rückgabewert der Anweisung `CALL TRANSACTION` wird im Systemfeld `SY-SUBRC` als numerischer Fehlercode abgelegt. Der Wert `0` bedeutet, dass während der Verarbeitung kein Fehler aufgetreten ist. Werte ungleich `0` zeigen eine Fehlersituation an, der wohl häufigste Fehler wird der Fehler `1001` sein. Dieser bedeutet, dass ein in der Transaktion auftauchendes Dynpro nicht in der Batch-Input-Tabelle vorgesehen ist (z.B. ein unerwartet erscheinendes Pop-Up-Fenster).

Die Verarbeitung des Batch-Input bricht ab, wenn das Ende der BTC-Tabelle erreicht, die Transaktion beendet bzw. globaler ausgedrückt, ein COMMIT WORK abgesetzt wurde (auf diesen Punkt wird in Abschnitt 6.9 noch ausführlicher eingegangen).

Mit dem Funktionsbaustein BAPI_MESSAGE_GET_DETAIL können sowohl Kurz- als auch Langtext einer Meldung ermittelt werden. Die in der Nachricht definierten Platzhalter werden hierbei durch die als Parameter übergebenen Variablen ersetzt.

Das BAPI hat folgende Aufrufschnittstelle:

```
FUNCTION BAPI_MESSAGE_GET_DETAIL
   IMPORT
      ID
      NUMBER
      LANGUAGE
      TEXTFORMAT
      MESSAGE_V1
      MESSAGE_V2
      MESSAGE_V3
      MESSAGE_V4
   EXPORT
      MESSAGE
      RETURN
   TABLES
      TEXT.
```

In den Feldern ID, NUMBER, LANGUAGE sowie MESSAGE_V1 bis _V4 können die entsprechenden Felder der Struktur BDCMSGCOLL übergeben werden. Im Parameter TEXTFORMAT kann spezifiziert werden, in welchem Format die Rückgaben generiert werden sollen. In Frage kommen hier die Werte HTM für HTML, RTF für Rich Text Format, SCR für SAPScript sowie ASC für die Darstellung im ASCII-Format. Welches Format hier gewählt wird, hängt vom jeweiligen Anwendungsfall ab.

Bei erfolgreicher Verarbeitung wird im Rückgabeparameter MESSAGE der Kurztext der Nachricht sowie im Tabellenparameter TEXT der Langtext der Nachricht in Form von max. 255-Zeichen langen Zeilen zurückgeliefert. Schlägt die Verarbeitung fehl, wird der aufgetretene Fehler im Rückgabewert RETURN, welcher den Aufbau der Struktur xx hat, beschrieben.

Weitere Informationen zu diesem Funktionsbaustein können der Dokumentation zum Funktionsbaustein entnommen werden.

6.6 Erzeugung einer Batch-Input-Mappe

Alternativ zur direkten Ausführung von Batch-Inputs können diese auch in Form von so genannten Batch-Input-Mappen im System gesammelt und zu einem späteren Zeitpunkt ausgeführt werden. Dieses Vorgehen ist insbesondere bei Datenübernahmen, dem eigentlichen Anwendungsgebiet von Batch-Input, sinnvoll. In einer solchen Mappe können die BTC-Abläufe von mehreren Transaktionsaufrufen abgelegt werden. Diese Abläufe müssen nicht zwangsweise die gleiche R/3-Transaktion ausführen. Bei Datenübernahmen oder –migrationen sind Batch-Input-Mappen mit mehreren tausend Dynpros und hunderten Transaktionsaufrufen keine Seltenheit. Bei diesen Datenmengen wird häufig von der Methode Gebrauch gemacht, an Stelle einer großen BTC-Mappe eine Vielzahl kleinerer Mappen, die gegebenenfalls auch parallel abgespielt werden können, anzulegen.

Im System werden die BTC-Mappen in der Transaktion `SM35` verwaltet (siehe Abbildung 6.3). BTC-Mappen können hierbei die Zustände.

- neu
- fehlerhaft
- verarbeitet
- in Bearbeitung
- im Hintergrund
- in Erstellung
- gesperrt

annehmen. Gruppiert nach diesen Zuständen werden Mappen in der Transaktion `SM35` angezeigt.

Batch-Input-Mappen werden von Programmen erzeugt. Diese bedienen sich hierzu verschiedener Funktionsbausteine, die allesamt in der Funktionsgruppe `SBDC` abgelegt sind.

Ähnlich wie beim Schreiben einer Datei, muss eine Batch-Input-Mappe vor dem Schreiben geöffnet werden und nach Beendigung aller Schreibvorgänge geschlossen werden. Hierfür stehen die Funktionsbausteine `BDC_OPEN_GROUP` sowie `BDC_CLOSE_GROUP` zur Verfügung. Eine Mappe wird mit einem Mappennamen erstellt, der nicht eindeutig sein muss. SAP vergibt für die erstellten Mappen eindeutige Bezeichner in Form einer ID.

Zwischen diesen beiden Funktionsaufrufen können in die Mappe beliebig viele Batch-Input-Abläufe über Aufrufe des Funktionsbausteins `BDC_INSERT` eingefügt werden. Die Schnittstellen dieser Funktionsbausteine ist im Abschnitt 6.10 dargestellt.

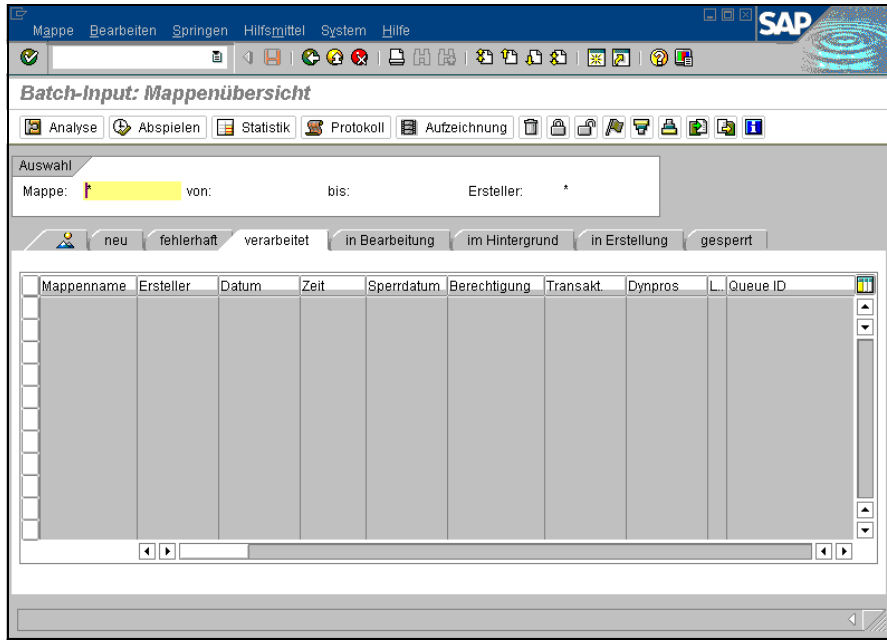

Abbildung 6.3
Transaktion SM35: Batch-Input Mappenübersicht (© SAP AG)

Während der prinzipielle Programmablauf eines mit `CALL TRANSACTION` arbeitenden Programmes, unter Verwendung der in Abschnitt 6.4.2 definierten Routinen, folgenden Aufbau hat,

```
DATA: L_BTC_DATA LIKE BDCDATA OCCURS 0.


PERFORM BTC_REFRESH..
PERFORM BUILD_BTC.     "eigene Routine zum Aufbau des BTC-Ablaufes
PERFORM BTC_TABLE_GET TABLES L_BTC_DATA.
CALL TRANSACTION '<TCODE>'
                 USING L_BTC_DATA.
```

ist der übliche Aufbau eines mit Mappen arbeitenden Programmes schematisch dargesellt

```
DATA: L_BTC_DATA LIKE BDCDATA OCCURS 0.

CALL FUNCTION 'BDC_OPEN_GROUP'

PERFORM BTC_REFRESH.
PERFORM BUILD_BTC.     "eigene Routine zum Aufbau des BTC-Ablaufes
PERFORM BTC_TABLE_GET TABLES L_BTC_DATA.
```

```
CALL FUNCTION 'BDC_INSERT'

CALL FUNCTION 'BDC_CLOSE_GROUP'.
```

Insbesondere bei Schnittstellenprogrammen, die von außerhalb des R/3-Systems aufgerufen werden, findet sich oft auch eine Kombination aus diesen beiden Möglichkeiten. Im Falle des Fehlschlagens eines `CALL TRANSACTION`-Aufrufes werden die fehlerhaften Abläufe zur späteren Verarbeitung in einer BTC-Mappe abgelegt.

6.7 Programmgesteuertes Abspielen von Batch-Input-Mappen

Um Batch-Input-Mappen automatisch zu starten, stellt SAP einen Report namens `RSBDCSUB` zur Verfügung. Im Selektionsbild dieses Programms kann eine Eingrenzung der gespeicherten Batch-Input-Mappen über den Namen, das Erstellungsdatum sowie den Status erfolgen. Alle Mappen, die den jeweiligen Kriterien entsprechen, werden anschließend vom Report gestartet. D.h. Mappen mit gleichem Namen werden auch zusammen verarbeitet, wenn keine weitere Einschränkung über das Erstellungsdatum erfolgt.

Werden keine Einschränkungen gemacht, können mit diesem Programm alle noch nicht verarbeiteten Mappen abgespielt werden.

Sollen Mappen programmgesteuert gestartet werden, erfolgt dies, indem der Report `RSBDCSUB` über die `SUBMIT`-Anweisung der ABAP-Sprache gestartet wird. Weiterhin kann der Report auch als Job eingeplant werden und über eine Reportvariante die zu verarbeitenden Mappen eingeschränkt werden.

6.8 Der Batch-Input-Recorder

Da das Erstellen von Batch-Input-Tabellen recht aufwendig ist und viel Routinearbeit beinhaltet, wurde von SAP schon früh ein Werkzeug zur Verfügung gestellt, mit dem einfach die Informationen für einen Batch-Input aufgenommen werden und anschließend zur späteren Weiterverarbeitung genutzt werden können. Dieses Werkzeug wird Batch-Input-Recorder genannt und steht im R/3-System mit der Transaktion `SHDB` zur Verfügung (Abbildung 6.4). Im Batch-Input-Recorder können Standard-Transaktionen aufgerufen und dabei sämtliche Dialogschritte aufgenommen und zunächst in einer internen Struktur abgespeichert werden. Aus den hier gewonnenen Informationen kann anschließend entweder ein ABAP-Programm, ein Funktionsbaustein oder eine Batch-Input-Mappe erstellt werden.

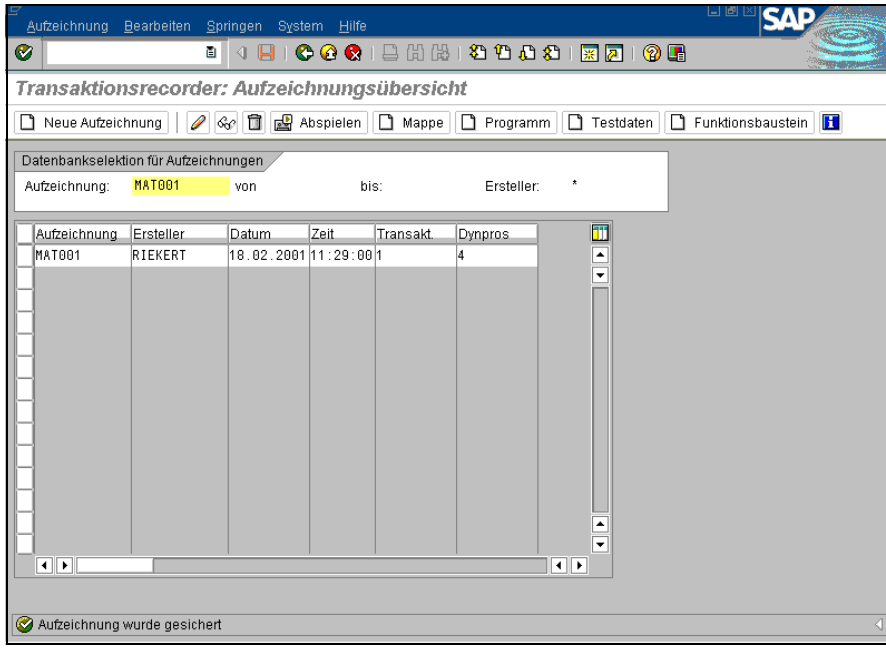

Abbildung 6.4
Benutzeroberfläche des Batch-Input-Recorders (© SAP AG)

In der Liste der Aufzeichnungen werden neben dem Namen der Aufzeichnung und dem Namen des Erstellers mit Erstellungszeitpunkt auch die Anzahl der in der Aufzeichnung gespeicherten Transaktionsaufrufe sowie die Anzahl der innerhalb der Aufzeichnung prozessierten Dynpros ausgegeben. Im Beispiel handelt es sich um eine sehr einfache Aufzeichnung aus dem Bereich des Materialstamms.

6.8.1 Anlegen und Bearbeiten einer Aufzeichnung

Eine neue Aufzeichnung kann über die Schaltfläche »Neue Aufzeichnung« am linken Rand der Schaltflächenleiste begonnen werden. Im daraufhin erscheinenden Fenster (Abbildung 6.5) muss der Name der Aufzeichnung sowie der Transaktionscode der aufzuzeichnenden Standard-Transaktion (im Beispiel die Transaktion »Materialstamm ändern«, `MM02`) festgelegt werden. Über die Schaltfläche »Aufzeichnung beginnen« gelangt der Benutzer anschließend in die Standard-Transaktion und kann diese ganz normal durchlaufen. Sämtliche Eingaben (mit Ausnahme von Werthilfen usw.) werden hierbei vom System registriert. Der einzige Unterschied während der Transaktionsverarbeitung, die für den Benutzer sichtbar wird, sind Meldungen in der Statuszeile beim Dynpro-Wechsel, die anzeigen, dass die Aufzeichnung der Transaktion aktiv ist.

Mit Beendigung der Transaktion gelangt der Benutzer wieder in die Maske des Batch-Input-Recorders zurück und kann die aufgezeichnete Sequenz speichern oder weiterverarbeiten.

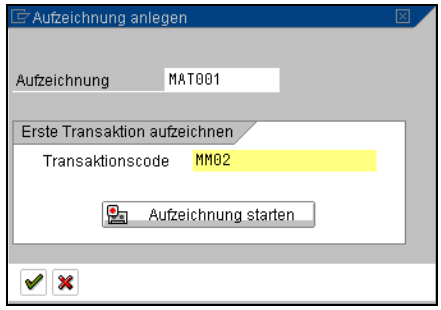

Abbildung 6.5
Attribute einer neuen Aufzeichnung (© SAP AG)

Innerhalb des Batch-Input-Recorders kann die Aufzeichnung angezeigt und modifiziert werden. Hierfür stehen die zweite und dritte Schaltfläche der Schaltflächenleiste des Batch-Input-Recorders zur Verfügung. Soll die Aufzeichnung verändert werden, gelangt der Benutzer in die in Abbildung 6.6 gezeigte Maske. Hier werden alle Schritte, die der Batch-Input-Recorder aufgezeichnet hat, zur Bearbeitung angezeigt.

Die erste Zeile der Aufzeichnung bedeutet hierbei den Transaktionsaufruf der Transaktion MM02 selbst. Die Zeilen 2 bis 5 behandeln das Einstiegsdynpro der Transaktion. Die Verarbeitung beginnt mit der Spezifikation des angezeigten Dynpros (dem Dynpro `0060` in der Funktionsgruppe `MGMM`). Anschließend wird über den reservierten Feldnamen `BDC_CURSOR` festgelegt, in welchem Feld der Maske der Eingabecursor stehen soll. Diese Information ist für einen vollständig dunkel ablaufenden Batch-Input normalerweise unerheblich und kann in diesem Fall ohne Probleme aus der Aufzeichnung gelöscht werden. Relevant ist das Setzen des Eingabecursors lediglich, wenn die Batch-Input-Tabelle nur benutzt werden soll, um sichtbar in eine Transaktion zu springen oder wenn eine Drucktaste auf einem Dynpro betätigt werden soll. In diesem Fall muss der Cursor auf die Drucktaste gesetzt und anschließend der Funktionscode der Drucktaste ausgelöst werden.

Über den ebenfalls reservierten Feldnamen `BDC_OKCODE` wird der Funktionscode festgelegt, mit dem das PAI-Ereignis ausgelöst werden soll. Während der Verarbeitung eines Dynpros wird dieses Feld nur einmal gesetzt. An welcher Position innerhalb der Einträge für ein Dynpro der OK-Code definiert wird, spielt hierbei keine Rolle. R/3 verarbeitet alle Zeilen, die zu einem Dynpro gehören und löst anschließend ein PAI aus. Im Beispiel wird der OK-Code `/00` gesetzt, welcher gleichbedeutend mit der ↵-Taste ist. Über das Konstrukt `/xx` können

ansonsten die Funktionstasten angesprochen werden. Der Wert `/01` würde demnach ein Auslösen der Funktionstaste F1 bedeuten.

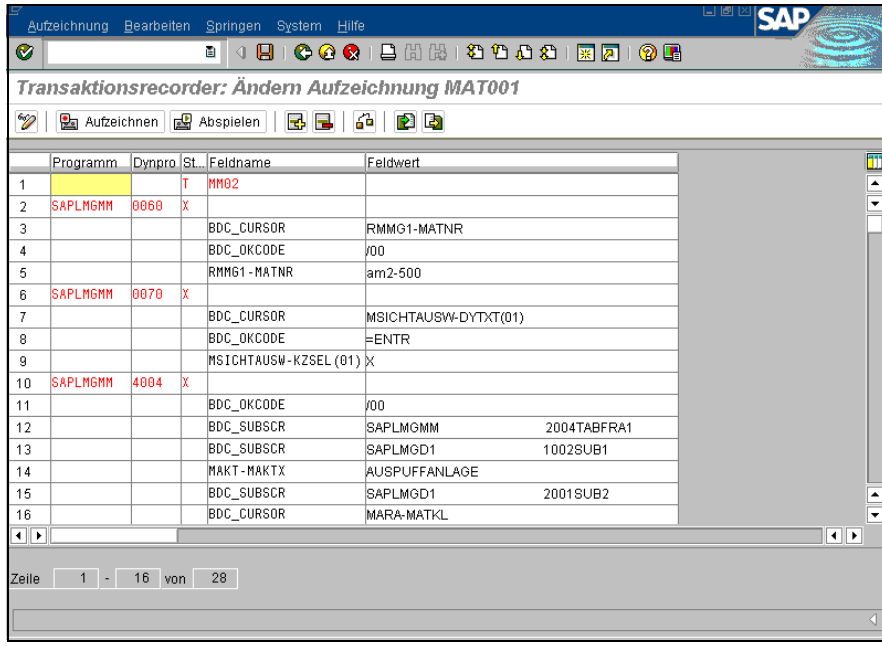

Abbildung 6.6
Ändern einer Aufzeichnung (© SAP AG)

Auf den folgenden Dynpros der Transaktion taucht weiterhin noch der reservierte Feldname `BDC_SUBSCR` auf (z.B. Zeile 12 der Aufzeichnung). Hier kann definiert werden, welchen Subscreen die Aufzeichnung im jeweiligen Subscreen-Bereich der Transaktion erwartet. Der dargestellte Ablauf würde beispielsweise abbrechen, wenn zur Laufzeit im Subscreen-Bereich `TABFRA1` des Dynpros `SAPLMGMM 4004` nicht der Screen `SAPLMGMM 2004` angezeigt würde. Diese Anweisungen sind jedoch für den Batch-Input-Ablauf nicht zwingend erforderlich und können demnach im Beispiel ebenfalls ohne Probleme gelöscht werden.

In der in Abbildung 6.6 dargestellten Maske können aufgezeichnete Zeilen verändert oder auch Zeilen gelöscht oder hinzugefügt werden. Um die Modifikationen zu testen, kann die Tabelle auch erneut abgespielt werden.

6.8.2 Weiterverarbeiten der Aufzeichnung

Die so erzeugte Aufzeichnung kann anschließend weiterverarbeitet werden. Dem Anwender bieten sich hierbei drei Möglichkeiten, die gewonnenen Informationen zu nutzen:

- Speichern einer Batch-Input-Mappe
- Erzeugung eines Batch-Input-Programmes
- Generieren eines Funktionsbausteins

Speichern einer Batch-Input-Mappe

Eine einfache Möglichkeit, die über die Aufzeichnung gewonnenen Informationen abzulegen ist die, den Ablauf in Form einer Batch-Input-Mappe zu speichern. Im Batch-Input-Recorder ist dies über die Schaltfläche »Mappe« in der Schaltflächenleiste möglich.

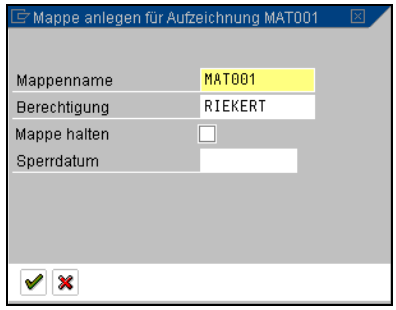

Abbildung 6.7
Attribute beim Generieren einer BTC-Mappe (© SAP AG)

Im daraufhin erscheinenden Fenster (siehe Abbildung 6.7) können der Name der Mappe, sowie die dargestellten übrigen Attribute der BTC-Mappe (siehe Abschnitt 6.6) festgelegt werden. Beim Bestätigen des Fensters mit dem grünen Häckchen wird im Beispiel die BTC-Mappe `MAT001` im System generiert.

Generieren eines Batch-Input-Programms

Speziell für einmalige Datenübernahmen aus Fremdsystemen ist es sinnvoll, aus der Aufzeichnung ein ablauffähiges Programm zu generieren. SAP bietet hierbei, neben der Möglichkeit, das Batch-Input-Pogramm für die während der Aufzeichnung erfassten Daten zu generieren, die Option an, das Programm derart zu erzeugen, dass die Transaktionsdaten der Aufzeichnung aus einer Datei eingelesen werden.

Der Weg der Weiterverarbeitung der Batch-Input-Aufzeichnung ist hierbei in beiden Fällen identisch. Nachdem im Batch-Input-Recorder die Schaltfläche »Programm« betätigt wurde, erscheint das in Abbildung 6.8 dargestellte Fenster. Neben dem Namen des zu erzeugenden Programms wird festgelegt, ob die Batch-Input-Daten aus einer Importdatei gelesen oder aus der Aufzeichnung übernommen werden sollen.

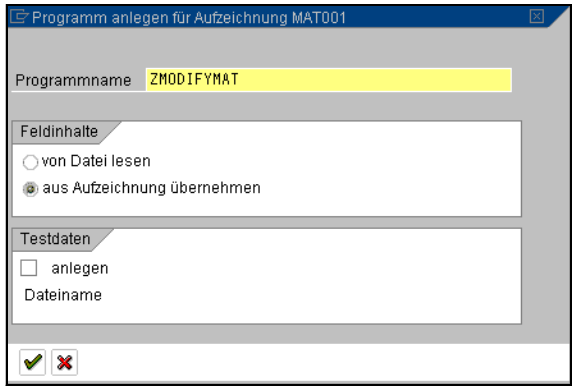

Abbildung 6.8
Generieren eines Batch-Input-Programms (© SAP AG)

Ersterer Fall ist – wie erwähnt – bei der Datenübernahme aus einem Fremdsystem sinnvoll. Soll diese Methode verwendet werden, kann in diesem Fenster auch die Importdatei definiert werden.

Die Daten des Programms aus der Aufzeichnung zu übernehmen ist insbesondere dann sinnvoll, wenn das Programm als solches nicht direkt weiterverwendet werden soll, sondern lediglich als Skelett dient, auf dessen Basis ein eigenes Programm entwickelt werden soll.

Nach Betätigen des grünen Häckchens erscheinen die üblichen Dynpros, die zur Anlage eines ABAP-Reports notwenig sind. Im Anschluss daran, steht das generierte Programm unmittelbar zur Verwendung im System zur Verfügung.

Generieren eines Funktionsbausteins

Soll die aufgezeichnete Sequenz häufig und in verschiedenen Kontexten ausgeführt werden, bietet sich hingegen an, einen Funktionsbaustein zu generieren, der die Transaktionsdaten über seine Aufrufschnittstelle bekommt und den Batch-Input wie aufgezeichnet durchführt.

Im dargestellten Fenster (Abbildung 6.9) muss der Name des Funktionsbausteins sowie die Funktionsgruppe, in dem der Funktionsbaustein abgelegt werden soll, definiert werden. Zusätzlich muss der Kurztext zum Funktionsbaustein erfasst werden. Aus diesen Infomationen wird ein Funktionsbaustein generiert, der den aufgezeichneten Batch-Input-Ablauf kapselt und die benötigten Informationen über die Schnittstelle des Funktionsbausteins übergeben bekommt.

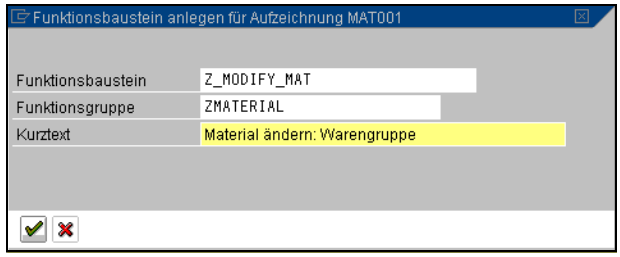

Abbildung 6.9
BTC als Funktionsbaustein ablegen (© SAP AG)

In das dargestellte Fenster gelangt der Benutzer, indem im Batch-Input-Recorder die Schaltfläche »Funktionsbaustein« in der Schaltflächenleiste angewählt wird.

6.9 Tipps, Tricks & Fallen

6.9.1 Fehlersuche in Batch-Input-Abläufen

An verschiedenen Stellen dieses Abschnitts wurden Situationen aufgezeigt, an denen ein Batch-Input scheitern kann. Oftmals ließe sich der aufgetretene Fehler sehr einfach lokalisieren, wenn man sehen könnte, was im Batch-Input-Ablauf genau passiert, wie das System im konkreten Fall reagiert und was schließlich zum Abbruch führt. Im Batch-Input ist dies kein Problem, da beim Start einer Mappe der Aufruf-Modus jeweils übergeben werden kann. Auch die `CALL TRANSACTION`-Anweisung bietet diese Möglichkeit über den Parameter `MODE`, über den die Sichtbarkeit der Transaktionsverarbeitung gesteuert werden kann.

Wird in diesem Parameter der Wert »`N`« übergeben, erfolgt der Transaktionsablauf auf jeden Fall für den Benutzer unsichtbar. Tritt eine der beschriebenen Fehlersituationen auf, bricht die Verarbeitung jedoch ab. Der Benutzer hat keine Möglichkeit der Einflussnahme. Dieser Wert ist der im produktiven Einsatz eines Programms gebräuchlichste. Der Benutzer wird in keinem Fall mit Programmsituationen konfrontiert, die er nicht kennt oder nicht lösen kann. Zur Fehlersuche bieten sich jedoch für den Parameter `MODE` die Werte »`E`« bzw. »`A`« an. Im ersten Fall gelangt der Benutzer in die gerufene Transaktion, falls ein Fehler während der Verarbeitung auftritt. Im zweiten Fall erfolgt die Verarbeitung in der gerufenen Transaktion in jedem Fall sichtbar. Der Benutzer sieht jeden ausgeführten Dialogschritt und alle gesetzten Feldinhalte. Speziell der zuletzt genannte Mode kommt daher für den produktiven Einsatz der Endanwender nicht in Frage. Aber auch der Mode »`E`« hat seine Tücken. Gelangt der Benutzer sichtbar in die Transaktion wird der Batch-Input– egal, wie der Benutzer die Transaktion verlässt – als erfolgreich beendet an das Batch-Input-Programm zu-

rückgemeldet. Eine Fehlerbehebung auf dieser Ebene ist daher bei Verwendung dieses Modus ausgeschlossen. Für den produktiven Betrieb bietet sich daher nur der Mode »N« an.

Falls jedoch Probleme auftreten, wäre es jedoch sehr sinnvoll, die Fehlersituation in der produktiven Umgebung durch sichtbares Abspielen des Batch-Inputs transparent machen zu können. Da jedoch in Programmen im Produktivsystemen keine Programmänderungen durchgeführt werden sollen, müsste stets eine Programmänderung im Entwicklungssystem mit anschließendem Transport ins Produktivsystem vorgenommen werden. Hierdurch wäre jedoch der Modus für alle Benutzer des Batch-Input-Programms geändert.

Sinnvoller erscheint hier, den Aufruf-Mode einer Transaktion in Form einer Variable an `CALL TRANSACTION` zu übergeben. Falls der Programmierer – was ebenfalls die Regel sein dürfte – keine Berechtigung hat, im Produktivsystem Variablen im Debugger zu ändern, bieten sich die in Kapitel 2 genannten Möglichkeiten, das Programm von außen beinflussen zu können, an.

Um den Aufruf-Modus eines Batch-Input dynamisch zu gestalten, bieten sich die folgenden Methoden an, die hier – ergänzend zu den Ausführungen in Kapitel 2 - in der konkreten Anwendung jeweils kurz skizziert werden.

- Funktionscode
- Steuertabelle
- Benutzerparameter
- Detaillierte Informationen zu den jeweiligen Verfahren, können in Kapitel 2 dieses Buches nachgeschlagen werden. An dieser Stelle soll jedoch nochmals exemplarisch gezeigt werden, wie die dort beschriebenen Techniken angewendet werden.
- Zugrunde gelegt werden soll ein Programmfragment, welches den Aufrufmode für den `CALL TRANSACTION`-Aufruf als Variable übergibt. Welcher Wert übergeben werden soll, wird erst unmittelbar vor Aufruf des Batch-Inputs in der `FORM`-Routine `GET_CALL_MODE` ermittelt.

```
DATA: L_CALLMODE TYPE C.
...
PERFORM GET_CALLMODE CHANGING L_CALLMODE.

CALL TRANSACTION <Tcode>
           USING BDC_TABLE
            MODE L_CALLMODE.
...
```

In den folgenden Abschnitten wird jeweils eine Implementierung für die verschiedenen Möglichkeiten der Ableitung des Aufruf-Modus vorgestellt.

Setzen des Aufruf-Modus über Funktionscodes

Bei der Methode, Programme über Funktionscodes zu beeinflussen, werden Funktionscodes genutzt, um den Wert einer oder mehrerer globaler Variablen zu setzen. Da für den Aufrufmode der CALL TRANSACTION-Anweisung drei Werte möglich sind, bietet es sich an, diese jeweils durch eigene Funktionscodes abzubilden.

Abgelegt werden soll der aktuelle Modus in der globalen Variable G_CALLMODE, welche wie dargestellt deklariert ist.

```
DATA: G_CALLMODE TYPE C VALUE 'N'.
```

Im PAI bzw. im Ereignisblock AT USER-COMMAND werden die drei Funktionscodes CM_A, CM_N sowie CM_E behandelt und der entsprechende Wert in die globale Variable übertragen.

```
...
CASE OK_CODE.
  ...
  WHEN 'CM_A'.
    G_CALLMODE = 'A'.

  WHEN 'CM_N'.
    G_CALLMODE = 'N'.

  WHEN 'CM_E'.
    G_CALLMODE = 'E'.
  ...
ENDCASE.
...
```

Die Implementierung der oben definierten Routine GET_CALLMODE müsste lediglich den Wert der globalen Variablen in den CHANGING-Parameter der Routine übertragen. Die abgebildete Implementierung verlässt sich jedoch nicht darauf, dass die Variable korrekt initialisiert wurde. Um Kurzdumps oder Programmabbrüche zu verhindern, wird der Wert für verdecktes Abspielen zurückgeliefert, wenn nicht explizit ein anderer Wert gesetzt wurde.

```
FORM GET_CALLMODE CHANGING P_CALLMODE TYPE C.
  IF NOT G_CALLMODE IS INITIAL.
    P_CALLMODE = G_CALLMODE.
  ELSE.
    P_CALLMOE = 'N'.
  ENDIF.
ENDFORM.
```

Möchte der Benutzer den Aufrufmodus des Batch-Inputs verändern, genügt es, im Fcode-Fenster im oberen Bereich des SAPGUI den entsprechenden Funktionscode anzugeben und mit ↵ zu bestätigen. Da diese Funktionalität im Normalfall nicht von jedem Benutzer gesehen werden soll, werden meist keine Menüeinträge hierfür erzeugt.

Der so eingestellte Wert für den Aufrufmodus bleibt solange erhalten, bis die Transaktion beendet wird oder explizit ein anderer Modus gesetzt wird.

Verwendung von Benutzerparametern

Noch einfacher als die Verwendung von Funktionscodes gestaltet sich in der Programmierung die Methode, den Aufrufmode für den Batch-Input aus einem Benutzerparameter zu ermitteln.

Hierzu muss zunächst ein Benutzerparameter erzeugt werden, aus dem der Wert ausgelesen werden kann. Dies geschieht im Object Navigator (Transaktion `SE80`). Über die Schaltfläche »Objekt bearbeiten« in der Schaltflächenleiste des Object Navigators gelangt man in das in Abbildung 6.10 dargestellte Fenster. Das Fenster enthält eine Reihe von Tab-Seiten, über die die verschiedensten Entwicklungsobjekte angezeigt, geändert bzw. erstellt werden.

Auf der Tab-Seite »Weitere« findet sich unter anderem die Möglichkeit, Parameter-Ids (hier SET/GET-Parameter-ID genannt) anzulegen. Hierzu muss das Auswahlfeld auf die Zeile der Parameter-Ids gesetzt werden und im Feld rechts daneben der Name der neuen ID, im Beispiel `ZCALLMODE`, eingetragen werden. Über die am unteren Fensterrand sichtbare Anlegen-Schaltfläche (4. von links), kann in das in Abbildung 6.12 gezeigte Fenster verzweigt werden.

Hier muss zu der neuen Parameter-ID ein Kurztext erfasst werden, der zusammen mit dem Wert der Parameter-ID in den Benutzerparametern angezeigt wird. Nach Betätigen von »Sichern« steht die Parameter-ID sofort zur Verfügung und kann in den Benutzerparametern mit Werten belegt werden.

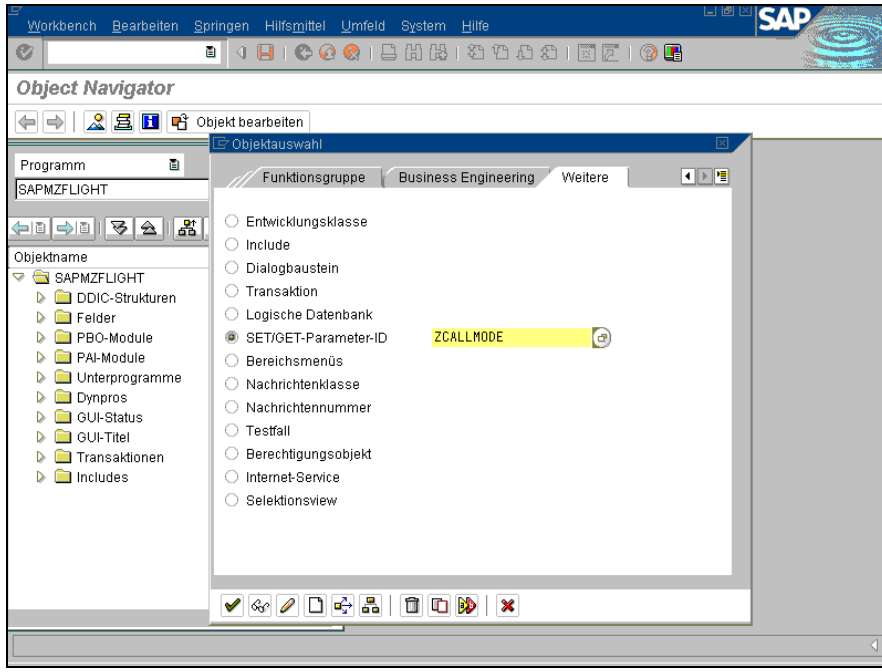

Abbildung 6.10
Anlegen von Parameter-Ids im Object Navigator (© SAP AG)

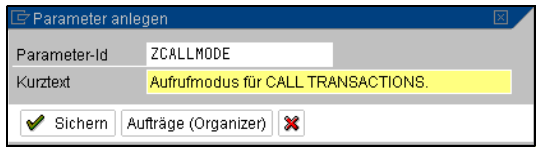

Abbildung 6.11
Parameter-ID anlegen (© SAP AG)

Während für die ABAP-Anweisungen `GET PARAMETER ID` und `SET PARAMETER ID` nicht zwingend notwendig ist, dass die ID auch tatsächlich existiert, ist dies für die Zuweisung von Werten in den Benutzerdaten des Benutzerstamms unbedingt erforderlich.

Möchte ein Benutzer den Aufrufmode für den Batch-Input setzten, kann er dies in seinen Benutzerdaten in der Transaktion `SU52` erledigen. Abbildung 6.14 zeigt einen Screenshot, in dem der Benutzerparameter `ZCALLMODE` auf den Wert »A« gesetzt wurde.

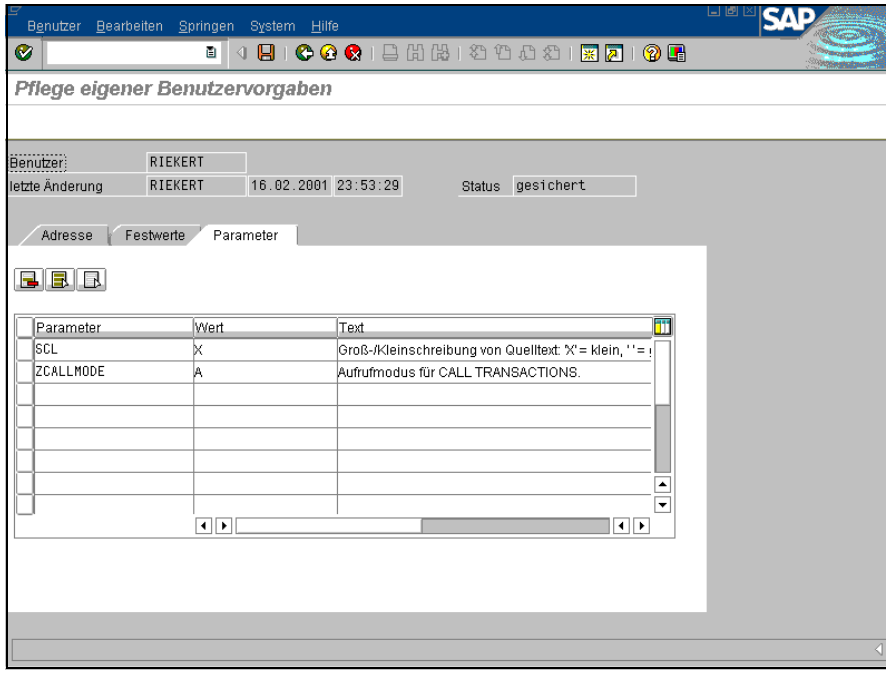

Abbildung 6.12
Benutzerparameter festlegen (© SAP AG)

Die Implementierung der Funktionalität schließlich beschränkt sich auf die Routine `GET_CALLMODE`, in welcher der Benutzerparameter ausgelesen und dem `CHANGING`-Parameter zugewiesen wird.

```
FORM GET_CALLMODE CHANGING P_CALLMODE TYPE C.
  GET PARAMETER-ID 'ZCALLMODE' FIELD P_CALLMODE.
  IF NOT P_CALLMODE EQ 'N' AND
     NOT P_CALLMODE EQ 'A' AND
     NOT P_CALLMODE EQ 'E'.
    P_CALLMODE = 'N'.
  ENDIF.
ENDFORM.
```

Auch die Implementierung dieser Routine geht nicht davon aus, dass der Benutzerparameter korrekt initialisiert ist. Im Fall eines ungültigen Wertes, wird auch hier der Wert `,N'` zurückgegeben.

Verwendung von Steuertabellen

Die wohl aufwendigste Methode, um den Aufrufmodus für die CALL TRANSACTION-Anweisung abzulegen, besteht in der Verwendung von Steuertabellen. Aufwendig nicht wegen der Implementierung, sondern im Wesentlichen wegen des Aufwands, eine transparente Tabelle anzulegen und diese mit Werten zu füllen. Für den Transport zu sorgen usw.. Sie kommt meist dann zum Zuge, wenn im Rahmen einer umfangreichen Programmierung bereits eine oder mehrere Steuertabellen angelegt wurden und somit der beschriebene Aufwand ohnehin anfällt.

Im diesem Fall soll hierfür eine generische Steuertabelle ZFLIGHT000 verwendet werden. Der Aufbau der Tabelle wird mit der in Tabelle 6.3 gezeigten Struktur festgelegt. Die Spalte KEY1 enthält hierbei den Transaktionscode, das Feld KEY2 den Benutzernamen, für den die Einstellung gelten soll. Wird als Benutzername der Wert »*« gespeichert, soll die Einstellung für alle Benutzer gelten, für die nicht explizit ein anderer Wert definiert wurde.

Feldname	Datentyp	Bedeutung
MANDT	MANDANT	Mandant
KEY1	CHAR 20	Keyfeld 1
KEY2	CHAR 20	Keyfeld 2
VALUE	CHAR 40	Wertfeld

Tabelle 6.3
Struktur der generischen Steuertabelle ZFLIGHT000

Neben der TABLES-Anweisung im Deklarationsteil des Programms muss bei Verwendung dieser Methode nur die Implementierung der Routine GET_CALLMODE angepasst werden.

```
FORM GET_CALLMODE CHANGING P_CALLMODE TYPE C.
* Erster Leseversuch mit konkretem Benutzernamen
  SELECT SINGLE VALUE
         INTO   P_CALLMODE
         FROM   ZFLIGHT000
         WHERE  KEY1 = SY-TCODE AND
                KEY2 = SY-UNAME.

  CHECK SY-SUBRC NE 0.

* Zweiter Versuch für alle Benutzer
  SELECT SINGLE VALUE
         INTO   P_CALLMODE
```

```
        FROM   ZFLIGHT000
        WHERE  KEY1 = SY-TCODE AND
               KEY2 = ,*'.

  CHECK SY-SUBRC NE 0.

* Sonst setzen des Default-Wertes
  P_CALLMODE = ,N'.
ENDFORM.
```

Das Programm versucht durch zweistufiges Lesen, den Aufrufmodus aus der Tabelle zu ermitteln. Höchste Priorität hat hierbei ein expliziter Tabelleneintrag für den angemeldeten Benutzer. Schlägt dieser Leseversuch fehl, wird mit dem generischen Benutzernamen »*« auf die Tabelle zugegriffen. Erst wenn auch diese SELECT-Anweisung keinen Datensatz zurückliefert, wird der Defaultwert »N« gesetzt.

6.9.2 Nicht Batch-Input-fähige Transaktionen

Mehrere LUW in einer Transaktion

Ein wesentlicher Grund für unerklärliche Abbrüche aus Batch-Input-Aufrufen ist der Umstand, dass manche Transaktionen aus mehreren SAP-LUW[2] bestehen. Eine SAP-LUW wird explizit durch den Aufruf der Anweisung COMMIT WORK im Erfolgsfall und ROLLBACK WORK im Fehlerfall abgeschlossen.

Ein Batch-Input-Ablauf kann sich jedoch nur über eine SAP-LUW erstrecken. Ein explizites Beenden einer LUW im Programmcode beendet demnach auch den Batch-Input. Dieses Verhalten lässt sich weder programmtechnisch, noch organisatorisch lösen. Transaktionen, in denen explizite Aufrufe von COMMIT WORK sowie ROLLBACK WORK enthalten sind, sind daher nicht für Batch-Input-Abläufe geeignet.

Variable Dynpro-Steuerung im Standard

In vielen Fällen sind die Abläufe im R/3 sehr komplex. An verschiedenen Stellen im R/3 finden sich daher Ansätze von SAP wieder, komplexe Abläufe flexibel zu gestalten. Hierzu werden umfangreiche Systeme von Steuertabellen verwendet, in denen sowohl die Dynpro-Reihenfolge als auch feldsteuernde Parameter definiert werden. Leider sind auch diese Steuertabellen in der Regel sehr umfangreich und komplex. In vielen Fällen werden ganze Dynpros aufgrund schwer nachvollziehbarer Gründe ausgetauscht, Felder werden ausge-

2. Nähere Informationen zu LUW und dem Verbucherkonzept in R/3 ist in Kapitel 8 dargestellt.

blendet oder sind in manchen Fällen Pflichtfelder, in anderen Fällen reine Ausgabefelder.

Um einen sicheren Batch-Input-Ablauf zu erstellen, müssten alle beeinflussenden Nebenbedingungen bei der Erstellung des Batch-Input-Ablaufs berücksichtigt werden. Aufgrund der Komplexität des Systems R/3 ist dies in vielen Fällen nicht möglich.

Bei der Analyse einer Transaktion, für die ein Batch-Input erstellt werden soll, müssen daher alle erkennbaren relevanten Randbedingungen ausgewertet und darauf reagiert werden. In der Regel genügt dieses Vorgehen, um einen ausreichend stabilen Batch-Input zu erstellen. Selbst bei sorgfältigster Arbeit ist man dabei jedoch nicht gegen Änderungen im Customizing gefeit, die den mühsam erstellten Batch-Input-Ablauf erneut zum scheitern bringt.

In manchen Fällen muss man jedoch auch einsehen, dass dies mit vertretbarem Aufwand nicht möglich ist und die Transaktion als nicht Batch-Input-geeignet bezeichnet werden muss. Anders als der im vorigen Abschnitt beschriebene Umstand ist der hier beschriebene Sachverhalt daher keine technische Einschränkung im System R/3, sondern liegt vielmehr auf der Anwendungsebene begründet.

Step-Loops und Table-Views

In Abschnitt 6.4.4 wurde bereits auf die Problematik von Step-Loops und Table-Views in Batch-Input-Abläufen hingewiesen. Wenn innerhalb eines Batch-Input auf Zeilen einer Step-Loops zugegriffen werden soll, muss vor dem Aufruf feststehen, in welcher Zeile einer Liste der gewünsche Satz dargestellt wird bzw. welche Zeile der Liste noch leer ist, um neue Sätze zu erfassen. Hinzu kommt das Problem, dass oftmals nicht sicher ist, wieviele Zeilen überhaupt auf dem Dynrpo vorhanden sind, wodurch ein Blättern in der Liste in so gut wie keinem Fall zu einem gesicherten Zustand führt.

Wirklich sichere Abhilfe kann hier nur in Transaktionen gesehen werden, in denen über eine Funktionalität der gewünschte Satz in die erste Zeile der dargestellten Liste gebracht werden kann. Bei SM-Aufträgen kann beispielsweise in der Vorgangsübersicht über den Funktionscode `OBSE` ein Fenster zur Anzeige gebracht werden, in dem über verschiedene Kriterien auf einen bestimmten Satz positioniert werden kann. Gelingt dies, kann der Batch-Input-Ablauf immer davon ausgehen, dass der zu bearbeitende Satz in der ersten Zeile der Liste dargestellt wird.

Bei der Programmierung von Batch-Input-Abläufen in Step-Loops oder Table-Views sollte daher immer nach solchen Möglichkeiten gesucht werden. Besteht eine solche Möglichkeit jedoch nicht und die genaue Position des zu bearbeitenden Satzes in der Liste kann nicht zusammen mit der Anzahl der dargestellten Zeilen vorausgesagt werden, besteht keine Chance, einen gesicherten Batch-Input-Ablauf zu erstellen.

An dieser Stelle nochmals ganz explizit der Hinweis: Achten Sie auf unterschiedliche Fenstergrößen. Sobald die Anzahl der sichtbaren Listenzeilen variieren kann, dürfen die Blätterfunktionen nicht genutzt werden, um auf einen bestimmten Satz zu positionieren.

Unterschiedliches Verhalten von Transaktionen im Dialog und Batch-Input

Zum großen Leidwesen von Programmierern von Batch-Input-Abläufen gibt es zusätzlich noch das Problem, dass sich das Verhalten von Transaktionen im echten Dialog, d.h. wenn ein Benutzer die Transaktion direkt bedient, von dem im Batch-Input unterscheiden kann. Für Programmierer von Dialogtransaktionen stehen hierbei die Systemfelder `SY-CALLD` sowie `SY-BATCH` zur Verfügung. Über sie kann ermittelt werden, ob ein Programm im Dialog, im so genannten Call-Modus (`CALL TRANSACTION`) oder im Batch-Input-Mode (als Mappe) abläuft. Häufig werden hierüber Dialogabläufe verändert oder Felder ein- bzw. ausgeblendet. Fehler dieser Art können am einfachsten durch Verwendung der in Abschnitt 6.9.1 dargestellten Methode lokalisiert und behoben werden.

6.10 Verwendete und wichtige Funktionsbausteine

6.10.1 BDC_CLOSE_GROUP

Schließt die zuvor mit `BDC_OPEN_GROUP` geöffnete Batch-Input-Mappe.

Import-Parameter	
Keine	
Export-Paramter	
Keine	
Exceptions	
NOT_OPEN	Keine BTC-Mappe geöffnet
QUEUE_ERROR	Fehler beim Lesen/Schreiben der Batch-Input-Queue (Details zum Fehler sind im SYSLOG enthalten)

6.10.2 BDC_INSERT

Fügt Ablauf-Daten für eine Dialogtransaktion in die geöffnete Batch-Input-Mappe ein.

Import-Parameter	
TCODE	Transaktionscode der aufzurufenden Dialogtransaktion
POST_LOCAL	Schaltet die lokale Verbuchung ein, d.h. die Verbuchung erfolgt nicht in einem Verbuchertask sondern im Dialogprozess, in dem auch die Transaktion selbst abläuft
PRINTING	Legt fest, ob Druckausgabe erzeugt werden soll oder nicht
Tabellen-Parameter	
DYNPROTAB	Batch-Input-Tabelle
Exceptions	
INTERNAL_ERROR	Interner Fehler
NOT_OPEN	Keine Batch-Input-Mappe geöffnet
QUEUE_ERROR	Fehler bei der Verarbeitung der Mappe
TCODE_INVALID	Ungültiger Transaktionscode
PRINTING_INVALID	Ungültiger Wert im Parameter PRINTING
POSTING_INVALID	Ungültiger Wert im Parameter POSTING

6.10.3 BDC_OPEN_GROUP

Öffnet eine neue Batch-Inputmappe und legt deren Eigenschaften fest.

Import-Parameter	
CLIENT	Mandant
DEST	Zielsystem (nicht mehr relevant)
GROUP	Name der Batch-Input-Mappe
HOLDDATE	Datum, bis zu dem die Mappe gesperrt werden soll
KEEP	Kennzeichen, um abgespielte Mappen zu erhalten

USER	Benutzerkennung
RECORD	Kennzeichen, Batch-Input-Aufzeichnung
Export-Parameter	
QID	ID der Batch-Input-Queue
Exceptions	
CLIENT_INVALID	Ungültiger Mandant
DESTINATION_INVALID	Zielsystem ist ungültig
GROUP_INVALID	Ungültiger Mappenname
GROUP_LOCKED	Mappe ist von anderem Programm gesperrt
HOLDDATE_INVALID	Ungültiges Sperrdatum
INTERNAL_ERROR	Interner Fehler
QUEUE_ERROR	Fehler beim lesen/schreiben der BTC-Queue
RUNNING	Mappe wird gerade abgespielt
SYSTEM_LOCK_ERROR	Systemfehler beim sperren der Mappe
USER_INVALID	Ungültiger Benutzer

7 Einplanen von Hintergrundjobs

7.1 Jobs in SAP R/3

In heutigen Client/Server-Systemen existiert neben dem, auf kurze Antwortzeiten optimierten Dialogbetrieb ein, im Hinblick auf Datendurchsatz optimierter, Batchbetrieb.

Die auszuführenden Aufgaben für den Batch-Mode werden in Form so genannter Jobs definiert und bis zur tatsächlichen Ausführung zwischengespeichert. Da ein Job immer im Hintergrund, also ohne Bezug zu einem GUI abläuft, werden die Ausgaben eines Jobs in Form einer Spoolliste im System abgelegt.

Ein Job in R/3 besteht neben den Verwaltungsinformationen, die unter anderem auch die Startoptionen des Jobs beinhalten (siehe unten), aus einem oder mehreren Verarbeitungsschritten, den so genannten Jobsteps. Bei diesen handelt es sich entweder um einen ABAP-Report, ein – vom Systemadminstrator definiertes externes Programm oder ein beliebiges Kommando auf der Ebene des Betriebssystems. Für jeden Jobstep einzeln können die Ausgabeparameter festgelegt werden, die zur Ausgabe der Spoolliste verwendet werden.

Wenn alle Schritte, die zu einem Job gehören, erfasst wurden, wird der Job abgeschlossen und mit Startoptionen versehen. Diesen Vorgang nennt man auch freigeben oder einplanen. Ein Job kann entweder bei der Freigabe sofort gestartet werden, zu einem späteren definierten Zeitpunkt oder bei Eintreten eines bestimmten Ereignisses. Näheres zu den hier verfügbaren Optionen wird an den entsprechenden Stellen in diesem Kapitel erläutert.

Die Jobs werden vom System abgelegt und zu gegebener Zeit gestartet. Hierbei werden alle zum Job gehörigen Steps nacheinander sequentiell abgearbeitet. Die Verarbeitung der Schritte erfolgt in speziellen Workprozessen den Batch-Prozessen. Den Bereich der Jobverwaltung und –ausführung in R/3 nennt man die Hintergrundverwaltung/Hintergrundverarbeitung.

In diesem Kapitel werden mehrere Arten gezeigt, wie Jobs und Jobschritte erzeugt werden können. Die Vielzahl von Optionen, speziell im Bereich der externen Kommandos, können hier jedoch nicht vollständig abgebildet werden. Die Ausführungen an dieser Stelle beschränken sich daher darauf, ABAP-Reports im Rahmen der Hintergrundverarbeitung zu verarbeiten.

7.2 Erstellen eines einfachen Jobs

Die Programmierung von Jobs in R/3 erfolgt grundsätzlich in drei Schritten. Am Anfang steht immer ein Aufruf des Funktionsbausteins JOB_OPEN, um einen neuen Job zu erstellen. Dieser Funktionsbaustein legt die Verwaltungsinformationen des neuen Jobs an und macht diesen über die eindeutige Jobnummer für die folgenden Funktionsaufrufe identifizierbar.

Im Anschluss daran werden die Verarbeitungsschritte des Jobs, die so genannten Jobsteps erzeugt. Dies erfolgt durch einen oder mehrere Aufrufe des Funktionsbausteins JOB_SUBMIT. Mit diesem Funktionsbaustein können Jobsteps aller Art angelegt werden. Die Aufrufschnittstelle enthält sowohl Parameter für Jobsteps, die einen ABAP-Report starten, als auch die entsprechenden Felder für Jobsteps mit einem externen Kommando- oder Programmaufruf.

Sobald alle Schritte an den Job angefügt wurden, folgt zuletzt das Schließen und Freigeben des Jobs. Damit verbunden in das Einplanen der Ausführungszeit bzw. der Umstände, zu denen der Job gestartet werden soll.

7.2.1 Erzeugen eines Jobs

Bevor Verarbeitungsschritte in einen Job eingefügt werden können, muss zunächst der Job als solcher erstellt werden. Dies erfolgt über einen Aufruf des Funktionsbausteins JOB_OPEN; der wie die übrigen hier erwähnten Funktionsbausteine in der Funktionsgruppe BTCH im R/3-Systems abgelegt sind.

Der Funktionsbaustein erhält im Aufrufparameter JOBNAME den bis zu 32-stelligen Namen des zu erstellenden Jobs. Der Parameter JOBGROUP hat im aktuellen Release keine Bedeutung mehr. Genauso verhält es sich mit den übrigen Parametern DELANFREP, SDLSTRTDT und SDLSTRTTM, die lediglich aus Kompatibilitätsgründen in der Schnittstelle des Funktionsbausteins enthalten sind. In künftigen Releaseständen von R/3 werden diese Parameter voraussichtlich entfallen.

```
FUNCTION JOB_OPEN
   IMPORT
      DELANFREP
      JOBGROUP
      JOBNAME
      SDLSTRTDT
      SDLSTRTTM
```

```
   EXPORT
      JOBCOUNT
   EXCEPTIONS
      CANT_CREATE_JOB
      INVALID_JOB_DATA
      JOBNAME_MISSING
```

Der Funktionsbaustein liefert im Erfolgsfall im Ergebnisparameter JOBCOUNT eine Nummer zurück, über die der Job eindeutig identifiziert werden kann. Ansonsten wird eine der definierten Ausnahmen ausgelöst. Über die beiden Felder JOBNAME und JOBCOUNT wird in den folgenden Funktionen auf den Job zugegriffen.

7.2.2 Erzeugen von Jobsteps

Nachdem der Job mit dem Funktionsbaustein JOB_OPEN erzeugt wurde, können Verarbeitungsschritte für den Job definiert werden. Jobsteps können an beliebige, noch nicht freigegebene Jobs angefügt werden. Es spielt also keine Rolle, ob der Job vom gleichen Programm angelegt wurde oder bereits früher definiert, jedoch nicht freigegeben wurde. Wichtig ist nur, dass der Zugriff auf einen bestimmten Job immer über die Felder JOBNAME und JOBCOUNT erfolgt. Ohne die interne Jobnummer kann niemals auf einen Job zugegriffen werden.

Das Anfügen von Steps an einen Job erfolgt durch einen Aufruf des Funktionsbausteins JOB_SUBMIT, welcher ebenfalls teil der Funktionsgruppe BTCH ist.

```
FUNCTION JOB_SUBMIT
   IMPORT
      ARCPARAMS
      AUTHCKNAM
      COMMANDNAME
      OPERATIONSYSTEM
      EXTPGM_NAME
      EXTPGM_PARAM
      EXTPGM_SET_TRACE_ON
      EXTPGM_STDERR_IN_JOBLOG
      EXTPGM_STDOUT_IN_JOBLOG
      EXTPGM_SYSTEM
      EXTPGM_RFCDEST
      EXTPGM_WAIT_FOR_TERMINATION
      JOBCOUNT
      JOBNAME
      LANGUAGE
      PRIPARAMS
      REPORT
      VARIANT
   EXPORT
      STEP_NUMBER
```

```
EXCEPTIONS
    BAD_PRIPARAMS
    BAD_XPGGLAGS
    INVALID_JOBDATA
    JOBNAME_MISSING
    JOB_NOTEX
    JOB_SUBMIT_FAILED
    LOCK_FAILED
    PROGRAM_MISSING
    PROG_ABAP_AND_EXTPG_SET
```

Die Parameter dieses Funktionsbausteins können in mehrere Gruppen aufgeteilt werden. Der Parameter AUTHCKNAM ist hierbei – neben dem Jobnamen (JOBNAME) und der Jobnummer (JOBCOUNT), über die der Job an sich identifiziert wird - als einziger Parameter in jedem Fall von Bedeutung. Er legt die R/3 Benutzerkennung fest, unter dessen Namen der Jobstep ausgeführt werden soll. Dies ist insbesondere für die Berechtigungsprüfung innerhalb des ABAP-Reports oder bei Aufruf des externen Kommandos notwendig Alle übrigen Parameter des Funktionsbausteins werden entweder benutzt, um die Ausführung eines ABAP-Reports und dessen Ausgabe zu steuern oder definieren ein – aus R/3-Sicht – externes Kommando oder Programm, welches ausgeführt werden soll.

Definition eines Jobsteps mit einem ABAP-Report

Die einfachere der beiden grundsätzlichen Möglichkeiten, einen Jobstep zu definieren besteht darin, einen ABAP-Report zur Ausführung zu bringen.

In der Aufrufschnittstelle des Funktionsbausteins JOB_SUBMIT ist der Parameter REPORT vorgesehen, um den Namen des Reports, der den Jobstep bilden soll, zu definieren. Da Reports in der Regel einen Selektionsbildschirm definieren, bei Jobs aber keine Möglichkeit besteht, diesen interaktiv zu versorgen, muss für den gewünschten Report eine Variante definiert werden, die die notwendigen Selektionsparameter enthält. Der Name dieser Variante wird dem Funktionsbaustein im Parameter VARIANT übergeben.

Zusätzlich dazu definiert der Parameter LANGUAGE das Sprachkennzeichen für die Sprache, die für den Report zum Ausführungszeitpunkt gesetzt werden soll.

Schließlich werden in den Parametern ARC_PARAMS sowie PRIPARAMS Strukturen für die Ausgabe der Report-Liste übergeben. Mit der Struktur ARC_PARAMS wird die Speicherung der Ausgabeliste in einem optischen Archiv definiert. Auf diese soll hier nicht näher eingegangen werden. Wichtiger ist die im Parameter PRIPARAMS übergebene Struktur, die die Druckaufbereitung und -ausgabe der Reportliste definiert.

Diese Struktur ist im Data Dictionary unter dem Namen PRI_PARAMS definiert. Der Aufbau dieser Struktur ist aus Tabelle 7.1zu ersehen.

Feldname	Datentyp	Bedeutung
PDEST	CHAR 4	Ausgabegerät
PRCOP	NUMC 3	Anzahl Ausdrucke
PLIST	CHAR 12	Name des Spoolauftrags
PRTXT	CHAR 68	Text für Beschriftung des Deckblatts
PRIMM	CHAR 1	Kennzeichen, ob Liste sofort ausgegeben werden soll
PRREL	CHAR 1	Kennzeichen, ob der Spoolauftrag nach der Ausgabe gelöscht werden soll
PRNEW	CHAR 1	Kennzeichen, ob ein neuer Spoolauftrag erzeugt werden soll
PEXPI	NUMC 1	Verweildauer des Auftrags im Spool
LINCT	INT	Seitenlänge der Liste in Zeilen
LINSZ	INT	Zeilenbreite der Liste in Zeichen
PAART	CHAR 16	Druckaufbereitung
PRBIG	CHAR 1	Selektions-Deckblatt
PRSAP	CHAR 1	Kennzeichen, ob Deckblatt gedruckt werden soll
PRREC	CHAR 12	Empfänger
PRABT	CHAR 12	Abteilung des Empfängers
PRBER	CHAR 12	Berechtigung
PRDSN	CHAR 6	Name des Spool-Datasets
PTYPE	CHAR 12	Typ des Spool-Auftrags
ARMOD	CHAR 1	Ablagemodus
FOOTL	CHAR 1	Kennzeichen, ob eine Fußzeile ausgegeben werden soll
PRIOT	CHAR 1	Priorität des Druckauftrags
PRUNX	CHAR 1	Hostspooler-Deckblatt
PRKEYEXT	CHAR 16	Schlüssel für Druckparameter
PRCHK	INT	Prüfsumme für Druck- und Archivierung

Tabelle 7.1
Aufbau der Dictionary-Struktur PRI_PARAMS

Von der Vielzahl der hier definierten Parameter ist lediglich die Angabe des Ausgabegerätes (PDEST) notwenig. Die übrigen Parameter werden bei Bedarf automatisch ermittelt.

Die übrigen Felder der Stuktur sind selbst sprechend und können bei Bedarf versorgt werden. Der Funktionsbaustein versucht intern durch einen Aufruf der Funktion GET_PRINT_PARAMETERS etwaige fehlende Angaben selbst zu ermitteln. Falls dies nicht gelingt, wird die Ausnahme BAD_PRIPARAMS ausgelöst.

Definition eines Jobsteps mit einem externen Kommando oder Programm

Anstatt eines ABAP-Reports kann einem Jobstep auch ein externes Kommando oder der Aufruf eines externen Programms zugeordnet werden. Der Unterschied zwischen einem Kommando und einem externen Programm liegt darin, dass ein Kommando ein vom Systemadmistrator vordefiniertes externes Programm oder Kommando beinhaltet. Der Erzeuger des Jobs hat hier nur die Möglichkeit, vordefinierte Programme aufzurufen. Dies ist – im Gegensatz zum Aufruf eines externen Programms – wesentlich sicherer, da die Aktivitäten des Jobs und damit des erstellenden Benutzers wesentlich besser gesteuert werden können. Nicht zuletzt ist es ein erhebliches Sicherheitsrisiko, wenn Benutzer beliebige Kommandos auf der Ebene des Betriebssystems aufrufen können.

Bei der Definition eines externen Kommandos oder externen Programmaufrufs im Dialog stehen mehrere Felder zur Definition der notwendigen Parameter zur Verfügung (siehe Abbildung 7.1). Bei den beiden Methoden besteht jedoch eine weitgehende Deckungsgleichheit der Parameter, so dass Parameter der Aufrufschnittstelle von JOB_SUBMIT teilweise für beide Varianten genutzt werden.

Die Unterscheidung, ob es sich um ein externes Kommando oder einen externen Programmaufruf handelt, wird über die Felder COMMANDNAME bzw. EXTPGM_NAME getroffen. In beiden Fällen werden in den Parametern EXTPGM_SYSTEM, EXTPGM_PARAM sowie EXTPGM_RFCDEST die Ausführungsparameter des Jobs übergeben. Weiterhin kann bei beiden Varianten über die Parameter EXTPGM_STDOUT_IN_JOBLOG, EXTPGM_STDERR_IN_JOBLOG gesteuert werden, ob die Ausgabe bzw. das Fehlerprotokoll des externen Kommandos oder Programms in das Protokoll des Jobs (Joblog) aufgenommen werden soll.

Über den Parameter EXTPGM_SET_TRACE_ON kann dabei festgelegt werden, ob der für den Aufruf eingeschaltet sein soll.

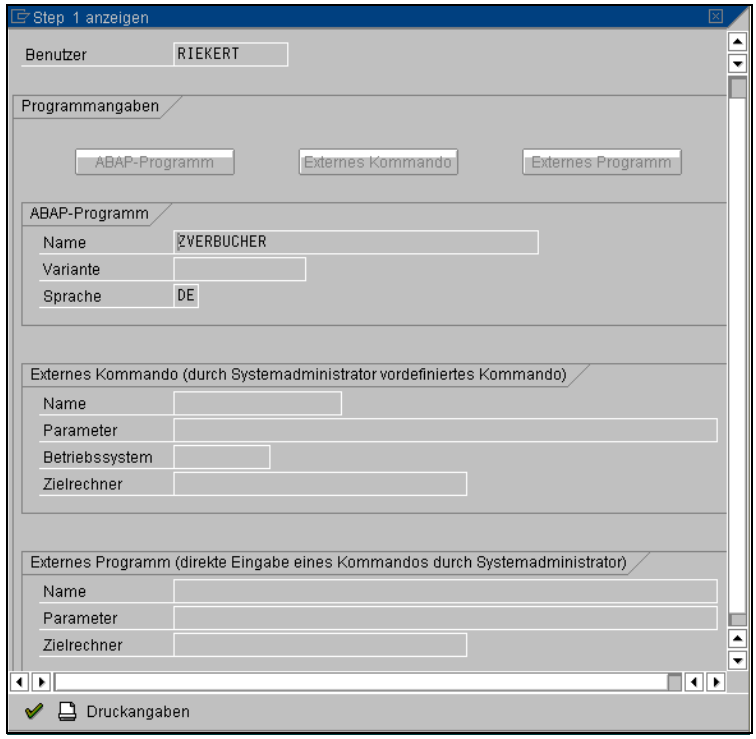

Abbildung 7.1
Definition eines Jobsteps im Dialog (Transaktion SM37) (© SAP AG)

Schließlich kann noch im Parameter EXTPGM_WAIT_FOR_TERMINATION ein Kennzeichen übergeben werden, ob das R/3-System auf die Beendigung des externen Programms warten soll, bevor mit dem nächsten Jobstep fortgefahren wird (»X«) oder nicht (Space).

Wenn der Verarbeitungsschritt erfolgreich in den Job eingefügt werden konnte, liefert der Funktionsbaustein im Rückgabeparameter STEP_NUMBER die laufende Nummer des erzeugten Steps innerhalb des Jobs zurück.

Auf diese Weise werden nacheinander alle Verarbeitungsschritte des Jobs angelegt. Wichtig hierbei ist, dass bei der Verarbeitung des Jobs alle Steps nacheinander in der angegebenen Reihenfolge ausgeführt werden. Bedingungen oder andere Verzweigungen sind innerhalb eines Jobs nicht vorgesehen und können allenfalls über mehrere Jobs realisiert werden.

7.3 Freigeben und Einplanen eines Jobs

Nachdem alle Verarbeitungsschritte des Jobs erzeugt wurden, kann der Job zur Ausführung freigegeben und eingeplant werden. Hierfür steht innerhalb der Funktionsgruppe BTCH der Funktionsbaustein JOB_CLOSE zur Verfügung.

Über die Parameter dieses Funktionsbausteins können alle Optionen, einen Job in der Hintergrundverarbeitung zu starten, definiert werden.

```
FUNCTION JOB_SUBMIT
   IMPORT
      AT_OPMODE
      AT_OPMODE_PERIODIC
      CALENDAR_ID
      EVENT_ID
      EVENT_PARAM
      EVENT_PERIODIC
      JOBCOUNT
      JOBNAME
      LASTSTRTDT
      LASTSTRTTM
      PRDDAYS
      PRDHOURS
      PRDMINS
      PRDMONTHS
      PRDWEEKS
      PREDJOB_CHECK
      PRED_JOBCOUNT
      PRED_JOBNAME
      SDLSTRTDT
      SDLSTRTTM
      STARTDATE_RESTRICTION
      STRTIMMED
      TARGETSYSTEM
      START_ON_WORKDAY_NOT_BEFORE
      START_ON_WORKDAY_NR
      WORKDAY_COUNT_DIRECTION
      RECIPIENT_OBJ
      TARGETSERVER
      DONT_RELEASE
   EXPORT
      JOB_WAS_RELEASED
   EXCEPTIONS
      CANT_START_IMMEDIATE
      INVALID_STARTDATE
      JOBNAME_MISSING
      JOB_CLOSE_FAILED
```

```
JOB_NOSTEPS
JOB_NOTEX
LOCK_FAILED
```

Es gibt eine ganze Reihe von Möglichkeiten, einen Job zu starten.

- Jobstart an einem bestimmten Wochentag
- Einplanen des Jobs an einem festen Termin
- Job sofort starten
- Starten eines Jobs über Ereignisse
- Job nach Verarbeitung eines anderen Jobs starten
- Starten eines Jobs beim Wechsel der Betriebsart

Sämtliche Varianten können durch den Aufruf des Funktionsbausteins JOB_CLOSE realisiert werden. Daher können die Parameter der Funktion ebenfalls in Gruppen eingeteilt werden. Die Parameter werden in oben genannter Reihenfolge geprüft, welche Varinante des Jobstarts gewünscht ist. Lediglich die erste erkannte Variante kommt hierbei zum Zuge.

Wie bei den übrigen Funktionsbausteinen der Funktionsgruppe BTCH erfolgt auch in diesem Fall die Identifikation des Jobs über den Namen (JOBNAME) sowie die Nummer des Jobs (JOBNUMBER).

Wird im Parameter DONT_RELEASE dem Funktionsbaustein der Wert »X« übergeben, wird der Job geschlossen und eingeplant, d.h. die Startbedingungen am Job hinterlegt, der Job jedoch nicht zur Verarbeitung freigegeben. Dies muss dann entweder über einen weiteren Aufruf des Funktionsbausteins JOB_CLOSE oder durch den Aufruf des Funktionsbausteins JOB_RELEASE erfolgen.

Die übrigen Parameter werden in den folgenden Abschnitten entsprechend ihrer Verwendung erläutert.

Jobstart an einem bestimmten Tag im Monat

Ein Job kann so eingeplant werden, dass er an einem bestimmten Tag der Woche gestartet wird. Die Nummer des Wochentages wird hierbei im Parameter START_ON_WORKDAY_NO an den Funktionsbaustein übergeben. Der Parameter WORKDAY_COUNT_DIRECTION definiert hierbei, ob die Tage zum Monatsende vorwärts oder rückwärts gezählt werden sollen.

Am definierten Tag wird der Job zu der im Parameter SDLSTRTTM definierten Uhrzeit gestartet. Über den Parameter START_ON_WORKDAY_NOT_BEFORT kann zusätzlich ein Datum definiert werden, vor dem der Job nicht gestartet werden soll. So kann gegebenenfalls eine Einplanung für einen späteren Monat erfolgen.

Wird im Parameter PRDMONTHS ein Wert ungleich null an den Funktionsbaustein übergeben, wird der Job periodisch immer am gleichen Tag des Monats ausgeführt.

Einplanen des Jobs an einem festen Termin

Eine sehr gängige Art der Jobeinplanung besteht darin, einen festen Termin zu definieren, zu dem der Job starten soll. Hierfür stellt die Schnittstelle des Funktionsbausteins die Parameter SDLSTRTDT für das Datum, sowie SDLSTRTTM für die Uhrzeit, zu der der Job gestartet werden soll, zur Verfügung.

Da die Hintergrundverarbeitung nicht die Ausführung zu einem exakten Termin garantiert, kann zusätzlich dazu in den Felden LASTSTRTDT und LASTSTRTTM ein Verfallsdatum definiert werden. Wurde der Job bis zu diesem Zeitpunkt noch nicht gestartet, wird er verworfen.

Job sofort starten

Die wohl mit Abstand einfachste Art der Einplanung eines Jobs ist die, den Job sofort zu starten. Hierfür genügt es, im Parameter STRTIMMED den Wert »X« zu übergeben. Der Job wird sofort eingeplant und so schnell wie möglich durch die Hintergrundverarbeitung gestartet.

Starten eines Jobs über Ereignisse

Jobs können auch über so genannte Events ausgelöst werden. Die hier gemeinten Ereignisse sind nicht mit den Programmereignissen in Reports zu verwechseln. Es handelt sich vielmehr um ein Konstrukt, einen bestimmten Zustand im System zu erkennen und darauf durch Starten von Jobs in der Hintergrundverarbeitung zu reagieren. Es muss sich hierbei um Hintergrundjobs handeln, nachdem die Ereignisse nicht primär durch Dialogprogramme, sondern durch Umstände wie z.B. das Starten (SAP_SYSTEM_START) oder Stoppen des R/3-Systems (SAP_SYSTEM_STOP) automatisch ausgelöst werden. Gleichwohl ist es möglich, Events gezielt durch einen Aufruf des Funktionsbausteins BP_EVENT_RAISE auszulösen.

Neben den vom System definierten Ereignissen, wie z.B. die beiden oben genannten, besteht die Möglichkeit, eigene Events im System zu registrieren. Dies erfolgt in der Transaktion SM62 (»Anzeige/Pflege von Eventbezeichnungen«). Abbildung 7.2 zeigt die Transaktion während eine Eventbezeichnung gepflegt wird.

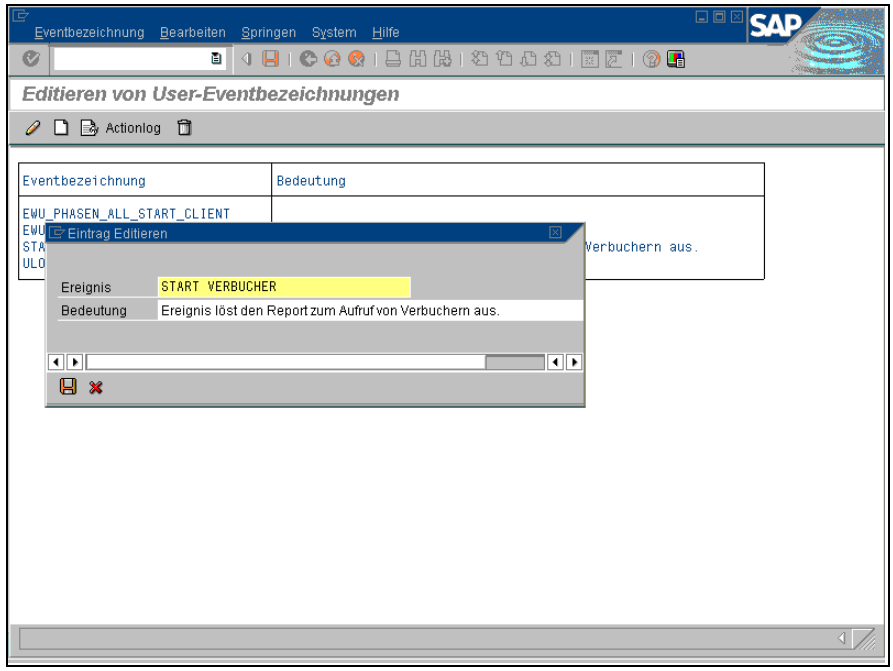

Abbildung 7.2
Pflegen von Eventbezeichnungen (SM62) (© SAP AG)

In der Transaktion lässt sich lediglich ein Eventname mit dazugehöriger Erläuterung definieren. Bei der Verwendung von Events besteht zusätzlich die Möglichkeit, den Event mit Parametern zu versorgen. Nur wenn der bei Aufruf von JOB_CLOSE definierten Eventparameter exakt mit den beim Auslösen des Events übergebenen Parametern übereinstimmt, wird der Job gestartet. Somit besteht eine feinere Möglichkeit, Jobs über Events gesteuert zu starten.

Beim Aufruf des Funktionsbausteins JOB_CLOSE werden der Eventname im Parameter EVENT_ID sowie die Ereignisparameter im Aufrufparameter EVENT_PARAM übergeben.

Zum Auslösen eines solchen Ereignisses wird der Funktionsbaustein BP_EVENT_RAISE verwendet.

```
FUNCTION BP_EVENT_RAISE
   IMPORT
      EVENTID
      EVENTPARAM
      TARGETINSTANCE
   EXCEPTIONS
      SUBMIT_FAILED
```

Der Funktionsbaustein erhält neben dem Eventnamen (EVENTID) ebenfalls die Parameter, anhand denen zusätzlich geprüft wird, ob der Job gestartet werden soll (EVENTPARAM), sowie die R/3-Instanz, in der das Ereignis ausgelöst werden soll (TARGETINSTANCE).

Gegebenenfalls können eine ganze Reihe von Jobs zu einem Ereignis definiert werden. Das Auslösen des Events startet dann eine ganze Jobkette. Die Jobs werden hierbei in der Reihenfolge, in der sie eingeplant wurden, ausgeführt.

Wird beim Freigeben des Jobs im Aufrufparameter des Funktionsbausteins JOB_CLOSE der Wert »X« übergeben, wird der Job periodisch eingeplant, d.h. bei jedem Auslösen des entsprechenden Events. Andernfalls wird der Job nur einmalig beim nächsten Auslösen des Ereignisses ausgeführt.

Job nach Verarbeitung eines anderen Jobs starten

Eine weitere Möglichkeit, einen Job starten zu lassen ist die, ihn an die erfolgreiche Ausführung eines anderen Jobs zu koppeln. Hierzu müssen in den Parametern PRED_JOBNAME sowie PRED_JOBCOUNT die Jobnummer und der Jobname des Vorgängerjobs übergeben werden. Über den Parameter PREDJOB_CHECKSTAT kann zusätzlich festgelegt werden, ob der Folgejob auf jeden Fall oder nur nach erfolgreicher Ausführung des Vorgängerjobs gestartet werden soll.

Starten eines Jobs beim Wechsel der Betriebsart

Mit der niedrigsten Priorität werden die Aufrufparameter des Funktionsbausteins JOB_CLOSE dahingehend geprüft, ob der Job beim Wechsel des Betriebsmodes des R/3-Systems gestartet werden soll.

Über Betriebsmodi kann im R/3-System die Anzahl und der Typ der Workprozesse geändert werden, um z.B. zwischen einem stark dialoglastigen Tagbetrieb in einen stark hintergrundverarbeitenden Nachtmode umzuschalten.

Jobs können so eingeplant werden, dass sie gestartet werden, wenn ein bestimmter Betriebsmode des R/3-Systems gesetzt wird. Der Name des Modes wird im Parameter AT_OPMODE an den Funktionsbaustein JOB_CLOSE übergeben. Intern wird der Wechsel des Betriebsmodes in Form eines Events gehandhabt. Zu jedem Betriebsmode wird automatisch ein gleichnamiges Ereignis definiert.

Jobs können einmalig beim nächsten Setzen eines bestimmten Betriebsmodes gestartet werden, können jedoch auch periodisch bei jedem Mal, wenn ein bestimmter Mode gesetzt wird, eingeplant werden. Hierfür muss im Parameter AT_OPMODE_PERIODIC der Wert »X« übergeben werden.

7.4 Eingeplante Jobs freigeben

Wurde ein Job durch einen Aufruf des Funktionsbausteins JOB_CLOSE zwar eingeplant, aber nicht freigegeben, kann dies durch einen späteren Aufruf des Funktionsbausteins JOB_RELEASE nachgeholt werden.

```
FUNCTION JOB_RELEASE
   IMPORT
      JOBNAME
      JOBCOUNT
   EXCEPTIONS
      MISSING_JOBNAME
      MISSING_JOBCOUNT
      MISSING_START_DATE
      STATUS_NOT_SCHEDULED
      CANT_ENQ_JOB
      CANT_START_JOB_IMMEDIATE
      NO_PRIVILEGE_TO_RELEASE_JOB
      CANT_RELEASE_JOB
      JOB_NOT_EXIST
      JOB_HAVE_NO_STEPS
      ERROR_JOB_MODIFY
```

Zur Identifikation des Jobs genügt auch hier der Jobname (JOBNAME) zusammen mit der Jobnummer (JOBCOUNT). Wurde der Job – vorrausgesetzt er existiert überhaupt – noch nicht eingeplant, d.h. mit Startoptionen versehen, oder besitzt der Benutzer, unter dessen Namen der Funktionsbaustein aufgerufen wird, nicht die Berechtigung, Jobs freizugeben, bricht der Funktionsbaustein mit einer Ausnahme ab.

7.5 Jobs periodisch wiederholen

Wie bereits im vorigen Abschnitt erwähnt, können Jobs zur einmaligen Ausführung oder als periodische Jobs eingeplant werden. Hierbei bedeutet periodisch nicht unbedingt, dass ein Job in regelmäßigen Abständen gestartet wird. Dies ist lediglich beim Einplanen des Jobs zu einem festen Termin, egal ob exakt als Datum gegeben oder als Tag im Monat, sowie bei Jobs, die sofort gestartet werden, der Fall.

Bei Jobs, die an ein Ereignis gekoppelt sind, bedeutet periodische Einplanung, dass der Job jedesmal, bei Eintreten des jeweiligen Events gestartet wird.

Bei der periodischen Einplanung von Jobs ist zu beachten, dass der Job bei der Einplanung lediglich für den nächsten Start eingeplant wird. Die jeweils folgende Freigabe erfolgt erst als letzter Schritt der Ausführung eines periodischen Jobs. Bricht ein periodisch eingeplanter Job ab, erfolgt keine folgende Einplanung und der Job wird im Folgenden nicht mehr gestartet.

7.6 Einplanen eines Reports als Job

Eine vereinfachte Form, einen Job zu erstellen und sofort zu starten, der lediglich einen ABAP-Report als Verarbeitungsschritt enthalten soll, bietet der Funktionsbaustein SIMPLE_BATCH_JOB_SUBMIT.

```
FUNCTION SIMPLE_BATCH_JOB_SUBMIT
   IMPORT
      PROGRAM
      VARIANT
   EXCEPTIONS
      SUBMIT_FAILED
```

Dem Funktionsbaustein kann lediglich der Name des Reports sowie der Name der Report-Variante, mit der der Report gestartet werden soll, übergeben werden. R/3 generiert daraus einen Job, der nur einen Step enthält und plant diesen mit sofortigem Start ein.

Auf diese Weise kann mit einem einzigen Funktionsaufruf eine Verarbeitung im Hintergrund gestartet werden.

7.7 Verwendete Funktionsbausteine

7.7.1 BP_EVENT_RAISE

Löst ein Ereignis aus und startet damit alle Jobs, die in Abhängigkeit zu diesem Ereignis definiert und freigegeben wurden.

Import-Parameter	
EVENTID	Names des Ereignisses
EVENTPARAM	Ereignisparameter, mit dem das Ereignis weiter eingeschränkt werden kann
TARGET_INSTANCE	R/3-Instanz, in der das Ereignis ausgelöst werden soll. In der Regel sollte dieser Parameter nicht verwendet werden
Exceptions	
SUBMIT_FAILED	Job konnte nicht erstellt werden

7.7.2 JOB_CLOSE

Mit dem Funktionsbaustein JOB_CLOSE wird ein zuvor mit JOB_OPEN erzeugter Job geschlossen und freigegeben. Der Funktionsbaustein bietet ein Reihe von Parametern, um sämtliche Startmöglichkeiten für Jobs zu realisieren.

Import-Parameter	
AT_OPMODE	Betriebsmode, bei dem der Job ausgelöst wird
AT_OPMODE_PERIODIC	Kennzeichen, ob der Job periodisch beim Betriebsartenwechsel gestartet werden soll
CALENDAR_ID	Kalender-ID
EVENT_ID	Ereignisname, bei dem der Job gestartet werden soll
EVENT_PARAM	Eventparameter
EVENT_PERIODIC	Kennzeichen, ob Job periodisch bei dem Event gestartet werden soll
JOB_COUNT	eindeutige Jobnummer
JOB_NAME	Name des Jobs
LASTSTRTDT	Verfallsdatum des Jobs
LASTSTRTTM	Uhrzeit am Verfallstag des Jobs
PRDDAYS	Startperiode des Jobs (Tage)
PRDHOURS	Startperiode des Jobs (Stunden)
PRDMINS	Startperiode des Jobs (Minuten)
PRDMONTHS	Startperiode des Jobs (Monate)
PRDWEEKS	Startperiode des Jobs (Wochen)
PREDJOB_CHECKSTAT	Kennzeichen, ob das Jobergebnis des Vorgängerjobs geprüft werden soll
PRED_JOBCOUNT	Jobnummer des Vorgängerjobs
PRED_JOBNAME	Name des Vorgängerjobs
SDLSTRTDT	Geplanter Startzeitpunkt des Jobs (Datum)
SDLSTRTTM	Geplanter Startzeitpunkt des Jobs (Uhrzeit)
STARTDATE_RESTRICTION	

Import-Parameter	
AT_OPMODE	Betriebsmode, bei dem der Job ausgelöst wird
STRTIMMED	Kennzeichen, ob der Job sofort gestartet werden soll
TARGETSYSTEM	Zielsystem, auf dem der Job laufen soll
START_ON_WORKDAY_NOT_BEFORE	
START_ON_WORKDAY_NUMBER	
WORKDAY_COUNT_DIRECTION	
RECIPIENT_OBJ	
TARGETSERVER	
DONT_RELEASE	Kennzeichen, ob der Job nicht freigegeben werden soll
Export-Paramter	
JOB_WAS_RELEASED	Kennzeichen, ob Job freigegeben wurde
Exceptions	
CANT_START_IMMEDIATE	Job kann nicht sofort gestartet werden
INVALID_STARTDATE	Ungültiger Startzeitpunkt
JOBNAME_MISSING	Jobname wurde nicht angegeben
JOB_CLOSE_FAILED	Job konnte nicht geschlossen werden
JOB_NOSTEPS	Der Job enthält keine Jobsteps
JOB_NOTEX	Der Job existiert nicht
LOCK_FAILED	Der Job konnte nicht gesperrt werden

7.7.3 JOB_OPEN

Mit dem Funktionsbaustein `JOB_OPEN` wird ein neuer Job erzeugt. Der Funktionsbaustein erhält mindestens den Namen des neuen Jobs, der nicht eindeutig sein muss und in der Regel auch nicht ist.

Die Felder `SDLSTRTDT` und `SDTSTRTTM`, mit denen in früheren Releaseständen das Startdatum des Jobs definiert werden konnte, sollen im aktuellen Release 4.6 nicht mehr verwendet werden. Das Startdatum des Jobs wird über die entsprechenden Parameter des Funktionsbausteins `JOB_CLOSE` definiert.

Über den Rückgabeparameter JOBCOUNT wird im Erfolgsfall die Jobnummer zurückgeliefert. Zusammen mit dem Jobnamen bildet die Jobnummer den eindeutigen Zugriffsschlüssel für die übrigen Funktionsbausteine.

Import-Parameter	
DELANFRREP	nicht mehr verwenden (!)
JOBGROUP	Jobgruppe, in der der neue Job angelegt werden soll
JOBNAME	Name des Jobs
SDLSTRTDT	nicht mehr verwenden (!)
SDLSTRTTM	nicht mehr verwenden (!)
Export-Paramter	
JOBCOUNT	Jobnummer des geöffneten Jobs
Exceptions	
CANT_CREATE_JOB	Job kann nicht erzeugt werden
INVALID_JOB_DATA	Ungültige Job Informationen
JOBNAME_MISSING	Jobname wurde nicht übergeben

7.7.4 JOB_RELEASE

Über den Funktionsbaustein JOB_RELEASE kann ein eingeplanter, aber noch nicht freigegebener Job zur Verarbeitung freigegeben werden. Voraussetzung hierfür ist, dass der Job existiert, mindestens einen Jobstep enthält und eingeplant ist. Weiterhin muss der Benutzer, unter dessen Namen der Funktionsbaustein aufgerufen wird, die Berechtigung zur Freigabe von Jobs besitzen.

Import-Parameter	
JOBNAME	Name des Jobs
JOBCOUNT	Jobnummer
Exceptions	
MISSING_JOBNAME	Jobname wurde nicht übergeben
MISSING_JOBCOUNT	Jobnummer fehlt
MISSING_START_DATE	Job enthält kein Startdatum
STATUS_NOT_SCHEDULED	Job ist nicht eingeplant

Import-Parameter	
JOBNAME	Name des Jobs
CANT_ENQ_JOB	Job kann nicht gesperrt werden
CANT_START_JOB_IMMEDIATELY	Job kann nicht sofort gestartet werden
NO_PRIVILEGE_TO_RELEASE_JOB	Keine Berechtigung zur Freigabe von Jobs
CANT_RELEASE_JOB	Job kann nicht freigegeben werden
JOB_NOT_EXIST	Job existiert nicht
JOB_HAVE_NO_STEPS	Job enthält keine Verarbeitungsschritte
ERROR_JOB_MODIFY	Fehler beim Ändern der Jobinformationen

7.7.5 JOB_SUBMIT

Mit dem Funktionsbaustein `JOB_SUBMIT` wird an einen, zuvor durch einen Aufruf des Funktionsbausteins `JOB_OPEN` erzeugten, Job ein Verarbeitungsschritt angefügt.

Über die entsprechenden Felder der Aufrufschnittstelle kann entweder ein externes Kommando, ein externes Programm oder ein ABAP-Report übergeben werden. Werden zuviele Felder gefüllt, bricht der Funktionsbaustein mit einer Exception ab.

Zurückgeliefert wird im Erfolgsfall die Nummer des neu erzeugten Jobsteps innerhalb des übergebenen Jobs.

Import-Parameter	
ARCPARAMS	Parameter für Archiv
AUTHCKNAM	Benutzer, unter dessen Namen der Job ausgeführt wird (für Berechtigungsprüfungen)
COMMANDNAME	Name des externen Kommandos
OPERATINGSYSTEM	Betriebssystem
EXTPGM_NAME	Name des externen Programms
EXTPGM_PARAM	Parameter für externes Programm
EXTPGM_SET_TRACE_ON	Kennzeichen, ob Trace-Informationen für den externen Programmaufruf gespeichert werden sollen

Import-Parameter	
ARCPARAMS	Parameter für Archiv
EXTPGM_STDERR_IN_JOBLOG	Kennzeichen, ob Fehlerliste des Programms ins Joblog übernommen werden soll
EXTPGM_STDOUT_IN_JOBLOG	Kennzeichen, ob Ausgabeliste des externen Programms im Joblog des Jobs abgelegt werden soll
EXTPGM_SYSTEM	Zielrechner für externes Programm
EXTPGM_RFCDEST	RFC-Destination, an der das externe Programm aufgerufen werden soll
EXTPGM_WAIT_FOR_TERMINATION	Kennzeichen, ob der Job warten soll, bis das externe Programm beendet ist
JOBCOUNT	Jobnummer
JOBNAME	Jobname
LANGUAGE	Sprache für Jobausführung
PRIPARAMS	Struktur mit Druckparametern für die Spoolliste des Jobsteps
REPORT	ABAP-Report für den Jobstep
VARIANT	Name der Report-Variante, die für den Aufruf des Reports verwendet werden soll
Export-Paramter	
STEP_NUMBER	Nummer des neu erzeugten Steps im Job
Exceptions	
BAD_PRIPARAMS	Ungültige Druckparameter
BAD_XPGFLAGS	Ungültige Flags für externes Programm
INVALID_JOBDATA	Ungültige Jobinformationen
JOBNAME_MISSING	Jobname wurde nicht übergeben
JOB_NOTEX	Job existiert nicht
JOB_SUBMIT_FAILED	Jobstep konnte nicht erzeugt werden
LOCK_FAILED	Job konnte nicht gesperrt werden

Exceptions	
ARCPARAMS	Parameter für Archiv
PROGRAM_MISSING	Es wurde kein Programm für den Jobstep übergeben
PROG_ABAP_AND_EXTPG_SET	Es wurde sowohl ein ABAP-Report als auch ein externes Programm für den Jobstep übergeben. Nur eine Angabe ist möglich

7.7.6 SIMPLE_BATCH_JOB

Der Funktionsbaustein `SIMPLE_BATCH_JOB` stellt eine vereinfachte Form dar, einen Job zu definieren, der aus nur einem Schritt besteht. Bei dem einen Schritt des Jobs muss es sich um einen ABAP-Report handeln .

Wenn der Job fehlerfrei erstellt und eingeplant werden konnte, liefert der Funktionsbaustein kein Ergebnis. Im Fehlerfall wird die Exception ausgelöst.

Import-Parameter	
PROGRAM	Name des ABAP-Reports
VARIANT	Programmvariante des ABAP-Reports
Exceptions	
SUBMIT_FAILED	Job konnte nicht erstellt werden

8 Programmierung mit Verbuchern

Wie bei allen Datenverarbeitungssystemen, die Daten speichern, bietet auch R/3 ein Transaktionskonzept, bei dem sichergestellt ist, dass Veränderungen am Datenbestand, die während der gesamten Verarbeitung auftreten, entweder vollständig oder gar nicht durchgeführt werden.

Gängiges Beispiel für ein solches Konzept sind Geldbewegungen auf Konten. Eine Transaktion besteht hierbei mindestens aus einer Sollbuchung und einer Habenbuchung (ggf. fallen weitere Daten zur Protokollierung z.B. in einem Journal an). Es muss hierbei sichergestellt werden, dass entweder beide Buchungen oder keine der Buchungen durchgeführt wird. Andernfalls würde ein inkonsistenter Datenbestand (in SAP-Sprache »Datenschiefstand«) produziert.

In R/3 wird dies erreicht, indem ein Transaktionskonzept auf der Anwendungsebene eingeführt wurde. Der Benutzer bewegt sich hierbei immer innerhalb einer Transaktion. Diese beginnt mit dem Programmstart auf höchster Ebene und endet entweder implizit mit dem Ende dieses Programms oder explizit durch Aufruf des Kommandos `COMMIT WORK` zum Bestätigen der Änderungen oder `ROLLBACK WORK`, um die Datenänderungen zu verwerfen. Dieses – vom Transaktionskonzept der darunter liegenden Datenbankprodukte abweichende – Konzept ist aufgrund der Client-Server-Architektur des Systems R/3 notwendig, welches im Folgenden kurz beschrieben werden soll. Eine ausführliche Beschreibung der Konzepte und Techniken kann im Buch »Die Technologie des Systems R/3« von Rüdiger Buck-Emden gefunden werden. Im Anschluss daran werden die Programmiertechniken für ABAP beschrieben, mit denen der Programmierer aktiv die Konzepte von R/3 nutzen kann, um asynchrone Programmabläufe bei der Verbuchung zu realisieren.

8.1 Architektur von R/3

8.1.1 System- und Rechnerarchitektur

Das System R/3 basiert auf einer ausgeklügelten, mehrschichtigen Client-Server-Architektur (siehe Abbildung 8.1). Basis des gesamten Systems bildet hierbei der so genannte Datenbank-Server (DB-Server), in dem die vollständige Datenhaltung mittels eines der unterstützten Datenbankprodukte erfolgt. Innerhalb der Datenbank-Server-Schicht des Systems ist keinerlei Anwendungslogik (Business-Logik) vorhanden, seine Aufgabe ist ausschließlich die konsistente Datenhaltung auf der Ebene des zugrunde liegenden Datenbank-Produktes. Der DB-Server bietet den Applikationsserver Schnittstellen an, um Daten aus der Datenbank zu lesen oder in der Datenbank zu speichern. Er reicht dabei das Transaktionskonzept der zugrunde liegenden Datenbank an die Applikationsserver durch. Dieses Transaktionskonzept findet in den so genannten Datenbank-LUW (siehe unten) Anwendung.

Die gesamt Anwendungs- (Business-) Logik des R/3-Systems liegt in den so genannten Applikationsservern. Es handelt sich hierbei um eine Einheit von einem Dispatcher und mehreren Workprozessen. Der Dispatcher übernimmt hierbei die Steuerung des Applikationsservers. Er startet die Workprozesse entsprechend der Konfiguration des Applikationsservers und verteilt die anfallenden Aufgaben auf die jeweiligen Workprozesse.

Diese kommunizieren mit den Präsentationsservern um die Dialogabfolge für die Benutzer abzuwickeln. Es handelt es sich hierbei klassischerweise um das SAPGUI von SAP. Dieses liegt im Release 4.6 nativ nur noch für die Microsoft-Windows-Betriebssysteme vor. Die übrigen Desktop-Betriebssysteme werden nur noch vom SAPGUI for JAVA bedient. Alternativ bietet SAP den Zugriff auf fast alle Transaktionen über einen normalen Internet-Browser.

Im Inter-/Intranet-Szenario bildet der seit längerem verfügbare Internet-Transaction-Server (ITS) die Ebene des Präsentationsservers. Tatsächlich kommuniziert der ITS mit den Applikationsservern wie ein SAPGUI. Über den Web-Browser werden die Anfragen der Browser-Anwender an den ITS übermittelt. Dieser reicht die Anfrage an den Applikationsserver weiter, empfängt von diesem die Antwort und bereitet diese in Form vorgefertigter HTML-Templates für den Benutzer auf, bevor er sie über den Web-Server zurück zum Browser überträgt.

Der Weg über den ITS findet in allen Szenarien der SAP, bei der Benutzer über einen Internet-Browser mit R/3-basierten Systemen kommunizieren, Anwendung. Dies betrifft insbesondere auch die SAP-Workplace.

Diese mehrstufige Architektur von R/3 findet sich im Übrigen in allen SAP-Systemen, denen ein R/3 zugrunde liegt wieder. Dies sind z.B. auch das SAP-Business-Warehouse (BW), Customer-Relationship-Management (CRM) oder auch der SAP-Workplace-Server.

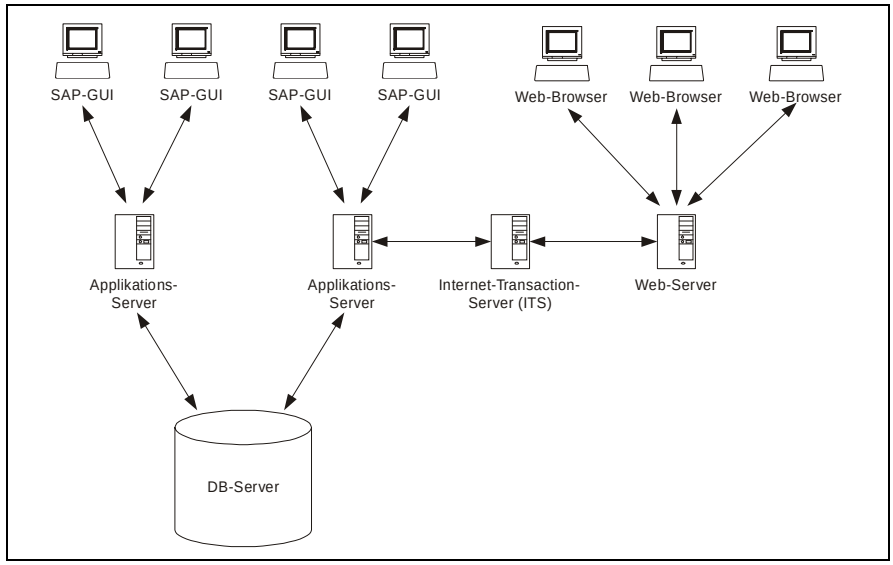

Abbildung 8.1
Systemarchitektur von R/3

Während der Datenbank-Server in jedem System genau einmal vorhanden ist, können bereits auf der Ebene der Applikationsserver mehrere Instanzen vorhanden sein. Die Anzahl der Applikationsserver ist hierbei nicht beschränkt und richtet sich zum einen nach der geforderten Leistung auf der Seite der Anwender und zum anderen an der Leistungsfähigkeit des zugrunde liegenden Datenbank-Servers.

Der ITS spielt in diesem Zusammenhang die Rolle eines normalen Benutzers, d.h. er kommuniziert mit dem Applikationsserver über das gleiche Protokoll (SAP verwendet hierfür das eigene DIAG-Protokoll) wie ein normales SAPGUI. In einer Systemlandschaft können daher auch mehrere ITS-Instanzen existieren, welche einen oder mehrere Web-Server bedienen.

Bei der hier beschriebenen Architektur handelt es sich um die softwaretechnische Systemarchitektur des Systems R/3. Die Trennung der verschiedenen Server-Typen muss nicht gleichzeitig auch eine Trennung der Hardware beinhalten. Die Trennung – zumindest von Datenbank-Server und Applikationsserver – hängt im Wesentlichen von der Hardware-Plattform und der Leistungsfähigkeit der verwendeten Systeme ab. In jedem Fall kann jedoch davon ausgegangen werden, dass auf dem Datenbank-Server auch gleichzeitig ein Applikationsserver installiert ist. Speziell für die Hintergrundverarbeitung ist es sinnvoll, diese auf dem Datenbank-Server durchzuführen, um die Menge der Daten, die über das Netzwerk transportiert werden müssen, gering zu halten.

8.1.2 Workprozesse

Die Verabeitung der Anwendungslogik erfolgt innerhalb der Applikationsserver in den so genannten Workprozessen. Es handelt sich hierbei um die Einheiten im System, in denen die Verarbeitung der ABAP-Programme und somit die gesamte Anwendungslogik von R/3 erfolgt.

Bei den Programmmodulen der Workprozesse handelt es sich jedoch auch nicht um monolitische Blöcke. Vielmehr bestehen die Workprozesse aus drei Modulen, dem Dialogprozessor, dem ABAP-Prozessor sowie der Datenbank-Schnittstelle (siehe Abbildung 8.2).

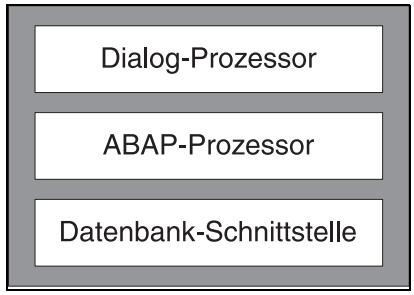

Abbildung 8.2
Aufbau von Workprozessen

Der Dialogprozessor bildet hierbei die Schnittstelle zum Präsentationsserver. Er beinhaltet die Dynpro-Steuerung und alle dazugehörigen Aufgaben. Zur Abwicklung dieser Aufgaben definiert ABAP einen eigenen Sprachschatz. Es handelt sich hierbei um die Anweisungen der Ablauflogik.

Über die Aufgaben des Dialogprozessors hinausgehende Anforderungen werden vom ABAP-Prozessor verarbeitet. Hier findet die gesamte Verarbeitung der Anwendungslogik des R/3-Systems statt. Der ABAP-Prozessor kommuniziert auf der einen Seite mit dem Dialogprozessor und auf der anderen Seite mit der dritten Komponente des Workprozesses, der Datenbank-Schnittstelle. In dieser Schicht sind die Sprachelemente zur jeweiligen Datenbank realisiert.

R/3 definiert verschiedene, auf die jeweiligen Aufgaben spezialisierte Typen von Workprozessen. Diese Spezialisierung spiegelt sich jedoch nicht im Aufbau der zugrunde liegenden Programme wieder. Alle Workprozesse bestehen aus den gleichen drei Komponenten. Die Prozesstypen stellen lediglich aufgabenspezifische Rollen dar, nach denen der Dispatcher, der Lastverteiler des Applikationsservers, die anfallenden Aufgaben verteilt.

R/3 definiert die folgenden fünf Prozesstypen:

- Dialogprozesse
- Batch-Prozesse
- Verbucher
- Spoolprozesse sowie
- Enqueue-Prozesse.

Wieviele Workprozesse auf einem Applikationsserver gleichzeitig ablaufen, hängt von der Leistungsfähigkeit der zugrunde liegenden Hardware ab. Welchen Typ diese Workprozesse haben, kann – bei Bedarf von verschiedenen Parametern abhängig – von den Systemadministratoren des R/3-Systems konfiguriert werden.

Jeder Benutzer, der sich an einem R/3-System anmeldet, tut dies an einem bestimmten Applikationsserver, der unter Umständen durch Lastverteilungsmechanismen (Load Balancing) ausgewählt wurde. Während der gesamten Anmeldung werden alle Anfragen von diesem einen Server abgearbeitet. Dialogprogramme oder Online-Reports werden hierbei von den Dialogprozessen verarbeitet. Jedesmal, wenn das SAPGUI mit dem Applikationsserver in Verbindung tritt, ist daran ein Dialogprozess beteiligt, der die Verarbeitung des jeweiligen Programmschrittes durchführt. Zunächst übernimmt der Dialogprozessor die Dialogsteuerung und nimmt bei Bedarf den ABAP-Prozessor zuhilfe.

Aufgrund der Architektur des Systems kann es sich hierbei jedesmal um einen anderen Workprozess handeln, denn die Workprozesse des R/3-Systems sind nicht an einen Benutzer gebunden (siehe Abbildung 8.3). Nach der Verarbeitung des jeweiligen Dialog- oder Programmschrittes werden die Ressourcen des Workprozesses wieder vollständig freigegeben und der Prozess steht für die Verarbeitung eines Dialogschrittes für einen anderen Benutzer zur Verfügung.

Dialogprozesse werden jedoch nicht nur von Benutzern mit SAPGUI in Anspruch genommen. Auch Anfragen, die über externe Interfaces wie z.B. Remote Function Calls (RFC) oder CPIC, die mit dem R/3-System kommunizieren, wenden von Dialogprozessen verarbeitet.

Programme, die als Hintergrundjobs eingeplant wurden, werden in so genannten Batch-Prozessen abgearbeitet. In der Hintergrundverarbeitung können ausschließlich Reports und Aufrufe zu externen Systemen eingeplant werden. Die Aufgaben der Batch-Prozesse sind darauf ausgerichtet, große Datenmengen zu verarbeiten. Sie beinhalten nicht die Möglichkeit, Dialoge anzuzeigen oder zu verarbeiten. Aus diesem Grund können Programme, die Batch-Input-Techniken verwenden (siehe Kapitel 6) nicht als Hintergrundjob eingeplant werden.

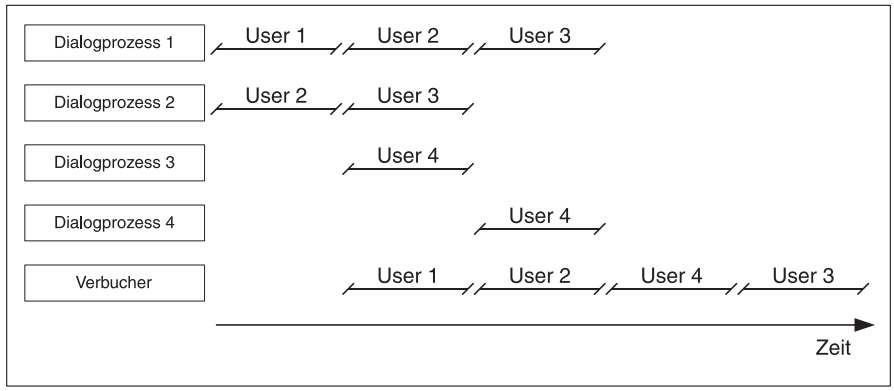

Abbildung 8.3
Verteilung der Benutzeranfragen auf die (Dialog-) Workprozesse

Am Abschluss jeder SAP-LUW (siehe Abschnitt 8.2) erfolgt die Verbuchung der durchgeführten Datenbankänderungen. Um diese – oftmals zeitintensiven – Operationen in der Datenbank durchzuführen, stehen spezielle Prozesse, die Verbucher zur Verfügung. Sie haben – wie Batch-Prozesse – keine Möglichkeit, eine Dialogverarbeitung durchzuführen. In der Regel dienen Verbucher lediglich dazu, Datenveränderungen in der Datenbank persistent zu speichern. SAP unterscheidet hierbei zwischen V1- und V2-Verbuchern. Der Unterschied zwischen diesen beiden Workprozesstypen ist der, dass V1-Verbucher zeitnah zum auslösenden `COMMIT WORK` aufgerufen werden, während V2-Vebucher - mit einer wesentlich niedrigeren Priorität ausgestattet – unter Umständen wesentlich später durchgeführt werden. Innerhalb von R/3 werden V1-Verbucher in der Regel verwendet, um die Anwendungsdaten einer Transaktion asynchron aber dennoch zeitnah zu speichern. V2-Verbucher hingegen werden verwendet, um unkritische Daten, wie beispielsweise die Fortschreibung von Datenverdichtungen, die nicht unmittelbar die Anwendungsdaten berühren, durchzuführen.

Sämtliche Druckausgaben – gleich auf welches Medium – werden von so genannten Spool-Prozessen verarbeitet. In diesen Prozessen findet die Aufbereitung und die Kommunikation mit den jeweiligen Ausgabegeräten (Drucker, Fax-Server o.ä.) statt.

Der Enqueue-Prozess schließlich ist dafür zuständig, die Sperren innerhalb des SAP-Systems zu verwalten. R/3 bietet dem Programmierer ein Sperrkonzept auf Anwendungsebene, d.h. Sperren werden nicht auf der Ebene der Datenbank-Tabelle oder auf Satzebene gesetzt sondern auf der Ebene der betriebswirtschaftlichen Objekte. Hierzu werden nicht die Sperrmechanismen der jeweiligen Datenbank-Produkte verwendet. In jedem R/3-System existiert genau ein Workprozess, der diese Sperrobjekte verwaltet. Näheres zum Sperrkonzept und dem Enqueue-Prozess kann aus Kapitel 9 entnommen werden.

Mit Ausnahme der beiden zuletzt genannten Workprozess-Typen können die Workprozesse gezielt zur Realisierung von asynchronen, d.h. voneinander unabhängigen Programmsträngen während der Verbuchung verwendet werden. Um die Konsistenz der Daten auch über parallele Ausführungspfade hinweg zu gewährleisten, stellt R/3 das Konzept von logischen Programmeinheiten zur Verfügung. Bevor auf die jeweiligen Programmiertechniken zur Nutzung dieser Möglichkeiten eingegangen wird, soll im Folgenden ein Überblick über diese Programmeinheiten (auch Logical Unit of Work, kurz LUW, genannt) gegeben werden.

8.2 Logical Units of Work

Beim Arbeiten mit R/3 bewegen sich die Benutzer immer innerhalb einer logischen Arbeitseinheit, der so genannten Logical Unit of Work (LUW). Eine LUW bildet auf Anwendungsebene eine Transaktion, an deren Ende die Datenänderungen, die während der LUW durchgeführt wurden, endgültig in der Datenbank gespeichert werden und somit für andere Benutzer sichtbar sind, oder vollständig verworfen werden. Der Begriff der SAP-LUW, wie diese Einheiten auf Anwendungsebene im SAP-Sprachjargon heißen, ist hierbei streng von den Transaktionen, die auf der Ebene der Datenbank, die hier Datenbank-LUW genannt werden, zu trennen.

8.2.1 Datenbank-LUWs

Fast alle relationalen Datenbanksysteme kennen das Konzept der Transaktionssicherheit. Dieses ist aus der Notwendigkeit heraus entstanden, dass Veränderungen am Datenbestand der Datenbank in der Regel nicht in einem Schritt durchgeführt werden, sondern mehrere Datenänderungen zusammengenommen einen semantischen Schritt bilden, der entweder vollständig oder gar nicht durchgeführt werden darf. Hierbei werden sämtliche Manipulationen am Datenbestand zunächst durch spezielle Techniken zwischengespeichert und erst am Ende der Datenbank-Transaktion für alle Benutzer der Datenbank sichtbar in den »echten« Tabellen gespeichert.

Diese Transaktionen zeichnen sich dadurch aus, dass sie die Sitzung genau eines Benutzers bzw. Prozesses widerspiegeln. Datenbank-Transaktionen können nicht unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden, genausowenig kann eine Datenbank-Transaktion den Besitzer – sprich den Workprozess, der die Transaktion gestartet hat – wechseln. In R/3 hat jeder Workprozess eine Verbindung zur Datenbank, bildet also – aus Datenbanksicht – eine Session, die Transaktionen durchführen kann.

Es gilt demnach die Einschränkung, dass jeder Workprozess des R/3-Systems genau eine Verbindung zur Datenbank hält (siehe Abbildung 8.4) und folglich zu einer Zeit nur eine Datenbank-Transaktion aktiv sein kann. Bevor eine neue

Transaktion gestartet werden kann, muss zuvor die aktive Transaktion beendet werden. Gleichzeitig gilt jedoch, dass ein Workprozess nicht exklusiv für einen Benutzer zur Verfügung steht, woraus sich ableitet, dass ein Transaktionskonzept auf Anwendungsebene nicht mit den Mitteln der Transaktionen auf Datenbank-Ebene abgewickelt werden kann. Selbst wenn ein Dialogprozess exklusiv für eine R/3-Transaktion zur Verfügung stünde, wäre das Verbucherkonzept in seiner realisierten Form nicht möglich, da spätestens in diesem Falle ein Wechsel des Workprozesses notwendig würde.

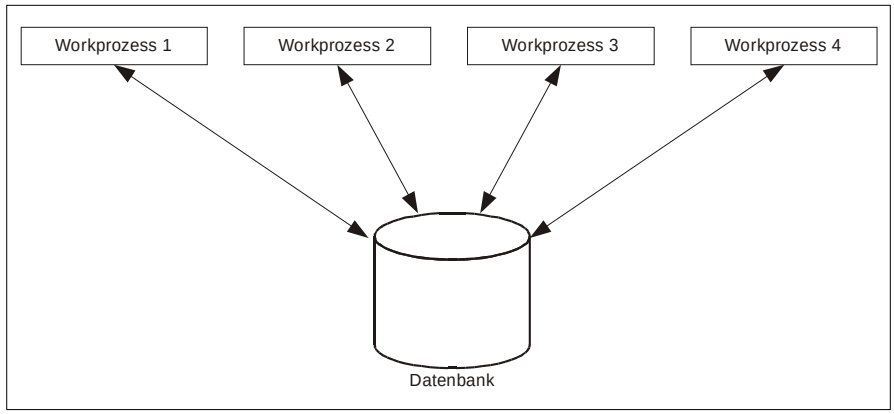

Abbildung 8.4
Zugriff der Workprozesse auf die Datenbank

In R/3 wird dieses Problem dadurch gelöst, dass ein eigenes Transaktionskonzept, welches auf den so genannten SAP-LUW basiert, etabliert wurde. Dieses verwendet im »kleinen« zwar die Funktionalität der Datenbank-Transaktionen, definiert im Ganzen gesehen jedoch einen eigenen Transaktionsbegriff. Tatsächlich ist es so, dass jeder Workprozess, eine neue Datenbank-LUW eröffnet, sobald er einen neuen Kontext (bei Dialogtransaktionen z.B. einen Dialogschritt) zur Ausführung zugewiesen bekommt. Sobald der jeweilige Dialogschritt abgeschlossen ist und der Workprozess zur Bearbeitung eines anderen Kontextes vorbereitet wird, wird auch die zugrunde liegende Datenbank-LUW abgeschlossen. Hierdurch wird gewährleistet, dass alle Veränderungen auf der Ebene der Datenbank konsistent durchgeführt werden können.

8.2.2 SAP-LUW

Auf Anwendungsebene hat der Begriff Transaktion einen wesentlich größeren Umfang, als auf Datenbank-Ebene. Mit Beginn einer Dialogtransaktion beginnt auch eine SAP-LUW. Diese endet entweder implizit mit dem Ende des Dialogprogramms oder explizit durch Aufruf der ABAP-Kommandos `COMMIT WORK` oder `ROLLBACK WORK` zum Bestätigen bzw. Abbrechen der Transaktion. Jeder Dialog-

schritt der SAP-LUW kann hierbei von einem anderen Workprozess und dementsprechend in einer eigenen Datenbank-LUW durchgeführt werden (Abbildung 8.5). Sämtliche Änderungen am Datenbestand werden hierbei in eigenen Tabellen des R/3-Systems mitgeführt und erst am Ende der Dialogtransaktion in einem speziellen Prozess, dem Verbucher-Task, in einer einzigen Datenbank-LUW transaktionssicher in den Anwendungstabellen der Datenbank fortgeschrieben.

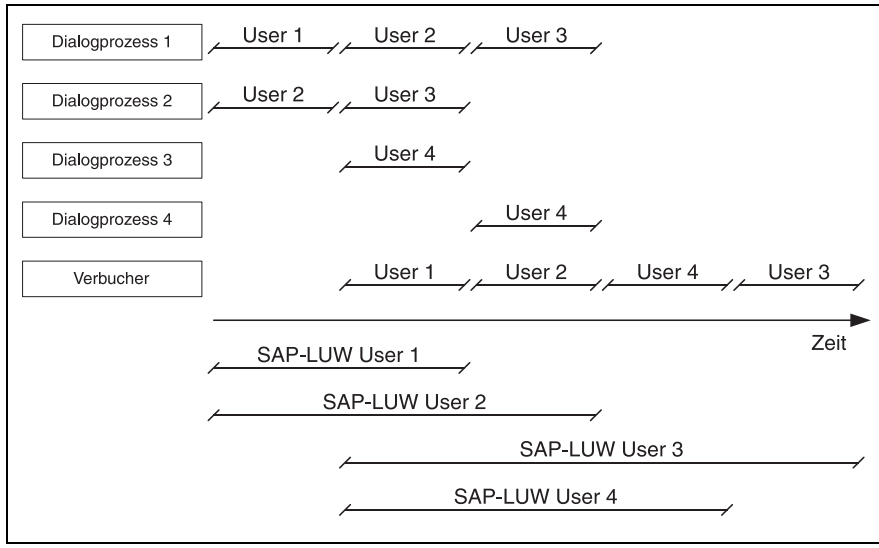

Abbildung 8.5
Verteilung der Dialogschritte auf die Workprozesse

Die Zeitabschnitte im oberen Bereich der Abbildung zeigen die einzelnen Dialogschritte der Benutzer. Diese sind gleichbedeutetend mit der Datenbank-LUW. Jedesmal, wenn ein Workprozess einen Dialogschritt abgearbeitet hat, wird die Datenbank-LUW abgeschlossen und mit Beginn der Verarbeitung eines Dialogschrittes für einen anderen Benutzer eine neue Datenbank-LUW gestartet. Die Abbildung zeigt weiterhin, dass jeder Dialogschritt eines Benutzers auf einem unterschiedlichen Workprozess ausgeführt werden kann. Hierbei gilt jedoch, dass jeder Benutzer an einem Applikationsserver angemeldet wird und dementsprechend nur die Prozesse dieses Servers zur Abarbeitung der Dialogschritte dieses Benutzers zur Verfügung stehen.

Jede SAP-LUW schließt mit einem Aufruf der Verbuchung ab, in dem die während der Transaktion protokollierten Datenänderungen in der Datenbank gesichert werden. In der Abbildung ist nur ein Verbucher-Prozess dargestellt. Die Anzahl der Dialogprozesse (Workprozesse) und Verbucher-Prozesse kann vom Administrator des R/3-Systems frei konfiguriert (bis zu der von der Hardware definierten maximalen Anzahl von möglichen Prozessen) werden. Die Dialog-

schritte der einzelnen aktiven Dialogtransaktionen werden hierbei nach Bedarf auf die zur Verfügung stehenden Workprozesse verteilt.

Im unteren Bereich der Abbildung sind die SAP-LUWs der jeweiligen Benutzertransakionen aufgetragen. Hierdurch wird deutlich, dass die SAP-LUW sich über mehrere Datenbank-LUW erstrecken kann. Die SAP-LUW endet hierbei immer mit einem Aufruf des jeweiligen Verbucher-Tasks. Zu beachten ist hierbei, dass der Begriff SAP-LUW nicht unbedingt gleichzusetzen ist mit dem Begriff Dialogtransaktion. Zum einen laufen auch Reports in SAP-LUWs, da auch hier Manipulationen am Datenbestand durchgeführt werden können, zum anderen ist es auch möglich, dass eine Dialogtransaktion (und auch ein Report) aus mehreren SAP-LUW besteht.

Soviel zur Theorie. In den folgenden Abschnitten werden die Programmiertechniken in ABAP vorgestellt, mit denen die vorgestellten Konzepte gezielt genutzt werden können.

8.3 Asynchrone Programmiertechniken

Basierend auf der oben beschriebenen Architektur können in R/3 verschiedene Wege genutzt werden, um den Programmablauf zu beeinflussen.

Diese Methoden können grob zwischen Techniken, die in der gleichen LUW (fortan meint LUW immer SAP-LUW) ablaufen und Techniken, die bewirken, dass der Programmlauf in einer neuen LUW durchgeführt wird, unterschieden werden.

Zur ersten Gruppe gehören Aufrufe, die beim `COMMIT WORK` im Dialogprozess durchgeführt werden, aber auch die so genannten Verbucherbausteine. Bei diesen handelt es sich um Funktionsbausteine oder `FORM`-Routinen, die in einem Verbucher-Prozess abgearbeitet werden. Auf der anderen Seite gibt es Möglichkeiten, den Programmlauf so zu steuern, dass die gerufene Routine in einer eigenen LUW abläuft. Unter diese Kategorie fallen Funktionsbausteine, die im Backgroud-Task gestartet werden. Diese Funktionsbausteine können im lokalen R/3-System aber auch in externen Systemen liegen.

8.3.1 Ablauf der Verbuchung

Am Ende einer Transaktion werden die durchgeführten Datenänderungen mit der ABAP-Anweisung `COMMIT WORK` permanent in der Datenbank des R/3-Systems fortgeschrieben. Dies erfolgt in mehreren Stufen.

Zunächst werden bei explizitem oder implizitem Auslösen von `COMMIT WORK` innerhalb des Dialogprozesses alle `FORM`-Routinen nacheinander abgearbeitet, welche mit dem Zusatz `ON COMMIT` der `PERFORM`-Anweisung aufgerufen wurden. Die Verarbeitung dieser Routinen erfolgt noch innerhalb des Dialogprozesses. Im

Anschluss daran beginnt die eigentliche Verbuchung. Hierzu werden in einem Verbucher-Workprozess die Verbuchungsbausteine mit höherer Priorität (V1-Verbucher) abgearbeitet. Diese Verbuchung läuft asynchron aber zeitnah mit der Dialogverarbeitung. Mit geringerer Priorität werden die V2-Verbucherbausteine in den entsprechenden Workprozessen verarbeitet.

Nach erfolgreicher Verarbeitung der V1-Verbucher werden etwaige noch bestehende Sperreinträge aufgehoben, um den produktiven Datenbestand wieder für andere LUW freizugeben. Nicht zuletzt aus diesem Grund heraus ist es anzuraten, Änderungen nicht an den operativen Unternehmensdaten in einem V2-Verbucherbaustein durchzuführen.

Wenn alle Verbucherbausteine erfolgreich verarbeitet wurden, ist die LUW vollständig abgeschlossen.

8.3.2 FORM-Routinen ON COMMIT

Die wohl einfachste Form, eine Unterroutine im Bereich der Verbuchung ablaufen zu lassen, besteht darin, eine FORM-Routine mit dem Zusatz ON COMMIT aufzurufen. Die Routine wird dabei nicht synchron zum Aufruf gestartet sondern am Ende der LUW, jedoch vor den Verbucherbausteinen.

Es handelt sich hierbei um eine sehr performante Art der asynchronen Programmierung, da für diese Aufrufe keine Parameter zwischengespeichert werden müssen. Dies resultiert aus der Restriktion, dass derartige FORM-Routinen keine Aufrufschnittstelle definieren können. Innerhalb der Routinen besteht lediglich die Möglichkeit, auf globale Datenobjekte des Programms zuzugreifen, wobei hierbei beachtet werden muss, dass die Variablen zum Ausführungszeitpunkt der Routine unter Umständen nicht den Wert enthalten, den sie beim Aufruf der Unterroutine enthalten haben. Die FORM-Routine arbeitet mit den Werten der globalen Variablen, die zum Zeitpunkt der Ausführung der Routine aktuell sind.

Das folgende Beispiel soll dies verdeutlichen.

```
...
DATA: G_INT    TYPE I.

G_INT = 1.
PERFORM WRITE_INT.
PERFORM WRITE_INT ON COMMIT.

G_INT = 5.
PERFORM WRITE_INT.
PERFORM WRITE_INT ON COMMIT.

G_INT = 7.

COMMIT WORK.
```

```
  ...

FORM WRITE_INT.
  WRITE / G_INT.
ENDFORM.
```

Die Ausgabe dieses Programmfragments würde im Ergebnis die Zahlenfolge

```
1
5
7
```

liefern.

Der erste – synchrone – Aufruf der Routine `WRITE_INT` würde den aktuellen Wert `1` der Variable `G_INT` zur Ausgabe bringen. Zum Zeitpunkt des zweiten Aufrufs der Routine hat die Variable zwar ebenfalls den Wert `1`, jedoch erfolgt die Ausführung dieses Aufrufs erst innerhalb der Verbuchung des Programms, zu dem die Variable den Wert `7` enthält.

Im Folgenden wird die Routine `WRITE_INT` nochmals synchron und asynchron aufgerufen. Hierbei erkennt man eine weitere Einschränkung der Verwendung von `ON COMMIT` zur Verbuchung. Jede `FORM`-Routine wird bei diesem Verfahren maximal einmal aufgerufen. Weitere `PERFORM ... ON COMMIT` werden vom System ignoriert, würden sie doch die gleichen Verarbeitungen auf den gleichen Daten erneut durchführen.

8.3.3 Programmieren mit Verbucherbausteinen

Anlegen von Verbucherbausteinen

Bei Verbucherbausteinen handelt es sich im Grunde um gewöhnliche Funktionsbausteine, deren Aufrufschnittstelle jedoch gewissen Einschränkungen unterworfen ist, die sich auf ihrem Charaketer als asynchron ablaufende Einheiten gründen. Aufgrund dieser Eigenschaft macht es keinen Sinn, Rückgabeparameter in Verbucherbausteinen zu definieren, da zum Zeitpunkt der Ausführung des Bausteins der aufrufende Kontext nicht mehr existiert und demnach die Rückgabeparameter nicht verarbeitet werden könnten. Aus dem gleichen Grunde dürfen Verbucherbausteine keine Ausnahmen definieren.

Da die Aufrufparameter für Verbucherbausteine im Rahmen der Verbuchung in Tabellen zwischengespeichert werden, müssen sämtliche Aufruf- sowie Tabellenparameter von Verbucherbausteinen mit Bezug zum Data Dictionary definiert werden. Die Verwendung von lokalen Datentypen und ABAP-Datentypen ist an dieser Stelle nicht zulässig.

Weiterhin müssen Verbucherbausteine für die Verarbeitung in Verbucherprozessen gekennzeichnet sein

SAP unterscheidet hierbei zwischen V1-Verbuchern und V2-Verbuchern. Diese Unterscheidung wird in den Eigenschaften des Funktionsbausteins getroffen (siehe Abbildung 8.6). Die Festlegung erfolgt hier durch die Definition des Prozesstypen für den Baustein.

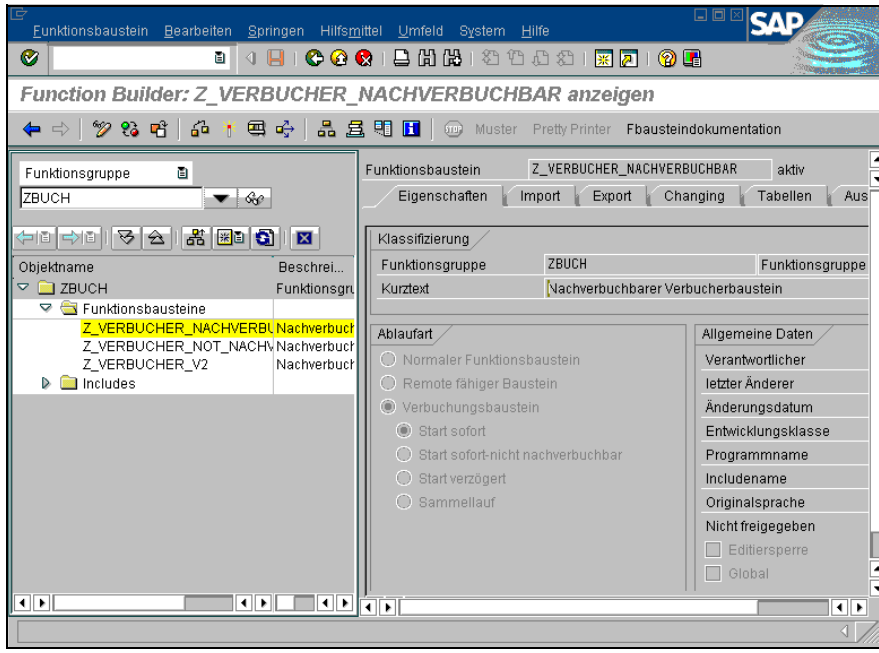

Abbildung 8.6
Verbucherbausteine anlegen (© SAP AG)

Hierbei stehen drei Prozesstypen für die Definition von Verbucherbausteinen zur Verfügung.

Bei Bausteinen, die den Prozesstyp »Normal« haben, handelt es sich um normale Funktionsbausteine. Sie können nicht asynchron als Verbucherbausteine aufgerufen werden. Gleichwohl können sie natürlich innerhalb der Verbuchung von einem Verbucherbaustein aufgerufen werden. Der Prozesstyp legt lediglich fest, ob intern die Mechanismen zum asynchronen Aufruf des Bausteins (z.B. Pufferung der Aufrufparameter) vorgesehen werden soll. Bei Funktionen, die von Verbucherbausteinen aufgerufen werden, ist dies nicht notwendig, da die Ausführung relativ zum Verbucherbaustein synchron erfolgt.

Der Prozesstyp »Verbuchung, Start sofort« kennzeichnet einen Baustein als V1-Verbucher. Er wird während der Verbuchung mit hoher Priorität abgearbeitet. Falls innerhalb des Verbucherbausteins ein Fehler auftritt, bricht die Verbuchung ab. Die entsprechenden Einträge der Verbuchungstabellen, über die der Start des Bausteins gesteuert wurde, bleiben jedoch erhalten. Der Benutzer, der den Verbucherbaustein

aufgerufen hat, wird mit einer Express-Mail auf die Fehlersituation hingewiesen und kann diesen in der Transaktion SM13 (siehe Abbildung 8.7) analysieren. Falls es sich lediglich um einen korrigierbaren Fehler handelt, kann der Benutzer den Verbucher erneut starten und ggf. zum positiven Abschluss bringen.

Verbucherbausteine vom Typ »Start sofort, nicht nachverbuchbar« sind ebenfalls V1-Verbucher. Im Gegensatz zu den zuvor genannten V1-Verbuchern besteht hierbei jedoch nicht die Möglichkeit der Nachverbuchung in der Transaktion SM13. Tritt während der Verarbeitung eines solchen Bausteins ein Fehler auf, wird die Verbuchung abgebrochen und die LUW beendet. Der Benutzer bekommt diesen Zustand ebenfalls per Express-Mail mitgeteilt, kann jedoch nicht mehr in den Ablauf eingreifen, um den Fehler zu korrigieren.

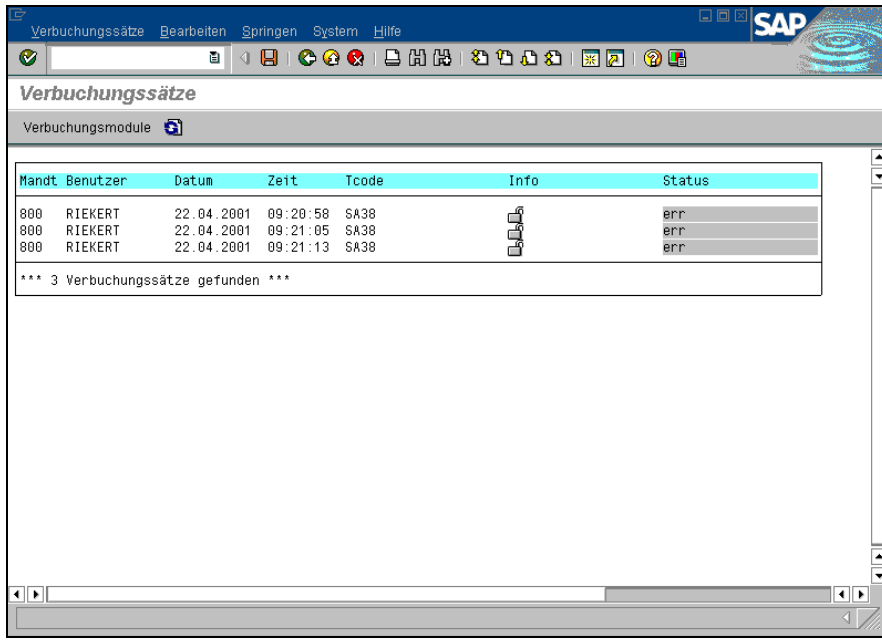

Abbildung 8.7
Verbuchungsmonitor (Transaktion SM13) (© SAP AG)

Ein Funktionsbaustein wird als V2-Verbucher definiert, indem er mit dem Prozesstyp »Verbuchung mit Start verzögert« angelegt wird. Diese Funktionsbausteine werden mit geringerer Priorität als V1-Verbucher ausgeführt. Sie laufen immer in einer eigenen Datenbank-LUW ab.

Die weitere Implementierung der Verbucherbausteine erfolgt identisch wie die von normalen Funktionsbausteinen. Der Programmierer sollte jedoch immer den asynchronen Programmablauf bedenken. Die Verwendung globaler Variablen und andere, von Umgebungsbedingungen abhängige Annahmen verbieten sich daher.

Aufruf von Funktionsbausteinen im Verbucher

Der Aufruf von Verbucherbausteinen ist fast identisch mit dem Aufruf normaler Bausteine. Soll beispielsweise der Funktionsbaustein Z_VERBUCHER im Rahmen der Verbuchung aufgerufen werden, würde der Aufruf folgendes Aussehen haben:

```
CALL FUNCTION 'Z_VERBUCHER' IN UPDATE TASK
     EXPORTING
        ....
     TABLES
     .....
     EXCEPTIONS
     ....
```

Der Aufruf unterscheidet sich lediglich durch den Zusatz IN UPDATE TASK von einem synchronen Aufruf. Die übrigen Möglichkeiten der Parameterübergabe sind von diesem Konzept unberührt. Im Gegensatz zu FORM-Routinen, die mit dem Zusatz ON COMMIT aufgerufen werden können, besteht bei Verbucherbausteinen demnach die Möglichkeit Parameter zu übergeben. Diese werden zum Zeitpunkt des Aufrufs in Verbuchertabellen zwischengespeichert. Anders als bei den zuvor genannten Unterprogrammen enthalten Parameter von Verbucherbausteinen während der Ausfühung des Bausteins die Werte, die zum Zeitpunkt des Funktionsaufrufes aktuell waren und nicht die Werte, die zum Zeitpunkt der Funktionsausführung aktiv sind.

Wird ein Funktionsbaustein im Rahmen einer Transaktion mehrfach aufgerufen, wird er auch im Rahmen der Verbuchung mehrmals mit den jeweiligen Parametern ausgeführt.

Ob ein Verbucherbaustein im V1-Verbucher oder im Rahmen der V2-Verbuchung abgearbeitet wird, entscheidet einzig und alleine der Prozesstyp des Funktionsbausteins in den Attributen (siehe oben). V2-Verbucher werden vom Dispatcher mit einer entsprechend geringeren Priorität ausgeführt. Weiterhin werden etwaige bestehende Satzsperren der LUW nach Abschluss der V1-Verbuchung aufgehoben. V2-Verbucherbausteine sollten daher lediglich Fortschreibungen und unkritische Datenänderungen durchführen.

Synchrone und asynchrone Verbuchung

Standardmäßig erfolgt die Verbuchung in den Schritten:

❶ Aufruf aller FORM-Routinen mit Zusatz ON COMMIT

❷ Aufruf aller V1-Verbucher

❸ Aufruf aller V2-Verbucher

Während Schritt 1 immer synchron im Dialogprozess, der die LUW abschließt (COMMIT WORK) ausgeführt wird, erfolgt die Abarbeitung der Verbucherbausteine in speziellen Workprozessen, den Verbuchern. Diese Ausführung erfolgt normalerweise asynchron, d.h. der Dialogprozess fährt mit der Verarbeitung fort, sobald Schritt 1 aus obiger Liste vollständig abgearbeitet ist. Die übrigen Schritte laufen asynchron in den Verbucherprozessen ab.

Es bestehen jedoch zwei Möglichkeiten, zu erzwingen, dass die Ausführung der Verbuchungs-Bausteine ebenfalls synchron zum auslösenden Dialogprozess erfolgt.

Eine LUW wird explizit durch den Aufruf der ABAP-Anweisung COMMIT WORK beendet. Dieser Aufruf löst das oben beschriebene Verfahren mit asynchroner Verbuchung aus.

Wird die COMMIT WORK-Anweisung mit dem Zusatz AND WAIT gestartet, wird der Dialogprozess so lange blockiert, bis sämtliche Verbucherbausteine abgearbeitet sind. Die Ausführung der Verbuchungsbausteine erfolgt jedoch in den Verbucherprozessen des Systems.

Daneben besteht eine weitere Möglichkeit, den Verbuchungsablauf dahingehend zu beeinflussen, dass die Verarbeitung der Verbuchungsbausteine synchron erfolgt.

Durch den Aufruf der ABAP-Anweisung

```
SET UPDATE-TASK LOCAL.
```

wird R/3 angewiesen, die Verbuchung nicht in Verbucherprozessen sondern in den auslösenden Dialogprozessen durchzuführen. Implizit bedeutet dies, dass die Dialogprozesse so lange blockiert sind, bis die Verbucher ausgeführt wurden.

Dieser Schalter wird am Ende der Verarbeitung von COMMIT WORK und ROLLBACK WORK wieder auf den Standardwert zurückgesetzt und muss daher in jeder LUW, wo dieses Verfahren gewünscht wird, explizit erneut gesetzt werden.

8.3.4 Funktionsaufrufe über RFC

Mit der Anweisung CALL FUNCTION können auch Funktionsbausteine außerhalb des R/3-Systems aufgerufen werden. Hierbei kann es sich um andere R/3-Systeme, R/2-Systeme oder um sonstige Fremdsysteme, die die RFC-Serverschnittstelle von SAP implementieren, handeln.

Ist das Zielsystem des Aufrufs ebenfalls ein R/3-System, kann es sich hierbei auch um das gleiche System wie das Aufrufende handeln. In diesem Fall erfolgt der Aufruf aus dem aktuellen R/3-System heraus wieder in das gleiche System hinein. Im Unterschied zu Verbucherbausteinen läuft hierbei der gerufene Funktionsbaustein in einer eigenen LUW.

Der aufgerufene Funktionsbaustein muss hierbei den bislang noch nicht erwähnten Prozesstyp »Remote Function Call unterstützt« besitzen. Dieser Typ kennzeichnet Funktionsbausteine als über RFC aufrufbar. Der Funktionsbaustein kann jedoch auch innerhalb eines R/3-Systems normal aufgerufen werden.

Der Aufruf des Funktionsbausteins im Zielsystem erfolgt durch Hinzufügen der Zieladresse an den `CALL FUNCTION`-Aufruf. Dies geschieht in der Form

```
CALL FUNCTION     <Funktionsname>
     DESTINATION <Zielsystem>
     ...
```

Das als `<Zielsystem>` angegebene System muss im Bereich der RFC-Destinationen in der Transaktion `SM59` gepflegt sein. Unter der Rubrik »interne Destinationen« kann hier das eigene System definiert werden.

Der Aufruf des Funktionsbausteins erfolgt hierbei synchron, d.h. das aufrufende Programm wartet, bis der Funktionsbaustein im Zielsystem vollständig abgearbeitet wurde und die Kontrolle zurückgibt.

Über den weiteren Zusatz `IN BACKGROUD TASK` kann der Funktionsbaustein wie ein Verbucherbaustein behandelt werden. Der Aufruf im Fremdsystem erfolgt hierbei asynchron im Rahmen der Verbuchung. Für die Aufrufschnittstelle des Funktionsbausteins bestehen die gleichen Einschränkungen wie für die Schnittstelle von Verbucherbausteinen, d.h. sie können weder Rückgabeparameter noch Ausnahmen definieren.

In beiden Fällen gilt, dass Tabellenparameter nicht als Referenz (»by reference«), sondern als Kopie (»by value«) erfolgt. Anders als Tabellenparameter in herkömmlichen Funktionsbausteinen wird hier keine Kopfzeile für die Tabelle mitdefiniert.

Ein Aufruf nach dem Muster

```
CALL FUNCTION     <Funktionsname>
     DESTINATION <Zielsystem>
     IN BACKGROUD TASK
     ...
```

würde – genauso wie obiger synchroner Aufruf – in einer eigenen LUW ablaufen. R/3 bündelt hierbei alle entfernten Funktionsaufrufe in ein System zu einer eigenen LUW, d.h. nicht jeder Funktionsaufruf bildet für sich genommen eine LUW im fremden System, sondern alle Funktionsaufrufe in einem Fremdsystem werden in einer LUW abgearbeitet.

Dieser Mechanismus des asynchronen Funktionsaufrufs in fremden Systemen wird in R/3 unter dem Begriff »transaktionaler RFC« (tRFC) zusammengefasst. Das Konzept der transaktionalen RFC beinhaltet zusätzlich, dass das Fremdsystem zum Zeitpunkt des Funktionsaufrufes nicht unbedingt verfügbar sein

muss. Ist dies nicht der Fall, plant R/3 den Funktionsaufruf im Rahmen der Hintergrundverarbeitung ein und versucht in Abständen von 15 Minuten bis zu 30 Mal die Funktion im fremden System zu starten. Ein anderes Konzept asynchroner Funktionsaufrufe, die eine Verfügbarkeit des fremden Systems voraussetzt, wird in R/3 unter dem Namen »asynchroner RFC« (aRFC) geführt. Nähere Informationen hierzu und zur Programmierung von RFC-Bausteinen im Allgemeinen findet sich in der SAP-Dokumentation »RFC-Programmierung in ABAP«.

Zusammenfassend kann gesagt werden, dass Funktionsbausteine in einer anderen LUW unter Zuhilfenahme der RFC-Schnittstelle von R/3 erfolgen können. Diese können entweder synchron über den Zusatz `DESTINATION` des Kommandos `CALL FUNCTION` oder asynchron als transaktionaler RFC über den Zusatz `DESTINATION ... IN BACKGROUND TASK` erfolgen. Der aufgerufene Funktionsbausein kann – je nach Definition der RFC-Ziele im jeweiligen System – im gleichen oder in anderen Systemen als dem aktuellen R/3-System erfolgen.

Programmieren mit Sperren

9

9.1 Allgemeines zu Sperrmechanismen

In einer Client/Server-Umgebung wie in R/3 arbeiten viele Benutzer mit dem gleichen Datenbestand. Sperrmechanismen verhindern hierbei, dass mehrere Benutzer zum gleichen Zeitpunkt versuchen, auf dieselben Anwendungsobjekte zuzugreifen. Die ist beim konkurrierenden Schreibzugriff mehrerer Benutzer unbedingt erforderlich, kann aber auch beim Lesenden Zugriff notwendig sein, um zu verhindern, dass ein anderer Benutzer die angezeigten Daten verändert.

Um den Datenbestand in der Datenbank konsistent zu halten, ist es daher nicht nur notwendig, ein Transaktionskonzept zu realisieren, um Datenänderungen zu bündeln, sondern auch, konkurrierende Zugriffe auf Anwendungsobjekte zu verhindern.

Die Datenbankprodukte bieten in der Regel Mechanismen, um einzelne oder mehrere Datensätze oder ganze Tabellen zu sperren. Es gibt im Wesentlichen zwei Gründe, warum diese Sperrmechanismen in R/3 keine Verwendung finden.

Zum einen haben diese Sperren in der Regel für die Dauer einer Datenbank-Transaktion, im Umfeld von R/3 also einer Datenbank-LUW Bestand. In R/3 ist der Kontext, währenddessen auf ein Anwendungsobjekt exklusiv zugegriffen werden soll, jedoch wesentlich länger als eine Datenbank-LUW. Daher muss die Möglichkeit bestehen, Objekte für die Dauer einer SAP-LUW gegen Zugriff von anderen Programmen zu schützen. Diese Zugriffe können unter Umständen von verschiedenen Workprozessen, in Ausnahmefällen sogar von verschiedenen Servern aus erfolgen. Hierfür bieten die Datenbank-Hersteller keine Lösung.

Zum anderen ist es auf der Ebene der Datenbank lediglich möglich, Sperren auf die im Meta-Modell der Datenbank enthaltenen Objekte zu setzten. Im Beispiel der R/3 zugrunde liegenden Datenbanken handelt es sich hierbei um Relationen, also Tabellen. Das Meta-Modell von R/3 basiert jedoch nicht auf Relatio-

nen, sondern auf Anwendungsobjekten, wie z.B. Aufträgen. Diese Anwendungsobjekte werden in der Datenbank zwar in Relationen abgebildet, jedoch ist dies aus der Sicht des R/3-Anwenders nicht die Ebene, auf der gearbeitet werden soll (Nicht umsonst gibt es Transaktionen um AUFTRÄGE zu bearbeiten und keine Transaktionen, um die Tabelle AUFK zu pflegen).

SAP trägt diesem Umstand Rechnung, indem das Sperrkonzept von R/3 nicht auf den Konzepten der Datenbank-Produkte beruht. Vielmehr wurde ein eigenes Sperrkonzept auf Anwendungsebene etabliert, welches auch die beiden zuvor genannten Forderungen erfüllen kann.

9.2 Der Sperrmechanismus von R/3

Wie bereits erwähnt, wird zur Realisierung des Sperrkonzepts in R/3 nicht auf die Möglichkeiten der jeweiligen Datenbankprodukte zurückgegriffen. Vielmehr handelt es sich um logische Sperren, die ebenfalls in Tabellen, welche im Memory des Servers, der die Sperren verwaltet, abgelegt werden. Das Konzept beruht auf einem kooperativen Stil der Programme untereinander. Es wird technisch nicht verhindert, auf gesperrte Sätze lesend oder schreibend zuzugreifen. Vielmehr vertraut das System darauf, dass Programme, die auf Sätze zugreifen wollen, Sperren anfordern bzw. anschließend wieder freigeben, sich also kooperativ gegenüber anderen Programmen im System verhalten.

Hierfür existieren so genannte Sperrobjekte, welche im Bereich des Data Dictionaries von R/3 angelegt und gepflegt werden. Sie definieren die Tabellen, die zu einem Sperrobjekt gehören sowie die Felder, über die die zu sperrenden Sätze der jeweiligen Tabellen eingeschränkt werden. Sperren können einzelne oder mehrere Sätze der Datenbank betreffen. Anders als bei Sperren der Datenbank-Produkte, können auch Sätze, die noch nicht in der Datenbank existieren mit einer Sperre versehen werden, bevor sie angelegt werden.

Mit den Informationen aus den Sperrobjekten werden zwei Funktionsbausteine generiert, mit denen Sperren gesetzt bzw. wieder freigegeben werden können. Näheres hierzu wird in Abschnitt 9.3 sowie Abschnitt 9.5 dargestellt.

Kern des SAP Sperrmechanismus bildet der so genannte Enqueue-Workprozess (siehe auch Kapitel 8). In jedem R/3-System gibt es genau einen Workprozess, der diese Rolle einnimmt, egal aus wievielen Servern das R/3-System besteht. Alleine diesem Workprozess obliegt die Verwaltung der Sperrtabelle, in welcher die aktiven Sperren abgelegt werden. Die durch die Sperrobjekte generierten Funktionsbausteine greifen über den Enqueue-Prozess auf die Sperrtabellen zu und erstellen oder löschen Einträge darin. Wird versucht eine Sperre auf ein Anwendungsobjekt zu setzen, auf dem bereits eine Sperre definiert ist, bricht der Baustein zum Setzen der Sperre mit einer Ausnahme ab. Das Prüfen, ob ein Anwendungsobjekt bereits gesperrt ist, erfolgt ebenfalls über den Weg, zu versuchen, eine Sperre zu setzten.

R/3 unterscheidet verschiedene Sperrmodi. Sperren können hierbei entweder als Lesesperre oder als Schreibsperre realisiert werden. Bei Schreibsperren wird wiederum zwischen normalen Schreibsperren und erweiterten Schreibsperren unterschieden.

Eine Lesesperre bedeutet hierbei, dass ein Programm ein Anwendungsobjekt dahingehend kennzeichnet, dass es darauf lesend zugreift. Andere Programme können gleichzeitig ebenfalls Lesesperren auf das gleiche Objekt setzten. Versucht ein Programm jedoch eine Schreibsperre auf das Objekt zu setzten, scheitert dies, da durch einen schreibenden Zugriff die von anderen Programmen angezeigten Daten verändert würden.

Andersherum kann keine Lesesperre gesetzt werden, wenn ein Objekt bereits mit einer Schreibsperre versehen ist.

Im Gegensatz zur erweiterten Schreibsperre kann ein Programm bei der normalen Schreibsperre ein Anwendungsobjekt mehrfach mit einer Sperre versehen. Wird das Objekt innerhalb eines Programms mehrfach gesperrt, wird die tatsächliche Sperre erst aufgehoben, wenn die Anzahl der Freigaben gleich der Anzahl der vorherigen Sperren ist. Sperraufrufe können also innerhalb eines Programms kumulliert werden. Wird eine Tabelle mit einer erweiterten Sperre ausgestattet, ist auch innerhalb eines Programms keine weitere Sperrung des Anwendungsobjekts möglich.

Beim Anlegen von Sperrobjekten im Data Dictionary kann sowohl für die Primärtabelle als auch jede Sekundärtabelle gesondert der Sperrmodus definiert werden.

9.3 Sperren von Anwendungsobjekten

9.3.1 Finden der benötigten Sperrobjekte

Die Programmierung mit Sperren beruht auf dem Setzen und Entfernen der notwendigen Sperreinträge durch den Enqueue-Prozess. Hierfür muss der Programmierer wissen, welche Sperrobjekte für welches Anwendungsobjekt zur Verfügung stehen und muss diese verwenden. Die Prüfung von existierenden Sperreinträgen erfolgt immer auf der Basis der Sperrobjekte. Wenn zwei Sperrobjekte für eine Tabelle definiert werden, müssen daher immer beide Sperren gesetzt werden, um eine vollständige Sperrung zu erreichen. Besser wäre es natürlich, wenn lediglich ein Sperrobjekt für das jeweilige Anwendungsobjekt verantwortlich wäre.

Nur stellt sich die Frage, wie der Programmierer herausbekommt, welches Sperrobjekt in welchem Fall zu verwenden ist.

Solange man sich im Bereich selbstdefinierter Anwendungsobjekte aufhält, ist die Antwort sehr einfach. Es müssen die Sperrobjekte, die selbst definiert worden sind, verwendet werden. Wie diese Sperrobjekte erzeugt werden, kann aus Abschnitt 9.4 dieses Kapitels ersehen werden. Anders verhält es sich, wenn im Umfeld von SAP Anwendungsobjekten programmiert werden soll. Hier muss bekannt sein, welche Standard-Sperrobjekte im jeweiligen Fall Anwendung finden.

Dies kann herausgefunden werden, indem in eine bekannte Standard-Transaktion gesprungen wird, die das Anwendungsobjekt auf die gewünschte Weise sperrt. In einem weiteren Modus wird in der Transaktion SM12 die Liste der gesetzten Sperreinträge angezeigt.

Abbildung 9.1 zeigt beispielhaft die Sperren, die gesetzt werden, wenn ein PM-Auftrag mit der Transaktion IW32 (»Auftrag ändern«) bearbeitet wird. Aus der Abbildung ist ersichtlich, dass die Standard-Transaktion während der Verarbeitung drei Sperren setzt, um konkurrierende Zugriffe auf Anwendungsobjekte zu verhindern. Neben dem Auftrag selbst (AUFK) werden die IH-Meldung, auf deren Basis der Auftrag eröffnet wurde (QMEL) sowie die zum Auftrag gehörenden Reservierungen (RKPF) mit einer Sperre versehen. Aus der Abbildung geht nur hervor, welche primären Tabellen gesperrt wurden, nicht jedoch, welche Sperrobjekte hierfür verwendet wurden.

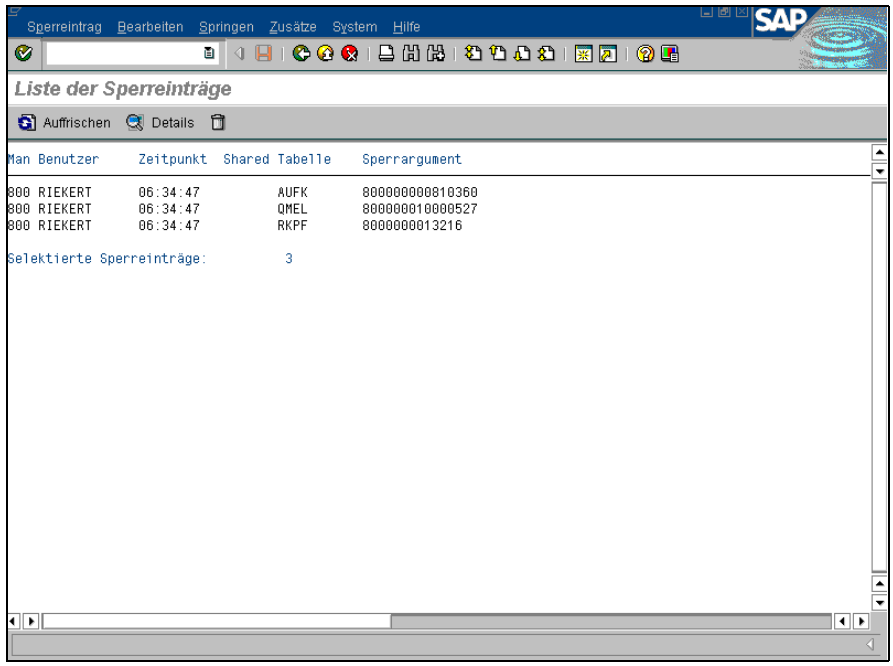

Abbildung 9.1
Sperrobjekte der Transaktion IW32 (© SAP AG)

Weitere Informationen zu den gesetzten Sperren können aus der Detailsicht der Sperren ermittelt werden. In diese gelangt man durch einen Doppelclick auf die entsprechende Zeile der Liste der Sperreinträge (Abbildung 9.2).

Aus dem erscheinenden Fenster können alle benötigten Informationen bezüglich der gesetzten Sperre ermittelt werden. Für den Programmierer besonders wichtig sind hierbei der Name des Sperrobjekts (im Beispiel `ESORDER` für den Auftrag) im unteren Bereich des Fensters sowie das Sperrargument unten im ersten Gruppenrahmen.

Über den Kumulations-Zähler kann hier gesehen werden, ob das Anwendungsobjekt mehrfach von der Anwendung gesperrt wurde. Im Beispiel ist dies nicht der Fall.

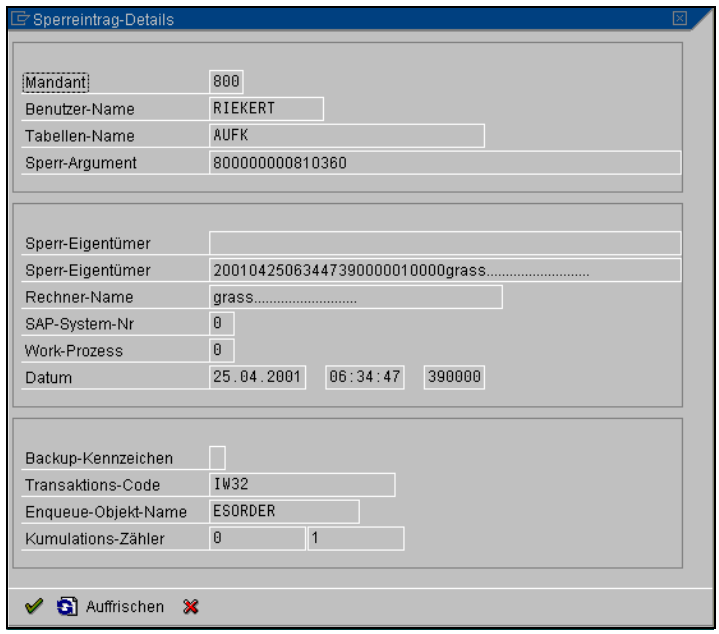

Abbildung 9.2
Detailansicht einer Sperre (© SAP AG)

Weiterhin ist aus der Abbildung ersichtlich, dass eine Sperre bis zu zwei Eigentümer haben kann. Es handelt sich hierbei um das Dialogprogramm, welches die Sperre angefordert hat sowie ggf. um den Verbucherbaustein, der die Sperre ebenfalls verwendet.

Um in einem eigenen Programm zu verhindern, dass ein anderer Benutzer das gleiche Anwendungsobjekt (hier Auftrag) parallel bearbeitet, müssen daher die gleichen Sperren gesetzt werden, wie im Standard. Es müssen jedoch nicht unbedingt alle Sperren gesetzt werden. Um eine Auftragsbearbeitung in der Trans-

aktion IW32 zu verhindern, genügt es, das Sperrobjekt ESORDER zu setzen. Allerdings könnte ein Benutzer dennoch die, dem Auftrag zugrunde liegende Meldung verändern und damit indirekt Einfluss auf den Auftrag nehmen.

9.3.2 Programmieren von Sperren

Betrachtet man sich das so gefundene Sperrobjekt im Data Dictionary (Transaktion SE11), kann man sehen, dass das Sperrobjekt ESORDER zwei Sperrparameter (MANDT und AUFNR) definiert. Die Sperrparameter sind beide aus der einzigen Tabelle des Sperrobjekts, der Tabelle AUFK, entnommen und entsprechen den gleichnamigen Feldern dort.

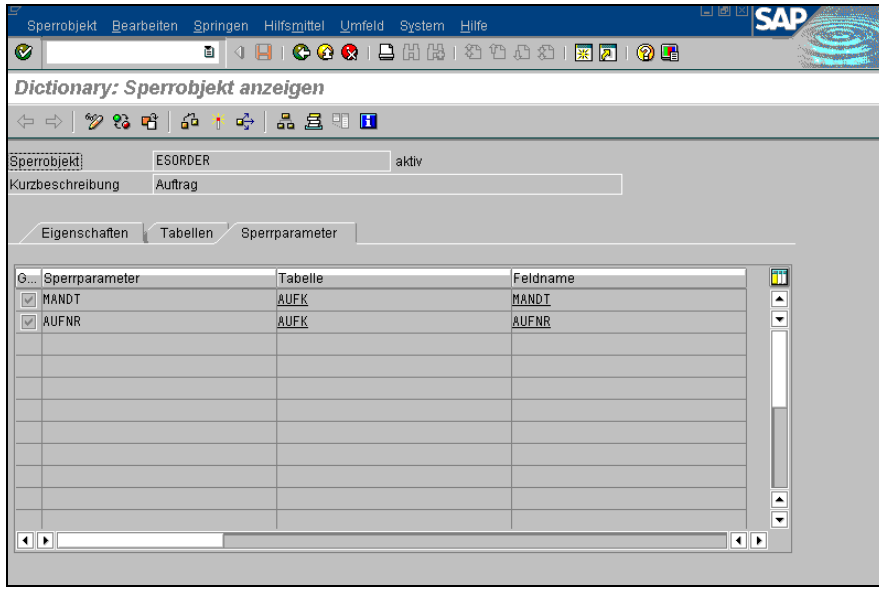

Abbildung 9.3
Sperrparameter für Sperrobjekt ESORDER (© SAP AG)

Zu jedem Sperrobjekt werden vom System zwei Funktionsbausteine zum Sperren bzw. Entsperren des Anwendungsobjekts generiert. Die Namen dieser Funktionsbausteine werden nach dem Muster

```
ENQUEUE_<Sperrobjektname>
```

für den Funktionsbaustein zum Setzen einer Sperre und entsprechend

```
DEQUEUE_<Sperrobjektname>
```

für den Funktionsbaustein zum Aufheben der Sperre gebildet.

Parameter von Enqueue-Bausteinen

Der Baustein zum Sperren von Aufträgen hat im Beispiel demnach den Namen ENQUEUE_ESORDER mit der im Folgenden dargestellten Schnittstelle.

```
FUNCTION ENQUEUE_ESORDER
   IMPORT
      MODE_AUFK
      MANDT
      AUFNR
      X_AUFNR
      _SCOPE
      _WAIT
      _COLLECT
   EXCEPTIONS
      FOREIGN_LOCK
      SYSTEM_FAILURE
```

Die Parameter der Aufrufschnittstelle des Enqueue-Bausteins kann hierbei in drei Gruppen unterschieden werden. Zunächst enthält die Schnittstelle für jede Tabelle, die im Sperrobjekt verwendet wird, einen Parameter, mit dem je Tabelle der Sperrmodus festgelegt werden kann. Im Beispiel verwendet das Sperrobjekt lediglich die Tabelle AUFK, weshalb hier lediglich der Parameter MODE_AUFK zu dieser Gruppe gezählt werden kann. Beim Anlegen des Sperrobjekts im Data Dictionary kann für den Sperrmodus jeder Tabelle eine Vorgabe gemacht werden. Diese wird beim Genenerieren des Funktionsbausteins als Default-Wert verwendet, kann aber bei jedem Enqueue-Aufruf separat gesetzt werden. Die möglichen Werte für den Sperrmodus sind in Tabelle 9.1 dargestellt.

Sperrmodus	Bedeutung
»S«	Lesesperre
»E«	Schreibsperre
»X«	erweiterte Schreibsperre

Tabelle 9.1
Sperrmodi

Die genaue Bedeutung der jeweiligen Werte kann Abschnitt 9.2 dieses Kapitels entnommen werden.

Den Sperrmodi folgen in der Aufrufschnittstelle des Enqueue-Bausteins die Sperrparameter des Sperrobjekts. Für jeden Parameter – außer dem Feld Mandant – existiert hier neben dem im Data Dictionary definierten Namen des Sperrarguments ein weiterer Parameter mit dem Namen X_<Sperrparameter>. Über dieses Feld wird – falls der Sperrparameter selbst initial übergeben wird – fest-

gelegt, ob der initiale Sperrparameter bedeutet, dass alle Werte gesperrt werden sollen (Space) oder ob tatsächlich nur die Objekte gesperrt werden sollen, die im dazugehören Feld keinen Wert enthalten (»X«).

Im Beispiel des Auftrags existiert auch nur ein Sperrparameter (AUFK), der dementsprechend mit den Schnittstellenparametern AUFK sowie X_AUFK versorgt wird. Diese Parameter definieren die zweite Gruppe innerhalb der Aufrufschnittstelle von Enqueue-Bausteinen. Die dritte Gruppe, die Verwaltungsfelder, sind für alle Sperrbausteine identisch und bestehen aus den drei Parametern _SCOPE, _WAIT und _COLLECT.

Mit dem Parameter _SCOPE kann der Gültigkeitsbereich der Sperre definiert werden. Hier sind die in Tabelle 9.2 dargestellten Werte gültig. Wird der Parameter nicht versorgt, wird als Default immer der Wert »2« angenommen.

Scope	Bedeutung
1	Sperre ist nur für den Dialogprozess gültig
2	Sperre wird an den Verbucher weitergegeben und dort aufgehoben
3	Sperre wird auch an den Verbucher weitergegeben und sowohl im Dialog als auch im Verbucher separat aufgehoben

Tabelle 9.2
Gültigkeitsparameter für Sperren.

Wenn das Setzen einer Sperre durch den Funktionsbaustein nicht gelingt, bricht der Baustein mit einer Exception (FOREIGN_LOCK) ab. Dies geschieht normalerweise sofort, wenn festgestellt wird, dass die Sperre nicht gesetzt werden kann. Durch den Parameter _WAIT kann - durch Übergabe des Wertes »X« - gesteuert werden, dass bei bereits vorhandener Sperre durch einen anderen Anwender, die Verarbeitung nicht sofort abgebrochen wird, sondern vielmehr mehrfach hintereinander der Versuch erfolgt, die Sperre zu setzen in der Hoffnung, dass die bereits existierende Sperre in diesem Zeitraum aufgehoben wird.

Über den Parameter _COLLECT schließlich kann gesteuert werden, ob die Sperranforderung sofort an den Enqueue-Server weitergeleitet werden soll (»X«) oder gebündelt. Näheres zu diesem Konzept ist in Abschnitt 9.4 zu finden.

Schnittstelle von Bausteinen zum Aufheben von Sperren

Die Schnittstelle von Bausteinen zum Aufheben von Sperren (Dequeue-Bausteine) unterscheidet sich nur minimal von der Aufrufschnittstelle von Enqueue-Bausteinen. Lediglich im Bereich der Verwaltungsparameter existieren hier Unterschiede.

```
FUNCTION DEQUEUE_ESORDER
   IMPORT
      MODE_AUFK
      MANDT
      AUFNR
      X_AUFNR
      _SCOPE
      _SYNCHRON
      _COLLECT
```

Der Parameter _WAIT des Enqueue-Bausteins entfällt im Zusammenhang des Entsperrens. Neu hinzugekommen ist jedoch der Parameter _SYNCHRON, über den gesteuert wird, ob der Dequeue-Baustein erst endet, wenn der Sperreintrag tatsächlich aus der Sperrtabelle gelöscht wurde (»X«) oder ob dieser Löschvorgang asynchron erfolgt. Im letzteren Fall kann es vorkommen, dass eine Sperre noch gefunden wird, obwohl bereits der Dequeue durchgeführt wurde. Läuft der Enqueue-Prozess auf dem lokalen Applikationsserver, wird dies in der Regel keinen Unterschied machen. SAP geht davon aus, dass der lokale Zugriff auf den Enqueue-Prozess innerhalb von weniger als einer Millisekunde erfolgt. Ein wenig anders kann dies aussehen, wenn der Enqueue-Prozess auf einem anderen Server läuft. Hier können die Zugriffszeiten zwischen 10 und ca. 80 Millisekunden liegen.

Der Dequeue-Baustein liefert kein Ergebnis zurück. Der Aufrufer hat also keine Chance, zu erfahren, ob eine Sperre existiert hat oder nicht. Weiterhin werden ausschließlich Sperren entfernt, die vollständig auf die übergebenen Parameter zutreffen. Dies betrifft sowohl die Sperrparameter, als auch die Sperrmodi. Wurde eine Sperre zu einem anderen Sperrmodus gesetzt, wird diese durch den Dequeue-Aufruf nicht aufgehoben.

9.4 Bündeln von Sperranfragen

In jedem R/3-System existiert auf genau einem Applikationsserver genau ein Workprozess, der als Enqueue-Prozess fungiert. Durch diese Architektur haben ausschließlich die Workprozesse auf diesem Server lokalen Zugriff auf den Sperrserver. Alle auf anderen Servern laufenden Prozesse müssen remote auf die Funktionalitäten des Sperrservers zugreifen. Diese Kommunikation kostet zum Teil erheblich Ressourcen. SAP geht davon aus, dass ein lokaler Aufruf des Sperrservers weniger als 1 Millisekunde in Anspruch nimmt, ein entfernter Aufruf jedoch zwischen 20 und 80 Millisekunden benötigt.

Setzt ein Programm mehrere Sperren unmittelbar hintereinander, wie die im Beispiel gezeigte Transaktion IW32, so besteht die Möglichkeit, die Sperr- (oder auch Entsperr-) Aufrufe zunächst lokal zwischenzuspeichern und in einem Schritt an den Sperrserver zu übertragen.

Dieses Vorgehen spart Ressourcen in erheblichen Umfang und verringert die Last auf dem Netzwerk.

Zur Realisierung diese Konzeptes stellen sowohl die Enqueue- als auch die Dequeue-Bausteine den Aufruf-Parameter _COLLECT zur Verfügung. Wird in diesem der Wert »X« übergeben (Default ist Space), so wird die Sperranforderung nicht sofort an den Enqueue-Server weitergegeben sondern zunächst lokal gepuffert. Dieser lokale Puffer wird auch »Lokaler Sperrcontainer« genannt.

Auf diese Weise können beliebig viele Enqueue- und Dequeue-Aufrufe gesammelt werden und schließlich durch einen Aufruf des Funktionsbausteins FLUSH_ENQUEUE an den Enqueue-Server übertragen werden.

```
FUNCTION FLUSH_ENQUEUE
   EXCEPTIONS
      FOREIGN_LOCK
      SYSTEM_FAILURE
```

Der Funktionsbaustein erhält keine weiteren Aufrufparameter, definiert aber die gleichen Ausnahmen, wie die Enqueue-Bausteine selbst.

Scheitert das Setzen einer, im Bündel enthaltenen Enqueue-Anweisung, wird keine der gebündelten Sperren gesetzt. Hierdurch kann eine Art Enqueue-LUW realisiert werden, ohne dass es diesen Begriff im SAP-Sprachgebrauch gibt.

Gebündelte Sperraufrufe können vor der Übertragung an den Sperrserver verworfen werden, indem der parameterlose Funktionsbaustein RESET_ENQUEUE aufgerufen wird. Analog zu den Dequeue-Bausteinen liefert dieser Funktionsbaustein keine Rückgabeparameter und auch keine Exceptions.

9.5 Anlegen von eigenen Sperrobjekten

Gelegentlich ist es notwendig, eigene Sperrobjekte zu definieren und zu verwenden. Hierbei muss jedoch immer zuvor geprüft werden, ob für den gewünschten Zweck nicht bereits Sperrobjekte im System vorhanden sind. Da es sich beim Sperrkonzept in R/3 um ein kooperatives Verfahren handelt, d.h. jedes beteiligte Programm ist angehalten sich kooperativ im Zusammenspiel mit anderen Programmen zu verhalten, macht es keinen Sinn für eigene Zwecke Sperrobjekte zu definieren, wenn nicht gewährleistet ist, dass andere Programme, die auf die gleichen Anwendungsobjekte zugreifen, dieselben Sperren verwendet.

Hieraus folgt, dass eigene Sperrobjekte ausschließlich für eigene Anwendungsobjekte in Frage kommen.

Bei Sperrobjekten handelt es sich um Objekte des Data Dictionaries. Demzufolge erfolgt die Anlage neuer Sperrobjekte immer in der Transaktion SE11.

Neue Sperrobjekte müssen zunächst mit einem Namen versehen werden. Dieser muss immer mit dem Buchstaben »E« beginnen. Anschließend folgt der Buchstabe »X«, »Y« oder »J«, um das Sperrobjekt im Kundennamensraum anzulegen.

Daran schließt sich der sprechende Teil des Namens für das Sperrobjekt an. Insgesamt kann der Name von Sperrobjekten bis zu 16 Stellen lang sein.

Nachdem im Eingangsbild der Transaktion SE11 der Name des Sperrobjekts definiert ist, kann das Sperrobjekt angelegt werden. Die Pflegetransaktion für Sperrobjekte besteht aus einem Dynpro mit einem aus drei Seiten bestehenden Tab-Strip. Auf der ersten Seite kann lediglich festgelegt werden, ob das Sperrobjekt auch über Remote Function Calls (RFC) ansprechbar sein soll. Desweiteren werden auf dieser Seite die Verwaltungsinformationen zum Sperrobjekt angezeigt. Auf eine Darstellung wurde daher an dieser Stelle verzichtet.

Auf der zweiten Seite des Tab-Strips mit dem Titel »Tabellen« können die Tabellen, die zum Sperrobjekt gehören, definiert werden (siehe Abbildung 9.4). Jedem Sperrobjekt liegt mindestens eine Datenbank-Tabelle zugrunde, die als Primärtabelle bezeichnet wird. Hierbei handelt es sich um die führende Tabelle, in der das Anwendungsobjekt abgelegt wird. Gegebenenfalls können weitere Tabellen als Sekundärtabellen dem Sperrobjekt zugewiesen werden.

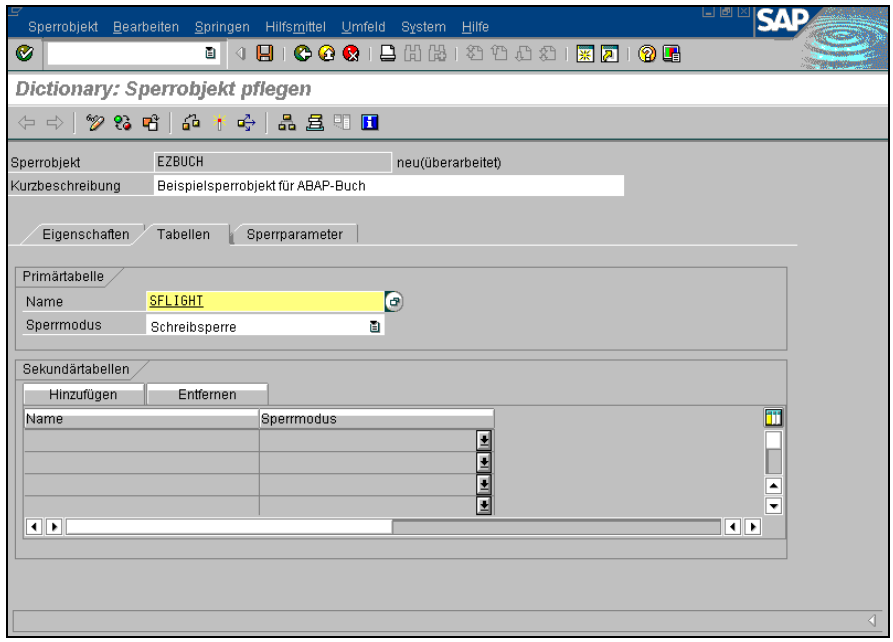

Abbildung 9.4
Sperrtabellen definieren (© SAP AG)

Für jede Tabelle einzeln kann der Vorschlagswert für den Sperrmodus definiert werden. Der hier festgelegte Wert wird als Default-Wert für die entsprechenden MODE_<Tabelle>-Parameter des Enqueue-Bausteins verwendet, kann beim Aufruf jedoch beliebig vom Programmierer überschrieben werden.

Zuletzt müssen noch auf der dritten Seite des Tab-Strips die Parameter für das Sperrobjekt definiert werden (Abbildung 9.5). Das System schlägt hier die Schlüsselfelder der verwendeten Tabellen vor. Der Name der Sperrparameter kann an dieser Stelle noch geändert werden. Er findet sich anschließend in der Aufrufschnittstelle des Enqueue- und des Dequeue-Bausteins wieder.

Über das Ankreuzfeld kann gesteuert werden, ob für den jeweiligen Sperrparameter ein Feld in der Aufrufschnittstelle der Funktionsbausteine angelegt werden soll.

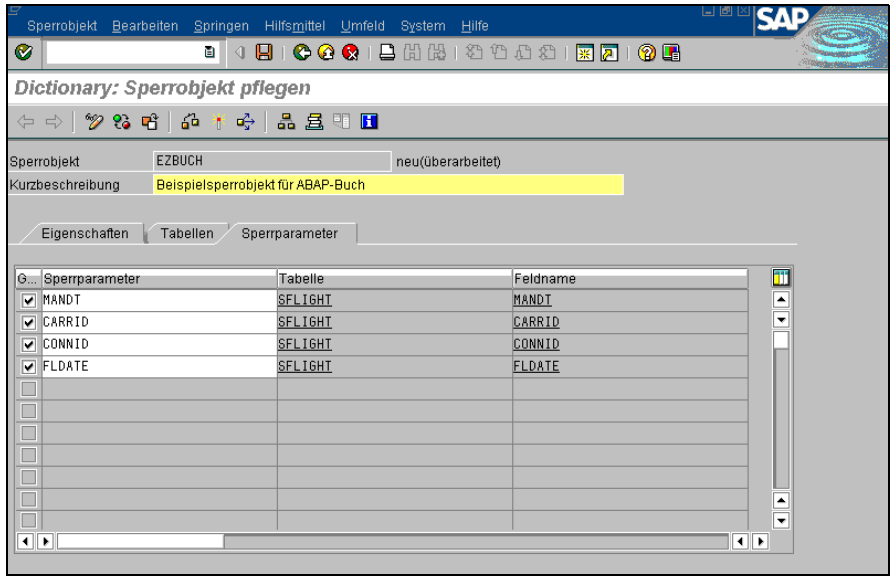

Abbildung 9.5
Definition der Sperrparameter (© SAP AG)

Beim Generieren des Sperrobjekts werden die Funktionsbausteine zum Sperren bzw. Aufheben der Sperren generiert. Die Funktionsbausteine werden dabei einer vom System bestimmten Funktionsgruppe zugeordnet. Sie dürfen weder manuell verändert, noch dürfen sie in eine andere Funktionsgruppe verschoben werden.

Der Transport der Funktionsbausteine darf ebenfalls nicht manuell erfolgen. Mit dem Transport und der Aktivierung des Sperrobjekts als Dictionary-Objekt werden im Zielsystem die dazugehörigen Funktionsbausteine angelegt. Die Zuordnung zur Funktionsgruppe kann hierbei eine andere sein, als im ursprünglichen System.

9.6 Verwendete Funktionsbauteine

9.6.1 DEQUEUE_<Sperrobjekt>

Mit den Dequeue-Bausteinen können zuvor gesetzte Sperren entfernt werden. Dequeue-Bausteine werden nicht vom Programmierer angelegt, sondern aus den Informationen des Sperrobjekts im Data Dictionary generiert. Manuelle Eingriffe dürfen hier nicht erfolgen. Dazu gehört auch das Umhängen des Bausteins in eine andere als die generierte Funktionsgruppe oder der Transport des Funktionsbausteins. Der Funktionsbaustein im Zielsystem wird beim Transport des Sperrobjekts im Zielsystem automatisch generiert.

Import-Parameter	
MODE_<Sperrtabelle>	Sperrmodus der beteiligten Tabellen (je Tabelle) »S« Lesesperre »E« Schreibsperre »X« erweiterte Schreibsperre
MANDT	Mandant
<Sperrparameter>	Sperrargument (je Sperrparameter)
X_<Sperrparameter>	Kennzeichen, ob Initialwert alle Werte beinhaltet oder exakt den Initialwert
_SCOPE	Gültigkeit der Sperre »1« Sperre gilt nur im Dialogprozess »2« Sperre wird an den Verbucher weitergereicht und dort Freigegeben »3« Sperre wird an den Verbucher weitergereicht, muss aber im Dialogprogramm und im Verbucher freigegeben werden
_SYNCHRON	Kennzeichen, ob Funktionsbaustein erst zurückkehren soll, wenn Sperreinträge gelöscht wurden (»X«) oder ob Löschen asynchron erfolgen soll (Space)
_COLLECT	Kennzeichen, ob die Sperrfreigabe gebündelt zum Enqueue-Server gesendet werden soll

9.6.2 ENQUEUE_<Sperrobjekt>

Generierte Funktionsbausteine zum Setzen von Sperren auf Anwendungsobjekte.

Mit dem Funktionsbaustein können sowohl ein als auch mehrere Anwendungsobjekte gesperrt werden. Da die Sperren nicht mit Datenbank-Sperren gelöst werden, können auch nicht existierende Sätze gesperrt werden.

Analog zu den Dequeue-Bausteinen dürfen die Enqueue-Bausteine nicht manuell vom Programmierer verändert werden oder in ein anderes System transportiert werden. Die Funktionsbausteine werden beim Transport der Sperrobjekte im Zielsystem automatisch generiert.

Import-Parameter	
MODE_<Sperrtabelle>	Sperrmodus der beteiligten Tabellen (je Tabelle) »S« Lesesperre »E« Schreibsperre »X« erweiterte Schreibsperre
MANDT	Mandant
<Sperrparameter>	Sperrargument (je Sperrparameter)
X_<Sperrparameter>	Kennzeichen, ob Initialwert alle Werte beinhaltet oder exakt den Initialwert
_SCOPE	Gültigkeit der Sperre »1« Sperre gilt nur im Dialogprozess »2« Sperre wird an den Verbucher weitergericht und dort Freigegeben »3« Sperre wird an den Verbucher weitergereicht, muss aber im Dialogprogramm und im Verbucher freigegeben werden
_WAIT	Falls die Sperre nicht gesetzt werden konnte wird gewartet und später erneut versucht, die Sperre zu setzen (»X«) oder sofortiger Abbruch (Space)
_COLLECT	Kennzeichen, ob die Sperrfreigabe gebündelt zum Enqueue-Server gesendet werden soll
Exceptions	
FOREIGN_LOCK	Sperre konnte aufgrund einer anderen Sperre nicht gesetzt werden
SYSTEM_FAILURE	Allgemeiner Systemfehler

9.6.3 FLUSH_ENQUEUE

Bei gebündelten Sperraufrufen wird durch den Aufruf dieses Funktionsbausteins die Übertragung der Aufrufe zum Sperrserver und somit das tatsächliche Setzen der Sperren ausgelöst.

Der Funktionsbaustein hat keine Aufrufparameter, definiert aber die gleichen Ausnahmen wie die Enqueue-Bausteine, nachdem erst beim tatsächlichen Setzen der Sperren Konflikte auftreten können.

Exceptions	
FOREIGN_LOCK	Sperre konnte aufgrund einer anderen Sperre nicht gesetzt werden
SYSTEM_FAILURE	Allgemeiner Systemfehler

9.6.4 RESET_ENQUEUE

Mit dem Funktionsbaustein `RESET_ENQUEUE` werden gebündelte und noch nicht zum Sperrserver übertragene Sperraufrufe verworfen. Dies betrifft sowohl Enqueue- als auch Dequeue-Aufrufe.

Der Funktionsbaustein definiert weder Aufruf- noch Rückgabeparameter. Es werden auch keine Exceptions ausgelöst.

Versenden von Mails

10

10.1 SAPOffice – Das Ablagesystem in R/3

Bestandteil jedes R/3-Basissystems ist das Ablagesystem SAPOffice. Dieses besteht zum einen aus dem Ablagesystem, bei dem Textobjekte (hier SAPOffice-Dokumente genannt) in Form von Ordnern organisiert abgelegt werden können. Zum anderen können diese Textobjekte über das integrierte Mailsystem an andere Benutzer versandt werden. Hierbei kann es sich sowohl um SAP-Benutzer innerhalb des gleichen oder anderer Systeme handeln. Auch der Versand an Empfänger im Internet ist hier möglich. Welche Empfänger durch das jeweilige SAPOffice erreicht werden können, hängt von der Konfiguration des R/3-Systems ab.

Ebenfalls von der Konfiguration des Systems hängt auch ab, ob Textobjekte als FAX-Mitteilung versendet werden können.

Das Ablage- und Nachrichtensystem SAPOffice bildet auch die Kernfunktionalität des SAP Workflow-Moduls. Nachrichten, die im Rahmen eines Workflows ausgetauscht werden, landen in einem speziellen Ordner des SAPOffice und können hier vom Benutzer weiterverarbeitet werden. Die Übertragung der Workitems findet hierbei ebenfalls über das integrierte Mailsystem statt.

Abbildung 10.1 zeigt die Ordnerübersicht von SAPOffice für einen Standard-Benutzer im R/3-Release 4.6C. In der Baumdarstellung auf der linken Seite sieht man die Ordner des Ablagesystems, auf die der Benutzer zugreifen kann. Diese Ordner sind auf der einen Seite private, d.h. nur für den jeweiligen Benutzer zugängliche, Ordner, wie z.B. der Ordner »Eingang« mit allen untergeordneten Ordnern. Hier befinden sich auch die Ordner des Workflows. Weiterhin zeigt die Abbildung eine Reihe von »öffentlichen« Ordnern. Dabei handelt es sich um Ordner, die mehreren Benutzern zugänglich sind. Die verschiedenen Benutzer sehen hierbei nicht Kopien der Informationen sondern den tatsächlichen Ordner selbst. In der Abbildung handelt es sich beispielsweise beim Ordner »Allgemei-

ne Ablage« um einen solchen öffentlichen Ordner. Je nach Berechtigung kann hier jeder Benutzer nach Belieben Dokumente lesen und anlegen.

Eine andere Unterteilung der dargestellten Ordner könnte die Unterscheidung in Ordner, die den Mailtransfer des Benutzers regeln und reine Ablageordner sein. Ausschließlich für den Nachrichtenaustausch sind die beiden Ordner »Eingang« und »Ausgang« bestimmt. Sie enthalten entsprechend die eingegangenen Nachrichten bzw. die für den Versand vorgesehenen Objekte.

Im rechten Teil der Abbildung ist im oberen Teil der Inhalt des in der Hierarchie ausgewählten Ordners (in diesem Fall »Dokumente«) dargestellt. Darunter sieht der Benutzer den Inhalt des oben ausgewählten Nachrichtenobjektes als Kurzdarstellung.

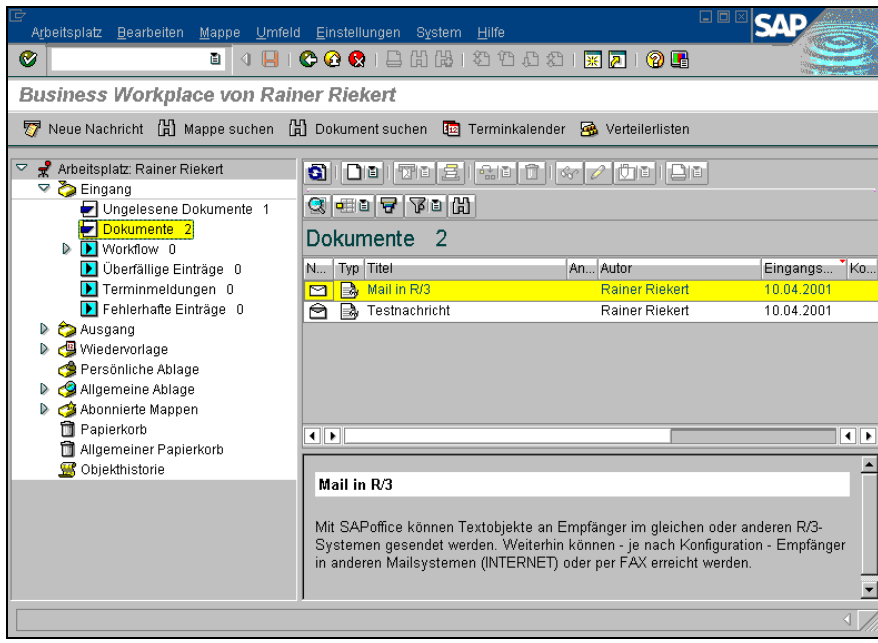

Abbildung 10.1
Ordnerübersicht im SAPOffice (© SAP AG)

Soweit zum SAPOffice. Dieses Kapitel soll zeigen, wie aus ABAP-Programmen heraus Nachrichten erzeugt und an einen oder mehrere Empfänger versendet werden können. Dies ist insbesondere bei der Programmierung von Hintergrundprogrammen und Jobs, aber auch bei Schnittstellenprogrammen, die von außerhalb des R/3-Systems aufgerufen werden und daher Verarbeitungsfehler ebenfalls nicht direkt an einen Benutzer melden können, sinnvoll.

Der Mailversand erfolgt in ABAP-Programmen durch einen Aufruf des Funktionsbausteins SO_OBJECT_SEND, der in der Funktionsgruppe SOA2 (»SAPOffice: Funktionen – Senden«) enthalten ist. Dieser wird im Folgenden zunächst in seiner Grundform vorgestellt und sukzessive in den folgenden Abschnitten erweitert. Die Möglichkeiten dieses Funktionsbausteins sind jedoch so vielfältig, dass eine vollständige Darstellung aller Möglichkeiten im Rahmen dieses Buches nicht möglich ist. Es werden jedoch die wesentlichen Funktionen zur Erzeugung und zum Versand von Nachrichten gezeigt.

10.2 Senden einer einfachen Textmitteilung

Das Versenden von Nachrichten über SAPOffice wird über den Funktionsbaustein SO_OBJECT_SEND, dessen Aufrufschnittstelle unten dargestellt ist, realisiert. Aus der Vielzahl der Parameter wird deutlich, dass es sich hierbei um einen sehr leistungsfähigen Baustein handelt, der sehr viele Variationsmöglichkeiten erlaubt. Im ersten Schritt soll der Funktionsbaustein lediglich genutzt werden, um eine einfache Textmitteilung an einen einzelnen Benutzer innerhalb des gleichen R/3-Systems zu versenden. Im Verlauf dieses Kapitels werden weitere Funktionalitäten aufgezeigt. Eine vollständige Darstellung aller Möglichkeiten ist im Rahmen dieses Buches jedoch nicht möglich.

```
FUNCTION SO_OBJECT_SEND
  IMPORTING
    FOLDER_ID
    FORWARDER
    OBJECT_FL_CHANGE
    OBJECT_HD_CHANGE
    OBJECT_ID
    OBJECT_TYPE
    OUTBOX_FLAG
    OWNER
    STORE_FLAG
    DELETE_FLAG
    SENDER
    CHECK_SEND_AUTHORITY
    CHECK_ALREADY_SENT
    GIVE_OBJECT_BACK
    ORIGINATOR
    ORIGINATOR_TYPE
    LINK_FOLDER_ID
  EXPORTING
    OBJECT_ID_NEW
    SENT_TO_ALL
    ALL_BINDING_DONE
    OFFICE_OBJECT_KEY
```

```
    ORIGINATOR_ID
  TABLES
    OBJCONT
    OBJHEAD
    OBJPARA
    OBJPARB
    RECEIVERS
    PACKING_LIST
    ATT_CONT
    ATT_HEAD
    NOTE_TEXT
    LINK_LIST
    APPLICATION_OBJECT
  EXCEPTIONS
    ACTIVE_USER_NOT_EXIST
    COMMUNICATION_FAILURE
    COMPONENT_NOT_AVAILABLE
    FOLDER_NOT_EXIST
    FOLDER_NO_AUTHORIZATION
    FORWARDER_NOT_EXIST
    OBJECT_NOT_EXIST
    OBJECT_NOT_SENT
    OBJECT_NO_AUTHORIZATION
    OBJECT_TYPE_NOT_EXIST
    OPERATION_NO_AUTHORIZATION
    OWNER_NOT_EXIST
    PARAMETER_ERROR
    SUBSTITUTE_NOT_ACTIVE
    SUBSTITUTE_NOT_DEFINED
    SYSTEM_FAILURE
    TOO_MUCH_RECEIVERS
    USER_NOT_EXIST
    ORIGINATOR_NOT_EXIST
    X_ERROR
    OTHERS
```

In diesem Abschnitt soll begonnen werden, eine einfache Textnachricht via SAP-Office an einen anderen Benutzer im R/3-System zu versenden. Von den vielen Parametern, die der Funktionsbaustein anbietet, werden hierfür lediglich die Parameter `OBJECT_HD_CHANGE`, `OBJECT_TYPE`, `OBJECT_CONT` sowie `RECEIVERS` benötigt.

Kopfstruktur OBJECT_HD_CHANGE

Im Parameter `OBJECT_HD_CHANGE` wird dem Funktionsbaustein die Kopfstruktur für das zu sendende SAPOffice-Dokument übergeben. Die zugrunde liegende Struktur ist im Data Dictionary unter dem Namen `SOOD1` definiert (Tabelle 10.1).

Feldname	Datentyp	Bedeutung
OBJLA	CHAR 1	Sprachkennzeichen des Dokuments
OBJSRT	CHAR 10	Feld nachdem Dokumente sortiert werden
OBJNAM	CHAR 12	Name des Dokuments
OBJDES	CHAR 50	Kurzbeschreibung des Dokuments
ARCHI	CHAR 1	Kennzeichen, ob das Dokument im Archiv abgelegt ist
DLDAT	DATS 8	Verfallsdatum des Dokuments
DLTIM	TIMS 6	Verfallsuhrzeit des Dokuments
OBJPRI	CHAR 1	Priorität des Dokuments
OBJSNS	CHAR 1	Sensititivtät des Dokuments
ACOUT	CHAR 1	Dokument ist außerhalb von SAPOffice zu erledigen
VMTYP	CHAR 1	Auführungstyp des Dokuments
SKIPS	CHAR 1	Überspringen des ersten Dynpros beim Ausführen
ACNAM	CHAR 54	Name des Auszuführenden Programms (Report, Funktionsbautein, Dialogbaustein oder Transaktion)
ACMEM	CHAR 32	Memory ID für Ausführung
OBJCP	CHAR 1	Kennzeichen, ob das Dokument verändert werden kann
FIXED	CHAR 1	Check-In/-Out-Flag
EXTHD	CHAR 1	Kennzeichen, ob objektspezifischer Kopf extern abgelegt ist
EXTCT	CHAR 1	Kennzeichen, ob Objektinhalt extern abgelegt ist
PCTRL	CHAR 1	Kennzeichen, ob das Dokument in einem fremden Eingangskorb gelöscht werden kann
FILE_EXT	CHAR 3	Dateierweiterung für PC-Dokumente

Feldname	Datentyp	Bedeutung
OBJLEN	CHAR 12	Länge des Dokumentinhalts
CREATOR	CHAR 4	Ersteller bei Dokumenten von MacIntosh-Rechnern
OWNNAM	CHAR 12	Name des Besitzer des Dokuments
PRTCT	CHAR 1	Kennzeichen, ob der Titel des Dokuments verändert werden darf

Tabelle 10.1
Kopfstruktur für SAPOffice-Dokumente (SOODl).

Die Struktur enthält die gesamten Kopfinformationen für Dokumente im Bereich SAPOffice. Beim Senden neuer Dokumente, werden jedoch nur wenige der Felder dieser Struktur benötigt.

Das einzige Feld, welches hier versorgt werden muss, ist das erste Feld OBJLA, in dem das Sprachkennzeichen der Sprache übergeben wird, in dem die Nachricht verfasst ist. Hier muss die Sprache in der einstelligen Form, d.h. »D« für Deutsch und nicht die neue zweistellige Schreibweise verwendet werden.

Alle übrigen Felder der Struktur sind zunächst optional. Sicherlich wird es jedoch Sinn machen, dem Dokument einen Titel zu geben, mit dem er im Eingangskorb des Empfängers aufgelistet wird. Dieser wird gegebenenfalls im Feld OBJDES abgelegt. Der interne Name des Dokuments wird im Feld OBJNAM definiert.

Auf die übrigen Felder der Struktur wird zum Teil in den folgenden Abschnitten noch näher eingegangen.

Typ des Dokuments (OBJ_TYPE)

Dokumente in SAPOffice können verschiedenartig sein. Je nachdem, wie der Inhalt des Dokuments strukturiert ist, werden unterschiedliche Editoren und Bearbeitungsmöglichkeiten angeboten.

Der Typ des jeweiligen Dokuments wird als dreistelliges Feld angegeben. Welche Dokumenttypen zur Verfügung stehen, wird im Rahmen des Customizing von SAPOffice definiert. Zusätzlich zu den Dokumenttypen, die im R/3 definiert werden, kann jede Dateierweiterung, die lokal auf dem PC des Benutzers definiert ist (soweit es sich um einen Windows-PC handelt), verwendet werden.

Für unstrukturierte Textdokumente innerhalb von R/3 – solche sollen hier ausschließlich betrachtet werden - wird hier der Typ »RAW« verwendet.

Dokumentinhalt (OBJECT_CONT)

Der eigentliche Inhalt des zu versendenden Dokuments wird in Form einer internen Tabelle im Parameter `OBJECT_CONT` an den Funktionsbaustein übergeben.

Feldname	Datentyp	Bedeutung
LINE	CHAR 255	Textzeile

Tabelle 10.2
Aufbau der Struktur `SOLI`

Diese Tabelle hat den Aufbau der Dictionary-Struktur `SOLI`, welche in Tabelle 10.2 wiedergegeben ist. Die Struktur besteht nur aus einem einzigen Feld `LINE`.

Empfänger (RECEIVERS)

Nachdem das zu versendende Dokument vollständig beschrieben ist, fehlen nur noch die Informationen, an wen die Mail geschickt werden soll. Die Empfänger der Mail werden dem Funktionsbaustein ebenfalls in Form einer internen Tabelle übergeben, welche den Aufbau der Data Dictionary-Struktur `SOOS1` hat. Diese Struktur enthält insgesamt 98 Felder. Eine vollständige Darstellung würde den Rahmen an dieser Stelle sprengen. Tabelle 10.3 beschränkt sich daher darauf die, für diesen Schritt benötigten Felder aufzulisten.

Feldname	Datentyp	Bedeutung
...		
DLINAM	CHAR 12	Name der Verteilerliste
RECNAM	CHAR 12	Name des Mailempfängers
SNDBC	CHAR 1	Senden als geheime Kopie (BCC)
SNDCP	CHAR 1	Senden als Kopie (CC)
SNDEX	CHAR 1	Kennzeichen, ob Mail als Express-Mail versendet werden soll
SNDPRI	CHAR 1	Priorität des Dokuments (1-9)
DELIVER	CHAR 1	Empfangsbestätigung angefordert
NOT_DELI	CHAR 1	Nicht-Empfang zurückmelden
READ	CHAR 1	Lesebestätigung angefordert
...		

Tabelle 10.3
Aufbau der Tabelle für Mailempfänger (Auszug)

Eine Mail kann entweder an einen einzelnen Benutzer (in diesem Schritt werden nur Benutzer innerhalb des gleichen R/3-Systems adressiert) oder an eine zuvor in Form einer internen Verteilerliste definierten Empfängerkreis gesendet werden. Für diese beiden Varianten stehen in der Struktur entsprechend die Felder DLINAM für eine Verteilerliste und RECNAM für einen einzelnen Mail-Empfänger zur Verfügung. Es darf jeweils nur eines dieser beiden Felder versorgt werden.

Für jeden Empfänger gesondert, können hier Eigenschaften für die Mail definiert werden. Über das Feld SNDCP wird der Empfänger als Kopieempfänger der Mail gekennzeichnet. Die Empfänger der Mail sehen am Mailkopf, wer ebenfalls Mailempfänger und Kopienempfänger ist. Über das Feld SNDBC kann ein Empfänger als geheimer Kopieempfänger klassifiziert werden. Die übrigen Empfänger können geheime Mailempfänger nicht erkennen.

Beispielsweise beim Abbruch der Verbuchung erhält der Anwender eine vom System generierte Mail, die den Verbuchungsabbruch mitteilt. Diese Mail wird als so genannte Express-Mail im System versandt. Beim Anwender erscheint ein PopUp, welches ihn auf den Eingang einer Express-Mail hinweist. Mit dem Funktionsbaustein SO_OBJECT_SEND können Express-Mails versandt werden, indem im Feld SNDEX der RECEIVERS-Tabelle ein »X« übergeben wird. Zusätzlich kann die Wichtigkeit der Mail in Form einer Priorität im Parameter SNDPRI definiert werden. Hier sind Werte zwischen 1 (hoch) und 9 (niedrig) definiert worden.

Zur Mailverfolgung können in den Feldern DELIVER, NOT_DELI und READ Zustellung bzw. nicht –Zustellung sowie eine Lesebestätigung angefordert werden.

Beispiel

In diesem Abschnitt soll ein vollständiges Beispielprogramm dargestellt werden, mit dem eine einfache Mail als Express-Mail versendet wird. Das Programm ist als Report ohne weitere Struktur realisiert.

```
REPORT ZMAIL.

DATA: G_OBJECT_HD_CHANGE   LIKE SOOD1,
      G_OBJECT_TYPE        LIKE SOOD-OBJTP.

DATA: G_OBJCONT            LIKE SOLI  OCCURS 0 WITH HEADER LINE,
      G_RECEIVERS          LIKE SOOS1 OCCURS 0 WITH HEADER LINE.

DATA: G_OBJECT_ID_NEW      LIKE SOODK,
      G_SENT_TO_ALL        LIKE SONV-FLAG.

* Versorgen des Headers
  G_OBJECT_HD_CHANGE-OBJLA  = SY-LANGU.
  G_OBJECT_HD_CHANGE-OBJDES = 'Testmail'(001).
```

```
* Objekttyp
  G_OBJECT_TYPE             = 'RAW'.

* Nachricht
  G_OBJCONT-LINE            = 'Zeile 1'.
  APPEND G_OBJCONT.
  G_OBJCONT-LINE            = 'Zeile 2'.
  APPEND G_OBJCONT.

* Empfänger
  G_RECEIVERS-RECNAM        = 'RIEKERT'.
  G_RECEIVERS-SNDEX         = 'X'.       "Als Express-Mail
  APPEND G_RECEIVERS.

  CALL FUNCTION 'SO_OBJECT_SEND'
    EXPORTING
      OBJECT_HD_CHANGE                = G_OBJECT_HD_CHANGE
      OBJECT_TYPE                     = G_OBJECT_TYPE
    IMPORTING
      OBJECT_ID_NEW                   = G_OBJECT_ID_NEW
      SENT_TO_ALL                     = G_SEND_TO_ALL
    TABLES
      OBJCONT                         = G_OBJCONT
      RECEIVERS                       = G_RECEIVERS
    EXCEPTIONS
      ACTIVE_USER_NOT_EXIST           = 1
      COMMUNICATION_FAILURE           = 2
      COMPONENT_NOT_AVAILABLE         = 3
      FOLDER_NOT_EXIST                = 4
      FOLDER_NO_AUTHORIZATION         = 5
      FORWARDER_NOT_EXIST             = 6
      NOTE_NOT_EXIST                  = 7
      OBJECT_NOT_EXIST                = 8
      OBJECT_NOT_SENT                 = 9
      OBJECT_NO_AUTHORIZATION         = 10
      OBJECT_TYPE_NOT_EXIST           = 11
      OPERATION_NO_AUTHORIZATION      = 12
      OWNER_NOT_EXIST                 = 13
      PARAMETER_ERROR                 = 14
      SUBSTITUTE_NOT_ACTIVE           = 15
      SUBSTITUTE_NOT_DEFINED          = 16
      SYSTEM_FAILURE                  = 17
      TOO_MUCH_RECEIVERS              = 18
      USER_NOT_EXIST                  = 19
      ORIGINATOR_NOT_EXIST            = 20
      X_ERROR                         = 21
      OTHERS                          = 22.
```

Das Programm zeigt, wie einfach im Grunde der Versand einer Mail in SAPOffice ist. Wie so oft verdeckt die Vielzahl der zur Verfügung stehenden Parameter die eigentlich einfache Funktionalität.

Das Beispielprogramm versorgt lediglich die notwendigen Felder der Kopfstruktur und der Empfängertabelle. Weiterhin werden exemplarisch zwei literale Textzeilen als Rumpf der Mail definiert.

Auf keinen Fall darf die Festlegung des Mailtypen (in der Regel »RAW«) unterbleiben. Der Funktionsbaustein würde sonst mit der Ausnahme PARAMETER_ERROR abbrechen.

Der Funktionsbaustein liefert in diesem Beispiel eine Struktur mit den Primärschlüsselfeldern des neu erzeugten SAPOffice-Dokuments sowie ein Flag, ob die Mail an alle Empfänger erfolgreich versendet werden konnte, zurück. Weitere Rückgabeparameter existieren, sollen hier aber nicht näher erläutert werden.

Der Primärschlüssel für SAPOffice-Dokumente ist in der Dictionary-Struktur SOODK (Tabelle 10.4) definiert und besteht aus drei Feldern.

Feldname	Datentyp	Bedeutung
OBJTP	CHAR 3	Objekttyp
OBJYR	CHAR 2	Jahr
OBJNO	CHAR 12	Objektnummer

Tabelle 10.4
Primärschlüssel für SAPOffice-Dokumente (Struktur SOODK)

10.3 Mails mit Ausführungsparametern

Mails in SAPOffice können mit so genannten Ausführungsparametern versehen werden. Darunter versteht man einen Mechanimus, der es erlaubt, direkt aus einer Mail heraus ein Programmmodul aufzurufen. Hierbei kann es sich um einen Report, einen Funktions- oder Dialogbaustein oder eine Dialogtransaktion handeln.

In der Ansicht der Mail gibt es im Menü »Dokument« die Option »Ausführen«, welche in diesem Fall aktiv ist. Je nach hinterlegter Funktion wird vom System das entsprechende Programmmodul aufgerufen und ggf. die entsprechenden Parameter versorgt.

Die Festlegung, welche Art von Programm aufgerufen wird, erfolgt im Feld VMTYP der im Parameter OBJECT_HD_CHANGE an den Funktionsbaustein übergebenen Kopfstruktur. Die hier möglichen Werte sind in Tabelle 10.5 dargestellt.

VMTYP	Bedeutung
»D«	Dialogbaustein
»F«	Funktionsbaustein
»R«	Report
»S«	Report mit Parameterübergabe im Memory
»T«	Dialogtransaktion
»U«	Dialogtransaktion mit Parameterübergabe im Memory

Tabelle 10.5
Mögliche Typen von Ausführungsparametern

Je nach Ausführungstyp haben weitere Felder der Kopfstruktur sowie der Inhalt der Tabellenparameter OBJPARA sowie OBJPARB unterschiedliche Bedeutungen. Im Folgenden soll kurz auf die einzelnen Varianten zur Steuerung der Ausführungsparameter eingegangen werden, bevor das Beispiel des vorigen Abschnitts exemplarisch erweitert wird, um den Aufruf einer Dialogtransaktion als Ausführungsparameter zu zeigen.

Zur Parameterübergabe an das auszuführende Programm (gleich welchen Typs) definiert die Aufrufschnittstelle von SO_OBJECT_SEND zwei Tabellenparameter OBJPARA und OBJPARB. Während die in OBJPARA definierten Einträge in Form von Benutzerparametern abgelegt werden, erfolgt die Weitergabe der internen Tabelle des Parameters OBJPARB direkt an das entsprechende Programm.

Der Tabellenparameter des Parameters OBJPARA entspricht der Dictionary-Struktur SELC, deren Aufbau in Tabelle 10.6 wiedergegeben ist.

Feldname	Datentyp	Bedeutung
NAME	CHAR 8	Name der Parameter-ID
OPTION	CHAR 4	hier keine Bedeutung
LOW	CHAR 40	Wert für den Benutzerparameter
HIGH	CHAR 40	ggf. Name der Parameter-ID

Tabelle 10.6
Aufbau der Data Dictionary-Struktur SELC

Der Aufbau der Tabelle entspricht somit dem Aufbau einer Selektionstabelle wie sie z.B. mit der RANGES-Anweisung von ABAP erzeugt werden kann. Das

Feld OPTION hat in hier beschriebenen Zusammenhang jedoch keinerlei Bedeutung und sollte daher leer gelassen werden.

Im Feld NAME wird der Name der Parameter-ID, die gesetzt werden soll, übergeben. Wird dieses Feld leer gelassen, wird stattdessen der im Feld HIGH übergebene Wert als Name der Parameter-ID interpretiert. Im Gegensatz zum Feld NAME, wo lediglich die ersten drei Stellen des übergebenen Wertes relevant sind, werden im Feld HIGH die ersten 20 Stellen verwendet.

Das Feld LOW schließlich enthält den eigentlichen Wert, der in den jeweiligen Benutzerparameter geschrieben werden soll. Durch einen speziellen Mechanismus ist es hier möglich, dynamisch Informationen des SAPOffice-Dokuments an das aufzurufende Programm zu übergeben. Im Einzelnen werden die in Tabelle 10.7 dargestellten Ersetzungen vor dem Setzen der jeweiligen Parameter-ID durchgeführt.

Wert	wird ersetzt durch	Bedeutung
&&FOLDER_ID&&	FOLDER_ID	ID der Mappe, in dem das Dokument abgelegt wurde
&&FORWARDER&&	FORWARDER	Name des weiterleitenden Benutzers
&&OBJNAM&&	OBJNAM	Objektname
&&OBJDES&&	OBJDES	Beschreibung (Titel) des Dokuments
&&CRONAM&&	CRONAM	Ersteller des Dokuments
&&CRDAT&&	CRDAT	Erstellungsdatum
&&CRTIM&&	CRTIM	Erstellungsuhrzeit
&&CHONAM&&	CHONAM	Letzter Änderer
&&CHDAT&&	CHDAT	Datum der letzten Änderung
&&CHTIM&&	CHTIM	Uhrzeit der letzten Änderung
&&LA_DAT&&	LA_DAT	Datum des letzten Zugriffs
&&DL_DAT&&	DL_DAT#	Verfallsdatum des Dokuments
&&OBJECT_ID&&	OBJECT_ID	Objekt-ID
&&OWNER&&	OWNER	Besitzer des Dokuments

Tabelle 10.7
Variable, die zur Laufzeit durch Dokumentinformationen ersetzt werden

Die Sätze der im Tabellenparameter OBJPARB übergebenen Tabelle werden unverändert an den aufzurufenden Dialog- bzw. Funktionsbaustein übergeben. Bei Aufruf von Reports oder Dialogtransaktionen wird dieser Tabellenparameter bei Bedarf über das ABAP-Memory an das aufzurufende Programm weitergegeben.

Der Tabellenparameter OBJPARB ist ebenfalls mit Bezug zu einer Dictionary-Struktur definiert. Es handelt sich hierbei um die Struktur SOOP1, deren Aufbau Tabelle 10.8 zeigt.

Feldname	Datentyp	Bedeutung
NAME	CHAR 30	Name des Paramters
VALUE	CHAR 255	Wert des Parameters

Tabelle 10.8
Dictionary-Struktur SOOP1

Ausführungstyp »Dialogbaustein«

Ausführungsparameter vom Typ »D« ermöglichen es, direkt aus der Mail heraus einen Dialogbaustein im R/3-System aufzurufen. Hierbei werden zunächst die im Tabellenparameter OBJPARA definierten Benutzerparameter gesetzt, bevor der eigentliche Dialogbaustein aufgerufen wird.

Der aufzurufende Dialogbaustein muss eine festgelegte Aufrufschnittstelle definieren. Diese besteht lediglich aus einem Tabellenparameter MSGDIAL, in dem die im Parameter OBJPARB an den Funktionsbaustein SO_OBJECT_SEND übergebene interne Tabelle übergeben wird.

Der Name des Dialogbausteins wird im Parameter ACNAM der Kopfstruktur übergeben. Vor dem Aufruf prüft das System die Existenz des Bausteins und bricht gegebenenfalls mit einer Meldung ab.

Ausführungstyp »Funktionsbaustein«

Mit dem Ausführungstyp »F« wird definiert, dass der im Feld ACNAM der Kopfstruktur übergebene Wert der Name eines aufzurufenden Funktionsbausteins ist. Analog zum Ausführungstyp »Dialogbaustein« werden zunächst die im Parameter OBJPARA definierten Benutzerparameter gesetzt, bevor der Funktionsbaustein mit der in OBJPARB übergebenen Tabelle als Parameter aufgerufen wird.

Die Aufrufschnittstelle des Funktionsbausteins muss daher folgenden Aufbau besitzen:

```
FUNCTION <Funktionsname>
  TABLES
    MSG_DIAL.
```

Ausführungstyp »Report«

Wird im Feld VMTYP der Kopfstruktur der Wert »R« oder »S« übergeben, wird der Wert ACNAM als Name eines ABAP-Reports interpretiert. Auch in diesem Fall werden die Sätze des Tabellenparameters OBJPARA in Benutzerparameter umgesetzt. Da Reports anders als Funktionsbausteine keine Aufrufschnittstelle definieren, besteht die Möglichkeit, die im Parameter OBJPARB übergebene Tabelle über das ABAP-Memory an den aufzurufenden Report weiterzugeben. Dies erfolgt jedoch nur beim Ausführungstyp »S« und auch nur dann, wenn im Feld ACMEM der Kopfstruktur ein Name für das ABAP-Memory übergeben wurde.

Anschließend wird der gewünschte Report über die SUBMIT-Anweisung des ABAP-Sprachschatzes gestartet. Über das Feld SKIPS der Kopfstruktur kann hierbei gesteuert werden, ob der Selektionsbildschirm des Reports übersprungen werden soll. Analog zu anderen Aufrufen mit SUBMIT funktioniert dies natürlich nur, wenn über die entsprechenden Mechanismen alle notwendigen Felder des Selektions-Dynpros versorgt wurden.

Ausführungstyp »Dialogtransaktion«

Analog zum Aufruf von Reports werden die Ausführungstypen »T« und »U« genutzt, um Dialogtransaktionen entweder ohne oder mit Nutzung des ABAP-Memories aufzurufen.

Von der Funktionalität her gleichen sich diese beiden Ausführungstypen mit dem Unterschied, dass der Inhalt des Feldes ACNAM nicht den Programmnamen, sondern den Transaktionscode der zu rufenden Dialogtransaktion enthält. Diese wird nach dem Versorgen der entsprechenden Benutzerparameter und ggf. des ABAP-Memories mit der CALL TRANSACTION-Anweisung von ABAP aufgerufen.

Auch hier besteht die Möglichkeit, das erste Dynpro der Transaktion zu überspringen. Dies wird über das Feld SKIPS der Kopfstruktur gesteuert.

Beispiel

Zur Demonstration der Möglichkeiten, eine Mail mit Ausführungsparametern zu versehen, soll das Beispielprogramm aus dem vorigen Abschnitt dahingehend erweitert werden, dass über die versendete Mail die Transaktion »Auftrag ändern« (Transaktionscode IW32) aufgerufen werden kann. Der Transaktion soll eine feste Auftragsnummer übergeben werden und der Eingangsbildschirm übersprungen werden.

Hierfür müssen zunächst weitere Felder der Kopfstruktur versorgt werden.

```
* Versorgen des Headers
  ...
  G_OBJECT_HD_CHANGE-VMTYP  = 'F'.
```

```
  G_OBJECT_HD_CHANGE-ACNAM  = 'IW32'.
  G_OBJECT_HD_CHANGE-SKIPS  = 'X'.
  ...
```

Der Ausführungstyp wird hier auf »F« gesetzt, da außer der Auftragsnummer, welche über einen Benutzerparameter gesetzt werden kann, keine Parameter übergeben werden müssen.

Der Transaktionscode IW32 wird im Feld ACNAM abgelegt. Über den Wert »X« im Feld SKIPS wird gesteuert, dass der erste Screen der Transaktion übersprungen werden soll.

Zuletzt muss noch der Benutzerparameter ANR, welcher zur Aufnahme einer Auftragsnummer definiert wurde und von der Transaktion IW32 verwendet wird, gesetzt werden. Um dies zu bewerkstelligen, muss eine weitere Variable als interne Tabelle deklariert werden.

```
REPORT ZMAIL.
  ...
  DATA: G_OBJPARA  LIKE SELC  OCCURS 0 WITH HEADER LINE.
  ...
```

In diese Tabelle wird vor dem Aufruf des Funktionsbausteins SO_OBJECT_SEND die Zeile, die das Setzen des Benutzerparameters ANR bewirkt, eingefügt.

```
* Benutzerparameter setzten
  G_OBJPARA-NAME           = 'ANR'.
  G_OBJPARA-LOW            = '810360'.
  APPEND G_OBJPARA.

  CALL FUNCTION 'SO_OBJECT_SEND'
    EXPORTING
      OBJECT_HD_CHANGE               = G_OBJECT_HD_CHANGE
      OBJECT_TYPE                    = G_OBJECT_TYPE
    IMPORTING
      OBJECT_ID_NEW                  = G_OBJECT_ID_NEW
      SENT_TO_ALL                    = G_SEND_TO_ALL
    TABLES
      OBJCONT                        = G_OBJCONT
      OBJPARA                        = G_OBJPARA
      RECEIVERS                      = G_RECEIVERS
    EXCEPTIONS
      ACTIVE_USER_NOT_EXIST          = 1
      COMMUNICATION_FAILURE          = 2
      COMPONENT_NOT_AVAILABLE        = 3
      FOLDER_NOT_EXIST               = 4
      FOLDER_NO_AUTHORIZATION        = 5
      FORWARDER_NOT_EXIST            = 6
      NOTE_NOT_EXIST                 = 7
```

```
OBJECT_NOT_EXIST                = 8
OBJECT_NOT_SENT                 = 9
OBJECT_NO_AUTHORIZATION         = 10
OBJECT_TYPE_NOT_EXIST           = 11
OPERATION_NO_AUTHORIZATION      = 12
OWNER_NOT_EXIST                 = 13
PARAMETER_ERROR                 = 14
SUBSTITUTE_NOT_ACTIVE           = 15
SUBSTITUTE_NOT_DEFINED          = 16
SYSTEM_FAILURE                  = 17
TOO_MUCH_RECEIVERS              = 18
USER_NOT_EXIST                  = 19
ORIGINATOR_NOT_EXIST            = 20
X_ERROR                         = 21
OTHERS                          = 22.
```

Der Aufruf des Funktionsbausteins SO_OBJECT_SEND wurde um den Tabellenparameter OBJPARA erweitert, um den gewünschten Benutzerparameter bei der Ausführung der Mail zu setzten.

10.4 Verwendete Funktionsbausteine

10.4.1 SO_OBJECT_SEND

Der Funktionsbaustein SO_OBJECT_SEND wird verwendet, um Dokumente aus dem Bereich von SAPOffice über das integrierte Mailsystem an einen oder mehrere Benutzer im gleichen oder anderen R/3-Systemen oder fremden Mailsystemen zu senden.

Import-Parameter	
FOLDER_ID	ID der Mappe des Dokuments
FORWARDER	Name des weiterleitenden Benutzers
OBJECT_FL_CHANGE	Mappeninformation für neues Objekt
OBJECT_HD_CHANGE	Kopfstruktur des Dokuments
OBJECT_ID	ID des Dokuments
OBJECT_TYPE	Dokumenttyp
OUTBOX_FLAG	Kennzeichen, ob Dokument in Ausgangskorb gespeichert werden soll
OWNER	Names des Besitzers

STORE_FLAG	Kennzeichen, ob Dokument gespeichert werden soll
DELETE_FLAG	Kennzeichen, ob Dokument gelöscht werden soll
SENDER	Name des Senders
CHECK_SEND_AUTHORITY	Kennzeichen, ob Sendeberechtigung des Benutzers geprüft werden soll
CHECK_ALREADY_SENT	Prüfen, ob Dokument bereits versendet wurde
GIVE_OBJECT_BACK	Kennzeichen, dass gesendetes Objekt vom Funktionsbaustein zurückgeliefert werden soll
ORIGINATOR	Vertreter
ORIGINATOR_TYPE	Benutzertyp des Vertreters
LINK_FOLDER_ID	undokumentiert
Export-Paramter	
OBJECT_ID_NEW	Key des gesendeten Dokuments
SENT_TO_ALL	Kennzeichen, ob der Versand an alle definierten Empfänger fehlerfrei war
ALL_BINDINGS_DONE	undokumentiert
OFFICE_OBJECT_KEY	undokumentiert
ORIGINATOR_ID	undokumentiert
Tabellen	
OBJCONT	Mailrumpf
OBJHEAD	Kopfzeilen der Mail
OBJPARA	Benutzerparameter für ausführbare Mails
OBJPARB	Parameter bzw. Memory-Parameter für ausführbare Mails
RECEIVERS	Empfänger der Mail
PACKING_LIST	undokumentiert
ATT_CONT	undokumentiert
ATT_HEAD	undokumentiert

NOTE_TEXT	undokumentiert
LINK_LIST	undokumentiert
APPLICATION_OBJECT	undokumentiert
Exceptions	
ACTIVE_USER_NOT_EXIST	Aktiver Benutzer ist kein SAPOffice Benutzer
COMMUNICATION_FAILURE	Kommunikationsfehler
COMPONENT_NOT_AVAILABLE	Die SAPOffice Komponente ist nicht aktiv
FOLDER_NOT_EXIST	Die Mappe existiert nicht
FOLDER_NO_AUTHORIZATION	Keine Zugriffsberechtigung für die Mappe
FORWARDER_NOT_EXIST	Der Weiterleitende existiert nicht
NOTE_NOT_EXIST	Die angegebene Notiz existiert nicht
OBJECT_NOT_EXIST	Das Objekt existiert nicht
OBJECT_NOT_SENT	Das Objekt wurde nicht versendet
OBJECT_NO_AUTHORIZATION	Keine Zugriffsberechtigung für das Objekt
OBJECT_TYPE_NOT_EXIST	Der angegebene Objekttyp existiert nicht
OPERATION_NO_AUTHORIZATION	Keine Berechtigung für diese Operation
OWNER_NOT_EXIST	Der verantwortliche Benutzer existiert nicht
PARAMETER_ERROR	Falsche Eingabe der Daten
SUBSTITUTE_NOT_ACITVE	Vertreter nicht aktiv
SUBSTITUTE_NOT_DEFINED	Vertreter nicht definiert
SYSTEM_FAILURE	Systemfehler
TOO_MUCH_RECEIVERS	Zu viele Empfänger definiert
USER_NOT_EXIST	Der SAPOffice Benutzer existiert nicht
ORIGINATOR_NOT_EXIST	Der SAPOffice Benutzer (Vertreter) existiert nicht
X_ERROR	Interner Fehler

Literaturverzeichnis

Buck-Emden, Rüdiger: Die Technologie des SAP-Systems: Basis für betriebswirtschaftliche Anwendungen, 4., aktualisierte und erw. Aufl., Bonn u.a.: Addison-Wesley-Longman, 1998.

Gamma, Erich, et al.: Design Patterns. Elements of Reusable Object-Oriented Software, 1. Aufl., Reading (Massachusetts): Addison-Wesley Publishing Company, 1995.

Keller, Horst; Krüger, Sascha: ABAP Objects. Einführung in die SAP-Programmierung, 1. Aufl. Bonn: Gallileo Press 2000.

Matzke, Bernd: ABAP/4. Die Programmiersprache des SAP-Systems R/3, 3., erweiterte Aufl., München u.a.: Addison-Wesley-Longman 1999.

SAP AG: Controls Technology, Walldorf: SAP AG 1999

SAP AG: R/3-Sicherheitsleitfaden: BAND I, R/3-Sicherheitsservices im Überblick, Version 2.0a, Walldorf: SAP AG, 1999.

SAP AG: R/3-Sicherheitsleitfaden: BAND II, R/3-Sicherheitsservices im Detail, Version 2.0a, Walldorf, SAP AG, 1999.

SAP AG: SAP-Bibliothek, R/3 Online Hilfe zu Release 4.6C, Walldorf: SAP AG 2000.

Stichwortverzeichnis

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z