Eine generische und objektorientierte Schnittstelle von TYCOON zum System SAP R/3

Diplomarbeit

Universität Hamburg
Fachbereich Informatik

Jens Latza
Sprützmoor 71
22547 Hamburg
Mat.-Nr.: 4340356

Rüdiger Lühr
Neuland 75
21614 Buxtehude
Mat.-Nr.: 4340407

Betreuung:

Prof. Dr. Florian Matthes
Technische Universität Hamburg-Harburg
Arbeitsbereich Softwaresysteme

Prof. Dr. Arno Rolf
Universität Hamburg
Fachbereich Informatik

16. Februar 1999

Diese Arbeit ist im Rahmen einer Gruppenarbeit entstanden. Die inhaltliche Erarbeitung der Diplomarbeit erfolgte gemeinsam. Die schriftliche Niederlegung teilt sich wie folgt auf:

- Gemeinsam: Abschnitt 1.1, Abschnitt 1.2, Abschnitt 3.4, Abschnitt 4.5, Abschnitt 6.1, Abschnitt 6.2, Abschnitt 7.1, Abschnitt 7.2, Abschnitt 7.4, Kapitel 8,
- Jens Latza:
  Abschnitte: 2.5 bis 2.7, 3.1, 3.3, 4.1 bis 4.4, 4.6, 5.1, 7.3.2, 7.3.4, 7.3.5
- Rüdiger Lühr:
  Abschnitte: 2.1 bis 2.4, 3.2, 4.7, 4.8, 5.2, 5.3, 7.3.1, 7.3.3, 7.3.6, 7.3.7

Die praktische Umsetzung teilt sich wie folgt auf:

- Gemeinsam: Architektur, C-Library, Klassendesign
- Jens Latza: Quelltextvorlagen, Generator, Metadaten, Marshalling
- Rüdiger Lühr: Typabbildung, Parametrisierbare Tabellen, Geschäftsobjekte, Ausnahmebehandlung

Zusammenfassung

Die SAP ist mit ihrem Produkt R/3 einer der Marktführer innerhalb des Marktsegments von betriebswirtschaftlicher Standardsoftware. Schnittstellen von Fremdsystemen zum R/3 sind oftmals notwendig, um durch das R/3 nicht abgedeckte Funktionalitäten zu integrieren. Probleme bisheriger R/3-Integratioen sind das geringe Abstraktionsniveau der Schnittstellen, das zu aufwendigen und unflexiblen Lösungen führt. Ansätze auf der Basis von Geschäftsobjektmodellen versprechen eine Komplexitätsverminderung der Integration von Softwaresystemen unter Bereitstellung eines objektorientierten Zugriffs.
In dieser Arbeit wird untersucht, ob sich die R/3-Geschäftsobjekte und die BAPI-Technologie der SAP für eine objektorientierte Integration mit dem R/3 eignen. Dabei werden die Probleme verdeutlicht, die sich aus der sukzessiven Weiterentwicklung der R/3-Geschäftsobjekte und der nicht objektorientierten R/3-Implementierung durch die SAP ergeben. Gleichzeitig wird jedoch mit der Generierung von *Stubs* eine Lösung aufgezeigt, eine generische R/3-Schnittstelle zu konstruieren, welche die komplexen Details einer R/3-Integration verbirgt. Zur Implementierung der R/3-Integration wird das persistente TYCOON-2-System benutzt, daß durch seine strikte Orientierung an objektorientierten Programmierparadigmen und seine Ausdrucksmächtigkeit eine leistungsfähige Umgebung darstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Das System R/3 der Firma SAP stellt das marktführende Produkt innerhalb des Marktsegments von betriebswirtschaftlicher Standardsoftware dar. Die Software R/3 eignet sich insbesondere für global agierende Konzerne. Der Umsatz der Firma SAP belief sich im Jahr 1997 auf 6 Mrd. Mark. Aus dieser marktdominierenden Stellung des R/3 heraus ergibt sich oftmals die Notwendigkeit der Integration existierender Softwaresysteme mit der Funktionalität des R/3. Dabei ist besonders die Integration mit den Finanzbuchhaltungs-, den Controlling- und den Personalwirtschaftsfunktionen des R/3 wichtig, da diese zu den am häufigsten eingesetzten R/3-Funktionalitäten gehören [König, Buxmann 97].

Weitere Gründe für eine R/3-Integration stellt die Existenz bereits vorhandener Systeme dar, die auch nach einer R/3-Einführung nicht abgelöst werden sollen. Außerdem existieren Systeme, die von R/3 nicht abgedeckte Funktionalität bereitstellen, wie z.B. Dokumentenmanagement und Groupwarefunktionalität. Eine nicht zu unterschätzende Größe stellt auch die Entwicklung alternativer Benutzungsoberflächen dar, weil zum einen die Bedienung des R/3 sehr komplex ist und zum anderen die R/3-Funktionalität auch über das Internet verfügbar sein soll.

Probleme dieser Integration sind bisher das niedrige Abstraktionsniveau der von der SAP angebotenen Schnittstellen. Dieses niedrige Abstraktionsniveau verursachte einen hohen Aufwand bei der Implementierung von Schnittstellen zum R/3, da zum einen die R/3-Schnittstellen schwierig zu benutzen waren, und zum anderen keine stabile Basis zu dem sich in Releases weiterentwickelnden R/3 darstellten.

Mit dem Release 3 des R/3 führte die SAP das Konzept des *Business Frameworks* ein: Das Ziel des *Business Frameworks* ist es, das bis dato eher als monolithisch einzustufende R/3 in eine Menge von Komponenten zu zerlegen, in denen Geschäftsobjekte die betriebswirtschaftliche R/3-Funktionalität bereitstellen. Die nach objektorientierten Prinzipien modellierten Geschäftsobjekte stehen dabei auch externen Softwaresystemen zur Verfügung und bieten eine objektorientierte Sichtweise auf die R/3-Funktionalität. Dieser objektorientierte Ansatz ermöglicht die Interaktion zwischen objektorientierten Systemen und dem R/3, ohne daß diese Interaktion mit einem semantischen Bruch einhergeht. Objektorientierte Systeme können in der Folge auf einem hohen Abstraktionsniveau auf das R/3 zugreifen. Dies ermöglicht wiederum eine nahtlose Integration von Softwaresystemen, die immer notwendig sein wird, da

gerade eine integrierte Standardsoftware wie das R/3 nicht die Gesamtheit aller gestellten Anforderungen in zufriedenstellendem Maß abdecken kann.

1.1 Ziel der Arbeit

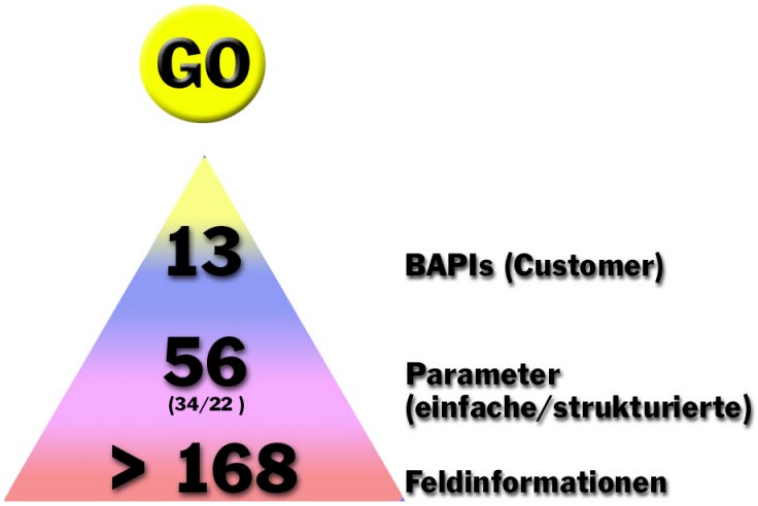

Abbildung 1.1: Parameterinformationen für die Benutzung eines Geschäftsobjekts

Ziel dieser Arbeit ist die Konstruktion einer objektorientierten Schnittstelle zum R/3. Als semantische Objekte seitens des R/3 werden die R/3-Geschäftsobjekte betrachtet, welche den Anspruch haben, die Funktionalität des R/3 in einem objektorientierten Modell abzubilden. Dabei soll auch auf das Problem der Generizität eingegangen werden, da die Funktionalität der R/3-Geschäftsobjekte von der SAP sukzessive erweitert wird. Eine generische Schnittstelle soll dabei so flexibel auf diese Erweiterungen reagieren können, daß keine grundlegenden Änderungen der Schnittstelle vorgenommen werden müssen, sondern diese den jeweiligen Entwicklungsstand im R/3 nach dem objektorientierten Paradigma zur Verfügung stellt.

Die Implementierung erfolgte innerhalb der objektorientierten Entwicklungsumgebung TYCOON-2. Die Wahl fiel auf TYCOON-2 auch unter dem Gesichtspunkt, daß innerhalb von TYCOON-1, dem nicht-objektorientierten Vorgänger von TYCOON-2, bereits eine Schnittstelle zum R/3 auf der Basis von R/3-Funktionsbausteinen entwickelt wurde.

Betrachtungsgegenstand dieser Arbeit sind - wie bereits erwähnt - die R/3-Geschäftsobjekte. Die R/3-Geschäftsobjekte kapseln die betriebswirtschaftliche Logik des R/3. Diese soll TYCOON-2-Anwendungen zur Verfügung gestellt werden. Konkret wird parallel zu dieser Arbeit am Arbeitsbereich Softwaresysteme der Technischen Universität Hamburg-Harburg an einer *E-Commerce-Lösung* gearbeitet, welche die durch diese Schnittstelle bereitgestellte Funktionalität nutzt.

Ziel dieser Arbeit und erhoffter Nebeneffekt durch die Orientierung an den R/3-Geschäftsobjekten ist weiterhin eine Erhöhung des Abstraktionsniveaus einer Schnittstelle zum R/3 und damit einhergehend eine Verminderung der Komplexität und eine leichtere Nutzbarkeit im

Vergleich zur TYCOON-1-Schnittstelle. Dies soll im wesentlichen durch die Programmierung eines Klassengenerators erreicht werden, der benötigte R/3-Geschäftsobjekte im TYCOON-2-System zur Verfügung stellt. R/3-Geschäftsobjekte, Funktionsbausteine und deren Parameter sollen bei dieser Abbildung zur Reduzierung von Fehlern als statisch typisierte TYCOON-2-Objekte dargestellt werden.

1.2 Aufbau der Arbeit

Der Entwurf einer objektorientierten Schnittstelle zum R/3 setzt detailliertes Wissen über das System R/3 voraus. Aus diesem Grund ist Kapitel 2 dem R/3-System gewidmet und beschreibt die Architektur, die semantischen Objekte im R/3 und die Möglichkeiten für einen externen R/3-Zugriff. Die Forderung nach einer R/3-Integrationslösung ensteht, wenn das R/3 Kundenanforderungen nicht oder nur ungenügend abdecken kann. Daher beschreibt diese Arbeit zum Abschluß des Kapitels 2 die Möglichkeiten, R/3 an kundenspezifische Anforderungen anzupassen. Dabei wird deutlich, daß R/3-Integrationslösungen notwendig sind, um kundenspezifische Anforderungen zu realisieren. Auf der anderen Seite sind diese Integrationen komplex und kostenintensiv, was durch Studien belegt wird.

Eine Reduzierung der Komplexität betriebswirtschaftlicher Anwendungen versprechen Ansätze auf der Basis von Geschäftsobjektmodellen. Es besteht die Hoffnung, daß sich diese Komplexitätsreduzierung auch die Komplexität der Schnittstellen zu externen Systemen mindert und so eine leichtere R/3-Integration erlaubt. Kapitel 3 stellt deshalb drei verschiedene Geschäftsobjektkonzepte vor und bewertet sie im Anschluß. Vorgestellt werden die Konzepte der Open Applications Group (OAG), der Open Management Group (OMG) und das IBM San Francisco *Framework* Die genannten Konzepte unterscheiden sich dabei hinsichtlich des gewählten Ansatzes als auch in der Nähe zur Implementierung.

Kapitel 4 ist dem Geschäftsobjektmodell im R/3 gewidmet. Dabei werden zunächst die R/3-Geschäftsobjekte im Rahmen des SAP-*Business Frameworks* vorgestellt. Daran anschließend wird auf die Implementierung der R/3-Geschäftsobjekte eingegangen. Hier spielen insbesondere die BAPIs (*Business Application Programming Interface*), die Methoden der Geschäftsobjekte, eine entscheidene Rolle. Mit dem R/3-Release 4 wurde die Programmiersprache des SAP-Systems um objektorientierte Eigenschaften erweitert. Diese Arbeit stellt OO-ABAP vor und untersucht, ob sich daraus eine vereinfachte Integration mit dem R/3 ergibt.

Kapitel 5 beschäftigt sich mit dem externen Zugriff auf die R/3-Geschäftsobjekte. Anhand von zwei kommerziellen Lösungen der Firma IBM und der SAP selbst werden existierende Zugriffsmöglichkeiten vorgestellt und ihre Vor- und Nachteile diskutiert. Die Ergebnisse dieser Untersuchung sind dabei in das Design der im Rahmen dieser Arbeit entwickelten Schnittstelle eingeflossen.

Im Kapitel 6 wird die Basisumgebung der entwickelten Schnittstelle vorgestellt: Die objektorientierte Entwicklungsumgebung TYCOON-2. Ausgehend von den Eigenschaften des TYCOON-2-Systems werden die semantischen Konstrukte der Programmiersprache des Systems TL-2 (*Tycoon Language*) vorgestellt. Insbesondere diese Konstrukte gestatten eine Umsetzung einer Schnittstelle zum R/3, welche in gängigen objektorientierten Programmiersprachen so nicht möglich wäre.

Kapitel 7 stellt die implementierte Schnittstelle zwischen dem TYCOON-2-System und dem R/3 vor. Hier wird zunächst die Schnittstellenarchitektur und daran anschließend die semantischen Objekte des R/3 beschrieben, die durch TL-2-Klassen im TYCOON-2-System repräsentiert werden. Dies sind vor allem die R/3-Geschäftsobjekte und die Funktionsbausteine mitsamt ihrer Parameter. Im zweiten Abschnitt dieses Kapitels wird auf die Implementierung im Detail eingegangen. Hierbei wird besonders auf die Unterschiede zwischen beiden Systemen in Fragen der Objektorientierung, der Persistenz und des Typkonzeptes eingegangen und die Schlußfolgerungen für die Erstellung einer Schnittstelle erklärt.

Kapitel 2

Architektur der Standardsoftware SAP R/3

Das System R/3 der Firma SAP stellt eines der komplexesten und wichtigsten betriebswirtschaftlichen Standardsoftwaresysteme weltweit dar. Die Software R/3 wird in „weit mehr als als 15000 Installationen weltweit“ [SAP 98b] eingesetzt.

Dieses Kapitel stellt nach einer allgemeinen Beschreibung des Systems R/3 die Architektur sowie die Entwicklungsumgebung vor. In der Entwicklungsumgebung des R/3 werden sowohl Programme als auch Funktionsbausteine definiert. Programme beinhalten die betriebswirtschaftliche Logik des R/3. Funktionsbausteine bilden das Konzept zur Wiederverwendung von Programmquelltext und können durch externe Anwendungen aufgerufen werden. Die RFC-Automation beschreibt den Zugriff auf Funktionsbausteine durch externe Systeme und stellt die wichtigste technologische Basis zur Integration mit dem R/3 dar.

Am Ende des Kapitels werden die Möglichkeiten der Anpaßbarkeit des R/3 an kundenspezifische Anforderungen beschrieben. Hierbei wird deutlich werden, daß R/3-Integrationslösungen vor allem dort benötigt werden, wo die R/3-Anpaßbarkeit auf ihre Grenzen stößt.

2.1 Die Standardsoftware R/3

Die Firma SAP[1] wurde 1972 von ehemaligen IBM-Mitarbeitern mit dem Ziel gegründet, integrierte betriebswirtschaftliche Standardsoftware zu entwickeln. Die für den damaligen Zeitpunkt revolutionäre Idee von Standardsoftware ermöglichte der SAP ein rasches Wachstum. Heute ist die SAP ein multinationales Unternehmen, dessen Umsatz im Jahr 1997 6 Milliarden Mark betrug. Die Firma bedient 7500 Kunden in über 85 Ländern. Unter den 500 umsatzstärksten Unternehmen weltweit arbeiten 90% mit dem SAP-System [Will et al. 96]

Die Standardsoftwareprodukte der Firma SAP sind das Mainframeprodukt R/2 und das *Client/Server*-System R/3, welches das Nachfolgeprodukt zu R/2 darstellt, seit 1991 auf dem Markt ist und sukzessive R/2 verdrängt.

[1] Das Kürzel SAP steht für Systeme, Anwendungen und Produkte.

R/3 läßt sich als Produkt für global agierende Firmen kennzeichnen, die mittels R/3 eine Integration und Vereinheitlichung ihrer DV-Infrastruktur erzielen möchten. Die Eigenschaften von R/3 lassen sich wie folgt umschreiben:

- Abbildung beliebiger Organisationsstrukturen
- Weite Anpaßbarkeit der im R/3 implementierten Geschäftsprozesse
- Anpaßbarkeit an nationales Steuerrecht und rechtliche Besonderheiten
- Mehrsprach- und Mehrwährungsfähigkeit
- Verteilungsfähigkeit und Portabilität

R/3 besteht aus Komponenten, welche die innerbetrieblichen Funktionsbereiche (z.B. Finanzwesen, Controlling, Vertrieb, Personalwirtschaft) abbilden. Die Komponenten sind innerhalb gewisser Grenzen unabhängig voneinander einsetzbar. Die Kernkomponenten des R/3 (u.a. Finanzwesen, Controlling) sind prinzipiell branchenneutral und zählen zu den am häufigsten eingesetzten Komponenten. Es existieren R/3-Module, die branchenspezifische Funktionalität in das R/3-System einbinden. Um diese Komponenten innerhalb eines Unternehmens einsetzen zu können, muß im Rahmen eines Softwareinführungsprojektes R/3 parametrisiert und an die unternehmensspezifischen Gegebenheiten angepaßt werden. Die Anwendungslogik des R/3 wird dabei durch circa 1000 vordefinierte Geschäftsprozesse bestimmt, aus denen die kundenrelevanten Teile ausgewählt und angepaßt werden. Diese Anpassung, das sogenannte *Customizing*, ist aufgrund der Komplexität des Systems nicht trivial und wird meistens durch spezialisierte Unternehmensberater vollzogen (siehe Abschnitt 2.7).

Vor dem Hintergrund eines ständig steigenden Konkurrenzdrucks und einer Globalisierung der Märkte existieren Bestrebungen, die Funktionalität des R/3 auch über das Internet verfügbar zu machen. Weiterhin wird durch die Bildung von *Business Komponenten* versucht, den monolithischen Block des Systems aufzubrechen, um einerseits externen Anwendungen eine Integration mit dem R/3 zu erlauben als auch andererseits die in sich sehr verwobenen und damit schwer wartbaren und einführbaren Komponenten des R/3 in leichter überschaubare und getrennt weiterzuentwickelnde Komponenten aufzulösen. Dieses Konzept wird von der SAP als *Business Framework* bezeichnet und wird in dieser Arbeit noch eingehender in Abschnitt 4.1 erläutert, da gerade dieses *Business Framework* eine objektorientierte Integration mit dem R/3 ermöglichen soll, welche das Ziel dieser Arbeit darstellt.

Um diese Möglichkeiten der Integration aufzuzeigen, wird jedoch zunächst der technische Hintergrund des R/3 eingehender erläutert. Dieser technische Hintergrund umfaßt die Architektur des R/3 als *Client/Server*-System, den essentiellen Begriff der SAP-Transaktion sowie die Integration des R/3 in andere Systeme.

2.2 Die *Client/Server*-Architektur des R/3

R/3 ist ein *Client/Server*-System, d.h. das System läßt sich gemäß dem *Client/Server*-Prinzip in die Komponenten *Client*, *Server* und einer zwischen den beiden Komponenten vermittelnden Kommunikationskomponente aufteilen. Zur genaueren Beschreibung der *Client/Server*-

Architektur sowie ihrer verschiedenen Formen sei hier auf die zahlreiche Literatur wie z.B. [Pour 97], [Dadam 96], [Niemann 95], [Sinha 92] verwiesen.

Das R/3 läßt sich prinzipiell allen Varianten der *Client/Server*-Architektur anpassen, eine besondere Rolle spielt jedoch die dreistufige *Client/Server*-Architektur. Bei der dreistufigen *Client/Server*-Architektur wird das R/3-System in die Komponenten Datenbank, Applikation und Präsentation gegliedert. Die Dienste der verschiedenen Komponenten werden jeweils durch einen oder mehrere *Server* angeboten. Insbesondere die Möglichkeit die R/3-Applikationsdienste auf mehrere *Server* zu verteilen, läßt eine gute Skalierbarkeit des R/3-Systems zu. Diese Architektur stellt dann keine dreistufige, sondern eine mehrstufige (*multi-tiered*) *Client/Server*-Architektur dar.

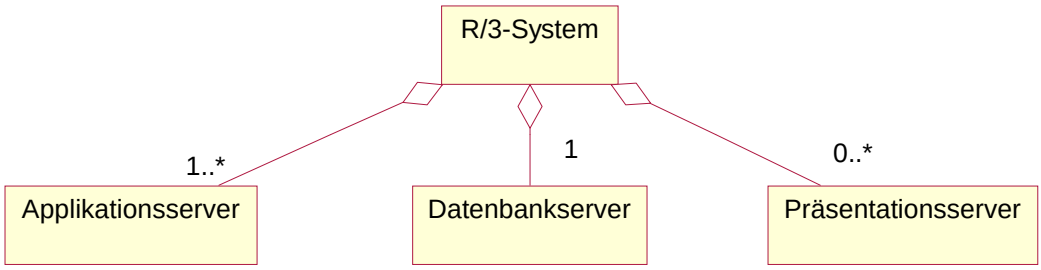

Abbildung 2.1: R/3-Systemarchitektur

Auf die drei verschiedenen Typen von *Servern* wird im folgenden eingegangen, wobei der Applikationsserver besonderer Beachtung bedarf und im nächsten Kapitel detailliert beschrieben wird.

Auf dem **Präsentationsserver** wird der sogenannte SAPGUI-Prozeß[2] ausgeführt, der die grafische Bedienoberfläche zum R/3 bildet. Die Funktionalität der grafischen Oberfläche ist dabei auf den verschiedenen von der SAP angebotenen Betriebssystemplattformen immer gleich. Der SAPGUI-Prozeß realisiert die von der SAP angebotene Mehrsprachfähigkeit in der Benutzung des R/3-Systems. Der SAPGUI-Prozeß kann gemäß der 3-Ebenen-Struktur sowohl auf einem oder mehreren zentralen Präsentationsservern als auch dezentral auf dem *Client* ablaufen. Heute wird meistens - da als *Clients* v.a. Workstations oder PCs benutzt werden - der dezentrale Ansatz gewählt. Der SAPGUI-Prozeß kommuniziert mit einem Dialog-Prozeß auf dem Applikationsserver. Dabei werden nicht aufbereitete Bildschirmbilder versandt, sondern lediglich Beschreibungen der darzustellenden Informationen. Dies reduziert den Datenstrom auf ein bis zwei Kilobyte pro Bildwechsel [Will et al. 96, S. 28].

Aufgrund des relationalen Datenmodells, welches dem R/3-System zugrunde liegt, können ausschließlich relationale Datenbanken als **Datenbankserver** eingesetzt werden, die über SQL angesprochen werden. Die Unterstützung für SQL wurde in die Programmiersprache ABAP integriert. Der Datenbankserver existiert in einem R/3-System genau ein einziges Mal. Die SAP begründet dies damit, daß die erhöhten Aufwände zur Sicherung der Datenbankkonsistenz bei mehreren Datenbankservern die Performanzvorteile überwiegen. Der Datenbankserver speichert die Anwendungsdaten, die im R/3 abgelegten Programme sowie Metadaten

[2] SAP Graphical User Interface

zu den Datenstrukturen, die diese Programme benutzen, in mehr als 4000 Datenbanktabellen. Dies ermöglicht den Aufbau eines aktiven *Repositories*, so daß z.B. die Erstellung eines Verwendungsnachweises für eine Datenstruktur möglich ist.

Als **Kommunikationskomponente** zwischen den einzelnen *Servern* wird das Transportprotokoll TCP/IP genutzt. Zwischen der Präsentationskomponente und der Applikationskomponente wird auf TCP/IP aufsetzend das proprietäre Protokoll DIAG verwendet; zwischen der Applikationskomponente und dem Datenbankserver wird das Remote-SQL-Protokoll des jeweiligen Datenbankherstellers verwendet.

Eine besondere Rolle für das Verständnis des R/3 spielt der (die) Applikationsserver eines R/3-Systems. Aus diesem Grund wird im folgenden Abschnitt auf diesen Servertyp gesondert eingegangen.

2.3 Der R/3-Applikationsserver

Der R/3-Applikationsserver wird auch als „Schaltzentrale des R/3“ [Will et al. 96, S. 28] bezeichnet, weil dieser die Anwendungsprogramme ausführt (interpretiert), welche die betriebswirtschaftliche Logik des R/3 beinhalten. Diese Anwendungsprogramme sind in ABAP, der plattformübergreifenden Programmiersprache des R/3, codiert. Das R/3-Laufzeitsystem ist in ANSI-C und C++ realisiert. ABAP-Programme und Benutzeroberflächen werden in der darunterliegenden Datenbank als portabler Bytecode gespeichert. Vor der Ausführung eines Programmes wird mit Hilfe eines Zeitstempels an Programmen und Objekten des ABAP *Dictionaries* die Existenz und die Aktualität überprüft und ggf. der Bytecode des Programms neu erzeugt. Der Bytecode wird durch die Komponenten ABAP-Prozessor und Dynproprozessor eines Workprozesses interpretiert. Auf dem R/3-Applikationsserver läuft eine konfigurierbare Anzahl von kooperierenden Prozessen mit jeweils spezialisierten Aufgaben, die sogenannten Workprozesse. Vermittler zwischen diesen lose gekoppelten Prozessen und zentraler Empfänger von Anforderungen an den Applikationsserver ist der *Dispatcher*, der pro Applikationsserver genau einmal vorhanden ist.

Der *Dispatcher* verwaltet die eingehenden Anforderungen in einer Warteschlange und weist diese den verschiedenen Workprozessen zu. Die von den Workprozessen ermittelten Ergebnisse werden wieder an den *Dispatcher* übermittelt, der wiederum diese an den Anforderungsinitiator weiterleitet.

Ein Workprozeß kann prinzipiell jede Anforderung übernehmen. Der Typ des Workprozesses wird durch den *Dispatcher* bestimmt.

Die Bestandteile eines Workprozeß sind (siehe Abbildung 2.3):

- der Taskhandler zur Koordination der Unterprozesse eines Workprozesses und zur Kommunikation mit dem *Dispatcher*
- der Dynproprozessor zur Erzeugung des Bildschirmlayouts, welches dem SAPGUI-Prozeß gesendet wird
- der ABAP-Prozessor zur Interpretation der als Bytecode gespeicherten Anwendungslogik

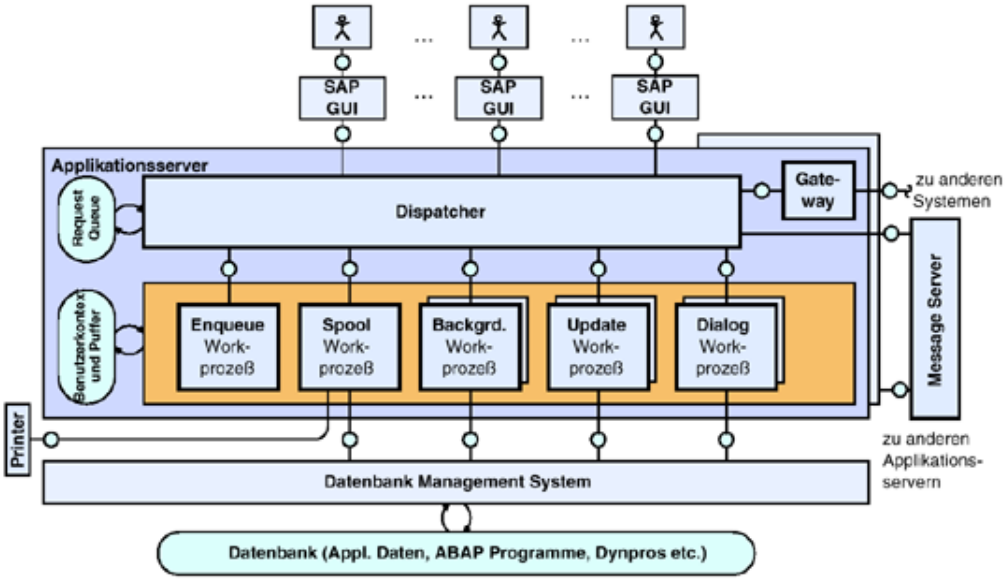

Abbildung 2.2: Struktur des Applikationsservers

Workprozesse müssen zur Erfüllung der ihnen zugewiesenen Aufgabe mit der Datenbank kommunizieren. Dies vollziehen sie aufgrund der enthaltenen Datenbankzugriffsfunktionalität ohne Hilfe des *Dispatcher*s.

Es existieren spezialisierte Workprozesse für die unterschiedlichen Typen von Anforderungen (z.B. Dialoganforderung, Hintergrundverarbeitung, Verbuchung). Für die Abarbeitung von Benutzeranforderungen durch die SAPGUI-Prozesse zeichnen Workprozesse des Typs Dialog verantwortlich.

Die Abarbeitung bezieht sich dabei auf genau einen Dialogschritt. Dabei wird die vom SAP-GUI-Prozeß erhaltene Dialogbeschreibung einschließlich der enthaltenen Daten ausgewertet und die Dialogbeschreibung des nachfolgenden Dialogschritts erstellt. Dabei bedient sich der Dialog-Workprozeß seiner Teilkomponenten Dynproprozessor zur Erzeugung der Dialogbeschreibung und ABAP-Prozessor zur Ausführung der im Dialogschritt enthaltenen Anwendungslogik. Der Folgedialog wird an den SAPGUI-Prozeß zurückgesandt, und damit steht der Dialogprozeß wieder für neue Dialoganforderungen zur Verfügung. Es existiert also keine fixe Zuordnung zwischen einem SAPGUI-Prozeß und einem Dialogprozeß. Die Zuordnung wird vielmehr dynamisch bei jeder SAPGUI-Dialoganforderung durch den *Dispatcher* hergestellt. Die SAP-GUI-Prozesse führen sogenannte SAP-Transaktionen aus. Die Semantik des Begriffs Transaktion im R/3 soll im folgenden genauer erläutert werden.

2.4 Transaktionen im R/3

Die betriebswirtschaftlichen R/3-Anwendungsprogramme greifen lesend und schreibend auf die R/3-Datenbank zu. Schreibende Zugriffe können also nebenläufig geschehen. Aus diesem

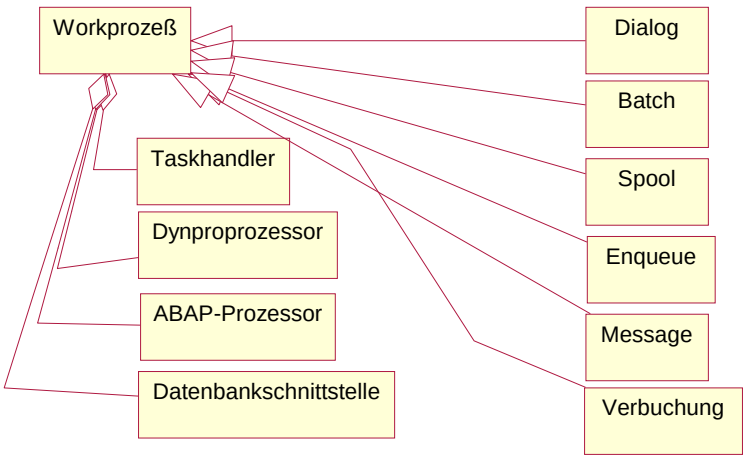

Abbildung 2.3: Bestandteile und Arten eines Workprozesses

Grund ist ein Transaktionskonzept zur Sicherung der Integrität der Daten notwendig. In Erweiterung zur Semantik einer Transaktion auf Datenbankebene definiert die SAP ein eigenes erweitertes Transaktionskonzept.

Eine **Datenbanktransaktion** ist eine Folge von Operationen, die eine Datenbank in ununterbrechbarer Weise von einem semantisch korrekten Zustand in einen (nicht notwendig verschiedenen) konsistenten Zustand überführt. Dabei müssen die Eigenschaften der Unteilbarkeit, Konsistenzerhaltung, der Isolation und der Dauerhaftigkeit erfüllt sein (ACID-Eigenschaften) [Lockemann, Schmidt 87, S. 404]. Im SAP-Umfeld wird eine Datenbanktransaktion als Datenbank-LUW[3] bezeichnet.

Betrachtungsobjekte im R/3 sind komplexe betriebswirtschaftliche Objekte, wie z.B. Buchungen oder Verkaufsaufträge. Komplex sind diese, da zum Ändern eines solchen Objektes eine Vielzahl von Tabellenzeilen in verschiedenen Tabellen zu ändern sind. Diese Änderung muß ebenfalls den ACID-Eigenschaften genügen, um keine Inkonsistenzen in der Datenbank entstehen zu lassen. Dies wird nicht durch das Datenbankmanagementsystem garantiert, sondern durch das R/3-System. R/3 faßt alle Änderungen an einem komplexen betriebswirtschaftlichen Objekt in einer SAP-LUW zusammen, die durch das Programm gesteuert in genau einer Datenbanktransaktion in die Datenbank geschrieben werden (**COMMIT WORK**), bzw. verworfen werden (**ROLLBACK WORK**). Technisch geschieht dies durch die Erzeugung von Änderungsaufträgen und Speicherung dieser in einer Tabelle, die dann nach einem **COMMIT** oder **ROLLBACK** durch den sogenannten Verbuchungsprozeß abgearbeitet werden. Die persistente Speicherung erfolgt im Normalfall also asynchron zur Ausführung des Programms. Alternativ kann der Anwendungsentwickler die **COMMIT**-Steuerung aber so beeinflussen, daß Datenbankänderungen synchron zur Ausführung des Programms ausgeführt werden.

Änderungsaufträge an die Datenbank entstehen im Verlauf der Ausführung einer SAP-Transaktion. Als SAP-Transaktion wird ein ABAP-Dialogprogramm bezeichnet, welches ein Benutzer bei der Bedienung des R/3 mittels der grafischen R/3-Bedienoberfläche SAPGUI abarbeitet. Dieses Programm besteht häufig aus verschiedenen Dialogschritten (ein einzelner

[3] Logical Unit Of Work

Dialog wird im folgenden als Dynpro bezeichnet). Dynpros enthalten sowohl die Definition der grafischen Oberfläche als auch eine Ablauf- und Verarbeitungslogik, welche die betriebswirtschaftliche Anwendungslogik des R/3 implementiert. Innerhalb dieser Anwendungslogik werden also z.B. Benutzereingaben überprüft, der Ablauf der Dialogschritte aufgrund der Benutzereingaben gesteuert und v.a. Änderungsanforderungen an die Datenbank erzeugt.

Im Unterschied zur SAP-LUW besitzt eine SAP-Transaktion nicht zwingend eine transaktionale Semantik. Eine SAP-Transaktion kann eine weitere SAP-Transaktion ausführen (**CALL TRANSACTION**), für die eine eigene SAP-LUW erzeugt wird. Ein Zurücksetzen (**ROLLBACK**) innerhalb dieser gerufenen Transaktion führt nicht zu einem Zurücksetzen der rufenden Transaktion. Dieses Konzept entspricht somit dem Konzept der *Open Nested Transaction* [Gray, Reuter 93]. Die transaktionale Semantik ist also nur garantiert, wenn eine SAP-Transaktion innerhalb genau einer SAP-LUW ausgeführt wird. Dies heißt wiederum auch, daß ein Programm, das Datenbankänderungen erzeugt und eine transaktionale Semantik innerhalb seiner Änderungen benötigt, einen strikt sequentiellen Ablauf haben muß. Der nebenläufige Zugriff der Workprozesse auf die Datenbank kann zu Konflikten führen,

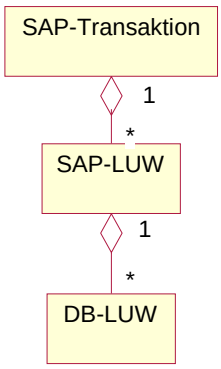

Abbildung 2.4: Transaktionsbegriff im R/3

die innerhalb einer SAP-LUW durch eine transaktionale Semantik aber keine Gefahr für die Datenintegrietät bedeuten. Um nicht bei jedem Konflikt mit einem Zurücksetzen (**ROLLBACK**) die Transaktion zu verwerfen, sondern zumindest den lesenden Zugriff erlauben, oder die weitere Ausführung der Transaktion für die Zeit der Sperrung zu verzögern, ist es möglich, sogenannte Sperrobjekte vom R/3-System anzufordern. Benötigt eine SAP-Transaktion exklusiven Zugriff auf betriebswirtschaftliches Objekte, so muß die Anwendungslogik explizit spezielle Sperrbausteine (*Enqueue*-Funktionsbausteine) aufrufen. Diese Sperren werden durch den *Enqueue*-Workprozeß verwaltet.

2.5 Die *Development Workbench*

Die *Development Workbench* stellt die integrierte Entwicklungsumgebung des R/3 dar, die es gestattet, sowohl Datendefinitionen und deren Inhalte zu bearbeiten als auch Programmentwicklung zu betreiben. Diese Programme benutzen die im ABAP *Dictionary* definierten Datendefinitionen. Eine Änderung einer Datendefinition hat also unmittelbar Auswirkungen

auf Programme, welche diese Datendefinition nutzen. Zusätzlich zu den Datendefinitionen werden sämtliche Programme im R/3 gespeichert und verwaltet. Alle Anwendungen des R/3-Systems sind innerhalb dieser Umgebung mit der Sprache ABAP entwickelt worden. Diese Integration von Programmen und Datendefinitionen erlaubt es, von einer Datendefinition des ABAP *Dictionaries* über den Verwendungsnachweis zu den Programmstellen zu verzweigen, in denen die Datendefinition benutzt wird. Dies gilt ebenso für die Navigation von Programmen zu den Definitionen der benutzten ABAP *Dictionaryobjekte*.

Betrachtungsgegenstand der *Development Workbench* sind Entwicklungsumgebungsobjekte. Entwicklungsumgebungsobjekte werden durch einen - aufgrund der Längenbegrenzung häufig kryptischen - Namen identifiziert. Das R/3-System stellt ein globalen Namensraum dar. Überschneidungen zwischen Kunden- und SAP-Entwicklungsobjekten werden durch die Bildung von getrennten Namensräumen verhindert. Änderungen an SAP-Entwicklungsobjekten sind möglich, jedoch auf keinen Fall zu empfehlen, da zum einen die Stabilität des R/3 nicht mehr garantiert werden kann und zum anderen Probleme beim R/3-Releasewechsel entstehen können.

Für eine komplette Abbildung aller Entwicklungsumgebungsobjekt(-typen) sei auf [Matthes, Ziemer 98] verwiesen. Für das Verständnis dieser Arbeit sind nur eine Teilmenge von Typen erforderlich, die im folgenden beschrieben werden.

Entwicklungsumgebungsobjekte lassen sich in zwei Gruppen unterteilen:

- ABAP *Dictionaryobjekte*
- Entwicklungsklassenobjekte

Im folgenden soll auf diese beiden Gruppen von Entwicklungsumgebungsobjekten eingegangen werden.

2.5.1 Typkonzept und ABAP *Dictionaryobjekte*

ABAP kennt analog zu gängigen Programmiersprachen eine Menge von Basistypen, die innerhalb von Programmen nutzbar sind. Diese sind in der Tabelle 2.1 mit ihren Eigenschaften wie Wertebereich, Länge und Initialwert erläutert.

Die Semantik eines Typs ist jedoch nicht mit statisch typisierten Programmiersprachen vergleichbar. „ABAP unterstützt fast alle denkbaren Konvertierungen (zwischen verschiedenen Typen) automatisch“ [Matzke 98, S. 70]. Das ABAP-Typkonzept verhindert also die Fehler, die aus der Nutzung falscher Typen zur Laufzeit entstehen, nur bedingt. Ein weiterer Unterschied ist, daß nur die Typen Fließkommazahl und Ganzzahl direkt auf die zugrundeliegende Maschine abgebildet werden. Alle anderen Typen sind durch Zeichenketten repräsentiert, wobei der Aufbau der Zeichenkette bestimmten Regeln folgt (z.B. JJJJMMTT bei einem Datum-Typ). Mit diesen skalaren ABAP-Basistypen lassen sich beliebig komplexe strukturierte Typen definieren, die auch systemweit, also auch durch andere Anwendungen benutzt werden können (**TYPE-POOLS**).

ABAP kennt neben den **TYPE-POOLS** zwei strukturierte Datentypen. Dieses sind die Feldleiste und die interne Tabelle:

Basistyp	Bedeutung	Initialwert	Länge
C	Zeichenkette	""	1-65536
D	Datum	"00000000"	8
T	Zeit	"00000"	6
N	Numeric, Zeichenkette aus Ziffern	"0"	1-65536
X	Hexadezimalzahl	"00"	1-65536
F	Gleitkommazahl	0.0	8
P	Gepackte Zahl im BCD-Format	0	1-16
I	Ganzzahl	0	4

Tabelle 2.1: ABAP-Basistypen

Feldleiste Eine Feldleiste ist vergleichbar mit einem **RECORD**-Typ, d.h sie besitzt eine Feldstruktur. Die Felder einer Feldleiste haben dabei einen skalaren Typ oder alternativ einen strukturierten Typ. Mit Feldleisten lassen sich also beliebig komplexe und auch rekursive Typen aufbauen.

Interne Tabelle Eine interne Tabelle ist mit einer Aggregation von Feldleisten vergleichbar. Der Datentyp einer internen Tabelle wird dabei durch den Zeilentyp, Schlüssel und Zugriffsart bestimmt. Ein Zeilentyp ist dabei gleichzusetzen mit den Feldern einer Feldleiste. Über den Schlüssel können die Felder bestimmt werden, die eine Tabellenzeile eindeutig identifizieren. Die Zugriffsart bestimmt die interne Darstellungsform als unsortierte, sortierte oder Hash-Tabelle. Mit dem ABAP-Datentyp interne Tabelle lassen sich Tabellenvariablen typisieren. Diese Tabellenvariablen werden dabei leicht irreführend genauso wie ihr Typ als interne Tabelle bezeichnet. Die Anzahl der Tabellenzeilen einer Tabelle ist nicht statisch, sondern wird dynamisch vom R/3-Laufzeitsystem angepaßt. Auf solchen Tabellen sind Operationen wie Suchen, Lesen und Ändern von Tabellenzeilen möglich. Die Lebenszeit einer internen Tabelle ist auf die Programmlaufzeit begrenzt. Eine interne Tabelle dient damit zur temporären Speicherung von Informationen. Um Informationen persistent zu speichern, müssen diese in die R/3-Tabellen geschrieben werden. Eine direkte Abbildung einer internen Tabelle auf eine persistente R/3-Tabelle (im folgenden als transparente Tabelle bezeichnet) ist jedoch nur dann möglich, wenn die Felder des Zeilentyps ausschließlich skalaren Charakter haben. Hat die interne Tabelle z.B. einen rekursiven Zeilentyp, ist es Aufgabe des Anwendungsentwicklers, diese Struktur geeignet aufzulösen und in den transparenten Tabellen zu speichern.

Tabellen-, Strukturdefinitionen und zugehörige Metadaten werden zentral durch das ABAP *Dictionary* gespeichert. Diese Definitionen können als Feldleisten oder als Zeilentyp für interne Tabellen innerhalb von ABAP-Programmen genutzt werden.

Tabelle Eine Tabelle besteht aus einem oder mehreren Tabellenfeldern, die über einen Namen angesprochen werden. Felder haben skalaren Charakter. Die Definition rekursiver Typen ist im ABAP *Dictionary* nicht möglich (im Gegensatz zu internen Tabellen). Es lassen sich Fremdschlüsselbeziehungen zwischen Tabellenfeldern definieren; die Überprüfung der Fremdschlüsselbeziehung obliegt jedoch dem Anwendungsprogramm und wird nicht automatisch durch das R/3 vorgenommen.

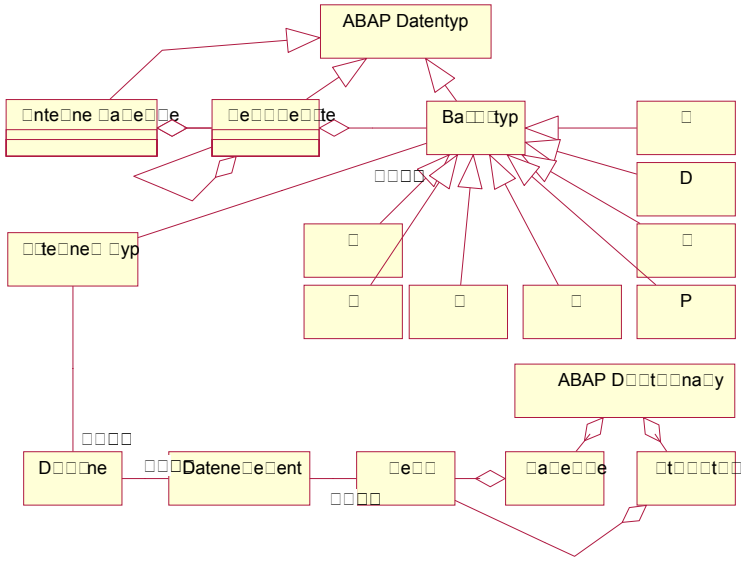

Abbildung 2.5: ABAP-Typsystem

Das R/3-System kennt verschiedene Arten von Tabellen, die sich hinsichtlich der Abbildung auf eine SQL-Tabelle der darunterliegenden Datenbank unterscheiden:

- eine transparente Tabelle wird direkt auf eine SQL-Tabelle abgebildet
- beim Typ Pooltabelle werden Tabellenzeilen verschiedener Tabellen in eine SQL-Tabelle abgebildet
- der Typ Clustertabelle ermöglicht die Speicherung verschiedener Tabellen in einer SQL-Tabelle

Struktur Strukturen werden analog zu Tabellen definiert, mit dem Unterschied, daß aus ihnen keine SQL-Tabellen in der Datenbank erzeugt werden. Strukturen enthalten also nur zur Laufzeit eines Programmes Daten und stellen somit lediglich eine in allen Programmen verfügbare Definition eines **RECORD**-Typs dar. Der Typ eines Tabellenfeldes oder des Feldes einer Struktur kann entweder durch eine direkte Zuweisung von Datentyp und Länge, oder - das ist der Normalfall - durch die Zuweisung zu einem Datenelement erfolgen, welches implizit den Typ bestimmt.

Datenelemente haben im Unterschied zu den ABAP-Basistypen weitere Eigenschaften, die bei der Programmierung von Anwendungen sinnvoll sind und diese erleichtern. Diese zusätzlichen Eigenschaften werden durch ein dreistufiges Typkonzept bestehend aus Datenelement, Domäne und externer Typ bestimmt:

Datenelement Datenelemente beschreiben die betriebswirtschaftlichen Aspekte eines Typs wie betriebswirtschaftliche Bedeutung und die angezeigte Feldbezeichnung, falls das Feld auf einem Dynpro verwandt wird.

Domäne Domänen beschreiben die primär technischen Aspekte wie den Wertebereich. Dies sind u.a. eine Festlegung einer Feldlänge und einer optionalen Wertetabelle. Domänen werden wiederum auf einen externen Typ abgebildet.

Externe Typen Externe Typen dienen der Abbildung der Domänen auf die von der Datenbank angebotenen Typen. Die ABAP-Basistypen werden ebenfalls auf die externen Typen abgebildet. Die externen Typen orientieren sich einerseits an betriebswirtschaftlichen Typen (Mandant, Währung) als auch an technischen Typen (Zeichenkette, Fließkommazahl). Im Unterschied zu den ABAP-Basistypen definieren die genannten ABAP *Dictionaryobjekte* also zusätzliche Eigenschaften eines Typs.

2.5.2 Entwicklungsklassenobjekte

Entwicklungsklassenobjekte nutzen die im ABAP *Dictionary* gepflegten Informationen zu Datendefinitionen und sind - wie der Name bereits andeutet - in Entwicklungsklassen organisiert. Eine Enwicklungsklasse stellt ein Mittel zur Strukturierung von Entwicklungsumgebungsobjekten dar. In einer Entwicklungsklasse sollten alle Objekte zusammengefaßt werden, die eine „funktionale Einheit“ [SAP 98f] bilden. Dies wären am Beispiel einer SAP-Transaktion die zugehörigen Programmelemente, Bildschirmdefinitionen usw.

Im folgenden werden bestimmte für diese Arbeit relevante Entwicklungsklassenobjekte erläutert. Dabei ist das Programmobjekt zu erwähnen, das - wie sich später zeigen wird - zur Implementierung von Geschäftsobjekten geschickt eingesetzt wird, sowie der Funktionsbaustein, der das bisher wichtigste Mittel zum externen Zugriff auf das R/3 darstellt.

2.5.2.1 Programmobjekt

Das wohl umfangreichste Entwicklungsklassenobjekt stellt das Programmobjekt dar. Programmobjekte sind in der portablen Programmiersprache des R/3-Systems ABAP erstellt und beinhalten die gesamte Anwendungslogik des R/3. Die Bearbeitung und Entwicklung geschieht innerhalb der R/3 Entwicklungsumgebung, der *Development Workbench*. Der Programmtext von ABAP-Programmen wird wie ein ABAP *Dictionaryobjekt* in Tabellen der Datenbank abgelegt. ABAP-Programmtexte werden durch das R/3 kompiliert und in einem plattformunabhängigen Bytecode ebenfalls in der Datenbank gespeichert und durch eine virtuelle Maschine, dem ABAP-Prozessor, ausgeführt.

R/3-Anwendungsprogramme laufen auf den Workprozessen der Applikationserver ab. Programme lassen sich unterteilen in die Ablauflogik (ausgeführt auf dem Dynpro-Prozessor) und die Verarbeitungslogik (ausgeführt auf dem ABAP-Prozessor):

Ablauflogik Die Ablauflogik reagiert auf Aktionen, die aus der Abarbeitung der Dynpros resultieren. Die Ablauflogik wird durch das R/3 zu fest definierten Zeitpunkten selbst gerufen. Diese Zeitpunkte sind unmittelbar bevor ein Dialog für den Nutzer sichtbar wird (*Process before output*), bzw. nachdem ein Dialog durch den Anwender verlassen wurde (*Process after input*). An dieser Stelle werden bevorzugt Aufgaben wie die Prüfung der Benutzereingaben, sowie die dynamische Anpassung von Dynproelementen vorgenommen.

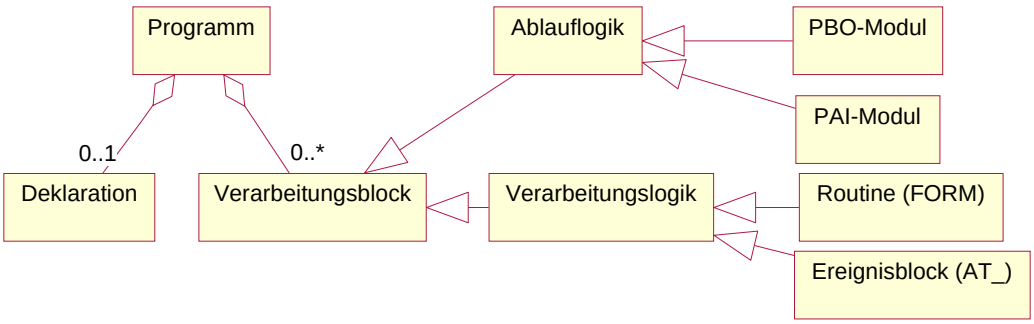

Abbildung 2.6: Programmobjekt

Verarbeitungslogik Die Verarbeitungslogik repräsentiert die eigentliche Anwendungslogik. Hier werden Benutzereingaben von den Dynpros verarbeitet und Datenbankinhalte aufgrund betriebswirtschaftlicher Logik verändert.

Sowohl die Ablauflogik als auch die Verarbeitungslogik werden zur besseren Strukturierung des Programmtextes innerhalb des Programms in Form von Verarbeitungsblöcken abgelegt, die ineinander schachtelbar sind. An Routinen (FORM) lassen sich Aktualparameter übergeben. Mit einer Funktionssignatur ist dies aber nur bedingt vergleichbar, da die Formalparameter nicht typisiert werden müssen. Der Aufruf dieser Blöcke ist dabei auch aus anderen Progammen heraus möglich (ABAP-Kommando **PERFORM**).

Außer den Verarbeitungsblöcken besitzt ein Programm einen Deklarationsteil für globale Daten, auf dessen Daten alle Verarbeitungsblöcke innerhalb des Programms zugreifen können. Außerhalb des Programms bleiben diese Daten verborgen, wenn z.B. ein anderes Programm eine Routine aufruft.

Abhängig vom Programmtyp unterscheidet sich die Struktur der Verarbeitungsblöcke eines Programms:

Ein Programm des Typs 1 läßt sich direkt ausführen und wird auch als *Report* bezeichnet. Ein Programm des Typs M (Modulpool) stellt die bereits erwähnte SAP-Transaktion dar und läßt sich nicht direkt ausführen. Der Einsprungpunkt in einen solchen Modulpool wird durch den Programmnamen und die Nummer des Startdynpros festgelegt. Typ-F-Programme stellen die Funktionsgruppen dar, die eine Menge von Funktionsbausteinen sowie zugehöriger globaler Daten aufnehmen. Diese Funktionsbausteine sind Thema des nächsten Abschnitts.

2.5.2.2 Funktionsbaustein

Funktionsbausteine stellen ein Mittel zur Modularisierung eines ABAP-Programms dar. Sie sind ABAP-Quelltext, der von beliebigen Programmen aus aufgerufen werden kann. Im Unterschied zu Unterroutinen (FORMs) wird der Zugriff jedoch nur über eine definierte Schnittstelle zugelassen. Gemeinsame Speicherbereiche zum Austausch von Daten zwischen rufendem Programm und Funktionsbaustein sind also nicht möglich und tragen so zur anwendungsunabhängigen Nutzung von Funktionsbausteinen bei. Das *Development Workbench*-Werkzeug

Funktionsbibibliothek verwaltet zentral die im R/3 verfügbaren Funktionsbausteine und deren Schnittstellen.

Das aufrufende Programm muß nicht notwendigerweise ein ABAP-Programm sein, sondern kann eine externe Anwendung sein, dies wird als entfernter Funktionsaufruf[4] bezeichnet. Umgekehrt lassen sich genauso aus ABAP-Programmen Funktionsbausteine aufrufen, die nicht als ABAP-Quelltext im R/3 vorliegen, sondern durch eine Funktion eines externen Programms realisiert sind. Alternativ dazu kann der Funktionsbaustein in einem anderen erreichbaren R/3-System definiert sein. Im R/3 existieren circa 32000 von der SAP ausgelieferte Funktionsbausteine, von denen circa 4500 entfernt aufrufbar sind. Daraus läßt sich bereits ablesen, daß Funktionsbausteine die von der SAP favorisierte Möglichkeit zur Wiederverwendung von Quelltext ist.

Über das Konzept der Funktionsbausteine steht prinzipiell die gesamte Funktionalität des R/3-Systems zur Verfügung, d.h. es ist der Zugriff auf die betriebswirtschaftliche Logik, auf Tabellen der Datenbank und sogar SAP-Transaktionen möglich. SAP-Transaktionen lassen sich durch die Benutzung des Funktionsbausteins **RFC_CALL_TRANSACTION** entfernt ausführen. Dabei wird als Laufzeitparameter eine Tabelle übergeben, welche die einzelnen Bildschirmelemente der Dialoge und ihre Werte enthält.

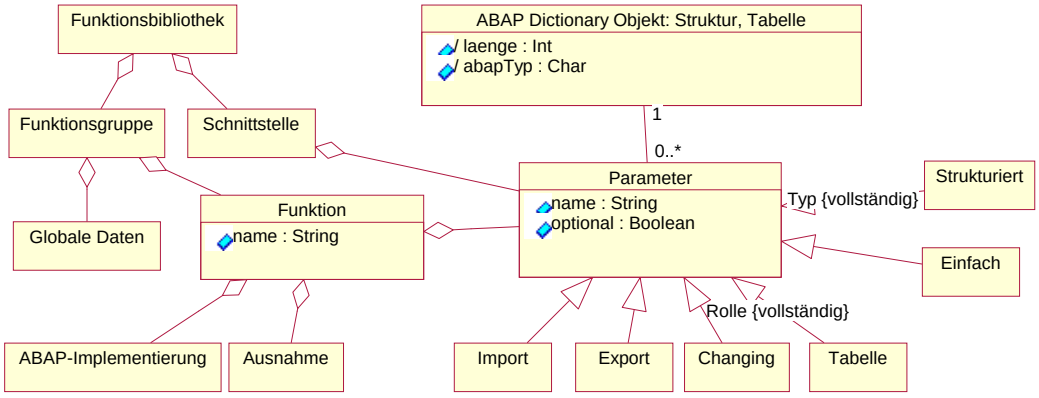

Abbildung 2.7: Die R/3-Funktionsbiliothek

Ein Funktionsbaustein ist Bestandteil einer Funktionsgruppe, die übergreifende Typdefinitionen und Deklarationen für alle enthaltenen Funktionsbausteine speichern kann. In einer Funktionsgruppe werden inhaltlich zusammengehörige Funktionsbausteine zusammengefaßt. Ein Funktionsbaustein wird über seinen Namen identifiziert und besitzt Möglichkeiten für die Behandlung von Ausnahmen.

Ausnahmen ermöglichen es, dem aufrufenden Programm auf aufgetretende Fehler innerhalb des Funktionsbausteins zu testen und geeignet zu reagieren. Im Quelltext des Funktionsbausteins wird eine Ausnahme mit dem ABAP-Schlüsselwort **RAISE** ausgelöst. Im aufrufenden Programm wird die Ausnahme durch einen Wert der Systemvariaben **SY-SUBRC** von ungleich null signalisiert. Bei entfernten Funktionsbausteinaufrufen existieren zwei vordefinierte Ausnahmen:

[4] RFC - Remote Function Call

- **SYSTEM_FAILURE** wird ausgelöst, falls auf der Empfängerseite ein System-Abbruch erfolgt ist.
- **COMMUNICATION_FAILURE** wird ausgelöst, falls ein Problem beim Aufbau der Verbindung oder bei der Kommunikation aufgetreten ist.

```
DATA: "Datendeklarationen:
"Deklaration einer internen Tabelle zur
"Aufnahme der Feldinformationen
fields LIKE rfc_fields OCCURS 10 WITH HEADER LINE,
tablelength LIKE RFC_FIELDS-INTLENGTH,   "Länge der Tabelle in Bytes
tabname LIKE X030L-TABNAME value 'KNA1'  "Name der Tabelle
CALL FUNCTION 'RFC_GET_STRUCTURE_DEFINITION' "Aufruf Funktionsbaustein
     EXPORTING
          TABNAME  = tabname             "Importparameter Tabellenname
    IMPORTING
         TABLENGTH = tablelength         "Exportparameter Tabellenlänge
     TABLES
         FIELDS    = fields              "Tabellenparameter Feldinfo
     EXCEPTIONS                          "Mögliche Ausnahmen"
          TABLE_NOT_ACTIVE  = 1
          OTHERS            = 2.
IF SY-SUBRC  =  1.                       "Ausnahme ausgelöst?
        "Ausnahmebehandlung für TABLE_NOT_ACTIVE
END IF.
LOOP AT fields.
"Bearbeite altuelle Tabellenzeile, die Info für ein Feld
"der Tabelle enthält
ENDLOOP.
```

Abbildung 2.8: Aufruf eines Funktionsbausteins in ABAP

Die Schnittstelle eines Funktionsbausteins wird durch seine Parameter definiert. Ein Parameter wird über seinen Namen angesprochen und läßt sich hinsichtlich seiner Rolle und seines Typs unterscheiden:

Typ des Parameters Der Parameter kann typtechnisch entweder einfach oder strukturiert sein. Einfache Parameter beziehen sich dabei zur Festlegung ihrer Typeigenschaften auf das Feld einer im ABAP *Dictionary* angelegten Tabelle oder Struktur. Durch die vorhandene Zuordnung dieses Feldes zu einem Datenelement läßt sich der ABAP-Basistyp ermitteln. Strukturierte Parameter gestatten einen feldweisen Zugriff und beziehen sich auf eine im ABAP *Dictionary* angelegte Tabelle oder Struktur. Strukturierte Parameter sind deshalb nur mit Einschränkung mit den ABAP-Datentypen Feldleiste und interne Tabelle vergleichbar, da die Felder eines Parameters skalaren Charakter haben.

Bei einfachen Parametern besteht die Möglichkeit diese per Wert oder alternativ per Referenzsemantik zu übergeben. Weiterhin können Parameter als optional gekennzeichnet werden oder mit einem Vorgabewert belegt werden.

Rolle des Parameters Parameter werden hinsichtlich ihrer Rolle unterschieden, den der Parameter für den Baustein besitzt.

- Importparameter dienen der Übergabe von Werten durch das aufrufende Programm an den Funktionsbaustein.
- In Exportparametern werden die vom Funktionsbaustein ermittelten Ergebnisse abgelegt.
- Changingparameter vereinigen die Eigenschaften von Import- und Exportparametern, d.h. der Funktionsbaustein kann den Wert eines Parameters ändern und an das rufende Programm zurückgeben.
- Tabellenparameter sind strukturierte Parameter, die mehrere gleichartige Zeilen einer zugrundeliegenden Struktur- oder Tabellendefinition enthalten. Tabellenparameter werden aus Gründen der Performanz immer mit Referenzsemantik übergeben und sind so mit den Changingparametern vergleichbar.

Im Abbildung 2.8 ist der Aufruf des Funktionsbausteins **RFC_GET_STRUCTURE_DEFINITION** illustriert, der zu einem gegebenen Namen einer ABAP *Dictionary*-Tabelle oder Struktur die Felder sowie Informationen zu diesen Feldern ermittelt.

Ein Funktionsbaustein läßt sich entfernt ausführen[5]. Der Parameter **DESTINATION** spezifiziert durch einen logischen Namen das Ziel des Aufrufs. Ziele werden durch Einträge in der Tabelle *RFCDES* definiert. Ein besonderes Ziel stellt **BACK** dar: Wenn der ausgeführte Funktionsbaustein durch einen anderen entfernt ausführbaren Funktionsbaustein aufgerufen wurde, verzweigt **BACK** in den rufenden Funktionsbaustein zurück und stellt somit einen Funktionsrückruf dar. Das gezeigte Programmbeispiel zeigt einen lokalen und synchronen Aufruf eines Funktionsbausteins. Zusätzlich zum synchronen Aufruf existieren weitere Aufrufsemantiken, die im folgenden dargestellt werden:

CALL FUNCTION func STARTING NEW TASK Der asynchrone Aufruf führt den Funktionsbaustein asynchron zum Programmfluß des rufenden Programms in einem neu gestarteten Workprozeß aus. Der Aufrufer wird also in diesem Fall nicht blockiert. Die Beendigung des Aufrufs kann durch den Aufruf einer Unterroutine dem Aufrufer signalisiert werden (**PERFORMING formName ON END OF TASK**). Die Ergebnisse des Funktionsbausteins (=Exportparameter) einschließlich ausgelöster Ausnahmen können durch das ABAP-Schlüsselwort **RECEIVE RESULTS FROM FUNCTION** erhalten werden.

CALL FUNCTION func IN BACKGROUND TASK Der transaktionale Aufruf führt den Funktionsbaustein ebenfalls asynchron aus, besitzt jedoch eine *exactly-once*-Semantik. Die durch den Funktionsbaustein ausgeführten Datenbankänderungen werden zunächst in den Tabellen *ARFCSSTATE* und *ARFCSDATA* gespeichert, um erst bei einem **COMMIT** des aufrufenden Programms ausgelesen und persistent in die Datenbank geschrieben zu werden. Der transaktionale Aufruf stellt den Mechanismus bereit, um Datenbankänderungen auf einem entfernten Sytem mit transaktionaler Semantik durchzuführen.

[5]Der Aufruf wird in einem anderen SAP-System oder in einer externen Anwendung ausgeführt.

CALL FUNCTION func IN UPDATE TASK Der Aufruf als Verbuchungsbaustein stellt den lokalen transaktionalen Funktionsbausteinaufruf dar. Abhängig von den Einstellungen des Funktionsbausteins wird die Verbuchung, d.h. das persistente Schreiben der Änderungen in die Datenbank gesteuert.[6] Die transaktionale Semantik wird dadurch erreicht, daß der gerufene Funktionsbaustein in derselben SAP-LUW ausgeführt wird wie der Aufrufer.

Wie bereits in Abschnitt 2.4 festgestellt ist die Ausführung aller Datenbankänderungen in einer einzigen SAP-LUW die Voraussetzung für eine transaktionale Semantik. Lediglich die Aufrufsemantiken **IN BACKGROUND TASK** und **IN UPDATE TASK** besitzen also eine transaktionale Semantik. Als Einschränkung ist es nicht möglich, Ergebnisparameter durch den Funktionsbaustein zu empfangen. Dies ist bei der Programmierung von Datenbankänderungen durch Funktionsbausteine zu berücksichtigen.

2.6 Zugriff auf R/3 durch Fremdsysteme

R/3 ist ein prinzipiell offenes System, d.h. es werden von der SAP Schnittstellen zu Fremdsystemen angeboten, die eine Interaktion mit dem R/3 ermöglichen. Eine im Rahmen dieser Arbeit zu entwickelnde objektorientierte Integration mit dem R/3 muß sich dabei auf eine der folgenden geschilderten Mechanismen abstützen.

R/3-Schnittstellen lassen sich in zwei Klassen unterteilen.

- Schnittstellen zur Unterstützung der direkten Programm-zu-Programm-Kommunikation zwischen R/3 und Fremdsystem
- Schnittstellen zur Erzeugung und Verarbeitung von Dateien

Die Schnittstelle zur dem R/3 unterliegenden SQL-Datenbank ist zwar prinzipiell auch als Schnittstelle zu Fremdsystemen geeignet, aber die SAP warnt eindringlich vor dieser Methode, da die Benutzung zum einen vom Releasestand des R/3 abhängt und zum anderen durch falsche Manipulation von Tabellen die Stabilität des gesamten R/3 in Frage stellen kann.

Die oben aufgeführten Dateischnittstellen werden von uns in diesem Abschnitt nicht betrachtet, da sie v.a. für unregelmäßige Interaktion (z.B. periodische Übertragung von Buchungen) sowie mengenmäßig umfangreiche Interaktionen (wie z.B. bei der Altdatendatenübernahme bei einer Einführung von R/3) geeignet sind. Für eine objektorientierte Integration mit dem R/3 kommt also nur der Mechanismus der direkten Programm-zu-Programm-Kommunikation in Frage.

Als Kommunikationsbasis für die direkte Programm-zu-Programm-Kommunikation benutzt die SAP das 1987 von der IBM entwickelte *Common Programming Interface - Communications (CPI-C)*, deren elementare Funktionalität in der Form von Schlüsselworten in die Programmierungsparache ABAP integriert wurde.

[6] Die Verbuchung ist alternativ als V1- oder V2-Komponente möglich.

ABAP-Typ	C-Typ
C (Zeichenkette)	unsigned char
D (Datum)	unsigned char [8]
T (Zeit)	unsigned char [6]
N (Numeric)	unsigned char
X (Hexadezimalzahl)	unsigned char
F (Gleitkommazahl)	double
P (Gepackte Zahl)	unsigned char
I (Ganzzahl)	int

Tabelle 2.2: Umsetzung von ABAP-Basistypen in der RFC-C-Bibliothek

Auf den CPI-C aufsetzend existiert das *Queue Application Programming Interface (Q-API)* , das die gepufferte asynchrone Datenübertragung zwischen dem R/3 und einem Fremdsystem ermöglicht.

Ebenfalls auf CPI-C basierend existiert die RFC-Automation-Schnittstelle, welche die hochsprachlichste Schnittstelle zur Anbindung von Fremdsystemen darstellt, da bei dieser Schnittstelle das Schreiben eigener Kommunikationsroutinen entfällt und diese durch die RFC-Schnittstelle zur Verfügung gestellt werden.

Neben der RFC-Schnittstelle wird die SAP-Automation-Schnittstelle beschrieben, die im Gegensatz zur RFC-Automation nicht Funktionsbausteine als Grundlage hat, sondern als Grundlage die Kommunikation zwischen SAPGUI und Applikationsserver nutzt.

2.6.1 RFC-Automation

Die RFC-Automation steht bereits seit Release 2.1 des R/3 zur Verfügung. Die Grundlage der RFC-Automation stellen die in Kapitel 2.5.2.2 beschriebenen Funktionsbausteine dar. Durch diese Schnittstelle lassen sich einerseits die Funktionsbausteine im R/3 durch Fremdsysteme aufrufen. Andererseits besteht die Möglichkeit, eine externe Funktion aus dem ABAP-Programmquelltext heraus aufzurufen.

Die RFC-Schnittstelle wird durch eine Programmierbibliothek für die Programmiersprachen C, C++ und sowie ActiveX-Komponenten für *Windows*-basierte Entwicklungsumgebungen zur Verfügung gestellt. Die RFC-Schnittstelle ist (mit Ausnahme der ActiveX-Komponenten) plattformunabhängig. In Anlehnung an die Funktionalität von Funktionsbausteinen im R/3 bietet die Schnittstelle folgende Funktionalitäten:

- Aufbau und Abbau von Verbindungen zum R/3
- Strukturen, die den Funktionsbaustein im R/3 repräsentieren
- Füllen und Auslesen der Parameter, die für den Funktionsbaustein benötigt werden. Als Parameter werden dabei die bekannten Typen wie einfache Parameter, strukturierte Parameter und Tabellen unterstützt. Für die elementaren ABAP Datentypen ist eine Abbildung auf C-Datentypen gemäß Tabelle 2.2 definiert.

```
RFC_HANDLE handle; //Handle für Verbindung mit R/3-System
RFC_RC rfc_rc; //Struktur für Ausnahmebehandlung
RFC_PARAMETER exporting[32]; //Datenfeld mit Importparametern
RFC_PARAMETER importing[32]; //Datenfeld mit Exportparametern
RFC_TABLE tables[32]; //Datenfeld mit Tabellenparametern
char* exception_ptr = NULL; //signalisiert Ausnahme

handle = RfcOpenEx(connect_param, &error_info); //Verbindung öffnen
//Exportparameter initialisieren
importing[0].name =  "RFCSI_EXPORT"; //Name des Parameters
importing[0].nlen =    12;  //Länge des Namens len("RFCSI_EXPORT")
//Zeiger auf Puffer, in dem R/3 die Info ablegt
importing[0].addr =   &rfcsi;
importing[0].leng =   sizeof(rfcsi); //Laenge des Puffers
importing[0].type =   RFCTYPE_CHAR;  //Typ des Parameters
importing[1].name = NULL;            //keine weiteren Exportparameter
exporting[0].name = NULL;            //keine Importparameter
tables[0].name    =   NULL;          //keine Tabellenparameter

//Aufruf des Funktionsbausteins mit den Parametern
rfc_rc = RfcCallReceive( handle, "RFC_SYSTEM_INFO", exporting,
importing, tables, &exception_ptr ); //Aufruf des Funktionsbausteins
if( rfc_rc != RFC_OK )     //Aufruf erfolgreich ?
{
  if ((rfc_rc == RFC_EXCEPTION) ||     (rfc_rc == RFC_SYS_EXCEPTION))
                rfc_error( exception_ptr ); //Ausnahme in Fkt.baustein
  else          rfc_error( "RfcCallReceive" ); //Ausnahme in RFC-C-Lib
}
else {  display_rfcsi( &rfcsi ); //Struktur mit Info ausgeben }
```

Abbildung 2.9: RFC-Automation: Aufruf eines Funktionsbausteins

- Synchroner, asynchroner und transaktionaler Aufruf von Funktionsbausteinen
- Entgegennahme von Aufrufen aus dem R/3 und Abbildung auf Funktionen der externen Anwendung

Die Unterschiede zwischen der internen[7] und externen Schnittstelle für Funktionsbausteine bestehen darin, daß als Parameter keine Changingparameter unterstützt werden und daß durch den Funktionsbaustein erzeugte Ausnahmen als Zeichenketten an das aufrufende Programm übergeben werden. Weiterhin muß im Programmtext explizit eine Verbindung zum entfernten R/3-System geöffnet werden.

Einen beispielhaften Aufruf eines Funktionsbausteins zeigt der Programmausschnitt eines C-Programms in Abbildung 2.9, welcher synchron den Funktionsbaustein **RFC_SYSTEM-**

[7] Aufruf des Funktionsbausteins aus einem ABAP-Programm

_INFO ruft, der technische Eigenschaften des gerufenen Systems als Ergebnis zurückgibt. Dies sind z.B. die Systemnummer und der Rechnername, auf dem der Applikationsserver läuft.

2.6.2 SAP-Automation

Die Schnittstelle SAP-Automation ist aus der Notwendigkeit heraus entstanden, daß es externen Anwendungen möglich sein muß, SAP-Transaktionen fernzusteuern und somit eine alternative Benutzerschnittstelle zum R/3 realisieren zu können. Alternative Benutzerschnittstellen sind insofern relevant, als daß die Bedienung von SAP-Transaktionen für nicht speziell geschulte Anwender zu kompliziert ist. Weiterhin wird es v.a. mit dem Aufkommen internetbasierter Anwendungen und Kioskanwendungen notwendig, Zugriff auf das R/3 zu erhalten, ohne dessen Benutzerschnittstelle SAPGUI nutzen zu müssen.

Die SAP-Automation-Schnittstelle arbeitet auf dem Datenstrom zwischen dem SAPGUI-Prozeß und dem Applikationsserver. Im Gegensatz zur RFC-Automation sind die Betrachtungsgegenstände nicht Funktionsbausteine, sondern die Dynpros[8] der SAP-Transaktionen. Die SAP-Automationsschnittstelle gestattet es, programmgesteuert SAP-Transaktionen im R/3 auszuführen. Die SAP-Automation bietet die Bearbeitung einzelner Dynpros, sowie in Abhängigkeit von dem vom R/3 erhaltenen Antwortdynpro die dynamische Änderung des Ablaufs der SAP-Transaktion. Die Anwendungslogik der SAP-Transaktionen steht somit vollkommen zur Verfügung.

Die Schnittstelle wird durch Programmierbibliotheken für die *Windows*-Plattform zur Verfügung gestellt und ist damit grundsätzlich erst einmal nicht plattformunabhängig. Für andere Plattformen besteht aber über den *GUI Terminal Server* die Möglichkeit auf ein *Windowssystem* zuzugreifen und dessen SAP-Automations-Schnittstelle zu nutzen. Der Zugriff kann dabei entweder über eine serielle Verbindung oder aber über eine *Sockets-Netzwerkverbindung* geschehen, so daß diese Schnittstelle plattformübergreifend wird. Die dabei verwendete Programmiersprache ist eine untypisierte Makrosprache, deren Kommandos wiederum auf die Primitive der SAP-Automationsschnittstelle abgebildet werden.

Die Kommunikation mit dem R/3 über den SAPGUI wird durch eine Protokollkomponente (itsgui.dll) realisiert, die sowohl durch den SAPGUI (front.exe) als auch durch die SAP-Automationsschnittstelle (merlin.dll) genutzt wird. Die SAP-Automations-Schnittstelle stellt die Daten, die zwischen der Protokollkomponente und dem R/3-Applikationsserver ausgetauscht werden, durch die Struktur **IT_EVENT** dar. Diese Struktur bildet die grafischen Elemente des aktuell bearbeiteten Dialogschrittes ab. Versenden und Manipulation dieser Struktur durch den SAP-Automation-Klienten simuliert dem R/3 gegenüber ein SAPGUI-Prozeß und das Bearbeiten eines Dynpros einer SAP-Transaktion.

Die Schnittstelle besitzt zwei elementare Funktionen: Die Funktion **It_GetEvent** empfängt eine **IT_EVENT**-Struktur vom Applikationsserver. Analog dazu versendet **It_SendEvent** die besagte Struktur, nachdem diese durch die Anwendungslogik manipuliert wurde. Abbildung 2.10 zeigt die Definition der **IT_EVENT**-Struktur.

Die **IT_EVENT**-Struktur besteht aus:

[8] Als Dynpro wird eine Bildschirmmaske eine SAP-Transaktion bezeichnet

```
typedef struct IT_EVENT_t {
 long cbSize;   //Größe der Struktur
 long Version;  //Versionsnummer der Struktur
 long eventtype;//Ereignistyp: z.B. Bildschirm, Funktionstaste,  Menü
 HANDLE hMerlin; // Verbindungshandle
 IT_SCREEN screen;//Steuerelemente des Dynpros
 IT_PFKEYS pfkeys;//Struktur, die gedrückte Funktionstaste beschreibt
 IT_MENUS menus;//Struktur, die Menü beschreibt
union {
int key;        //Code für Taste
HANDLE hMenu;   //Menünachricht
long lPos;      // Scrollnachricht
      };
char okcode[MAX_OK];           //OK-Code
char szMessage[MAX_MESSAGE];   //Meldungstext bei Eventtyp Message
unsigned short  nDiagVersion;  //Version DIAG-Protokoll
char szDB[128];                //Name der R/3-Datenbank
char szCPU[128];               //Name der CPU
unsigned short  nModeNumber;   //Nummer des Modus
char szTCode[64];
char szUsername[64];           //Name des benutzen R/3-Users
char szClient[64];             //Mandant
char szNormTitle[MAX_TITLE];   //Titel des Dynpros
int iSessionId;
                        } IT_EVENT;
```

Abbildung 2.10: SAP-Automation: Die **IT_EVENT**-Struktur

- einem Handle für den *SAP Automation-Thread*
- Felder mit Datenstrukturen, welche jeweils die auf dem Dynpro befindlichen Dynproelemente (**screen**) mit ihren Eigenschaften und aktuellen Werten repräsentieren, die verfügbaren Funktionstasten (**pfkeys**) und Menüs (**menus**).
- einem Ereignistyp, der u.a. angibt, ob die Struktur eine Dynprodefinition enthält, ob sie einen Funktionscode (durch Wahl eines Menüeintrags) oder den Druck einer Funktionstaste darstellt
- einem Wert, der den Funktionscode, den gedrückten Menüeintrag oder Funktionstaste darstellt

Die Entwicklung von Anwendungen auf der Ebene von Dynpros kann sehr zeitaufwendig und fehlerträchtig sein, da auf einem sehr niedrigen Abstraktionsniveau (Dynpros) gearbeitet wird. Um diesen Aufwand zu vermindern, existiert der *GUI Code Generator*, der es erlaubt, aus einer aufgezeichneten SAP-Transaktion *Visual-Basic*-Programmtext zu generieren, der sich dann individuell anpassen läßt. Den dynamischen Ablauf einer Kommunikation über die SAP-Automation Schnittstelle zeigt Abbildung 2.11.

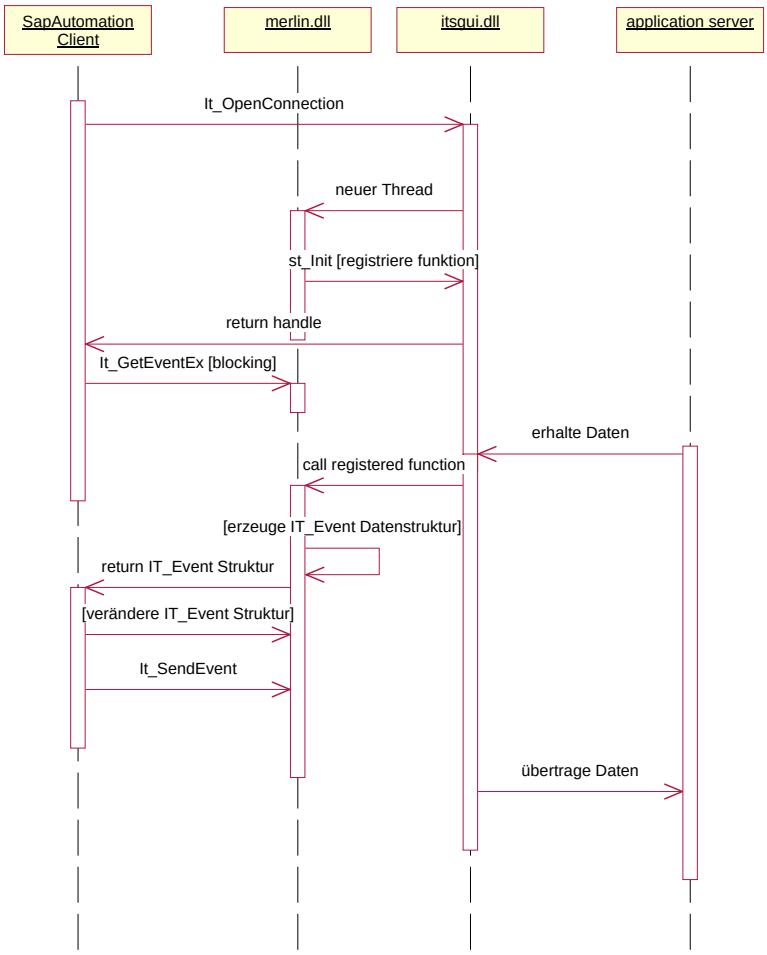

Abbildung 2.11: Dynamischer Ablauf einer SAP-Automation-Kommunikation

2.6.3 RFC- und SAP-Automation im Vergleich

Die RFC-Automation stellt eine der wichtigsten Schnittstellen für Fremdsysteme dar, die auf das R/3 zugreifen wollen. Die Relevanz der Schnittstelle wird zusätzlich durch die Tatsache unterstrichen, daß die im Abschnitt 5.1 beschriebenen Ansätze sich sämtlich auf diese Schnittstelle abstützen.

Durch die Vielzahl der zur Verfügung stehenden Funktionsbausteine ist eine große Menge der R/3-Funktionalität über die RFC-Automation zugreifbar. Durch die typisierte Schnittstelle von Funktionsbausteinen ist weiterhin ein sicherer Aufruf möglich. Trotz der Vielzahl von Funktionsbausteinen existiert immer noch eine große Menge an R/3-Funktionalität, die nicht durch vorhandene Funktionsbausteine abgedeckt wird. Diese Funktionalität ist v.a. innerhalb von SAP-Transaktionen vorhanden. Die RFC-Automation gestattet zwar, Transaktionen zu rufen und fernzusteuern[9] und so die Anwendungslogik der SAP-Transaktionen zu nutzen; der Ablauf der Transaktion ist aber dort a priori festgelegt, da dem R/3 die Inhalte der von der SAP-Transaktion prozessierten Dynpros als Tabelle beim Aufruf übergeben werden müssen.

Das Schreiben eigener Funktionsbausteine, welche die Funktionalität der SAP-Transaktionen bereitstellen, ist so ohne weiteres nicht möglich, da der Programmtext der SAP-Transaktionen nur rudimentär dokumentiert ist und sehr unübersichtlich ist. Insbesondere die betriebswirtschaftlichen Prüfungen in der Transaktionslogik werden zudem stark durch die *Customizing*-Einstellungen beeinflußt und erschweren zusätzlich das Verständnis des Programmtextes.

Der Ansatz der SAP-Automation stellt mit seinem Fokus auf die Dynpros der SAP-Transaktionen auf den ersten Blick eine interessante Alternative zu Anforderungen dar, in den die Funktionalität des R/3 nur über die SAP-Transaktion und nicht über einen Funktionsbaustein zur Verfügung steht[10]. Insbesondere kann hier interaktiv auf die SAP-Transaktionen zugegriffen werden. Die Probleme dieser Schnittstelle ergeben sich jedoch aufgrund des niedrigen Abstraktionsgrades (Dynpros von SAP-Transaktionen). Dynpros stellen keine stabile Schnittstelle zum R/3 dar. Die Probleme beginnen bereits an der Stelle, an dem über die Möglichkeiten des *Customizings* Felder und Dynpros ausgeblendet oder in ihren Eigenschaften verändert werden. Die Probleme setzen sich bei unterschiedlichen Releases des R/3 und verschiedenen Anmeldesprachen fort.

Abschließend läßt sich sagen, daß die RFC-Automation seit Release 2.1 die dominierende Schnittstelle zur Integration mit dem R/3 ist und bleiben wird. Die SAP-Automation eignet sich für eng umrissende, nicht über die RFC-Automation realisierbare Problemstellungen. Die Änderungen an den Dynpros im Verlauf der R/3-Versionsstände als auch die Möglichkeiten, diese Dynpros über das *Customizing* zu modifizieren, machen die Programmpflege existierender SAP-Automation-Anwendungen sehr schwierig. Die Entwicklung von Anwendungen basierend auf SAP-Automation ist ohne die Nutzung des Quelltextgenerators für *Visual Basic* sehr fehleranfällig. Dessen Funktionalität steht jedoch leider nur *Windows*-Plattformen zur Verfügung.

[9] Über den Aufruf des Funktionsbausteins **RFC_CALL_TRANSACTION**

[10] Dies ist ein sehr häufiger Fall.

2.7 Anpassung von R/3 an kundenspezifische Anforderungen

SAP als komplexe betriebswirtschaftliche Standardsoftware deckt mit seinem modulartigen Aufbau eine Vielzahl von betrieblichen Anforderungen ab. Trotz der großen Funktionsvielfalt der Module ist es unumgänglich, R/3 an einen Betrieb anzupassen. Anpassung oder Erweiterung von R/3 ist auch dann erforderlich, wenn bereits etablierte Systeme einzubinden sind oder Software für spezielle Anwendungsbereiche in das R/3-System integriert werden soll. Für ein Unternehmen stellt sich daher die Frage, wie umfangreich Anpassungen durchgeführt werden und außerdem, ob und im welchem Umfang Erweiterungen notwendig sind. Im Gegensatz zu inidvidueller, auf ein Unternehmen abgestimmter Software, ist Standardsoftware nachträglich auf die Bedürfnisse einer Unternehmung einzustellen.

Bei der Auswahl der einzusetzenden Komponenten des R/3-Systems besteht die Möglichkeit statt einer bestimmten Komponente andere Software einzusetzen und diese in das R/3-System zu integrieren. Die Auswahl der jeweils geeignetesten Software für ein Einsatzgebiet statt die Mitbenutzung einer Komponente der Standardsoftware, die den Anforderungen nicht genügt, kann die Qualität der gesamten eingesetzten Software erhöhen. Um aber die gleiche Qualität der Integration zu erreichen, wie sie die R/3-Komponenten untereinander haben, ist ein hoher Aufwand zu betreiben, der bei einer Beschränkung auf R/3-Komponenten nicht anfällt. Die Zeit, die durch die höhere Qualität und die individuelle Anpassung der dann im eingesetzten System abgebildeten Prozesse gewonnen wird, kann langfristig den zu betreibenden Aufwand rechtfertigen. In Abschnitt 2.7.4 wird diese Problematik genauer behandelt.

Zusammengefaßt gibt es vier Arten R/3 an kundenspezifische Anforderungen anzupassen bzw. zu erweitern:

1. Anpassung von R/3 mit Hilfe des *Customizing*.
2. Erweiterungen des R/3-Standards durch Customer-Exits und ABAP Dictionary Elementen
3. Anwendungsentwicklung in der in R/3 integrierten Programmiersprache ABAP.
4. Anwendungsentwicklung in einer anderen Sprache als ABAP und Nutzung der offenen Schnittstellen zum R/3.

Auf die beiden letzten Arten, der Anwendungsentwicklung, wird in Abschnitt 2.7.4 eingegangen. Im folgenden werden die ersten beiden Möglichkeiten beschrieben, wobei die Grenzen dieser R/3 eigenen Verfahren verdeutlicht werden. Der folgende Abschnitt über das *Customizing* orientiert sich sehr eng an [SAP 98f].

2.7.1 Anpassungen des R/3 über das Customizing

Unter *Customizing* faßt SAP eine Methodik zusammen, die bei der Einführung des Systems, bei Erweiterungen und bei einem Systemwechsel bzw. einer Systemerneuerung Unterstützung leistet. Empirische Untersuchungen zeigen, daß in den meisten Betrieben ein *Customizing* durchgeführt wird [König, Buxmann 97], so daß eine genauere Betrachtung der Möglichkeiten und vor allem der Grenzen des *Customizing* angebracht ist:

Das *Customizing*

- liefert mit dem Vorgehensmodell den Strukturplan für die Einführung und Erweiterung des R/3-Systems
- gibt Empfehlungen für die Systemeinstellungen und bietet Werkzeuge für die Systemeinstellungen und deren Dokumentation, um diese umzusetzen
- liefert mit dem *Customizing*-Projekt Steuerungsinstrumente für die Verwaltung, die Bearbeitung und die Auswertung von Einführungs- oder Erweiterungsprojekte
- unterstützt die Übernahme der Systemeinstellungen vom Testsystem in das Produktivsystem
- liefert Werkzeuge für den System-Upgrade und den Release-Wechsel (Transportsystem)

Das *Customizing* selber geschieht nicht durch ein Programmieren im eigentlichen Sinne, sondern dadurch, daß Tabelleneinträge gepflegt werden. Die Komponenten von R/3 reagieren dabei auf Änderungen in diesen Tabellen. Dadurch wird verhindert, daß der Programmcode selber verändert werden muß. Wegen der Komplexität des *Customizing* wird es in Unternehmen oftmals mit Hilfe spezialisierter Unternehmensberater durchgeführt. Aufgrund dieser Komplexität wird das *Customizing* u.a. unterstützt durch das in den folgenden Abschnitten beschriebene Vorgehensmodell und den Einführungsleitfaden.

2.7.1.1 Das Vorgehensmodell

Das Vorgehensmodell ist das wesentliche Element des *Customizing*. Es strukturiert R/3-Einführungsprojekte, indem es die Einführung in vier Phasen untergliedert. Jede dieser Phasen beschreibt grundsätzlich die auszuführenden Tätigkeiten und erfordert am Ende eine Qualitätssicherung, bevor die nächste Phase begonnen werden kann.

In jeder Phase wird das von der SAP ausgelieferte System weiter spezialisiert und auf ein Unternehmen eingestellt, so daß am Ende des *Customizing* ein einsetzbares, für ein bestimmtes Unternehmen konfiguriertes, System hergestellt ist. Die vier Phasen gliedern sich folgendermaßen:

Organisation und Konzeption In der ersten Phase des Vorgehensmodells wird ein Sollkonzept erarbeitet. Mit Hilfe eines von der SAP ausgelieferten Referenzmodells, welches auf einer betriebswirtschaftlichen Ebene den Leistungsumfang und die Geschäftsprozesse der R/3-Komponenten beschreibt, muß herausgefunden werden, wie die unternehmensspezifischen Prozesse und Funktionen mit den R/3-Anwendungskomponenten unterstützt werden, um die Unternehmensziele und die dafür optimale Aufbauorganisation und Ablauforganisation abzubilden.

Andere durchzuführende Tätigkeiten in dieser Phase sind das Entwerfen der Schnittstellen und Systemerweiterungen, die Schulung der Projektmitarbeiter und das Einrichten eines Testsystems, an dem alle wichtigen Abläufe und ausgewählten Funktionen in der zweiten Phase überprüft werden.

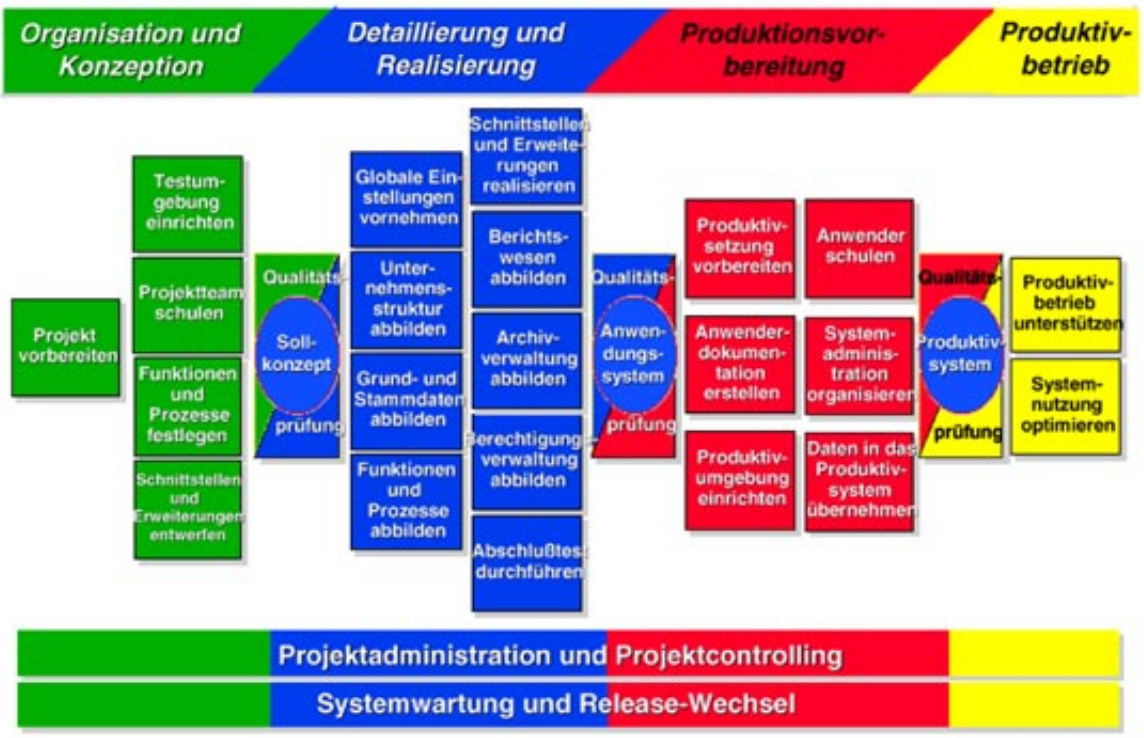

Abbildung 2.12: *Customizing* R/3: Das Vorgehensmodell

Detailierung und Realisierung Das Ergebnis dieser Phase ist die Umsetzung des in der vorherigen Phase erarbeiteten Sollkonzeptes. Das Testsystem aus der vorherigen Phase wird evaluiert und ausgiebig getestet, so daß am Ende dieser Phase ein Produktivsystem hergestellt ist, das bis auf der in Phase drei durchzuführenden Tätigkeiten schon dem endgültigen eingesetzten System entspricht.

Produktionsvorbereitung Das in Phase zwei erarbeitete System wird vervollständigt, d.h., daß u.a. die Datenübernahme geplant und durchgeführt wird, aber auch Benutzer eingerichtet und Schulungen der Anwender durchgeführt werden. Als Ergebnis dieser Phase wird das System geprüft und freigegeben.

Produktionsanlauf Das Ergebnis der Phase Produktionsanlauf ist die Sicherstellung eines reibungslosen Produktivbetriebs einschließlich einer Betreuung der Anwender bei der Systemnutzung und ein weiter optimierter Systemeinsatz. Jede dieser vier Phasen ist sehr komplex und besteht aus einer Vielzahl von Schritten, die durchgeführt werden müssen. Unterstützung für die Bearbeitung der Phasen bietet der Einführungsleitfaden, dessen Funktionalität im nächsten Abschnitt beschrieben wird.

2.7.1.2 Der Einführungsleitfaden

Der Einführungsleitfaden ist ein umfassendes Informationssystem, das alle notwendigen Aktivitäten mit ausführlichen Erläuterungen und Hinweisen beschreibt. Durchzuführende Aktivitäten werden aufgelistet, bereits durchgeführte Schritte werden als „erledigt“ gekennzeichnet. Den Aktivitäten sind Transaktionen für das *Customizing* zugeordnet. Der Benutzer kann direkt aus der Liste eine Aktivität auswählen und startet die entsprechende Anwendung, die für die Ausführung der Aktivität benötigt wird. Zu jeder Aktivität existiert eine ausführliche Dokumentation, die auch durch den Einführungsleitfaden bereitgestellt wird. Der Einführungsleitfaden ist somit das zentrale Element des *Customizing*, um dieses geordnet durchzuführen.

Da das *Customizing* sehr spezifisch für eine Unternehmung ist und es sowohl für die Einführung, als auch für die Systemerweiterung zuständig ist, gibt es verschiedene Ebenen von Einführungsleitfäden, die sich durch die Einstellung von diversen Parametern vom SAP-Referenz-IMG ableiten (IMG = Implementation Guide):

SAP-Referenz-IMG Der SAP-Referenz-IMG enthält die möglichen Systemeinstellungen und durchzuführende *Customizing*-Aktivitäten aller Komponenten. Aus ihm wird der Unternehmens-IMG generiert, nachdem die benötigten Anwendungen und die Länder ausgewählt sind, in denen das R/3-System eingesetzt werden soll.

Im Referenz-IMG können globale Felder eingestellt und Transaktionsvarianten erstellt werden: Mit Transaktionsvarianten können Transaktionsabläufe durch die Vorbelegung von Feldern mit Werten, durch Ausblenden und Ändern der Eingabebereitschaft von Feldern und durch das Ausblenden ganzer Bilder vereinfacht werden. Bildschirmbilder mit ausgeblendeten Feldern zeigen nur die sichtbaren Felder an und passen ihre Größe der anzuzeigenden Felder an. Außerdem können Felder systemweit (globale Felder) mit Werten vorbelegt werden. Sie können ebenso ausgeblendet oder deaktiviert werden.

Unternehmens-IMG Der Unternehmens-IMG faßt alle durchzuführenden Aktivitäten für eine Unternehmung zusammen und ist somit die Referenz für das *Customizing* einer Unternehmung. Er dient als Ausgangspunkt für die Generierung der Projekt-IMGs, welche diesen weiter unterteilen. Der Unternehmens-IMG ist nur einmal im System vorhanden und wird aus dem SAP-Referenz-IMG generiert. Er ist eine Teilmenge vom SAP-Referenz-IMG. Falls sich die bei der Generierung des Unternehmens-IMGs eingestellten Anforderungen geändert haben, kann er aus dem SAP-Referenz-IMG neu generiert werden.

Projekt-IMG Aus dem Unternehmens-IMG wird für jedes *Customizing*-Projekt jeweils ein Projekt-IMG generiert, der wiederum eine Teilmenge des Unternehmens-IMG ist. Er dient der weiteren Unterteilung des *Customizings*. Für die Generierung des Projekt-IMG müssen wie bei der Erstellung des Unternehmens-IMG die einzustellenden Komponenten und die das Projekt betreffenden Länder ausgewählt werden. Er erhält dann nur die der Auswahl entsprechenden Aktivitäten.

Release-spezifischer IMG Der release-spezifische IMG wird für das Unternehmen auf Basis des Unternehmens-IMGs und für jedes *Customizing*-Projekt auf Basis des Projekt-IMG generiert. Der release-spezifische IMG faßt alle Aktivitäten aus dem Unternehmens-

IMG oder aus dem Projekt-IMG zusammen, die für das jeweilige *Release* oder *Update* nötig sind.

Für Eingriffe, die über das *Customizing* hinausgehen, bietet die SAP sogenannte Customer-Exits an, die es dem Kunden erlauben, an von der SAP vorgesehenen Stellen eigenen Programmcode einzufügen. Im folgenden Abschnitt werden die Möglichkeiten der Customer-Exits genauer beschrieben.

2.7.2 Erweiterung des R/3 durch Customer-Exits

Customer-Exits ermöglichen es, R/3 an von der SAP vordefinierten Stellen, d.h. in Programmen, Menüs und Bildschirmbildern, zu erweitern. Die SAP bezeichnet die Customer-Exits als „leere Modifikationshülsen“ [SAP 98h], die im Standard vorgedacht sind und mit kundenspezifischer Logik ausgestaltet werden können. Die aufwärtskompatiblen Customer-Exits ermöglichen eine strikte Trennung von SAP- und Kundenlogik. Erweiterungswünsche können bei der SAP beantragt werden, die dann entweder die gewünschte Funktionalität in der nächsten Systemerneuerung implementiert oder dem Kunden an den erforderlichen Stellen neue Customer-Exits anbietet. Customer-Exit stellen die einzige Möglichkeit dar, R/3 versionsunabhängig zu erweitern. Für alle Customer-Exits ist hervorzuheben, daß sie nicht für die Implementierung beliebiger Funktionalität bestimmt sind. Dies liegt an den Eigenschaften der verschiedenen Arten von Customer-Exits, die im folgenden genauer beschrieben werden.

2.7.2.1 Funktionsbaustein-Exits

Funktionsbaustein-Exits dienen dazu, bestimmte von der SAP vorgedachte Funktionalitäten, die das R/3-System an genau festgelegten Stellen erweitern, zu implementieren. Da die Funktionsbaustein-Exits einen vordefinierten Zweck erfüllen sollen, besteht oftmals die Möglichkeit, Beispielcode zu übernehmen, der in das eigene Programm transferiert und an die eigene Bedürfnisse angepasst werden kann. Die Existenz von Beispielcode verdeutlicht, daß Funktionsbaustein-Exits normalerweise für einen streng definierten Aufgabenbereich vorgesehen sind. Funktionsbaustein-Exits sind Aufrufe von Funktionsbausteinen an festgelegten Stellen der Anwendungslogik und erfordern eine Programmierung in ABAP. Die aufgerufenen Funktionsbausteine sind bereits von der SAP angelegt. Sie enthalten eine *include*-Anweisung und können selbst nicht verändert werden. Die *include*-Anweisung hat als Parameter einen Programmnamen und sorgt dafür, daß an dieser Stelle, ein zugehöriges Programm aufgerufen wird, wenn dieses zuvor vom Kunden angelegt wurde. Die *include*-Anweisung erzeugt ohne gültigen Parameter (d.h. bei nicht vorhandenem Programm) keinen Laufzeitfehler und ist damit geeignet Änderungsmöglichkeiten optional anzubieten. Da die SAP eine strikte Trennung von Namensräumen vornimmt und es nicht möglich ist, Programme mit einem Namen anzulegen, der im SAP Namensraum angesiedelt ist, hat die SAP bereits einen Namen für dieses Programm gewählt, der den Bedingungen für den Kundennamensraum entspricht. Das Erstellen eines Programms mit dem von der SAP definierten Namen ermöglicht also die Nutzung des zugrundeliegenden Funktionsbaustein-Exit.

Durch die von der SAP eingefügten Funktionsbausteinaufrufe ist auch die Schnittstelle eines solchen Aufrufes festgelegt. Die Ein- und Ausgabeparameter sowie Tabellenparameter, die

das vom Kunden zu erstellende Programm verarbeiten kann, sind nicht veränderbar. Unter Umständen ist man durch dieses von der SAP auferlegte Korsett nicht in der Lage, beliebige Funktionalität, d.h. andere oder zusätzliche Funktionalität, als von der SAP geplant wurde, zu implementieren. Um dennoch eigene Daten an den Funktionsbaustein zu übergeben bzw. eigene Ergebnisse aus dem Funktionsbaustein zu sichern, müssen globale Felder benutzt werden. Die Benutzung von globalen Feldern stellt programmiertechnisch keine elegante Lösung dar, ist aber die einzige Möglichkeit die engen Grenzen eines Funktionsbaustein-Exit zu erweitern.

Funktionbaustein-Exits nehmen unter den Customer-Exit eine Sonderrolle ein, da sie auch im Zusammenhang mit Menü-Exits und Bild-Exits benutzt werden. Nur sie ermöglichen es, eigene Programmierlogik in den standardisierten Code einzufügen.

2.7.2.2 Menü-Exits

Menü-Exits sind vorhandene nicht aktivierte Menüeinträge, die unsichtbar für den Benutzer sind. Ein Menü-Exits wird erst sichtbar, nachdem er aktiviert wird. Der Text eines solchen Menüeintrages ist frei wählbar. Es existieren Menü-Exits mit und ohne zugehörigem Funktionsbaustein-Exit:

Menü-Exits mit Funktionsbaustein-Exit Nach der Aktivierung des Menü-Exits wird bei Auswahl des daraufhin angezeigten Menüpunktes der zugehörige Funktionsbaustein-Exit angesprochen. Um einen solchen Menü-Exit nutzen zu können, muß also auch ein Funktionsbaustein-Exit bearbeitet werden. Dabei ist es oft üblich, daß mehrere Menü-Exits einem Funktionsbaustein-Exit zugeordnet sind. Der Programmierer muß also durch eine Abfrage innerhalb des Funktionsbausteins auf die jeweilige Menüauswahl reagieren. Jedem Menüpunkt hinterliegt ein Funktionscode, der in der Programmierung für die Bestimmung des gewählten Menüeintrages benutzt wird, um individuell auf eine Auswahl reagieren zu können.

Menü-Exits ohne Funktionsbaustein-Exit Mit Hilfe dieser Menü-Exits läßt sich die angezeigte Menüstruktur verändern. Nach dessen Aktivierung erscheint ein neuer Menüeintrag im „alten“ Standard-Menü. Nach Auswahl dieses Menüeintrages wird das angepaßte Menü angezeigt. Auf diese Weise kann keine neue Funktionalität angeboten werden, als die, welche in der originalen Menüstruktur vorgesehen war. Im Gegensatz zu den Menü-Exits mit Funktionsbaustein existiert kein zugehörigen Funktionsbaustein, der bei der Auswahl eines neuen Eintrages aufgerufen werden kann. Der Nutzen dieser Art von Menü-Exits liegt also in der möglichen Umstrukturierung des Standard-Menüs, so läßt sich zum Beispiel die Übersichtlichkeit eines Menüs verbessern, indem nur noch die Einträge angezeigt werden, die innerhalb einer Unternehmung genutzt werden.

2.7.2.3 Bild-Exits

Bild-Exits können dafür genutzt werden, in den Bildschirmbildern von der SAP vorgesehene Bereiche selbst zu gestalten. Die hierzu nötigen Felder[11] werden auf einem eigenen Bildschirm-

[11] Bildschirmbilder bestehen aus Feldern, denen unterschiedliche Attribute zugeordnet werden, z.B. Bearbeitungsfeld, Anzeigefeld, Auswahlfeld usw.

bild (Subscreen) angeordnet.

Bild-Exits sind nur im Zusammenhang mit Funktionsbaustein-Exits nutzbar, da den eingefügten Feldern globale Variablen aus dem aufrufenden Programm nicht bekannt sind. Um dennoch die Werte aus dem umgebenden Programm nutzen zu können, existieren in den PBO/PAI-Modulen Funktions-Exits. Diese werden dazu benutzt, bestimmte Werte an die eigene Programmlogik des *Subscreens* zu übertragen bzw. daraus zu übernehmen. Da auch hier Funktionsbaustein-Exits benutzt werden, sind alle mögliche Parameter bereits von der SAP implementiert und können nicht verändert oder angepaßt werden.

2.7.3 Erweiterung von Datenelementen und Strukturen

Datenelemente dienen der Festlegung von Feldeigenschaften. Zu den Feldinformationen, die in den Datenelementen enthaltenen sind, zählen u.a. Kurztext, Schlüsselworttext, Dokumentation und Hilftexte. Bei der Bildschirmausgabe werden Felder nach den Infomationen ihres zugehörigen Datenelementes entsprechende dargestellt. Es gibt folgende Änderungs- bzw. Erweiterungsmöglichkeiten, die an Datenelementen durchführbar sind:

Ändern der Dokumentation Die Dokumentation, die über die Hilfefunktion zu einem Feld aufgerufen werden kann, ist veränderbar. Dabei ist es möglich, den originalen Hilfetext wegzulassen, abzuändern oder einen Bezug zum originalen Hilfetext anzubieten. Letzteres wird von der SAP empfohlen, da der Bezug auch bei einem Systemwechsel erhalten bleibt. Es ist außerdem generell möglich, über Zusätze zu Datenelementen einen an ein Bildschirmbild gebundenen Hilfetext anzubieten. Durch die Anpassung eines Hilfetextes können z.B. unternehmenspezifische Eigenschaften für die einzugebenden Werte erklärt werden.

Schlüsselwort-Exits Schlüsselwort-Exits beziehen sich auf Datenelemente. Datenelemente besitzen u.a. einen Schlüsselworttext und einen Kurztext, welche sich über einen Schlüsselwort-Exit ändern lassen.

Schlüsselworttexte sind Bezeichnungen der einzelnen Felder auf einem Bildschirm, z.B. „Kundennummer“ und „Buchungskreis“. Durch die Änderung von Schlüsselworttexten ist es möglich, Bezeichnungen auf den Bildschirmbildern an eigene Bedürfnisse anzupassen. Ein Kurztext eines Datenelementes wird im Hilfesystem benutzt und auf Anforderung des Benutzers angezeigt. Ein Kurztext kann zum Beispiel das Feld "Kundennummer" mit einigen Erläuterungen versehen, welche die Eigenschaften der einzugebenden Kundennummer spezifizieren. Dieser Kurztext läßt sich ebenso an den Sprachgebrauch eines Unternehmens anpassen.

Änderungen, die mit Hilfe von Schlüsselwort-Exits vorgenommen wurden, wirken sich immer global auf das System aus. Schlüsselwort-Exits brauchen nicht aktiviert werden, d.h. Änderungen stehen sofort zur Verfügung. Bei einem Systemwechsel muß manuell eine Transaktion gestartet werden, welche die Schlüsselwort-Exits erneut erstellt. Schlüsselwort-Exits unterscheiden sich in der Hinsicht von den anderen Exits, da sie weder den SAP-Erweiterungen noch einer Entwicklungsklasse zugeordnet sind[12].

[12] Ab der Version 4 sind die Schlüsselwort-Exits den Customer-Exits zugeordnet. Es ist anzunehmen, daß die SAP in Zukunft auch die Schlüsselwort-Exits dem einheitlichen Konzept der Projektverwaltung für Customer-

Feld-Exits Feld-Exits sind Funktionsbausteine, die zu einem Datenelement erzeugt werden können. Der Funktionsbaustein wird mit dem eingegebenen Wert als Parameter aufgerufen. Innerhalb dieses Funktionsbausteines können Einschränkungen für ein Datenelemente vorgenommen werden, z.B. die Überprüfung von Gültigkeitsbereichen oder Berechtigungen einzelner Benutzer. Es werden zwei Arten von Feld-Exits unterschieden:

- **Globale Feld-Exits** sind nicht an ein Bildschirmbild gebunden. Der Funktionsbaustein wird für jedes Bildschirmbild aufgerufen, in dem das Datenelement benutzt wird.
- **Selektive Feld-Exits** müssen einzelnen Bildschirmbildern zugeordnet werden. Nur auf diesen Bildschirmbildern wird in den entsprechenden Funktionsbaustein verzweigt. Um unterschiedliche Funktionalität auf verschiedenen Bildschirmbildern mit gleichem Datenelementbezug zu bekommen, ist es möglich, pro Datenelement sogenannte Exitnummern zu vergeben. Hinter jeder Exitnummer steht dann ein eigener Funktionsbaustein.

Includes bei Tabelleninhalten (Tabellen-*Appends*) Tabellen im R/3 lassen sich auf zwei Arten erweitern: Einige Tabellen im ABAP *Dictionary* enthalten von der SAP definierte *Include*-Befehle für Kundenerweiterungen. Wie auch bei den meisten anderen Erweiterungen können keine eigenen *Includes* erstellt werden. *Include*-Befehle werden dazu benutzt, eigene Felder innerhalb dieses *Includes* anzulegen.

Im Gegensatz zu den von der SAP vorgedachten Erweiterungsmöglichkeiten für Tabellen über die *Include*-Befehle, lassen sich mit Tabellen-*Appends* alle Tabellen erweitern. Eine *Append*-Struktur wird genau einer Tabelle oder Struktur zugeordnet. Eine Tabelle kann mehrere *Append*-Strukturen besitzen. Da *Append*-Strukturen im Namensraum des Kunden angelegt werden, sind diese bei einer Systemerneuerung vor Änderungen geschützt.

2.7.4 Erfahrungen mit Eigenentwicklungen im R/3-System

Eigene Erfahrungen und Untersuchungen ([König, Buxmann 97], [Pressmar, Scheer 98]) zeigen, daß Anpassungen von R/3 an eigene betriebswirtschaftliche Bedürfnisse mit einem hohen finanziellen und zeitlichen Aufwand verbunden sind. Es gibt Ansätze der SAP (*Business Engineer* in [SAP 98e]) und anderen Herstellern (Siemens R/3 Live Kit in [Siemens 97]), vorkonfigurierte R/3-Systeme anzubieten, die mit einem reduzierten Aufwand auf einen Betrieb abgebildet werden können. Sie sollen die Komplexität des *Customizing* vereinfachen und vor allem die Einführung des R/3-Systems in ein Unternehmen beschleunigen.

Beliebige Erweiterungen und Anpassungen von R/3 allein durch das *Customizing* sind, wie auch aus dem vorherigen Abschnitt ersichtlich, nicht möglich. Das *Customizing* bietet nur einen beschränkten von der SAP definierten Handlungsspielraum. Nur durch die Entwicklung eigener Anwendungen in ABAP oder in einer anderen Programmiersprache sind beliebige Erweiterungen möglich. SAP R/3 ist durch die zahlreich vorhanden Schnittstellen (RFC-Automation, SAP-Automation) ein offenes System (siehe auch Abschnitt 2.6). Es läßt sich theoretisch jede gewünschte Funktionalität und jedes beliebige Produkt an R/3 ankoppeln.

Exits unterwerfen will. Zur Zeit scheinen die Schlüsselwort-Exits eher zufällig zu den Customer-Exits zu gehören

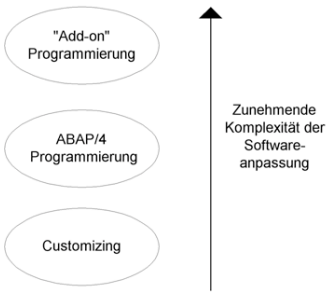

Abbildung 2.13: *Customizing* R/3: Vereinfachtes Schichtenmodell der Anpassungalternativen

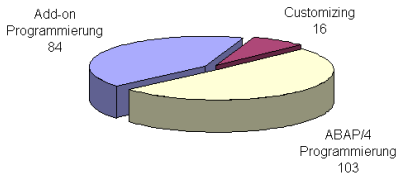

Abbildung 2.14: *Customizing* R/3: Absolute Häufigkeiten der durchgeführten Anpassungsmaßnahmen

In der Studie [König, Buxmann 97] wird gezeigt, daß dies in der Praxis auch umgesetzt wird, und die Anwendungsentwicklung bzw. die Integration von anderen Systemen durchgeführt wird. Die Studie beschäftigt sich mit der Frage, ob Standardsoftware flächendeckend oder nur für ausgewählte Bereiche eingesetzt werden soll: Von den Antworten der Unternehmen, die in dieser Studie ausgewertet wurden und die das R/3-System einsetzen, hat sich nur ein kleiner Teil (unter 10 Prozent) der Unternehmen auf das *Customizing* beschränkt (siehe Abbildung 2.14). Es werden drei Arten von Anpassungalternativen beschrieben (siehe Abbildung 2.13), die eine Vergröberung der am Anfang dieses Abschnittes beschriebenen vier Möglichkeiten der Anpassung darstellen: Das Customizing, die Programmierung in ABAP und die *Add-On*-Programmierung.

Die als *Add-On*-Programmierung bezeichnete Programmierung, welche die Entwicklung von Anwendungen bedeutet, die an das R/3 über die vom R/3 zur Verfügung gestellten Schnittstelllen angebunden werden, wurde von circa einem Drittel der Unternehmen durchgeführt. Es handelt sich bei den Eigenentwicklungen um die Anbindung fremder Systeme an R/3 oder um eine Erweiterung in einer beliebigen Programmiersprache. Weniger als 25 Prozent der Unternehmen würden sich rückblickend wieder für diese Art von Erweiterung entscheiden. Von den Unternehmen, die sich für eine Programmierung in ABAP entschieden haben, würden rückblickend nur noch circa 70 Prozent dies wiederholen. Die Studie kommt zu folgender Aussage: „daß die Einführungsprojekte, die die Alternative *Add-On*-Programmierung gewählt haben, zu einem erheblich vermehrten Aufwand führen, der sich in höheren Personalkosten sowie einer bedeutend längeren Einführungdauer niederschlägt.“ Dies wird bestärkt durch die Aussage der befragten Unternehmen: „daß sie zukünftig weniger komplexe Anpassungsmaßnahmen durchführen würden, da die Möglichkeiten einer besseren Aufgabenabdeckung durch das R/3-System aufgrund aufwendiger Erweiterungsmaßnahmen offenbar in keinem Verhältnis zum Aufwand steht, der für solche Anpassungen getätigt wird.“

Dem gegenüber steht der Wunsch der Unternehmen nach Integration an Stelle von nicht vereinbaren Insellösungen. Eine Vielzahl von Unternehmen hofft gerade durch die Einführung mehrerer Module[13] auf Synergieeffekte. Die Entscheidung für konzernweite Einführungen durch zentrale Instanzen in der Organisation, welche die gesamte Unternehmung im Auge

[13] 75 Prozent aller befragten Unternehmen nutzt mehr als drei Module.

behalten, verstärkt diesen Fokus noch.

Zusätzlich haben Hersteller anderer Software grundsätzlich das Problem der aufwendigen Integration. Aufgrund der Marktführerschaft von der SAP mit ihrem R/3-System liegt es speziell für Hersteller branchenspezifischer Produkte im eigenen Interesse, eine Integration mit dem R/3-System anbieten zu können. Ansonsten kämen sie bei einer eventuellen Erweiterung des R/3-Systems innerhalb einer Unternehmung durch Fremdprodukte, mit ihrem Produkt gar nicht erst zum Einsatz.

Daß die befragten Unternehmen den Abdeckungsgrad besonders von nicht so häufig eingeführten Modulen (MM, SD, PP) relativ schlecht bewerten (siehe Abbildung 2.15), läßt darauf schließen, daß andere Software zum Einsatz kommt, die in das R/3 integriert werden muß. Dabei ist es prinzipiell unerheblich, ob es sich dabei um einen Hersteller von Software handelt oder um ein Unternehmen, welches das R/3-System bei sich einführen möchte. Hier entsteht also ein Zwiespalt, der sich darin zeigt, daß Unternehmen einerseits das Bedürfnis haben, Fremdprodukte in das R/3-System zu integrieren, aber für die anderseits der Aufwand für diese Integration zu hoch ist. Dieser Zwiespalt wird in dieser Arbeit als Integrations-

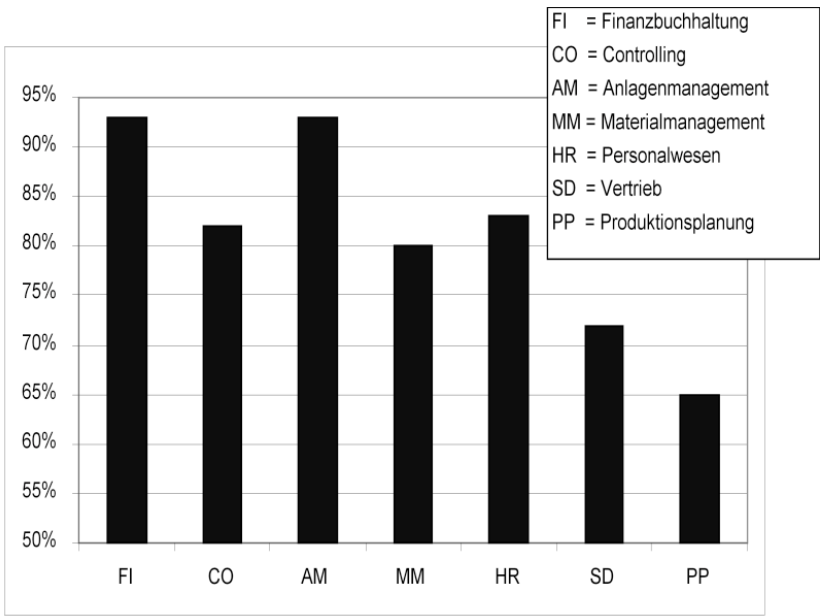

Abbildung 2.15: Abdeckungsgrad der Anforderungen durch R/3-Module

problem bezeichnet. Das Problem wird dadurch verstärkt, daß Erweiterungen, die über das *Customizing* hinausgehen, bei einem Releasewechsel nur mit erhöhtem Aufwand in das neue System zu überführen sind [Dömer 97]. Dies erzeugt zusätzlich Probleme, die man durch die Benutzung einer Standardsoftware vermeiden wollte.

Geschäftsobjekte versprechen die leichte Integration bzw. Verknüfung verschiedener Systeme und Nutzung der betriebswirtschaftlichen Konzepte. Welche Modelle hinter den Geschäftsobjekten stehen und ob sie zur Lösung des Integrationsproblems beitragen können, wird in den nächsten Kapiteln behandelt.

Kapitel 3

Geschäftsobjektkonzepte

Die Standardsoftware R/3, bestehend aus verschiedenen hoch integrierten Modulen, wird von der SAP werbewirksam als „flexibel und individuell anpassbar“ dargestellt [SAP 98b]. In Abschnitt 2.7 wird das *Customizing* beschrieben, durch welches das R/3 parametrisiert und an eine Unternehmung angepaßt werden kann. Die Anpassungsmöglichkeiten sind allerdings vorgedacht und können nur im geringen Ausmaß erweiterte Anforderungen erfüllen. In Abschnitt 2.7.4 zeigt sich, daß Erweiterungen (eigene Programmentwicklung) des R/3 zeit- und kostenintensiv sind und von einer zunehmenden Anzahl von Unternehmen augrund der schlechten Erfahrungen bei bisherigen Erweiterungen abgelehnt werden. Neben dem R/3 wird in den gleichen Unternehmen Software anderer Hersteller eingesetzt, so daß es durchaus einen Integrationsbedarf gibt. Die Tendenz geht zur Vereinfachung des *Customizing* und der Anpassung, löst aber das eigentliche Integrationsproblem nicht (siehe Abschnitt 2.7.4).

Die Erweiterung des R/3 und die Integration anderer Systeme in R/3 soll sich durch die Benutzung von Geschäftsobjekten (engl. *Business Objects*) einfacher gestalten [SAP 97a]. Geschäftsobjekte sind Gegenstand von Standardisierungsbemühungen verschiedener Hersteller und Organisationen, deren Umsetzung sich anders gestaltet. Eine Auswahl von verschiedenen Modellen wird im folgenden vorgestellt.

Behandelt werden die Geschäftsobjektansätze der Object Management Group (OMG) [OMG 99] und der Open Applications Group (OAG)[OAG 99]. Als Beispiel einer praxisnahen Umsetzung von Geschäftsobjekten wird das *San Francisco Framework* von IBM [SF 98] vorgestellt, deren Lösung durch die aktive Mitarbeit an dem Standardisierungsverfahren der OMG Ähnlichkeiten zu den Konzepten der OMG aufweist [Heiderich 98].

Neben diesen Ansätzen existieren noch eine Reihe von anderen Modellen für Geschäftsobjekte [1], die nicht weiter erwähnt werden. Die ausgewählten Ansätze haben für die Arbeit durch ihre Nähe zu SAP eine höhere Relevanz, da sich die SAP gemäß eigener Aussagen [SAP 97a] an den Standards der OAG und der OMG orientiert. Die SAP ist außerdem Mitglied beider Organisationen. Abschließend werden die Gemeinsamkeiten und Unterschiede aufgezeigt, bevor auf das Objektmodell und die Geschäftsobjekte der SAP im darauffolgenden Kapitel 4 eingegangen wird.

[1] (z.B. *Business Objects* nach Shelton [Shelton 97], Convergent Engineering [EE 97], *Business Objects* nach T. Erler [Erler 98])

3.1 Das Geschäftsobjekt-Konzept der OMG

Die Object Management Group (OMG) ist eine herstellerübergreifende und nicht kommerzielle Vereinigung mit dem Ziel, Objekttechnologie zu fördern, Komplexität und Kosten von Softwaresystemen zu reduzieren und die Weiterentwicklung von Softwareanwendungen zu beschleunigen. Die OMG stellt dafür die Architektur für die Verteilung und Zusammenarbeit objektorientierter Softwarebausteine in vernetzten, heterogenen Systemen bereit. Die OMG besteht aus mehr als 800 Mitgliedern zu denen staatliche Einrichtungen und namhafte Softwarehersteller wie z.B. SAP, IBM und Microsoft zählen. Die Standardisierungsbemühungen der OMG erstrecken sich auch auf den Bereich der Geschäftsobjekte. Aus diesem Grund wurde die *Business Objekt Domain Task Force* (BODTF) gegründet, die sich mit dem Einsatz von Geschäftsobjekten in Informationssystemen beschäftigt, mit dem Ziel, einen allgemein anerkannten Standard zu entwickeln.

Die OMG begründet den Bedarf an einem Geschäftsobjektkonzept folgendermaßen: Informationssysteme wachsen nicht mit dem Unternehmen, sind unflexibel und kostenintensiv, da deren Wartung und Pflege problematisch und zeitaufwendig ist. Bisherige Informationssysteme gestalten sich unübersichtlich und sind schwer vom Anwender zu verstehen. Aktuelle Applikationen entsprechen nicht den Anforderungen oder dem Geschäftsmodell einer Unternehmung, die dazu noch meistens geschlossene Umgebungen darstellen und bestehende Systeme schwer integrieren können [Group 96].

Durch die Wahl eines *multi-tiered Client/Server*-Modells, also die Trennung von Präsentations-, Anwendungs- und Datenbankebene, soll ein genaueres Abbild der Realität möglich sein. Geschäftsobjekte sollen auf dieser Basis eine *Plug-and-Play*-Fähigkeit besitzen, d.h. sie sind in ein laufendes verteiltes System integrierbar und können untereinander kommunizieren und kooperieren. Geschäftsobjekte sind unter Einhaltung der Kooperationsfähigkeit mit anderen Geschäftsobjekten unabhängig voneinander entwickelbar. Geschäftsobjekte lassen sich gezielt zu Applikationen zusammenstellen, die individuellen betriebswirtschaftlichen Anforderungen genügen und/oder einem speziellen Geschäftsmodell entsprechen. Geschäftsobjekte stellen selbständige wiederverwendbare Komponenten dar, die mittels Vererbung, Delegation, und Konfiguration in ihren Fähigkeiten erweitert bzw. angepaßt werden. Gleichzeitig ist das Geschäftsobjektmodell anderen Technologien nicht verschlossen, so ist das Modell datenbank- und programmunabhängig. Die Funktionalität bereits eingesetzter Softwaresysteme (*Legacy Applications*) kann durch Geschäftsobjekte abgebildet werden, indem Geschäftsobjekte die Funktionalität eines alten Systems kapseln und Methoden anbieten, welche die Funktionalität anderen Anwendungen zur Verfügung stellen.

Die OMG verspricht sich von diesen Anforderungen Zeitersparnis bei der Entwicklung und eine Erleichterung der Wartung und Pflege von Anwendungen. Geschäftsobjekte bieten eine betriebswirtschaftliche Sicht auf ein Informationsssystem und abstrahieren von darunterliegenden Technologien, wodurch sich die Entwicklung, Implementierung und Erweiterung der Geschäftsobjekte vereinfachen soll.

Ein Geschäftsobjekt wird von der OMG folgendermaßen definiert:

> A **business object** is defined as a representation of a thing active in the business domain, including at least its business name and definition, attributes, behavior,

> relationships, rules, policies and constraints. A business object may represent, for example, a person, place, event, business process or concept. Typical examples of business objects are: employee, product, invoice and payment [Group 96, Seite 19].

Eine Geschäftsobjektklasse besitzt demnach einen Namen zur eindeutigen Identifikation, Attribute zur Beschreibung ihres Zustandes sowie Methoden und Schnittstellen, die ihr Verhalten repräsentieren. Außerdem enthält es Beziehungen zu anderen Klassen und Regeln, die das Verhalten eines Geschäftsobjektes überwachen und den Rahmen vorgeben, innerhalb dessen es agieren und sich verändern kann. Ein Geschäftsobjekt stellt einen ausführbaren Softwarebaustein dar. Einem Geschäftsobjekt können beliebig viele visuelle Ausprägungen zugeordnet sein, die entsprechend dem jeweiligen Kontext das Objekt visuell präsentieren.

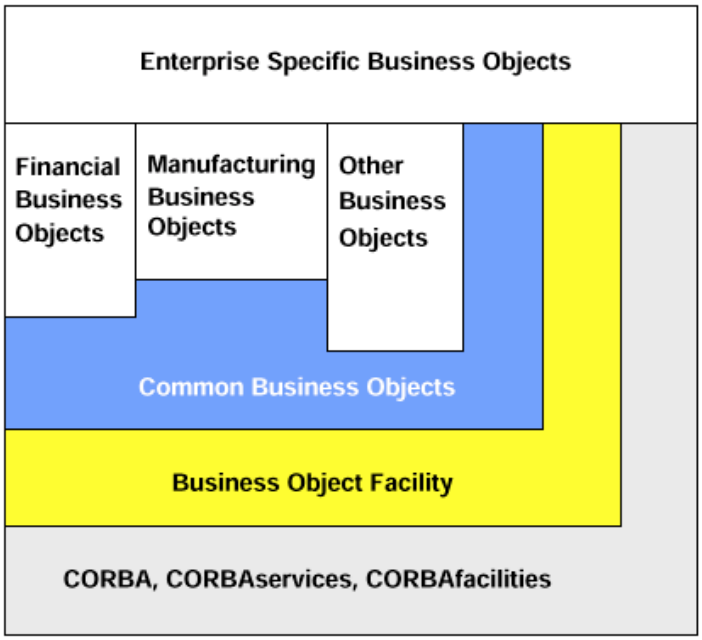

Abbildung 3.1: Das Schichtenmodell der OMG

Die OMG formuliert ein Schichtenmodell (siehe Abbildung 3.1), dessen wesentlichen Bestandteile die *Business Object Facility* (BOF) und die *Common Business Objects* sind. Die *Common Business Objects* setzen dabei ihrerseits auf den Standards der OMG zur Entwicklung von verteilten betriebswirtschaftlichen Anwendungen auf (**C***ommon* **O***bject* **R***equest* **B***roker* **A***rchitecture* (CORBA)):

Schicht 1 (CORBA): CORBA ist ein Standard der OMG für die Kommunikation in verteilten Umgebungen. Weitere Ziele des Standards sind Plattform- und Programmiersprachenunabhängigkeit. Wesentlicher Bestandteil von CORBA ist ein ORB (Objekt Request Broker), der die Verwaltung von verteilten Objekten und Methodenaufrufen in heterogenen Systemen regelt [Group 96]. Ein ORB ermöglicht die Kommunikation zwischen lokalen und entfernten Objekten, wobei alle Verwaltungsaufgaben, wie Objekterzeugung, -speicherung, Verzeichnisdienste usw. durch den ORB geregelt werden.

Ein *Client* kann über ein Netzwerk eine Objektreferenz auf ein entferntes Objekt erhalten, Methodenaufrufe an diesem Objekt ausführen und Ergebnisse zurückgeliefert bekommen. Durch die Beschreibungssprache IDL (*Interface Definition Language*) ist es möglich, die Schnittstellen (Operationen und Attribute) der CORBA-Objekte sprachunabhängig zu beschreiben. Objekte, die mit Hilfe eines ORB zur Verfügung gestellt werden, geben ihre Schnittstelle in IDL bekannt.

Die unterste Ebene des Laufzeitsystems der OMG besteht aus einem ORB[2], der folgende Aufgaben hat:

- Nachrichtenvermittlung zwischen ORB-Objekten
- Bereitstellung von Basismechanismen für den transparenten Aufruf und Empfang von Nachrichten von lokalen und verteilten Objekten ohne Kenntnis des Kommunikationsmechnismus auf Seite des Klienten.
- Ortsunabhängige Speicherung und Aktivierung von Objekten

Schicht 2 (Business Object Facility): Die *Business Object Facility* (BOF) ist für die Persistenz, die Überwachung der Regeln eines Geschäftsobjektes zuständig und gewährleistet zur Laufzeit die Anzeige und Änderung von Beziehungen zwischen Geschäftsobjekten. Zusätzlich vereinfacht die BOF den Nachrichtenaustausch zwischen Geschäftsobjekten, indem sie von den Kommunikationsdiensten der unteren Schicht abstrahiert und den Geschäftsobjekten Dienste für den synchronen und asynchronen Nachrichtenaustausch anbietet. Wie in Abbildung 3.1 dargestellt, soll die BOF zusätzlich einen transparenten Zugriff auf die u.a. durch CORBA angeboten Dienste gewährleisten [Eeles, Sims 98]:

- der *Lifecycle Services* für das Erzeugen, Freigeben, Aktivieren und Deaktivieren von Objekten,
- ein *Event Management Service*, der für die Ereignisverwaltung zuständig ist,
- einem *Transaction Service* für die transaktionale Veränderung von Objekten,
- ein *Naming Service* für die Zuordnung eines eindeutigen Namen zu einem Object und
- einen *Trader Service*, der ein Objekt zur Laufzeit ausfindig macht.

Schicht 3 (Geschäftsobjekte): Die oberste Schicht bilden die bereits am Anfang definierten Geschäftsobjekte. Die OMG unterteilt Geschäftsobjekte in drei Arten:

Common Business Object (CBO): *Common Business Objects* sind horizontale Geschäftsobjekte, die übergreifend allen betrieblichen Funktionsbereichen gemeinsam sind. Das ist z.B. das Geschäftsobjekt „Vertrag“, das mindestens zwei Partner und einen Handelsgegenstand beinhaltet.

Vertikale Geschäftsobjekte: Hiermit werden die Geschäftsobjekte bezeichnet, die spezifisch innerhalb einer Branche sind, z.B. für den Finanzierungs- oder den Produktionsbereich.

Unternehmensspezifische Geschäftsobjekte: Geschäftsobjekte, die sich durch Subtypisierung aus den *Common Business Objects* erzeugen lassen.

[2] Angedacht ist von der OMG auch die Benutzung von COM/DCOM von Microsoft, oder DSOM von IBM.

Das Schichtenmodell der OMG hat somit zwei wesentliche Teile, und zwar einen technisch und einen betriebswirtschaftlich orientierten.
Ersterer besteht aus der *Business Object Facility*, die technisch orientierte Standards einschließt (wie z.B. die Kommunikation in verteilten Systemen) und diese den Geschäftsobjekten zur Verfügung stellt. Geschäftsobjekte repräsentieren die betriebswirtschaftliche Sicht und nutzen die von der BOF angebotenen Dienste transparent für den Entwickler von Geschäftsobjekten. Durch CORBA und die zahlreichen Implementierungen von ORBs durch verschiedene Hersteller ist die technologische Infrastruktur für die BOF und die darauf aufsetztenden Geschäftsobjekte vorhanden. Die OMG konzentriert ihre Bemühungen um eine Standardisierung auf den Bereich der BOF und der CBOs [Heiderich 98].

3.2 IBM San Francisco - Ein *Framework* für Geschäftsobjekte

San Francisco ist ein von der IBM entwickeltes *Framework* für die Implementierung von Geschäftsobjekten. IBM arbeitet aktiv am Standardisierungsprozeß für Geschäftsobjekte der OMG mit. In Anlehnung an deren Standardisierungsprozeß reichte die IBM einen Teil dieses *Frameworks* (*Base Layer*) als Vorschlag ein [Heiderich 98]. Daher entsprechen die Anforderungen der IBM an Geschäftsobjekte denen der OMG, wobei IBM mit einer bereits fertigen Implementation von Geschäftsobjekten dem Standard der OMG vorgreift. Die Übernahme eines in Zukunft erscheinenden Standards ist nicht ausgeschlossen. Ziel des *San Francisco*-Projektes ist es, den Anwendungsprogrammierern die schnelle Entwicklung von verteilten, objektorientierten Komponenten zu ermöglichen. Dies geschieht auf Basis einer objektorientierten Infrastruktur und zugehörigen Anwendungslogik, die vom Entwickler erweitert werden muß. IBM geht davon aus, daß die abgebildeten Prozesse, zusammen mit den zur Vefügung gestellten Geschäftsobjekten, 40 Prozent einer typischen Anwendung abdecken, so daß 60 Prozent einer Anwendung selber zu entwickeln sind [Wolff 98].

IBM entwickelt das *Framework* für Unternehmen, die für sich selbst individuelle betriebswirtschaftliche Software entwickeln wollen. Dadurch, daß die IBM die Basistechnologie zur Verfügung stellt, können die Kosten einer solchen Entwicklung gegenüber einer kompletten Neuentwicklung reduziert werden. Dies gelingt zum einen dadurch, daß die IBM vorhandene Java-Dienste[3] in das Framework integriert, so daß der Benutzer des *Frameworks* von den Details dieser verschiedenen Dienste abstrahieren kann. Zum anderen stellt das *Framework* ein Programmiermodell zur Verfügung, daß den Programmierer entlastet, da der objektorientierte Entwurf bereits durchgeführt ist. Es stehen damit Klassen zur Vefügung, die durch Verfeinerung in ihren Funktionalitäten erweitert werden können. Durch die Verwendung der plattformunabhängigen Programmiersprache Java und der integrierten Netzwerkfunktionalität in Java eignet sich das *Framework* besonders für betriebswirtschaftliche Anwendungen in heterogenen Umgebungen. Dabei ist das *Framework* aber speziell für kleine bis mittlere Lösungen vorgesehen, da die Entwicklung umfangreicher Anwendungen nicht dem zu erwartenden Nutzen entspricht. Vielmehr kann sich ein Unternehmen bei der Entwicklung einer Anwendung auf seine Kernkompetenzen konzentrieren und in anderen Bereichen Anwendungen anderer Hersteller integrieren, die auf Basis des *Frameworks* entwickelt wurden.

[3] Java Mail API, JNDI Java Naming and Directory Interface, Java Transaction Service, Java IDL, Java Messaging Service, Java Database Connectivity und IIOP bzw. Java Remote Method Invocation

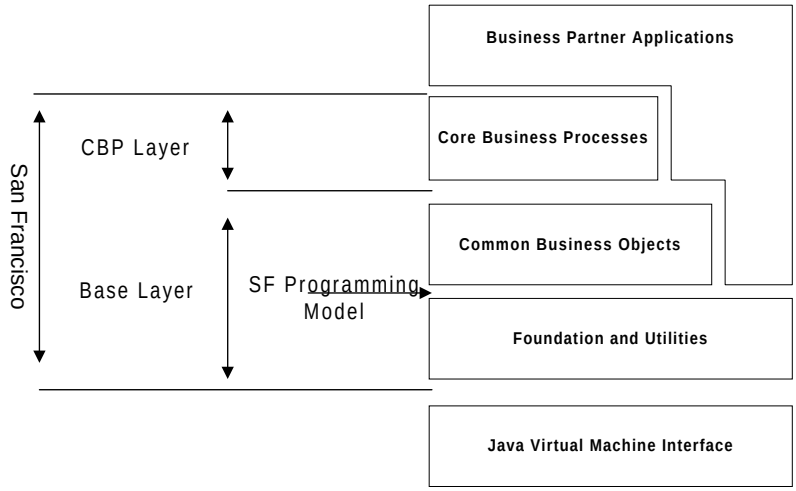

Abbildung 3.2: Das Schichtenmodell des *San Francisco Frameworks*

Das für das *Framework* entwickelte Schichtenmodell in Abbildung 3.2 ähnelt stark dem Geschäftsobjektmodell der OMG. Viele der angebotenen Dienste sind denen der OMG ähnlich. Zum Beispiel basieren Transaktionsdienste, Kommunikationsdienste zwischen verteilten Objekten oder die Verwaltung von persistenten Objekten auf den entsprechenden OMG-Ansätzen (*Object Service Definitions* der OMG). Anderseits bietet das *Framework* keinen CORBA-fähigen ORB an. Vielmehr werden die Funktionalitäten der OMG mit denen von JAVA abgeglichen, wobei gleichzeitig die Anforderungen der OMG in einigen Bereichen verändert wurden. Das Schichtenmodell in Abbildung 3.2 gestaltet sich folgendermaßen:

Base Layer: Aufbauend auf der virtuellen Maschine für Java bietet der *Base Layer* mit dem *Common Business Objects Layer* und der unterliegenden Schicht *Foundation und Utilities* eine Abstraktion von der Komplexität der verteilten Kommunikation auf verschiedenen Plattformen.

- **Foundation and Utilties:** In dieser Schicht befinden sich zum einen die Klassen für die Unterstützung des *San Francisco* Programmiermodells (Foundation) wie die persistente Speicherung und die Organisation der Objekte [Wolff 98]. Die oben erwähnten Dienste werden über diese Schicht angeboten. Zum anderen werden hier Werkzeuge für die Administration (Installation, Konfliktmanagement und Sicherheitseinstellungen) angeboten, die z.B. betriebssystemeigene Funktionalitäten kapseln.
- **Common Business Object Layer:** IBM unterteilt diese in *General Business Objects* (Geschäftspartner, Adresse, Währung usw.), *Financial Business Objects* (Bankkonto, Verkaufsauftrag usw.) und nach *Generalized Mechanisms* (z.B. die Berechnung des Saldos eines Kontos).

Core Business Processes Layer: Zweck dieser Schicht ist es, grundlegende Geschäftsprozesse zur Verfügung zu stellen. Die Geschäftsprozesse können an definierten Stellen

erweitert werden. Zur Zeit stehen Prozesse für das Führen von Buchungsjournalen, das Erstellen und Verwalten von Kontenplänen oder Buchungen, Auftragsabwicklung und Warenwirtschaft zur Verfügung. Jeder dieser Prozesse stellt grundlegende Funktionalitäten zusammen mit den zugehörigen allgemeinen Geschäftsobjekten zur Verfügung.

3.3 OAG - Geschäftsobjektdokumente

Die Open Applications Group (OAG) wurde 1995 von den neun weltweit größten Anbietern betriebswirtschaftlicher Anwendungssoftware gegründet[4]. Diese Gruppe hat sich zum Ziel gesetzt, Industriestandards zur Integration betriebswirtschaftlicher und externer Anwendungen verschiedener Hersteller zu entwickeln. Die OAG berücksichtigt bei der Standardisierung etablierte Technologien, wie z.B. CORBA, OLE usw. Die OAG konzentriert sich neben der Integration von verschiedenen Systemen auf der Basis von Geschäftsobjektektdokumenten auch z.B. auf die Integration von Unternehmen unter Benutzung von EDI (*Electonic Data Interchange*) oder EFT (*Electronic Funds Transfer*). Im folgenden werden die Geschäftsobjektdokumente beschrieben, die in einem von der OAG veröffentlichten Strategiepapier [OAG 98] spezifiziert sind: Geschäftsobjektdokumente (*Business Object Document* (BOD)) ermöglichen die Nutzung und Integration von Produkten verschiedener Hersteller. Dadurch sollen kundenspezifische Anforderungen an eine Software besser erfüllt werden. Ziel der Spezifikation ist es, den Integrationsprozeß für alle zu optimieren und kostenintensive Einzellösungen zu vermeiden.

Die Integration auf Basis von Geschäftsobjektdokumenten wird von der OAG als *Open Applications Integration* bezeichnet und bezieht sich auf Funktionen wie Kundenauftragsbehandlung, die Führung von Bestandsverzeichnissen und Rechnungsschreibung. Diese Funktionen werden von der OAG analysiert und in Form eines BOD gekapselt. BODs abstrahieren von den Daten, Prozessen und der Architektur verschiedener Anwendungen. Ein BOD stellt ein standardisiertes Austauschformat dar, mit dessen Hilfe eine Anforderung (*Business Service Request*) einer betriebswirtschaftlichen Anwendung an eine andere gestellt werden kann.

Ein BOD besteht aus zwei Hauptstrukturen, die sich wiederum in Unterstrukturen aufteilen. Innerhalb jeder Struktur sind Felder spezifiziert, die genau den Typ und die Länge eines Feldes festlegen. Die zwei Hauptstrukturen eines BODs sind die *Control Area* und die *Business Data* (siehe Abbildung 3.3):

Control Area Die *Control Area* setzt sich aus einem **Sender**, **Receiver** und einem **Business Service Request** zusammen. In der Struktur **Sender** wird spezifiziert, von welchem System und von welcher Anwendung ein Anforderung ausgeht. Die Struktur **Receiver** wird dazu benutzt, das zuständige System zu adressieren, das die Anforderung durchzuführen hat. Das System, das die Anforderung erhält, entnimmt die Informationen für die Art der Anforderung aus der Unterstruktur **Business Service Request**, die sich aus einem Nomen und einem Verb zusammensetzt. Das Nomen spezifiziert den Typ (z.B. *Customer*, *Invoice*) an dem eine Aktion auszuführen ist, die durch das Verb spezifiziert wird (z.B. *add*, *sync*, *post* usw.). Abhängig von Nomen und Verb sind die Unterstrukturen der *Business Data*-Struktur bestimmt.

[4]Mitglieder: SAP AG, IBM, Oracle Corporation usw.

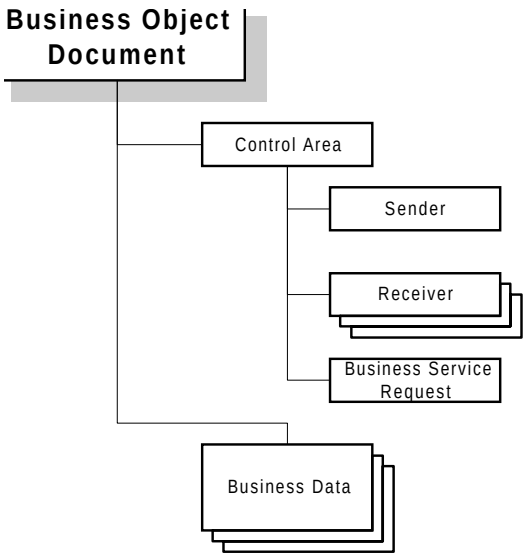

Abbildung 3.3: Struktur eines **Business Object Documents**

Business Data Die benötigten Daten für eine Anforderung befinden sich im Feld *Business Data* (siehe anschließendes Beispiel). Strukturen sowie Felder innerhalb dieser Strukturen können als optional gekennzeichnet werden. Mit Hilfe eines BOD kann eine Anwendung nur jeweils eine Anforderung an ein anderes System schicken. Für jede weitere Anforderung muß ein neues BOD erstellt werden. Es ist allerdings möglich, mehrere gleiche Anforderungen in einem BOD zu akkumulieren (z.B. eine Operation auf mehreren Kundendaten).

Beispiel: Der *Business Service Request* **SYNC CUSTOMER** dient dazu, Kundendaten innerhalb verschiedener Systeme zu synchronisieren, d.h. mit dieser Anforderung kann ein Kunde in einem anderen System angelegt oder verändert werden. **SYNC CUSTOMER** benutzt drei Strukturen **PARTNER**, **ADDRESS** und **CONTACT**, die im BOD in der Struktur *Business Data* enthalten sind (siehe Abbildung 3.4). Die beiden Strukturen **ADDRESS** und **CONTACT** sind optional und enthalten die Adresse bzw. Kontaktinformationen wie z.B. die Telefonummer eines Ansprechpartners. Die Struktur **PARTNER** repräsentiert einen Geschäftspartner und enthält neben dem Typ des Partners u.a. seinen Namen und eine Identifikationsnummer. Der Typ gibt an, ob es sich bei dem Partner um einen Kunden, Verkäufer oder Lieferanten handelt.

Ein BOD enthält also im Unterschied zu den Geschäftsobjekten der OMG keine Geschäftslogik, sondern nur Daten, die für die Ausführung einer Anforderung erforderlich sind. Das hat den Vorteil, daß sich die OAG weniger technische Details zu berücksichtigen hat, da im Grunde nur geklärt werden muß, wie ein BOD zwischen zwei Systemen ausgetauscht werden kann. Aus dem Ansatz der OAG resultiert eine lose Kopplung zwischen verschiedenen Systemen, die relativ leicht, aber auf einem niedrigen Abstraktionsnivau, implementiert werden kann. Es

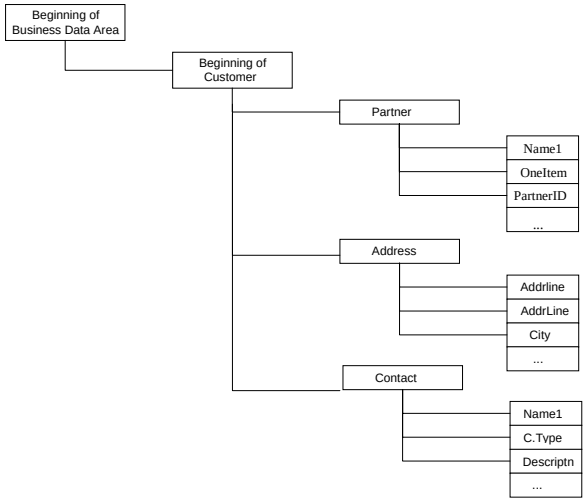

Abbildung 3.4: OAG: *Business Data Area* von **SYNC CUSTOMER**

ist also nicht möglich wie bei der OMG eine Referenz auf ein Objekt zu erhalten und daran eine Methode auszuführen.

3.4 Bewertung

Semantische Objekte aller drei Ansätze sind Geschäftsobjekte. Jedoch bestehen große Unterschiede hinsichtlich der Umsetzung eines Geschäftsobjekts. Die OMG benutzt zur Definition der Geschäftsobjekte den von ihr u.a. definierten Standard CORBA zur Kommunikation in veteilten Systemen. Ein Geschäftsobjekt bietet die Möglichkeit, Methoden auszuführen und auf Ereignisse zu reagieren. Die Standardisierungbemühungen konzentrieren sich dabei auf Basisdienste wie *Transaction Sercices* oder *Event Management Services*. Konkrete Geschäftsobjekte mit einer spezifizierten Funktionalität werden nicht definiert. Dies erscheint auch schwierig, da die Anforderungen an ein Geschäftsobjekt zu unterschiedlich sind, als daß sie statisch formuliert werden können. Ein konkretes Geschäftsobjekt wird durch eine Subtypbildung von einem allgemeinen Geschäftsobjekt (*Common Business Object*) erzeugt. Dies impliziert nicht zwingend den Einsatz einer objektorientierten Implementierung des Geschäftsobjekts, wie die CORBA-Basisdienste zeigen. Damit ist es im Prinzip möglich, vorhandene nichtobjektorientierte Softwaresysteme über Geschäftsobjekte zu kapseln und hiermit eine bessere Integration mit solchen Systemen zu erreichen. Geschäftsobjektkonzepte trennen die Prozesslogik, die in jedem Anwendungsfall spezifisch ist, von der Logik des Geschäftsobjektes, welche eine grundlegende einheitliche Semantik in den Systemen besitzen. Das Modell der OMG formuliert eine enge Kopplung zwischen den beteiligten Systemen.

Demgegenüber steht das Konzept der *Business Object Documents* der OAG. Ein *Business Object Document* stellt die Kapselung eines Geschäftsvorfalls in ein standardisiertes Austauschformat dar. Das *Business Object Document* ist damit damit kein Objekt im Sinne der OMG. Methoden sind auf einem *Business Object Document* nicht ausführbar, d.h. sie be-

sitzen kein eigenes Verhalten. Ein *Business Object Document* benötigt im Modell der OAG auch kein Verhalten, da ein *Business Object Document* gewissermaßen die Parameter einer spezifischen Methode auf einem spezifischen Objekttyp darstellt. Das Verhalten wird durch die Anwendung definiert, die als Empfänger des *Business Object Documents* auftritt. Partner einer Interaktion sind also nicht Objekte wie im OMG-Konzept, sondern Anwendungen. Mit dem Objektbegriff der OMG geht ein Referenzsemantik einher. Im OAG-Konzept besitzen nur die interagierenden Amwendungen Zugriff auf ein Geschäftsobjekt, das *Business Object Document* unterstützt nur die Kommunikation und stellt mit seiner definierten Feldstruktur eine Teilmenge der Attribute eines Geschäftsobjekts dar. Die komplexe Dienststruktur, die dem OMG-Modell zugrundeliegt, wird im Vorschlag der OAG nicht benötigt. Die Standardisierungsbemühungen beschränken sich deshalb vor allem auf die Definition der Strukturen der *Business Object Documents*. Das Modell der OAG impliziert eine lose Kopplung der interagierenden Softwaresyste.

Ein weiterer Unterschied zwischen OMG- und OAG-Modell besteht in der Granulariät der Modellierung. Ein *Business Object Document* definiert abhängig vom Typ mehr oder minder komplexe festgelegte Datenstrukturen. Die Semantik eines *Business Object Documents* ist somit festgelegt. Anpassungen an konkrete Anforderungen bedingen eine Änderung der Struktur des *Business Object Documents*. Die Semantik eines OMG-Objekts wird in einer sprachunabhängigen *Interface Defintion Language* festgelegt. Die IDL definiert, welche Methoden ein Objekt versteht.

Bei den Konzepten der OMG und der OAG steht die Integration von Softwaresystemen im Vordergrund. Im Gegensatz dazu bietet die IBM mit dem San Francisco-*Framework* eine konkrete Implementierung eines Geschäftsobjektmodells. Das *Framework* bietet ein Programmiergerüst für die Entwicklung kleiner bis mittlerer Anwendungen basierend auf Geschäftsobjekten. Wesentlicher Betrachtungsgegenstand ist also die Neuerstellung von Anwendungen, die mit Geschäftsobjekten operieren. Für die Integration von Softwaresystemen stellt Java jedoch auch die geeignete Umgebung dar, da es durch seine Plattformunabhänigkeit und vorhandenen Dienste Probleme überwinden hilft, die aus der Heterogenität der eingesetzten Systeme entstehen.

In Tabelle 3.1 sind die wesentlichen Merkmale der einzelnen Konzepte gegenübergestellt. Im folgendem Kapitel wird das Objektmodell des R/3-Sytems vorgestellt. In Abschnitt 4.5 wird das R/3-Objektmodell in die in diesem Abschnitt vorgestellten Konzepte eingeordnet.

	OMG	IBM	OAG
Konzept	Geschäfts-objekte	Geschäfts-objekte	Standardisiertes Austauschformat (BOD)
Ziel	Definition eines Objektmodells und Vermeidung von mono-lithischen Systemen	Entwicklungs-unterstützung für kleine bis mittelgroße Anwendungen	Interoperabilität verschiedener Systemen
Geschäftslogik	Integriert in Geschäfts-objekte und zugehörigen Prozessen	Integriert in Geschäfts-objekte und zugehörigen Prozessen	Enthalten in diensterbringen-den Systemen
Sprachabhängigkeit	Nein	Ja (JAVA)	Nein
Systemabhängigkeit	Nein	Nein	Nein
Entwicklungsstand	Definition von Allgemeinen Geschäfts-objekten	Evaluation Kit (V1R2)Juli 1998	Definition von Geschäftsobjekt-dokumenten

Tabelle 3.1: Vergleich der vorgestellen Geschäftsobjekt Konzepte

Kapitel 4

Geschäftsobjekte und Objektmodell im R/3-System

Nachdem im vorangegangenen Kapitel verschiedene Ansätze von Geschäftsobjektkonzepten erläutert wurden, soll im folgenden das Geschäftsobjektmodell im R/3 vorgestellt werden. Da die Geschäftsobjekte eine objektorientierte Nutzung dieser Objekte auf Implementierungsebene nahelegen, wird in diesem Kapitel allgemeiner das Objektmodell der SAP mitsamt der Programmierung der Geschäftsobjekte vorgestellt. Die Ebene der Implementierung ist insofern relevant, als daß mit dem Release 4 objektorientierte Erweiterungen in die Programmiersprache ABAP Einzug gehalten haben. Aus diesem Grund werden zum Ende dieses Kapitels die Konzepte von OO-ABAP und die Eignung von OO-ABAP für die Implementierung von Geschäftsobjekten untersucht.

4.1 Das *Business Framework*

Das Objektmodell des R/3 ist eng mit dem SAP-Konzept des *Business Frameworks* verbunden. Das *Business Framework* ist ein Konzept, welches mit der Komponentisierung des R/3-Systems und der Offenlegung stabiler Schnittstellen zu externen Anwendungen bessere Möglichkeiten zur Integration mit R/3 schafft, als auch die Anpaßbarkeit an Kundenanforderungen verbessert.

Bisherige Möglichkeiten der Integration basieren meist auf der Nutzung von Funktionsbausteinen, die durch externe Anwendungen aufgerufen werden. Probleme dieser Integration sind, daß die Schnittstellen der Funktionsbausteine zwischen verschiedenen Releases des R/3 Änderungen unterworfen sind. Ein weiteres Problem ist, daß nur ein geringer Teil der R/3-Funktionalität über Funktionsbausteine verfügbar ist. Das *Business Framework* formuliert als Lösung dieser Problematik Schnittstellen auf der Basis von Geschäftsobjekten, bei denen die Stabilität der Schnittstelle zugesichert wird. Die R/3-Geschäftsobjekte werden dabei sukzessive von der SAP definiert und implementiert.

Der zweite Problembereich - die Verbesserung der Anpaßbarkeit und Einführung des R/3 - soll durch eine Komponentisierung des R/3 in *Business Komponenten* erreicht werden. Als *Busi-*

ness Komponenten sind dabei die Module des R/3-Systems zu verstehen. Konkret verwirklicht ist dies mit Release 4 zwischen den Modulen des Finanzwesens (FI), Personalwirtschaft (HR) und Logistik (LO). In der Zukunft ist geplant, auf dieselbe Weise branchenspezifische Komponenten als auch Internetanwendungskomponenten in das R/3 einzubinden. Die Integration der Komponenten wird technisch über *Application Link Enabling (ALE)* erreicht, der SAP-Variante von *Electronic Data Interchange* (EDI). ALE „ermöglicht die Verteilung betriebswirtschaftlicher Prozesse und Funktionen auf mehrere, lose gekoppelte R/3-Systeme“ [SAP 98k]. Die *Business Komponenten* als auch die Geschäftsobjekte werden in R/3 im *Business Object Repository* verwaltet, auf das im Abschnitt 4.4 eingegangen wird. Das *Business Framework* ist als Architektur zu verstehen, die es ermöglicht, über beliebige *Frontends* auf die integrierte betriebswirtschaftliche Funktionalität des R/3 zuzugreifen.

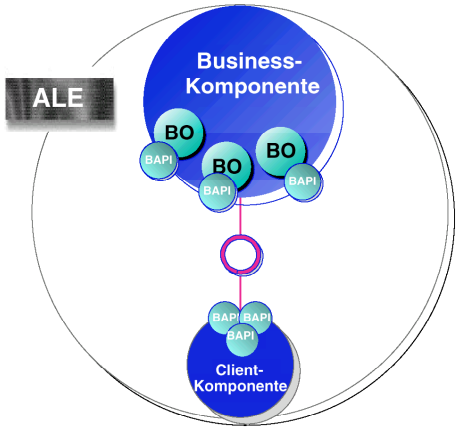

Abbildung 4.1: SAP-*Business Framework*

4.2 Geschäftsobjekte im R/3

Die R/3-Geschäftsobjekte stellen den von der SAP favorisierten Ansatz zur Integration mit dem R/3 dar, da diese eine langfristig stabile Schnittstelle aufweisen [SAP 98m]. Fremdsysteme (und natürlich auch Eigenentwicklungen in ABAP) bietet sich so eine brauchbare objektorientierte Programmierschnittstelle. Geschäftsobjekte sind als Entwicklungsklassenobjekte Teil des R/3-Repositories und werden daher in der *Development Workbench* gepflegt. Die Geschäftsobjekte im R/3 können durch eigene Geschäftsobjekte überladen und erweitert werden. Es besteht so die Möglichkeit, nicht vorhandene Funktionalität im R/3 über die Definition von Geschäftsobjekten selbst zu entwickeln und objektorientiert zu nutzen. Die Komplexität solcher Eigenentwicklungen sollte dabei jedoch nicht unterschätzt werden. Insbesondere sind detaillierte Kenntnisse des Aufbaus der R/3-Tabellen und deren Zusammenwirken erforderlich.

Geschäftsobjekte stehen seit Release 3.0 zur Verfügung, können jedoch durch externe Anwendungen erst seit Release 3.1 und der Erstellung der BAPIs durch die SAP genutzt werden. Seitdem Geschäftsobjekte im R/3 vorhanden sind, werden diese durch die R/3-Komponente *Business Workflow* genutzt. *Business Workflow* ist ein anwendungsübergreifendes Werkzeug, wel-

ches die „automatisierte Steuerung und Bearbeitung von anwendungsübergreifenden Abläufen“ [SAP 98j] bietet. *Business Workflow* stützt sich auf existierende Funktionalitäten in Form von SAP-Transaktionen und Funktionsbausteinen ab und ist deshalb von der Architektur her oberhalb der R/3-Komponenten anzusiedeln. Die Verknüpfung der durch *Business Workflow* gesteuerten Aktivitäten mit den entsprechenden Bearbeitern wird durch eine Integration mit dem R/3-Organisationsmanagements ermöglicht. Die Aktivitäten werden dabei in abstrakter Form, d.h im Normalfall ohne weitere Programmierung auf den R/3-Geschäftsobjekten definiert.

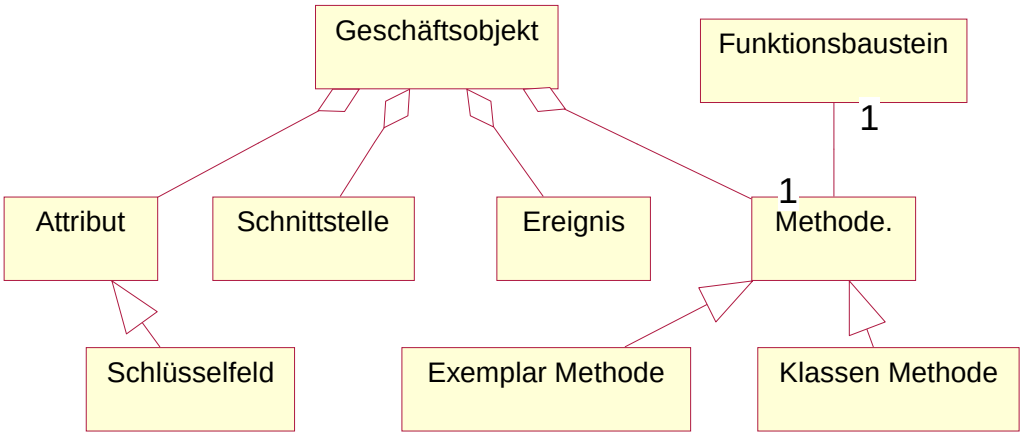

Abbildung 4.2: Geschäftsobjekte im R/3

Die fachliche Einordnung des Geschäftsobjektes in das R/3-System wird durch ein zugeordnetes (SAP-SERM-)Datenmodell verdeutlicht. Dieses Datenmodell ist Teil des SAP-Referenzmodells, welches die dem R/3 innewohnenden Geschäftsregeln und Geschäftsprozesse grafisch in verschiedenen Diagrammtypen[1] darstellt. Das zugewiesene Datenmodell dient somit der Dokumentation und hat keine Auswirkungen auf die Programmierung oder die benutzten Datenstrukturen.

Die Implementierung eines Geschäftsobjektes wird im Abschnitt 4.6 erläutert. Hier soll zunächst das fachliche Konzept der Geschäftsobjekte vorgestellt werden. Ein Geschäftsobjekt besteht dabei aus Grunddaten, Schnittstellen, Schlüsselfeldern, Attributen, Ereignissen und Methoden:

Grunddaten Grunddaten enthalten allgemeine Angaben zum Objekttyp wie Name (welcher den Objekttyp identifiziert) und den Namen des implementierenden Programms. Die Grunddaten der Objekttypen werden in der Datenbanktabelle **TOJTB** gehalten. Weitere Grunddaten sind Supertyp, Name eines verbundenen Datenmodells sowie Informationen zum Versionsstand wie z.B. letzte Änderung und Freigabestatus.

Schnittstellen Geschäftsobjekte implementieren Schnittstellen. Ein Schnittstelle stellt - im Unterschied zum Schnittstellenbegriff in Java - eine Sammlung von Attributen, Methoden und Ereignissen dar, die das Geschäftsobjekt sich verpflichtet, zu implementieren.

[1] u.a. Organisationsdiagramme, Ereignisgesteuerte Prozeßketten, SAP-SERM-Diagramme

Alle Geschäftsobjekte müssen standardgemäß die *IFSAP*-Schnittstelle implementieren. Die *IFSAP*-Schnittstelle definiert zwei Methoden:

- Die Methode **Display** sollte durch ein implementierendes Geschäftsobjekt durch einen Aufruf der entsprechenden SAP-Transaktion abgebildet werden.
- Die zweite Methode **ExistenceCheck** dient der Überprüfung, ob ein durch seine Schlüsselfelder spezifiziertes Geschäftsobjekt im R/3-System vorhanden ist. Schnittstellen können als abstrakt gekennzeichnet werden. In diesem Fall muß das Geschäftsobjekt die Methoden der Schnittstelle geeignet implementieren.

Schlüsselfelder Schlüsselfelder dienen zur Identifikation eines Geschäftsobjektes im R/3. Da dem R/3 die Semantik des relationalen Datenmodells zugrunde liegt, beziehen sie sich auf Primärschlüsselfelder in Datenbanktabellen, in denen das R/3 die Informationen speichert (Assoziative Identifikation). Ein Geschäftsobjekt wird also durch die Werte seiner Schlüsselfelder und dem Namen seines Geschäftsobjekttyps identifiziert. Geschäftsobjekte sind immer persistent. Es besteht nicht die Möglichkeit, transiente Geschäftsobjekte zu erstellen.

Attribute Ein Attribut modelliert die Eigenschaften eines Geschäftsobjektes. Es existieren zwei verschiedene Typen von Attributen, die sich hinsichtlich ihrer Implementierung unterscheiden:

- Ein Datenbankfeld-Attribut läßt sich direkt auf ein Feld einer Datenbanktabelle des ABAP *Dictionaries* abbilden. Die Schlüsselfelder des Geschäftsobjektes bestimmen den zu lesenden Datensatz, in dem das Feld vorhanden ist.
- Ein virtuelles Attribut läßt sich durch Berechnung und/oder Auswertung von Datenbankinhalten zur Laufzeit ermitteln. Die Berechnung wird durch die Implementierung einer Attributzugriffsmethode definiert.
  Virtuelle Attribute sind nicht wie die Datenbankfeld-Attribute auf die beschriebenen skalaren Typen eingeschränkt, sondern können auch den Typ einer Tabelle besitzen. Auf diese Weise ist es möglich, Aggregationen von Objektreferenzen zurückzugeben und so Assoziationen zwischen Objekttypen abzubilden. Eine Objektreferenz ist dabei als Tupel von Schlüsselfeldern zu verstehen, welche ein Geschäftsobjekt identifizieren.

Ereignisse Ein Ereignis signalisiert eine Zustandsänderung an einem Objekt. Dieses Signal wird vom Geschäftsobjekt selber gegeben und steht systemweit zur Verfügung. Objekte, die an Ereignissen eines Geschäftsobjektes interessiert sind, können sich durch einen Ereignismanager registrieren lassen, der diese dann benachrichtigt.

Die Methoden (BAPIs) eines Geschäftsobjektes sind von besonderer Relevanz. Deshalb wird ihnen das folgenden Abschnitt gewidmet.

4.3 BAPIs

BAPIs (*Business Application Programming Interface*) sind Methoden von Geschäftsobjekten und stellen die elementare Schnittstelle zum Zugriff auf Geschäftsobjekte dar; nur über die

BAPIs ist es möglich, Objekte zu erzeugen oder zu verändern. Ein schreibender Zugriff auf die Geschäftsobjektrepräsentation über Attribute ist nicht vorgesehen. Implementiert werden BAPIS durch entfernt aufrufbare Funktionsbausteine. BAPIs stellen eine Teilmenge der gesamten Objektmethoden der Geschäftsobjekte dar. BAPIs zeichnet dabei aus, daß sie im Gegensatz zu normalen Objektmethoden keine Benutzerdialoge enthalten. Implementierungen, die Benutzerdialoge enthalten, sind für externe Anwendungen nicht steuerbar und somit nicht innerhalb des *Business Framework*-Konpepts zur Integration externer Anwendungen nutzbar. Die gängige Methode **DISPLAY** ist ein Beispiel für eine Objektmethode, die keinen BAPI darstellt. Sie wird bei vielen Geschäftsobjekten durch den Aufruf der entsprechenden SAP-Transaktion implementiert.

BAPIs werden von der SAP seit dem Release 3.1g entwickelt. Im Release 3.1g sind rund 220 und im Release 4.0b bereits über 450 BAPIs implementiert. Daran ist bereits die Wichtigkeit der BAPIs erkennbar. BAPIs können dabei sowohl von der SAP als auch durch andere Hersteller implementiert werden. Die SAP definiert Standard-BAPIs mit einer Namenskonvention. Drei der Standard-BAPIs werden im folgenden beispielhaft beschrieben, wobei die BAPIs in Klassen- und Exemplarmethoden unterschieden werden:

- Die Klassenmethode **CreateFromData** entspricht dem Konstruktor in objektorientierter Sprachen und liefert bei Erfolg die Schlüsselfelder des neu erzeugten Geschäftsobjekts zurück. Als Parameter erwartet sie Strukturen von Feldern, welche die Attribute des neu zu erzeugenden Objekts definieren.
- Die Klassenmethode **GetList** liefert eine Liste aller Schlüsselfelder der im R/3 persistent gespeicherten Geschäftsobjekte eines Geschäftsobjekttyps zurück. Exemplarmethoden gestatten es Informationen zu einem Geschäftsobjekt abzufragen, bzw. es persistent zu ändern.
- Die Exemplarmethode **GetDetail** ermöglicht das Lesen von Detailinformationen zu einem Geschäftsobjekt, welches durch seine Schlüsselfelder spezifiziert wird

Ein BAPI wird auf einen entfernt aufrufbaren Funktionsbaustein (RFC) abgebildet. Deshalb gilt für die Schnittstelle eines BAPIs dasselbe, was bereits im Abschnitt 2.5.2.2 bei der Schnittstelle eines Funktionsbausteins erläutert wurde. Ein BAPI läßt sich also durch den Namen des Funktionsbausteins und durch seine Import, Export und Tabellenparameter beschreiben. Bei exemplarabhängigen BAPIs sind die Schlüsselfelder eines Geschäftsobjekttyps als Importparameter abgebildet und vor dem Aufruf geeignet zu füllen. Die Prüfung, ob ein Benutzer die Berechtigung zum Zugriff auf ein Geschäftsobjekt hat, wird durch die BAPI-Logik überprüft.

Ein (schreibender) BAPI bildet die transaktionale Änderung genau eines Geschäftsobjektes ab. Er besitzt daher die Semantik einer SAP-LUW, wobei die Transaktionsteuerung durch das rufende Programm vorgenommen wird. Diese Transaktionsteuerung durch das rufende Programm ist eine Neuerung des BAPI-Konzepts in R/3-Release 4. Bis dato stellte die Ausführung jedes BAPI für sich eine SAP-LUW dar. Die Aggregation mehrerer BAPIs zu einer SAP-LUW (=Transaktion) war nicht möglich, da der BAPI die Transaktionsteuerung übernahm. Die Transaktionssteuerung in Form der BAPIs **BAPI_TRANSACTION_COMMIT** und **BAPI_TRANSACTION_ROLLBACK** ist seit Release 4 aus dem BAPI herausgezogen. Grundsätzlich muß jedoch bei externen Anwendungen als BAPI-Aufrufer

beachtet werden, daß sich die Transaktionssteuerung auf alle getätigten BAPI-Aufrufe einer R/3-Verbindung bezieht. Diese Art von Steuerung ist also problematisch, wenn mehrere Prozesse auf derselben R/3-Verbindung Aufrufe tätigen.

Die Ausführung von BAPIs geschieht generell synchron. Ab Release 4.0 können BAPIs jedoch im Kontext des *Application Link Enabling (ALE)* auch für die asynchrone Datenübertragung verwendet werden. Dazu wird statt des BAPIs ein generierter Funktionsbaustein aufgerufen, der die Parameter des BAPIs in eine EDI-Nachricht verpackt[2], diese Nachricht an das Zielsystem gesendet, wo wiederum ein für diesen BAPI generierter Funktionsbaustein die EDI-Nachricht dekodiert und den BAPI aufruft. Diese Methodik, die asynchrone Ausführung von BAPIs, empfiehlt die SAP für alle schreibenden BAPIs. Der Grund liegt in der fehlenden Transaktionssemantik des synchronen Funktionsbausteinaufrufes. Wie bereits in Abschnitt 2.5.2.2 gezeigt, stellt der transaktionale Funktionsbausteinaufruf die einzige Möglichkkeit des entfernten Funktionsbausteinaufrufes unter Beibehaltung einer transaktionalen Semantik dar. ALE wendet genau diese Möglichkeit an, um die Transaktionalität des Aufrufes sicherzustellen. Nichtsdestotrotz fokussieren alle gängigen SAP-Beispiele zur BAPI-Programmierung auf den synchronen Aufruf, obwohl dies mit den genannten Unsicherheiten verbunden ist.

Im Unterschied zu Funktionsbausteinen werden für die BAPI-Funktionsbausteine und ihre Schnittstellen von der SAP die Stabilität und Abwärtskompatibilität zugesichert, d.h. „Schnittstellendefinition und Parameter (eines BAPIs) bleiben langfristig stabil“ [SAP 98m]. Wie sich bei der Entwicklung dieser Schnittstelle gezeigt hat, ist dies nicht so und verursacht auch beträchliche Schwierigkeiten bei der Entwicklung einer R/3-Schnittstelle. Diese Probleme und deren Lösung werden detailliert in der Beschreibung der Schnittstelle in Kapitel 7 erläutert.

BAPIs werden durch Funktionsbausteine implementiert. Dokumentiert werden die Geschäftsobjekte, ihre Eigenschaften und Methoden im *Business Object Repository*, welches im nächsten Kapitel beschrieben wird.

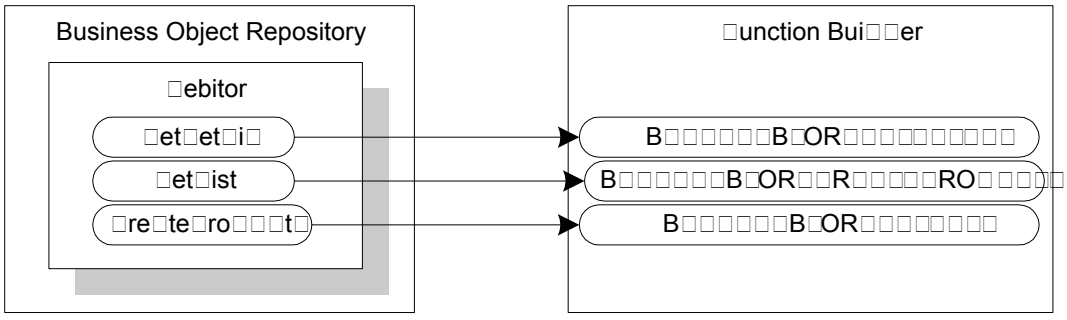

Abbildung 4.3: Geschäftsobjekte und Funktionsbausteine

[2]Im SAP-Umfeld wird diese EDI-Nachricht als *IDoc = Intermediate Document* bezeichnet

4.4 *Business Object Repository*

Die SAP-Geschäftsobjekttypen und ihre Beziehungen untereinander werden im *Business Object Repository (BOR)* definiert. Es existieren circa 170 Geschäftsobjekte, die sukzessive durch die Implementierung von BAPIs durch die SAP für Fremdsysteme zur Verfügung gestellt werden. Das *Business Object Repository* stellt analog zur Funktionsbibliothek für die Funktionsbausteine den zentralen Ort der *SAP-Business Komponenten* und Objekttypen dar. An dieser Stelle wird der Begriff Objekttyp benutzt, da das BOR sowohl betriebswirtschaftliche Objekttypen (die Geschäftsobjekte im eigentlichen Sinne), Strukturobjekttypen und technische Objekttypen darstellt und anbietet. Strukturobjekttypen[3] und technische Objekttypen[4] werden durch die R/3-Komponente *Business Workflow* benutzt und haben für die Integration des R/3 mit externen Anwendungen keine Bedeutung. Relevant sind lediglich Objekttypen, die BAPI-Methoden anbieten. Das BOR bietet eine fachliche Sicht auf die R/3-Objekttypen. Es gestattet, die Definition von Objekttypen anzusehen, als auch Objekttypen anzulegen.

Die Aufgaben des BOR sind:

- die Identifizierung und Beschreibung der Objekttypen
- die Erstellung von Exemplaren der Objekttypen
- die Dokumentation der Schnittstellen
- die Bereitstellung einer Schnittstelle zu anderen Repositories wie dem ARIS Toolset

Die im R/3 verfügbaren Objekttypen sind im BOR in einer hierarchischen Struktur angeordnet. Die Hierarchie orientiert sich dabei auf der ersten Ebene an den R/3-Anwendungskomponenten (Module), welche die *Business Komponenten* darstellen. Anwendungskomponenten sind beispielsweise Finanzwesen oder *Treasury*. Auf den untergeordneten Ebenen wird nach Teilfunktionen der jeweiligen Anwendungskomponente differenziert. Teilfunktionen des Finanzwesens sind z.B. die Debitoren-, Kreditoren- und Anlagenbuchhaltung. Innerhalb der verschiedenen Teilfunktionen des R/3 sind die Objekttypen angeordnet. Von mehreren R/3-Teilfunktionen genutzte Objekttypen wie z.B. das Debitorenkonto erscheinen in mehreren Ästen dieser Hierarchie (siehe Abbildung 4.5). Geschäftsobjekte bilden eine Teilmenge der verfügbaren Objekttypen. Die Modellierung der Geschäftsobjekte ist noch nicht abgeschlossen. So existieren Ungenauigkeiten in den vorhandenen Geschäftsobjekten. Gemäß den Metainformationen im BOR wird z.B. ein Kreditor durch die **CREDITORID** identifiziert. In der Realität wird zur Identifizierung eines Kreditors jedoch auch der Buchungskreis benötigt. Diese Ungenauigkeiten verursachen Schwierigkeiten bei der Erstellung einer generischen R/3-Schnittstelle, wenn diese die Informationen im BOR als Grundlage nutzen will.

Das BOR modelliert vier verschiedene Arten von Verknüpfungen:

- Schnittstellenimplementierung zwischen einem Objekttyp und einer oder mehreren Schnittstellen
- Assoziation zwischen Objekttypen

[3] Objekttypen aus der SAP-Organisationsmodellierung wie Buchungskreis, Werk und Verkaufsorganisation
[4] Kalender, *Workitems* oder archivierte Dokumente

- (Einfach-)Vererbung zwischen Objekttypen
- Komposition zwischen Objekttypen

Innerhalb der hierarchischen Struktur des BOR werden die Beziehungen Vererbung und Komposition dargestellt. Eine Assoziationsbeziehung ist implizit über die Signatur der Methoden und die Attribute des Objekttyps ersichtlich. Eine Methode, die eine Assoziation modelliert, ist beispielsweise **Customer.getSalesAreas**, die eine Tabelle von Schlüsselfeldern der Vertriebsorganisationen zu einem Kunden liefert.

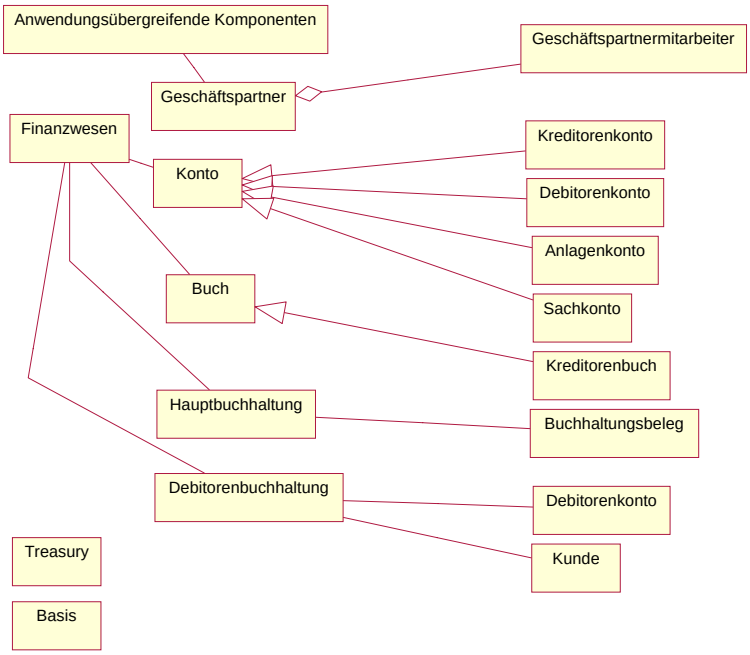

Abbildung 4.4: Geschäftsobjekte im *Business Object Repository*

Wie bereits im Abschnitt 4.3 angedeutet, stellt das BOR keine statische Beschreibung der vordefinierten R/3-Geschäftsobjekte dar, sondern kann durch die Erstellung neuer Objekttypen und der Implementierung selbst entwickelter BAPIs erweitert werden. Vorhandene Objekttypen können jedoch nicht verändert werden. Ist es erforderlich, von der SAP vordefinierte Objekttypen in ihrer Semantik zu verändern, weil z.B. die BAPI-Schnittstelle eines Objekttyps erweitert werden soll, so ist ein neuer Objekttyp anzulegen, der als Subtyp des vorhandenen Geschäftsobjekts auftritt und die vorhandene Implementierung des Supertyps erbt. Innerhalb dieses Subtyps kann eine erweiterte Funktionalität in Form neuer BAPIs entwickelt werden. Ein Überladen von Methoden ist jedoch nicht erlaubt. Zusätzlich zu dieser Subtypbildung im BOR muß zusätzlich der neu erstellte Objekttyp in einer Datenbanktabelle registriert werden. Die SAP bezeichnet dies als Registration eines Delegationstyps. (SAP-)Anwendungen, die mit dem Supertyp operieren, arbeiten hiernach mit dem selbst definierten Objekttyp (dem Delegationstypen).

Die Definitionen der Geschäftsobjekte im BOR lassen sich über den Aufruf von Funktions-

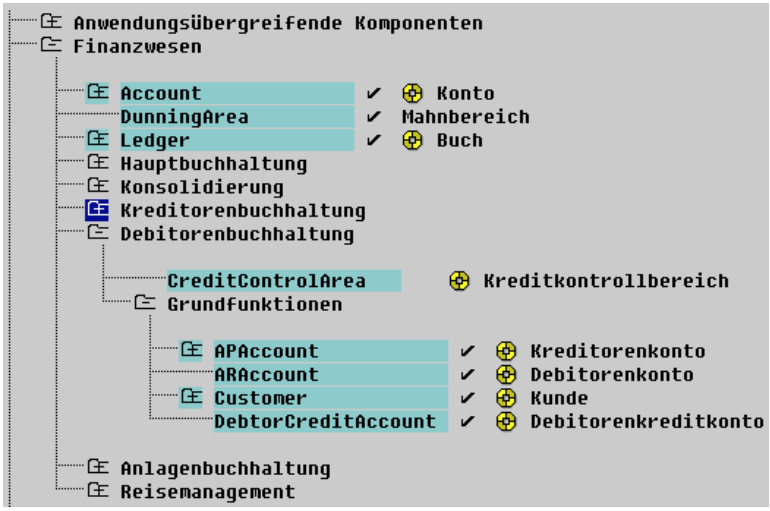

Abbildung 4.5: Das Geschäftsobjekt Kunde im *Business Object Repository*

bausteinen durch Anwendungen auslesen[5]. Prinzipiell besteht so die Möglichkeit, Schnittstellendefintionen auf Basis der ermittelbaren Informationen zu generieren.

Das BOR bietet weiterhin die Funktionalität Objekte zu erzeugen, Attribute zu lesen und Methoden aufzurufen. Diese Programmierung mit Geschäftsobjekten wird in den nächsten zwei Abschnitten beschrieben.

4.5 Geschäftsobjektkonzepte im Vergleich

In Abschnitt 3.4 sind zwei grundlegende Geschäftsobjektkonzepte beschrieben, die sich jeweils aus den Standardisierungsbemühungen der OMG und der OAG ergeben. Das R/3 Geschäftsobjektmodell ähnelt im derzeitigen R/3-Releasestand stark den Vorschlägen der OAG. Im OAG-Entwurf werden *Business Object Documents* zur Interaktion zwischen den Systemen benutzt. Diese Dokumente definieren dabei ein standardisiertes Austauschformat, welches allen beteiligten Systemen bekannt ist und von diesen benutzt wird. Überträgt man den Begriff des *Business Object Documents* in das R/3-Umfeld, so läßt sich die Ähnlichkeit mit den Methoden (BAPIs) der R/3-Geschäftsobjekte erkennen. BAPIs werden in ihrer Semantik vor allem durch den Aufbau ihrer Parameter bestimmt.
Die OAG strebt aufgrund der Unterschiedlichkeit der Systeme keine Interaktion der Systeme auf Ebene von Objekten an. Dies ermöglicht die Konstruktion von Schnittstellen, die ohne größere Schwierigkeiten von allen Systemen benutzt werden können. Auf der anderen Seite stellt das Modell der *Business Object Documents* keine objektorientierte Integration im eigentlichen Sinne dar, da ein *Business Object Document* lediglich eine Datenkapselung anbietet, jedoch selber kein Verhalten implementiert.

Mit einem BAPI stellt die SAP einer externen Anwendung kein Geschäftsobjekt im Sinne der OMG zur Verfügung, sondern nur die Möglichkeit, die Funktionalität der R/3-Geschäftsob-

[5] Funktionsbausteine **SWO_BUSINESS_OBECT_GET** und **RPY_MODELTYPE_READ**

jekte zu nutzen. Das fachliche R/3-Objektmodell im *Business Object Repository* entspricht den Anforderungen der OMG-Spezifikation. *Plug-and-Play*-fähige Geschäftsobjekte im Sinne der OMG abstrahieren vom hinterliegenden System, das dadurch an Bedeutung verliert. Demgegenüber erfordert die Nutzung der BAPIs eine genaue Kenntnis der im R/3-System verborgenen Prozesslogik und des Datenmodells, da sich daran die BAPI-Semantik sowie die Signaturen der BAPIs eng orientieren[6].
So erfordert die Benutzung der BAPI-Schnittstelle die genaue Kenntnis des R/3-Organisationsmodells mit Begriffen wie z.B. Geschäftsbereich, Buchungskreis und Verkaufsorganisation. Weiterhin besitzt ein R/3-Geschäftsobjekt verschiedene komplexe Teilsichten (u.a. Vertriebs- und Buchhaltungssicht), auf die sich über einen BAPI nur in Auszügen zugreifen läßt. Dieses Wissen besitzen nur spezialisierte SAP-Berater, so daß eine R/3-Integration weiterhin deren Spezialkenntnisse erfordert.

Ein gänzlich anderer Aspekt ist der, daß mit der Abstraktion vom hinterliegenden System eines Geschäftsobjekts das System beliebig austauschbar wird. Dies kann nicht im Interesse des Systemherstellers SAP sein, da damit die Bindung an sein Produkt R/3 geschwächt wird, was auch eine Minderung des kommerziellen Erfolgs des R/3 bedeutet.
Eine Komponentisierung des R/3 in *Business Komponenten* und Geschäftsobjekte wird also sicherlich nicht so weit gehen, daß das R/3-Geschäftsobjekt Kunde ohne größeren Aufwand mit Geschäftsobjekten in den Produkten der SAP-Mitbewerber kommunizieren kann. Nichtsdestotrotz ist die SAP Mitglied sowohl der OAG als auch der OMG und arbeitet nach eigenen Aussagen aktiv am Standardisierungsprozeß mit.

4.6 Implementierung von Objekttypen

Die R/3-Objekttypen sind gegenwärtig (Releasestand 4.0b) durch Makro-ABAP realisiert. Makro-ABAP simuliert eine objektorientierte Sprache mit Konstrukten wie Klassen und Schnittstellen. Die Simulation ist erforderlich, da die Programmiersprache ABAP selbst keine objektorientierte Semantik besitzt.
Die weiter unten im Abschnitt 4.8.2 dieser Arbeit beschriebene objektorientierte Erweiterung der Sprache ABAP, OO-ABAP, ist noch nicht vollständig implementiert und wird aus diesem Grund für die Implementierung von Objekttypen nicht benutzt. Es ist jedoch bereits absehbar, daß OO-ABAP das Makro-ABAP-Konzept in späteren R/3-Releases ablösen wird.
Die Simulation einer objektorientierten Sprache wird durch eine Menge von ABAP-Makros realisiert, die sich ihrerseits auf eine Menge von Funktionsbausteinen abstützen. Die Makroaufrufe werden zur Übersetzungszeit durch das R/3-System expandiert und transparent für den Entwickler gespeichert.

Einem BOR-Objekttyp ist genau ein Programm zugeordnet, welches die Implementierung des Objekttyps darstellt. Dieses Programm ist ein ABAP-Programm des Typs Subroutinenpool, welches - analog zu dem Modulpool einer SAP-Transaktion - nicht ausführbar ist und lediglich aus Subroutinen (FORMs) und programmlokalen Datendefinitionen besteht. Die Subroutinen lassen sich durch andere ABAP-Programme ausführen und stellen so die Funktionalität des Geschäftsobjekts dar. Innerhalb dieser Subroutinen ist ein Zugriff auf die Datendefinitionen

[6] Beispielsweise erfordert das Anlegen eines Kunden über den BAPI **Customer.CreateFromData** die Angabe eines R/3-Referenzkunden, der als Vorlage benutzt wird.

möglich, dem aufrufenden ABAP-Programm bleiben diese jedoch verborgen. Zwischen mehreren Manipulationen eines Objektes bleiben die Daten als faktisch statische Variablen des „Objektexemplars“ erhalten und können so einen Objektzustand abbilden, der gegenüber dem rufenden Programm gekapselt ist.

Die Funktionsbausteine, auf die sich die Makrodefinitionen abstützen, sind in der Funktionsgruppe **SWC** organisiert. Die Variablen dieser Funktionsgruppe stellen globale Variablen für die zugehörigen Funktionsbausteine dar und realisieren so ein Laufzeitsystem für die Ausführungszeit eines Makro-ABAP-Programmes. Funktionsbausteine der Funktionsgruppe **SWC** lassen sich prinzipiell wie andere Funktionsbausteine auch, über den Mechanismus der RFC-Automation aufrufen. Damit wäre eine Nutzung dieser Schnittstelle auch für externe Anwendungen möglich. Praktisch stellt dies jedoch ein Problem dar, da zum einen diese Schnittstelle nicht dokumentiert ist und zum anderen im jetzigen Releasestand noch fehlerhaft ist[7]. Das Auslesen von Metainformationen zu Geschäftsobjekten und BAPIs funktioniert jedoch zuverlässig und wird sowohl durch die im Rahmen dieser Arbeit erstellte Schnittstelle als auch durch kommerzielle Schnittstellen vollzogen.

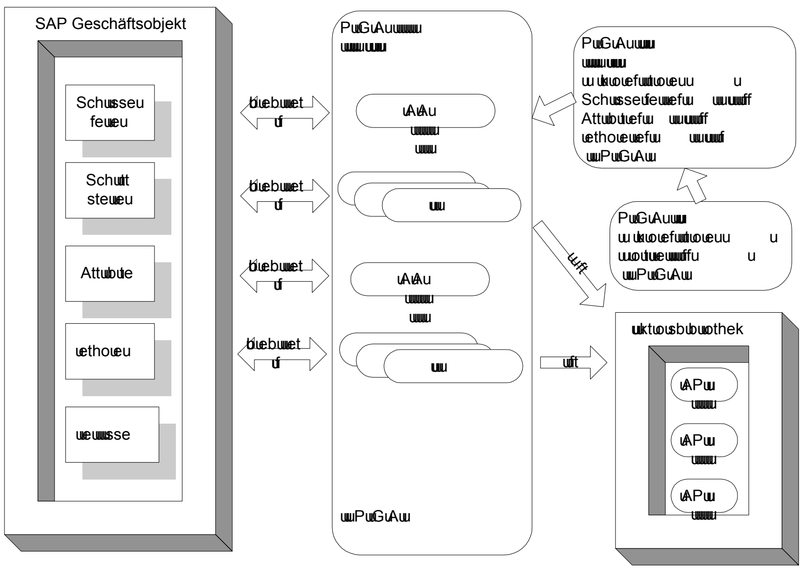

Abbildung 4.6: Das R/3-Objektmodell

Die für die Implementierung und Nutzung von Objekttypen erforderlichen Makrodefinitionen werden in Form der zwei *Includeprogramme* **OBJECT** und **CNTN01** zur Verfügung gestellt:

OBJECT enthält Makros, welche die Definition einer Klasse und den Zugriff auf Schlüsselfelder, Attribute und Methoden eines Objekts in gültige ABAP-Konstrukte umsetzen. Die ABAP-Konstrukte für die Schlüsselfelder und Attribute eines Objekttyps stellen dabei Datendeklarationen dar; der Zugriff auf Attribute und die Definition von Methoden werden auf Unterprogramme (ABAP-Kommando **FORM**) abgebildet.

[7] So scheitert z.B. das Auslesen von Attributen eines Objekts mit dem SAP BAPI-ActiveX-Control mit einer Schutzverletzung in Visual Basic.

Abbildung 4.7: ABAP *Dictionary*-Strukturen **OBJ_RECORD** und **SWOTRTIME**

CNTN01 Das *Includeprogramm* **CNTN01** enthält Makrodefinitionen für den Zugriff auf den Parametercontainer. Da Makro-ABAP-Objekte nicht wirklich Daten kapseln können, wird ein sogenannter Container bei allen Aktionen auf diesem Geschäftsobjekt als Parameter übergeben. Der Parametercontainer speichert die Daten eines Geschäftsobjektes. Er kann sowohl Feldwerte, Strukturen als auch Referenzen auf andere Objekte enthalten. Einträge in diesem Container werden über ihren (Parameter- bzw. Attribut-)Namen identifiziert. Zum lesenden und schreibenden Zugriff auf die Containereinträge existieren wiederum Makros.

Von diesen Makrodefinitionen wird in dem Programm Gebrauch gemacht, welches die Implementierung des Geschäftsobjekts darstellt. Das Programm ist unterteilt in einem Deklarationsteil und einem Implementierungsteil:
Der Deklarationsteil bildet die Schlüsselfelder und Attribute als globale Variablen innerhalb des Programms ab. Im Implementierungsteil werden die Zugriffe auf virtuelle Attribute und Methoden des Objekts durch Unterprogramme realisiert. Die Implementierung des vordefinierten Objekttyps **Kunde** zeigt Anhang D. Das BOR unterstützt den Entwickler beim Anlegen des zu erstellenden Programmes, indem aus der fachlichen Sicht im BOR die zugehörigen Makro-ABAP-Programmfragmente erstellt werden.

4.7 Programmierung mit Geschäftsobjekten

Im folgenden wird die Programmierung mit Makro-ABAP im Detail vorgestellt. Um diese Art der Programmierung deutlicher zu machen, sei auf das Beispiel in Abbildung 4.8 verwiesen, welches zu einem Kunden im R/3 durch den Aufruf des BAPIs **Customer.GetDetail** Detailinformationen ausliest.

Eine Objektvariable wird durch eine Variable des Typs **SWC_OBJECT** repräsentiert. **SWC_OBJECT** wird durch die ABAP *Dictionary*-Struktur **OBJ_RECORD** (siehe Abbildung 4.7) spezifiziert. Der essentielle Eintrag in **OBJ_RECORD** stellt das Feld **HANDLE** dar. Dieses Feld wird durch das Makro-ABAP-Laufzeitsystem mit einer für die Laufzeit des Programmes eindeutigen Nummer belegt und dient im folgenden zur Identifikation des Objektes.

Das Makro-ABAP-Laufzeitsystem organisiert die Verwaltung der Objektvariablen durch die Tabelle **RUNTIME** des Typs **SWOTRTIME** (siehe Abbildung 4.7). Die Tabelle ist im Rahmenprogramm der Funktionsgruppe SWC enthalten und steht den Makro-ABAP Funktionsbausteinen als globale Variable zur Verfügung. In die Tabelle **RUNTIME** werden Objektvariablen nach ihrer Erzeugung (**SWC_CREATE_OBJECT**) eingetragen und nach ihrer Freigabe (**SWC_FREE_OBJECT**) wieder ausgetragen. Das Feld **HANDLE** aus der Struktur **OBJ_RECORD** taucht hier als Schlüsselfeld **OBJECT** wieder auf.

```
INCLUDE <OBJECT>.
DATA:
OBJ TYPE  SWC_OBJECT ,                    "Struktur OBJ_RECORD
OBJKEY LIKE SWOTOBJID-OBJKEY.             "char(70)
DATA:
"Schlüsselvar für Kundennr,Verkaufsorg,Vertriebsweg,Sparte
CUSTOMERNO LIKE BAPIKNA103-CUSTOMER VALUE '0000000033',
PI_SALESORG LIKE BAPIKNA102-SALESORG VALUE '0001',
PI_DISTR_CHAN LIKE BAPIKNA102-DISTR_CHAN VALUE '01',
PI_DIVISION LIKE BAPIKNA102-DIVISION VALUE '01'.
DATA:
PE_ADDRESS LIKE BAPIKNA101,  "Exportparam für BAPI ExistenceCheck
RETURN LIKE BAPIRETURN,      "Exportparam für BAPI ExistenceCheck
CUSTOMERNO1 LIKE KNA1-KUNNR, "Variable für Kundennr
CITY LIKE KNA1-ORT01,        "Variable für Ort des Kunden
SALESAREAS LIKE BAPIKNVVKY OCCURS 5 WITH HEADER LINE. "Tabelle für GetDetail
SWC_CONTAINER CONTAINER.                  "Dekl. Containervar
SWC_CREATE_CONTAINER CONTAINER.           "Init. Containervar
CONCATENATE CUSTOMERNO PI_SALESORG        "Konkat. Teilschlüssel
            PI_DISTR_CHAN PI_DIVISION     INTO OBJKEY.
SWC_CREATE_OBJECT OBJ  'KNA1' OBJKEY.     "Erzeuge Kundenobjekt
SWC_CALL_METHOD OBJ 'GetDetail' CONTAINER."Aufruf Methode(BAPI) GetDetail
SWC_GET_ELEMENT CONTAINER 'PEADDRESS' PE_ADDRESS."Detailinfo zu Kunden
SWC_GET_PROPERTY OBJ  'City' CITY.        "Attributzugriff City
SWC_GET_PROPERTY OBJ 'CustomerNo' CUSTOMERNO1.   "Attributzugriff CustomerNo
SWC_CALL_METHOD OBJ 'GetSalesAreas' CONTAINER.   "Aufruf BAPI GetSalesAreas
SWC_GET_TABLE CONTAINER 'SALESAREAS' SALESAREAS ."Auslesen Tabelle SALESAREAS
```

Abbildung 4.8: Programmierung mit Geschäftsobjekten in Makro-ABAP

Bei der Objekterzeugung durch **SWC_CREATE_OBJCT** werden als Parameter der Objekttyp (**SWOTRTIME-OBJTYPE**) sowie konkateniert die Werte der Schlüsselfelder des Geschäftsobjektes (**SWOTRTIME-OBJKEY**) angegeben. Der Objekttyp ist dabei als einfache Zeichenkette zu verstehen, die meist die Form „BUSXXXX“ hat und wobei „XXXX“ eine Nummer darstellt. Da bei diesem Aufruf die Schlüsselfelder anzugeben sind, ist es offensichtlich, daß Makro-ABAP vollständig auf den persistenten in der Datenbank gespeicherten Geschäftsobjekten des R/3 arbeitet. Innerhalb der Implementierung von Geschäftsobjekten ist über das Konstrukt **SELF** ein selbstbezüglicher Verweis auf das aktuelle Objekt möglich.

Mit Hilfe einer so erzeugten Objektvariablen lassen sich Attribute auslesen, Methoden des Objekts rufen als auch Ereignisse auslösen. Auf Objektattribute ist lediglich lesender Zugriff möglich, da die Manipulation der Attribute über den Aufruf von Methoden erfolgt, welche die persistente Repräsentation des Objektes ändern. Eine Übersicht über die ABAP-Makroaufrufe gibt Tabelle 4.1.

Charakteristisch an Makro-ABAP-Aufrufen ist der Aufruf von Makro-Funktionen mit jeweils einen Parameter für die Objektvariable (<**Objekt**>), der durchzuführenden Aktion

Erzeugen einer Objektreferenz
SWC_CREATE_OBJECT <Objekt><Objekttyp><Objektschlüssel>
Typ und Schlüssel lesen
SWC_GET_OBJECT_TYPE <Objekt><Objekttyp>
SWC_GET_OBJECT_KEY <Objekt><Objektschlüssel>
Aufruf einer Methode
SWC_CALL_METHOD <Objekt><Methode><Container>
SWC_CALL_METHOD SELF <Methode><Container>
Lesender Zugriff auf Attribute
SWC_GET_PROPERTY <Objekt><Attribut><Attributwert>
SWC_GET_TABLE_PROPERTY <Objekt><Attribut><Attributwert>
SWC_GET_PROPERTY SELF <Attribut><Attributwert>
SWC_GET_TABLE_PROPERTY SELF <Attribut><Attributwert>
Ausnahmen auslösen
EXIT_OBJECT_NOT_FOUND
EXIT_CANCELLED
EXIT_NOT_IMPLEMENTED
EXIT_PARAMETER_NOT_FOUND
EXIT_RETURN <Ausnahme><Var1><Var2><Var3><Var4>
Objektattribute aus der Datenbank auffrischen
SWC_REFRESH_OBJECT <Objekt>

Tabelle 4.1: Makro-ABAP-Funktionen

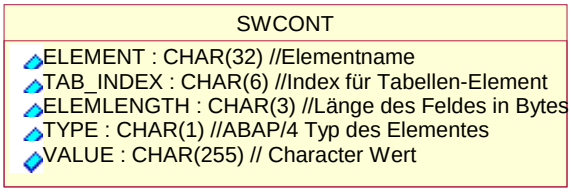

Abbildung 4.9: ABAP *Dictionary*-Struktur **SWOCONT**

sowie einer Puffervariablen, in welcher der zu lesende oder schreibende Wert gespeichert ist (<**Wert**>). Der Typ der Variablen sollte sich an dem zu erwartenden Typ orientieren. Dies zu gewährleisten ist Aufgabe des Anwendungsentwicklers. Durch das Laufzeitsystem erfolgt keine Überprüfung, ob der Puffervariablentyp im Einklang mit dem Ergebnistyp steht. Aufgrund der automatischen Typkonvertierungen in ABAP ist dies nicht zwingend notwendig. Dies kann jedoch zu sehr schwer auffindbaren Fehlern führen.

Fehler werden durch die Geschäftsobjektimplementierung als Ausnahmen bekanntgegeben (**EXIT_RETURN**). Der Anwendungslogik wird diese Ausnahmen über die Systemvariable **SY-SUBRC** signalisiert. In weiteren Systemvariablen ist erläuternder Text zum Fehler enthalten (Systemnachrichtenfelder **SY-MSGV**). Das Prüfen dieser Felder ist jedoch nach jedem Makro-ABAP-Aufruf durch das Anwendungsprogramm zu übernehmen. Zur Erleichterung der Fehlersuche ist es möglich, Methodenaufrufe und den Zugriff auf Attribute im Einzelschrittmodus auszuführen.

Bei einem Methodenaufruf werden die Methodenparameter und das Ergebnis (sofern vorhanden) in der Containertabelle abgelegt, bzw. aus ihr gelesen. Ebenso werden auch die virtuellen Attribute[8] aus Performanzgründen nach ihrer Ermittlung im Container abgelegt. Elemente in diesem Container werden über ihren Namen identifiziert. Dies sind die Namen von Methodenparametern oder Attributen. Diese Methodik wurde gewählt, da Makro-ABAP-Objekte keine Objekte im engeren Sinne darstellen. Sie sind nicht in der Lage, ihre Daten zu speichern. Die Datenstruktur des Containers wird durch die ABAP *Dictionary*-Struktur **SWCONT** (siehe Abbildung 4.9) definiert. Die Zeilen einer Tabelle werden im Container ebenfalls in mehreren Zeilen abgelegt. Das Feld **VALUE** hält eine Zeichenkettenrepräsentation des zu speichernden Wertes. Eine Übersicht über die Makro-ABAP-Funktionen für den Zugriff auf den Container zeigt Tabelle 4.2.

Das Lesen von Attributen und das Ausführen von Methoden an einem Objekt wird durch den generischen Funktionsbaustein **SWO_INVOKE** geleistet. Dazu wird dem Funktionsbaustein ein Objekthandle **OBJECT** sowie der Name des Attributs oder der Methode (**VERB**) übergeben. Die Ergebnisse werden im **CONTAINER** abgelegt und können wiederum über Makro-ABAP-Funktionen ausgelesen werden. Die Signatur des Funktionsbausteins zeigt Abbildung 4.10. Der Funktionsbaustein stellt somit einen reflexiven Mechanismus zur Ausführung beliebiger Aktionen auf einem Geschäftsobjekt dar.

R/3-Geschäftsobjekte sind in einen systemweiten Ereignis-Mechanismus eingebunden. Neben der Auslösung eines Ereignisses in der Implementierung des Geschäftsobjekts kann ein Ereig-

[8] Virtuelle Attribute sind Attribute, die durch eine Berechnungsvorschrift ermittelt werden

Container-Datenstruktur deklarieren und initialisieren
SWC_CONTAINER <Container>
SWC_CREATE_CONTAINER <Container>
Objektreferenz, Feldwert oder Tabelle in Container schreiben
SWC_SET_ELEMENT <Container><Element><Wert>
SWC_SET_TABLE <Container><Element><Wert>
Objektreferenz, Feldwert oder Tabelle aus Container lesen
SWC_GET_ELEMENT <Container><Element><FeldVariable>
SWC_GET_TABLE <Container><Element><TabellenVariable>
Element in Container löschen
SWC_DELETE_ELEMENT <Container><Element>

Tabelle 4.2: Makro-ABAP-Funktionen für den Containerzugriff

```
FUNCTION SWO_INVOKE.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"IMPORTING
*"  VALUE(ACCESS) LIKE  SWOTP-ACCESS DEFAULT 'C'
*"  VALUE(OBJECT) LIKE  SWOTRTIME-OBJECT
*"  VALUE(VERB) LIKE  SWOTINVOKE-VERB DEFAULT SPACE
*"  VALUE(PERSISTENT) LIKE  SWOTP-PERSISTENT DEFAULT SPACE
*"  VALUE(REQUESTER) LIKE  SWOTOBJID STRUCTURE  SWOTOBJID DEFAULT SPACE
*"  VALUE(SYNCHRON) LIKE  SWOTINVOKE-SYNCHRON DEFAULT '*'
*"  VALUE(UNSORTED_CONTAINER) LIKE  SWOTINVOKE-SYNCHRON DEFAULT SPACE
*"EXPORTING
*"      VALUE(RETURN) LIKE  SWOTRETURN STRUCTURE  SWOTRETURN
*"      VALUE(VERB) LIKE  SWOTINVOKE-VERB
*"TABLES
*"       CONTAINER STRUCTURE  SWCONT
*"----------------------------------------------------------------------
```

Abbildung 4.10: Funktionsbaustein **SWO_INVOKE** für den generischen Objektzugriff

nis von einem beliebigen Anwendungs- oder Systemprogramm (dem Ereigniserzeuger) durch Aufruf des Funktionsbausteins **SWE_EVENT_CREATE** erzeugt. Weitere Möglichkeiten der Ereigniserzeugung, die keine weitere Programmierung erfordern, sind in folgenden Fällen möglich:

- Bei Objekten, welche die Allgemeine Statusverwaltung nutzen
- Bei einer Anbindung an die Nachrichtensteuerung
- Beim Schreiben von Änderungsbelegen

Die allgemeine Statusverwaltung bietet die Möglichkeit, den aktuellen Bearbeitungszustand eines Anwendungsobjektes in Form von Stati (*Flags*) zu dokumentieren. Das Ändern von komplexen betriebswirtschaftlichen Objekten kann durch die Erstellung von Änderungsbelegen protokolliert werden. Diese vom R/3 durchgeführten Aktionen können mit dem Auslösen eines Ereignisses verbunden werden.

Interessenten eines Ereignisses (Ereignisverbraucher) registrieren sich in Ereignis-Verbraucher-Kopplungstabellen. Der dabei erzeugte Eintrag besteht dabei aus dem Geschäftsobjekt-Typnamen, dem Ereignisnamen sowie einem Funktionsbaustein, der im Fall des Ereigniseintritts zu rufen ist. Teil der Schnittstelle des registrierten Funktionsbausteins ist der bereits erwähnte Container-Parameter, der beim Aufruf des Funktionsbausteins mit Werten gefüllt ist, welche das ereignisauslösende Geschäftsobjekt identifizieren. Diese Informationen können dann im Verbraucherfunktionsbaustein entsprechend genutzt werden.

In Erweiterung zum eben beschriebenen Mechanismus, der die Ereignisbehandlung auf Ebene des Geschäftsobjekttyps beschreibt (Typkopplung), ist weiterhin eine Ereignisbehandlung auf Ebene konkreter Exemplare von Geschäftsobjekten möglich (Exemplarkopplung).

4.8 Objektorientierung und ABAP

In die Programmiersprache ABAP haben mit dem R/3-Release 4 objektorientierte Erweiterungen Einzug gehalten. Das ABAP des Release 4 wird im folgenden kurz als OO-ABAP bezeichnet.

Der Einzug eines neues Programmierparadigmas in ABAP bedeutet nicht die Ablösung der bisherigen ABAP-Programmieransätze, sondern eine Ergänzung, wie sie die Sprache ABAP seit ihrer Entstehung im R/2-System bereits mehrfach erfahren hat. Diese Erweiterungen sind noch nicht abgeschlossen. So sind Teile der OO-ABAP-Spezifikation bis jetzt nur in konzeptioneller Form beschrieben, aber noch nicht implementiert. Die in diesem Abschnitt vorgestellten Konzepte fußen dabei auf Informationen aus [Matzke 98], der R/3-Onlinehilfe [SAP 98g] und Berichten (*White Papers*) der SAP ([SAP 97a], [SAP 97b]), die auf dem WWW-Server der SAP zur Verfügung stehen.

Um die Objektorientierung in ABAP zu verstehen, sind diese Erweiterungen im Kontext der Historie und der Eigenschaften von ABAP zu sehen, die im folgenden Abschnitt skizziert werden sollen.

4.8.1 Historie und Eigenschaften von ABAP

ABAP steht für **A**dvanced **B**usiness **A**pplication **P**rogramming Language und hat ihre Ursprünge in einer einfachen Programmiersprache zur Erzeugung von Reports im R/2-System. Im Laufe der Zeit hat ABAP vielfache Erweiterungen erfahren und vereinigt so Sprachelemente aus imperativen, ereignisorientierten, datenbankorientierten und seit Release 4 auch objektorientierten Programmiersprachen. ABAP stellt damit eine hybride Sprache dar.

Die Komplexität von ABAP ergibt sich aus der Vielzahl von Sprachelementen, die zudem noch in einer großen Menge von Varianten anwendbar sind. Aus Gründen der Abwärtskompatibilität werden veraltete Sprachelemente nicht aus der Sprache entfernt, sondern lediglich in der ABAP-Sprachbeschreibung als obsolet gekennzeichnet, was zusätzlich die Komplexität der Sprache erhöht. Die Sprache ABAP läßt sich nicht durch den Anwendungsentwickler mit neuen Anweisungen erweitern. Um dennoch Funktionen, die in einer Vielzahl von ABAP-Programmen benötigt werden, zentral zu speichern, besteht die Mglichkeit diese in *Includeprogrammen* abzulegen.

Die ABAP Sprachelemente lassen sich wie folgt klassifizieren (In Klammern sind jeweils Beispiele der jeweiligen Sprachelemente angegeben):

Deklarative Schlüsselwörter definieren Datentypen oder deklarieren Datenobjekte (**TYPES, DATA, TABLES**).

Datenbankanweisungen greifen über die Datenbankschnittstelle schreibend und lesend auf Datenbanktabellen des zentralen Datenbanksystems zu (**INSERT, MODIFY**).

Operationale Schlüsselwörter ermöglichen Datenmanipulationen wie Berechnungen und Zuweisungen (**ADD, MOVE, COMPUTE**).

Steuernde Schlüsselwörter steuern den Ablauf eines ABAP-Programms innerhalb eines Verarbeitungsblocks abhängig von bestimmten Bedingungen und realisieren die aus imperativen Sprachen bekannten Kontrollstrukturn wie Schleifen und Vezweigungen. (**IF, WHILE, CASE**).

Aufrufende Schlüsselwörter rufen Verarbeitungsblöcke im gleichen oder anderen ABAP-Programmen auf oder verzweigen vollständig in andere Programme (**PERFORM, CALL, SUBMIT, LEAVE TO**).

Modularisierungs-Schlüsselwörter definieren Verarbeitungsblöcke in einem ABAP-Programm. Verarbeitungsblöcke sind Gruppen von Anweisungen, die während der Ausführung eines ABAP-Programms von anderer Stelle aufgerufen werden, stellen also Unterprogramme dar. Es wird unterschieden in Definierende Schlüsselwörter und Zeitpunkt-Schlüsselwörter.

- Zeitpunkt-Schlüsselwörter stellen die bereits erwähnten Sprachelemente ereignisorientierter Sprachen dar. Zeitpunkt-Schlüsselwörter ermöglichen es, Programmteile an Ereignisse zu binden. Als Ereignisse sind dabei z.B. der Druck von Funktionstasten oder auch der Beginn einer neuen Seite bei der Ausgabe eines Reports zu verstehen (**AT USER-COMMAND, TOP-OF-PAGE**).

- Definierende Schlüsselwörter kennzeichnen Unterprogramme, Funktionsbausteine, und Dialogmodule. Diese sind aus einem ABAP-Programm oder aus einer Bildschirmlogik aufrufbar (**FORM, FUNCTION, MODULE**).

OO-ABAP-Schlüsselwörter beschreiben die Definition von Klassen, die Erzeugung von Objekten und den Zugriff auf Objekte.

4.8.2 OO-ABAP

Seit dem Release 4 existieren die objektorientierten Erweiterungen von ABAP. Sie stellen also ein *Add-On* zum bisherigen Funktionsumfang der Sprache dar. Dieses *OO-Add-on* stellt nicht das bisherige Programmierkonzept des R/3 in Frage, sondern soll dieses lediglich ergänzen.

Gründe für die Entwicklung von OO-ABAP sind „Wiederverwendbarkeit, Pflegbarkeit und Qualität des Codes herbeizuführen“ [SAP 97a, S. 3]. Insbesondere die SAP-Transaktionen (=ABAP-Programme) beinhalten eine große Menge von Funktionalität, die nur schwer durch andere Programme nutzbar ist. Der Grund liegt darin, daß SAP-Transaktionen keine wirkliche Schnittstelle besitzen. Wiederverwendbarkeit ist also insbesondere für die Anwendungslogik in SAP-Transaktionen erforderlich. Aber auch Funktionalität, die über Funktionsbausteine bereitsteht, würde von den Zielen Wiederverwendbarkeit, Pflegbarkeit und Qualität profitieren. Funktionsbausteine sind durch ihre definierte Schnittstelle prinzipiell widerverwendbar. Problematisch ist jedoch die fehlende Strukturierung der rund 32000 Funktionsbausteine, die nur über die Funktionsgruppen in eine gewisse logische Ordnung gebracht werden. Ein weiteres Problem ist, daß die Schnittstellen von Funktionsbausteinen aufgrund von Weiterentwicklungen im R/3 nicht über mehrere SAP-Releases stabil sind. Eine Kapselung dieser Schnittstellen in stabile Methoden von Geschäftsobjekten würde dieses Problem lösen.

Ein objektorientiertes Modell der R/3-Funktionalität wurde bereits im Release 3 von der SAP auf der Basis von Geschäftsobjekten definiert. Das Ergebnis war das *Business Object Repository (BOR)*, welches bereits in Abschnitt 4.4 vorgestellt wurde. Die Geschäftsobjekte des BOR sind seitdem über Makro-ABAP in einer an objektorientierte Systeme angelehnte Art und Weise zugreifbar. Ein Problem ist jedoch, daß die R/3-Geschäftsobjekte nicht in das ABAP-Typsystem eingebunden sind und die Programmierung in der Folge sehr fehleranfällig und mühselig ist. Mit OO-ABAP versucht die SAP, dieses Manko zu beheben und dem fachliche objektorientierten Konzept des BOR auch eine objektorientierte Programmiersprache gegenüberzustellen.

4.8.3 OO-ABAP-Klassen

Kern der objektorientierten Erweiterung von ABAP bildet erwartungsgemäß der Begriff der Klasse. Eine OO-ABAP-Klasse kapselt Daten und Verhalten und stellt so eine wesentliche Erweiterung des R/3-Programmierkonzepts dar, welches sich bisher an Programmen und Funktionen orientiert. Illustriert werden die in den folgenden Anschnitten beschriebenen Konzepte durch das OO-ABAP-Beispiel am Ende des Kapitels.

Eine Klasse besteht aus Attributen, Methoden, Ereignissen sowie in der Klasse sichtbaren Typen und Konstanten. Attribute, Methoden und Ereignisse werden in ihrer Abhängigkeit von

Exemplaren einer Klasse differenziert. Exemplarabhängige Attribute, Methoden und Ereignisse sind auf Objekten einer Klasse definiert, während Klassenattribute, Klassenmethoden und Klassenereignisse an die Klasse gebunden sind:

Attribute speichern den Zustand eines Objekts und werden unter Bezugnahme auf in der Klasse definierte Typen, ABAP *Dictionary*-Typen, ABAP-Basistypen oder OO-ABAP-Klassen definiert. Objekte können also andere Objekte über ihre Attribute referenzieren. Der Zugriff auf die Attribute und Methoden kann durch verschiedene Sichtbarkeitsbereiche gesteuert werden (**PUBLIC, PROTECTED, PRIVATE**). Alternativ kann ein öffentliches Attribut (Attribut mit Sichtbarkeitsbereich **PUBLIC**) als von außen nicht veränderbar gekennzeichnet werden (*READ-ONLY*). Eine weitere Variante stellt das virtuelle Attribut dar: Über ein virtuelles Attribut einer Schnittstelle besitzt eine implementierende Klasse die Möglichkeit, das Attribut durch eine Methode zu implementieren. Dies ist dort sinnvoll, wo ein Attribut durch andere Objekteigenschaften abgeleitet werden kann.

Methoden manipulieren das Objekt, an dem sie aufgerufen werden. Methodendeklarationen in OO-ABAP orientieren sich an der Aufrufsemantik von Funktionsbausteinen, d.h. eine Methode besitzt als Schnittstelle benannte Import-, Export- und Tabellenparameter. Von den Funktionsbausteinen übernommen wurde das Konzept der Ausnahmen. Ausnahmen sind nicht Teil der Klassendefinition, sondern werden durch eine Methode individuell ausgelöst. Das Auftreten einer Ausnahme ist weiterhin durch das aufrufende Programm nach jedem Methodenaufruf zu überprüfen (Überprüfung der Systemvariablen **SY-SUBRC**). Die Ausnahmebehandlung ist somit nicht Bestandteil des sprachlichen Konzepts OO-ABAP. Ausnahmen stellen keine Objekte dar. Neben dem Auslösen von Ausnahmen können im Methodenkörper das Eintreten von Ereignissen bekannt gegeben werden.

Ereignisse Objekte können Ereignisse signalisieren, auf deren Eintreten andere Objekte reagieren können. Ereignisse werden innerhalb von Methoden der Klasse oder in einer abgeleiteten Klasse ausgelöst[9]. Ein Ereignis stellt also kein eigenes Objekt dar. Ein Ereignis kann dabei an ein Objektexemplar als auch an die Klasse gebunden sein. Ereignisverbraucher (andere Objekte) müsssen durch eine Registrierung ihr Interesse an einem Ereignis anmelden. Bei der Registrierung, die sowohl zur Laufzeit eines Programms als auch zur Definitionszeit der Klasse erfolgen kann, gibt der Ereignisverbraucher eine Methode an, die bei Eintreten des Ereignisses zu rufen ist. Die Kopplung zwischen dem Ereignisauslöser und dem Ereignisverbraucher wird analog zu Makro-ABAP durch das Laufzeitsystem vorgenommen.

Schnittstellen Das Konzept der Schnittstelle stellt die Definition einer abstrakten Schnittstelle dar. Eine Schnittstelle beinhaltet im Gegensatz zu einer Klasse aber keine Implementierung. Klassen, die diese Schnittstelle anbieten wollen, müssen diese geeignet implementieren. Das Konzept der Schnittstelle kann zur Simulation einer Mehrfachvererbung zwischen Klassen benutzt werden, wie es z.B. in Java ausgiebig praktiziert wird. Eine Schnittstelle ermöglicht es, an Objekten verschiedener Klassen, die in keiner Vererbungsbeziehung stehen, Manipulationen durchzuführen und somit generische

[9] Schlüsselwort **RAISE EVENT** mit einer bestimmbaren Anzahl von Parametern

Operationen anzubieten. Im Unterschied zu Java definiert eine OO-ABAP-Schnittstelle neben Methodensignaturen alle in einer Klassendefinition möglichen Elemente, also z.B. Attribute, Ereignisse und Typen. Schnittstellen lassen sich hierarchisch gliedern, indem eine Schnittstelle eine andere Schnittstelle implementiert [Arnold, Gosling 96].

4.8.4 Klassen, Objekte und Objekterzeugung

Eine Klasse stellt einen Typ dar. Um auf ein Objekt zuzugreifen, ist eine Referenzvariable mit der Klasse zu typisieren. Eine Referenzvariable stellt eine Erweiterung des bisherigen ABAP-Typkonzepts aus Basistypen, Feldleiste und interner Tabelle dar. Die Objekterzeugung wird durch Aufruf der polymorphen Funktion **CREATE_OBJECT** initiiert, die als Argument diese typisierte Referenzvariable erhält, über die im folgenden das Objekt zugreifbar ist. Anhand des Typs der Referenzvariablen erkennt das ABAP-Laufzeitsystem die für die Objekterzeugung zu nutzende Klassendefinition. Über diese Referenz ist jedoch kein Zugriff auf die Speicherrepräsentation des Objektes möglich. Diese wird durch das Laufzeitsystem verborgen. Die Identität des Objektes wird ebenfalls durch das Laufzeitsystem intern verwaltet. OO-ABAP benutzt also eine referentielle Identifikation gegenüber der assoziativen Identifikation in Makro-ABAP. OO-ABAP unterstützt bei OO-ABAP-Objekten eine statische Typisierung[10]. Das heißt, die Zuweisung zueinander inkompatibler Objektvariablen oder der Aufruf von Methoden an Objekten werden bereits zur Kompilierzeit überprüft. Einer Objektvariablen der vordefinierten Klasse **OBJECT** lassen sich beliebig typisierte Objektvariablen zuweisen. **OBJECT** stellt so implizit die Wurzel der OO-ABAP-Klassenhierarchie dar. Bei einer Zuweisung von Referenzen bleibt die Referenzsemantik erhalten. Ein Kopieren (*deep copy*) der beteiligten Objekte wird also nicht vorgenommen. Die Angabe eines *Copy*-Konstruktors wie in C++ ist nicht vorgesehen.

Die Lebensdauer eines Objektes ist durch die Lebensdauer des erzeugenden Programmes beschränkt. Wird ein Objekt nicht mehr durch mindestens eine Variable referenziert, wird es automatisch aus dem Speicher entfernt (*Garbage Collection*). OO-ABAP-Objekte sind nicht von sich aus mit Mechanismen zur persistenten Speicherung ausgestattet. Soll ein OO-ABAP-Objekt persistent gespeichert werden, so ist der Objektzustand durch den Entwickler in einer persistenten Datenbanktabelle zu speichern. Das Konzept der Persistenz ist also in ABAP und OO-ABAP gleich. Durch die Einbettung einer Teilmenge von SQL-Schlüsselworten bieten ABAP und OO-ABAP Persistenzabstraktion.

Eine OO-ABAP-Klasse stellt im Gegensatz zu anderen OO-Sprachen wie Smalltalk und TL-2 und in Anlehnung an C++ kein eigenständiges Objekt dar. Die Objekterzeugung wird also nicht durch eine Klassenobjekt (Metaklasse) vorgenommen, sondern ist durch das Laufzeitsystem verborgen. Auf die Objekterzeugung kann durch eine Konstruktormethode Einfluß genommen werden. Pro Klasse ist es möglich, genau eine parametriesierbare Konstruktormethode zu definieren. Die Attribute eines Objekts befinden sich nach der Objekterzeugung in einem Initialzustand, der durch die Initialwerte der zugehörigen ABAP-Basistypen definiert wird (siehe Abschnitt 7.3.2). Die Verwaltung erzeugter Objekte obliegt der steuernden Anwendung. Ein Extensionsbegriff wie in einem objektorientierten Datenbankmodell ist in OO-ABAP unbekannt.

[10] Im Gegensatz zu den ABAP-Basistypen, die beinahe beliebig ineinander konvertierbar sind

4.8.5 Vererbung und Polymorphismus

Zwischen OO-ABAP-Klassen läßt sich eine Vererbungsbeziehung definieren. Eine erbende Klasse übernimmt die Eigenschaften der Superklasse, d.h. Attribute sowie Methoden und ihre Implementierung. Ererbte Methoden lassen sich überschreiben.

OO-ABAP unterstützt die Einfachvererbung, d.h. eine Klasse kann maximal eine Superklasse besitzen. Mehrfachvererbung läßt sich über das Konzept der OO-ABAP-Schnittstellen emulieren. Allen OO-ABAP-Klassen gemeinsam ist das (transitive) Erben von der Klasse **OBJECT**, auch wenn dieses in der Klassendefinition nicht explizit formuliert wird. Im augenblicklichen Stand der Realisierung ist noch nicht klar, welches Protokoll oder welche Dienste die Klasse **OBJECT** seinen Subklassen bietet. Wie bereits erwähnt wurde, stellt eine OO-ABAP-Klasse einen Typ dar. Durch die Vererbung wird eine erbende Klasse Subtyp der Superklasse. OO-ABAP unterscheidet also nicht zwischen der Subtyprelation und der Vererbungsrelation.

Gemäß genereller Aussagen in [SAP 97a][S. 5] wird OO-ABAP Polymorphismus unterstützen. Genauere Aussagen werden an keiner Stelle der zur Verfügung stehenden Quellen gemacht.

4.8.6 Einbettung der OO-ABAP-Klassen in das R/3-System

OO-ABAP-Klassen besitzen zwei verschiedene Sichtbarkeitsbereiche im R/3-System:

- Eine lokale Klasse wird innerhalb eines ABAP-Programmes definiert und läßt sich auch nur im Kontext dieses Programmes verwenden.
- Eine öffentliche Klasse läßt sich durch alle Programme des R/3-Systems benutzen. Öffentliche Klassen können in beliebigen Programmen definiert werden. Es bietet sich jedoch an, das von der SAP entwickelte Entwicklungswerkzeug, den *Class Builder*, zu benutzen.

Der *Class Builder* speichert und verwaltet die systemweit zur Verfügung gestellten Klassendefinitionen und orientiert sich mit seinem zentralen Ansatz an der Funktionsbausteinbibliothek. In dieser Klassenbibliothek werden als Endziel die jetzigen Inhalte des *Business Object Repositories* (also die Geschäftsobjekte) integriert [SAP 97a, S. 16]. Durch eine Implementierung der R/3-Geschäftsobjekte als OO-ABAP-Klassen wird es möglich, auf die R/3-Funktionalität objektorientiert und typsicher zuzugreifen. Damit wäre die Fokussierung und Beschränkung auf die BAPIs aufgehoben, wie es bis jetzt für externe Anwendungen der Fall ist.

4.8.7 Zusammenfassung und Stand der Realisierung

Zusammenfassend läß sich sagen, daß OO-ABAP das objektorientierte Paradigma in die Sprache ABAP integriert. Aufgrund seiner Konzeption als Erweiterung von ABAP ist damit aber nicht ein radikaler Wechsel des Programmierstils im R/3 verbunden. Bei einem Vergleich mit Makro-ABAP fällt auf, daß OO-ABAP an vielen Stellen dieselbe Semantik besitzt. Der essentielle Unterschied ist aber, daß OO-ABAP vollständig in die Sprache ABAP integriert ist und somit OO-ABAP-Konstrukte auch statisch typprüfbar sind. Die fehlende Typsicherheit ist es, die Makro-ABAP als Gegenstand einer externen R/3-Integration unbrauchbar erscheinen

läßt. Durch seine Rolle als *Add-On* zu ABAP besitzt OO-ABAP nicht die konzeptuelle Eleganz und Striktheit anderer objektorientierter Programmiersprachen. OO-ABAP trägt damit die gesamte ABAP-Entwicklungsgeschichte in sich.

Die Beurteilung erschwert, daß große Teile des OO-ABAP-Konzepts noch nicht realisiert sind (Releasestand 4.0b). So ist es noch nicht möglich, öffentliche Klassen zu definieren oder eine Vererbungsrelation zwischen Klassen zu definieren. Da also die systemweite Klassenbibliothek nur in Fragmenten existiert, orientiert sich die Programmierung mit OO-ABAP nicht an den R/3-Geschäftsobjekten. OO-ABAP wird momentan im R/3 nur im Rahmen eines Beispiel zur MS-Office-Integration auf COM-Basis benutzt. Auch in der R/3-Dokumentation wird OO-ABAP lediglich als Möglichkeit beschrieben, „beliebige OLE2-fähige Desktop Office Anwendungen an das R/3-System anbinden“ [SAP 98g]. OO-ABAP wird also nicht in den eigentlichen Trägern der R/3-Anwendungslogik (SAP-Transaktionen und Funktionsbausteine) benutzt und stellt so (noch) nicht das Mittel zur objektorientierten Programmierung im R/3 dar. Für die Integration mit externen Anwendungen muß auch nach einer Einführung von OO-ABAP als Trägern der R/3-Anwendungslogik ein beträchtlicher Aufwand auf Seiten der externen Anwendung getrieben werden, um eine objektorientierte Integration zu ermöglichen. Der Grund liegt im niedrigen Abstraktionsgrad der vorhandenen RFC-Automation-Schnittstelle

Inwieweit OO-ABAP aber wirklich Einfluß auf die weitere Entwicklung des R/3 zu einem OO-System nehmen wird, kann aber bis jetzt noch nicht abschließend geklärt werden. Die Grund liegt darin, daß auch im System mit Releasestand 4.5, in das die Autoren dieser Arbeit kurzzeitig Einblick nehmen konnten, keine wesentlichen Erweiterungen von OO-ABAP stattgefunden haben. In Gesprächen mit R/3-Unternehmensberatern wurde zudem deutlich, daß das Kriterium Objektorientiertheit kein strategisches Thema für die Auswahl des R/3-Systems beim Kunden darstellt. R/3 ist ein betriebswirtschaftliches Standardsoftwaresystem und die potentiellen R/3-Käufer sind primär an der Abdeckung ihrer betriebswirtschaftlichen Anforderungen interessiert. Ob diese Abdeckung nun durch objektorientierte Techniken erreicht wird, ist zunächst ohne Bedeutung. Eine objektorientierte Entwicklung würde jedoch beträchtliche Vorteile in der Wartbarkeit und Verständlichkeit des R/3-Systems bringen.
Zum Abschluß dieses Kapitels soll anhand eines Beispiels die Definition eines Geschäftsobjektes **Kunde** und die Nutzung dieser Klasse gezeigt werden. Die Klasse kapselt die BAPI-Aufrufe **Customer.ExistenceCheck** und **Customer.GetDetail** in einer OO-ABAP-Klasse **Kunde**. Das Beispiel stellt somit dar, wie zukünftige OO-ABAP-Klassen die R/3-Geschäftsobjekte implementieren könnten. Zur Umgehung des Problems der sich ändernden Funktionsbausteinsignaturen müßten natürlich die Typen der Methodenparameter logisch von den Typen des implementierenden Funktionsbausteins getrennt werden.

```
REPORT  ZCUSTOMER00.
*****************Definition Klasse Kunde*********************************
CLASS KUNDE DEFINITION.
PUBLIC SECTION.
DATA:
CUSTOMERNO LIKE KNA1-KUNNR, "Öffentliche Attribute
SALES_ORGANIZATION LIKE  BAPIORDERS-SALES_ORG,
DISTRIBUTION_CHANNEL LIKE  BAPIORDERS-DISTR_CHAN,
DIVISION LIKE  BAPIORDERS-DIVISION.
CLASS-METHODS: "Klassenmethode
```

```
CHECKEXISTENCE IMPORTING CUSTOMERNO LIKE BAPI1007-CUSTOMER
                         SALES_ORGANIZATION LIKE BAPIORDERS-SALES_ORG
                         DISTRIBUTION_CHANNEL LIKE BAPIORDERS-DISTR_CHAN
                         DIVISION LIKE BAPIORDERS-DIVISION
                EXPORTING
                 CUSTOMER_DATA LIKE  KNA1
                 CUSTOMER_NUMBER_OUT LIKE  BAPI1007-CUSTOMER
                 BAPIRETURN LIKE  BAPIRETURN .
METHODS:
GETDETAIL EXPORTING ADDRESS LIKE BAPIKNA101
                    BAPIRETURN LIKE BAPIRETURN     .
ENDCLASS.
*****************Implementierung Klasse Kunde*****************************
CLASS KUNDE IMPLEMENTATION.
METHOD CHECKEXISTENCE.
CALL FUNCTION 'BAPI_CUSTOMER_CHECKEXISTENCE'
     EXPORTING
          CUSTOMERNO           = CUSTOMERNO
          SALES_ORGANIZATION   = SALES_ORGANIZATION
          DISTRIBUTION_CHANNEL = DISTRIBUTION_CHANNEL
          DIVISION             = DIVISION
     IMPORTING
          CUSTOMER_DATA        = CUSTOMER_DATA
          CUSTOMER_NUMBER_OUT  = CUSTOMER_NUMBER_OUT
          RETURN               = BAPIRETURN
     EXCEPTIONS
          OTHERS               = 1.
ENDMETHOD.
METHOD GETDETAIL.
CALL FUNCTION 'BAPI_CUSTOMER_GETDETAIL'
     EXPORTING
          CUSTOMERNO    = CUSTOMERNO
          PI_SALESORG   = SALES_ORGANIZATION
          PI_DISTR_CHAN = DISTRIBUTION_CHANNEL
          PI_DIVISION   = DIVISION
     IMPORTING
          PE_ADDRESS    = ADDRESS
          RETURN        = BAPIRETURN
     EXCEPTIONS
          OTHERS        = 1.
ENDMETHOD.
ENDCLASS.
*****************Programm*************************************
DATA:
KUNDEVAR TYPE REF TO KUNDE,
BAPIRETURN LIKE BAPIRETURN,
                CUSTOMER_DATA LIKE  KNA1,
```

```
CUSTOMER_NUMBER_OUT LIKE BAPI1007-CUSTOMER,
ADDRESS LIKE BAPIKNA101.
START-OF-SELECTION.
  CREATE OBJECT KUNDEVAR.
  KUNDEVAR->CUSTOMERNO = '0000000033'.
  KUNDEVAR->SALES_ORGANIZATION = '0001'.
  KUNDEVAR->DISTRIBUTION_CHANNEL = '01'.
  KUNDEVAR->DIVISION = '01'.
  CALL METHOD KUNDE=>CHECKEXISTENCE
                  EXPORTING CUSTOMERNO = '0000000033'
                  SALES_ORGANIZATION = '0001'
                  DISTRIBUTION_CHANNEL = '01'
                  DIVISION = '01'
                  IMPORTING BAPIRETURN = BAPIRETURN
                  CUSTOMER_DATA = CUSTOMER_DATA.
*               customer_number_out = customer_number_out.
  CALL METHOD KUNDEVAR->GETDETAIL
                IMPORTING ADDRESS = ADDRESS.
```

Kapitel 5

Techniken und Produkte zum objektorientierten R/3-Zugriff

5.1 Techniken des objektorientierten externen R/3-Zugriffs

Wie bereits in Abschnitt 4.3 festgestellt wurde, sind für den Zugriff auf die SAP-Geschäftsobjekte nur deren Methoden (BAPIs) relevant, da nur über die BAPIs die Geschäftslogik des R/3 erreichbar ist. Weiterhin ist jeder BAPI auf genau einen Funktionsbaustein der Funktionsbausteinbibliothek abgebildet. Dies sind die Prämissen für den objektorientierten externen R/3-Zugriff. Für den externen Zugriff auf die BAPIs der R/3-Geschäftsobjekte gibt es zwei prinzipielle Möglichkeiten:

Zugriff über das *Business Object Repository (BOR)* Das BOR stellt das Geschäftsobjekt sowie dessen Methoden dar. Die Methoden sind durch einen Namen spezifiziert. Beim Zugriff über das BOR wird vor einem Aufruf eines BAPIs die Schnittstelle des BAPIs erfragt, um eine Unabhängigkeit von den Details des diese Methode implementierenden Funktionsbausteins zu erreichen. Die Schnittstelle eines BAPIs wird durch den Namen des Funktionsbausteins, den Namen, Längen und Typen der Parameter beschrieben. Bei strukturierten Parametern (Strukturen oder Tabellen) sind zusätzlich die genannten Informationen zu den Feldern zu ermitteln. Daran anschließend werden die entsprechenden Parameterstrukturen erstellt und der Funktionsbaustein gerufen. Dies stellt das von der SAP favorisierte Vorgehen dar. Die für diesen Zweck benötigten Funktionsbausteine sind:

- **SWO_TYPE_INFO_GET** zum Auslesen von Informationen zu einem Geschäftsobjekt
- **RFC_GET_FUNCTION_INTERFACE** zum Ermitteln der Signatur eines Funktionsbausteins
- **RFC_GET_STRUCTURE_DEFINITION** zum Auslesen der Feldinformationen zu einem strukturierten Parameter

Direkter Zugriff auf den Funktionsbaustein Beim direkten Zugriff auf den Funktionsbaustein entfällt die Ermittlung der Schnittstelle eines BAPIs, da die für den Aufruf

benötigten Informationen fest in die Programmlogik des aufrufenden Programms eincodiert sind.

Ein Vergleich der beiden Methodiken zeigt folgende Probleme auf:

- Beim direkten Zugriff auf den Funktionsbaustein ist eine stabile Schnittstelle zum R/3 nicht gewährleistet, da die Stabilität der BAPI-Signaturen in der Praxis nicht gewährleistet ist. Dieses Problem wird noch eingehender in Abschnitt 7.3.5.1 erläutert.
- Durch das Lesen der Schnittstelle zur Laufzeit vor dem eigentlichen BAPI-Aufruf ergibt sich ein Performancenachteil, der noch dadurch verstärkt wird, daß die Schnittstellen des BAPIs häufig komplex (viele Parameter mit vielen Feldern) sind und daß die verschiedenen BAPIs meistens unterschiedliche Parameterstrukturen verwenden. Es ergibt sich demzufolge auch keine Verbesserung der Performanzeigenschaften, wenn die gelesenen Schnittstellendefinitionen für eine eventuelle Wiederverwendung zwischengespeichert werden.
- Durch ein Erfragen der jeweiligen Schnittstelle vor dem Aufruf werden zwar immer richtige Parameterstrukturen erstellt. Der Mechanismus verhindert jedoch nicht, daß bereits an einem R/3-System getestete Anwendungen auf einem anderen System fehlschlagen, wenn die BAPI-Signaturen in den Systemen unterschiedlich sind. Die SAP deckt die Schwächen diese Mechanismus selbst auf, indem sie drei verschiedene BAPI ActiveX-Quelltextbeispiele für die verschiedenen R/3-Releasestände anbietet.

Im folgenden werden zwei Produkte vorgestellt, wobei das BAPI ActiveX Control den Weg über das BOR beschreitet, und der *Access Builder* direkt auf den Funktionsbaustein zugreift. Der *Access Builder* umgeht das Manko der nicht stabilen BAPI-Funktionsbausteinsignaturen dadurch, daß er in der Lage ist, die Schnittstellendefinitionen bei Bedarf aus dem BOR zu lesen und in generierte Java-Proxyklassen umzusetzen. Generell ist es jedoch Aufgabe des Entwicklers, die Verträglichkeit des Quelltextes mit dem R/3-System sicherzustellen. R/3-Strukturen besitzen keinen Versionsstempel, so daß kein automatisches Prüfen dieses Stempels und ein Reagieren darauf möglich ist. Beiden gemeinsam ist das Ziel, die R/3-Gesschäftsobjekte der externen Anwendung zur Verfügung zu stellen.

5.2 BAPI Active X Control

Das BAPI Active X Control ist eine auf dem COM-Mechanismus basierende Komponente der Firma SAP zum Zugriff auf die R/3-Geschäftsobjekte und ihre BAPIs[1].

Das BAPI ActiveX Control besteht aus den drei Kernkomponenten:

- BAPI ActiveX Control Object
- Business Object

[1]Der Zugriff auf Attribute ist ebenfalls möglich, ist jedoch noch nicht stabil implementiert; das *Control* bricht hiernach ab

- Collection Object

Das **BAPI ActiveX Control Object** stellt die Kernkomponente zum Erzeugen von Geschäftsobjekten im R/3, bzw. für den Zugriff auf sie dar. Bevor auf die Funktionalität eines SAP-Geschäftsobjekts zugegriffen werden kann, muß mit der Methode

Object GetSAPObject(String ObjectType,Variant ObjectKey1,... ,Variant ObjectKey10)

eine lokale Referenz auf eine persistentes Exemplar eines SAP-Geschäftsobjekt kreiert werden. Der *ObjectType* bezeichnet den Namen des Geschäftsobjekttyps gemäß seiner Definition im BOR, die Schlüssel müssen zum Aufruf von exemplarabhängigen BAPIs mit den entsprechenden Werten besetzt werden. Voraussetzung für die Erzeugung einer R/3-Geschäftsobjektreferenz ist das vorherige Anmelden am R/3, da bei der Objekterzeugung dynamisch auf das Vorhandensein des Geschäftsobjekttyps und der konkreten Ausprägung anhand der übergegebenen Schlüsselfelder geprüft wird.

Für den Aufruf von BAPIs sind weiterhin die Parameter von essentieller Bedeutung. Die Import- und Tabellenparameter sind vor dem Aufruf zu füllen. Nach dem BAPI-Aufruf sind die Export- und Tabellenparameter auszuwerten.
Um ein polymorphes Stubobjekt für einen (strukturierten) Parameter eines BAPIs zu erhalten ist die Methode

Object DimAs(Object BusinessObject, String Method, String Parameter)

zu benutzen. Um ein Parameterstubobjekt zu erhalten, muß dem R/3 neben einer Geschäftsobjektreferenz, der Name der Methode sowie der Name des Parameters angegeben werden. Das Parameterstubobjekt besitzt **Value**-Methoden, die es ermöglichen, auf Feldinhalte lesend und schreibend zuzugreifen. Auf Felder wird dabei über einen Index, bzw. über den Feldnamen zugegriffen. Nötige Typkonvertierungen werden dabei automatisch durchgeführt[2].
Bei der Ausführung der Methode **DimAs** durch dasVisual Basic-Laufzeitsystem werden die beschriebenen Informationen aus dem R/3 gelesen und entsprechende Parameterstrukturen erzeugt.

Die durch **GetSAPObject** erzeugte **Geschäftsobjektreferenz** (*Business Object*) ist polymorph, d.h. abhängig von den übergebenen Parametern stellt sie ein Handle auf unterschiedliche SAP-Geschäftsobjekttypen dar. Der Aufruf einer Methode, die dieses Geschäftsobjekt nicht besitzt, erzeugt einen Laufzeitfehler und kann nicht zur Kompilierzeit getestet werden. Beim Auftreten eines Laufzeitfehlers wird eine OLE-Ausnahme ausgelöst. Der Fehler wird durch eine Zahl (Fehlercode) genauer spezifiziert. Wie bereits geschildert sind Fehler auch bei einer bereits an einem R/3-System getesteten Anwendung nicht unwahrscheinlich, da sich über die verschiedenen SAP-Releasestände hinweg Geschäftsobjekte ändern können.

Metainformationen könnnen durch den Aufruf der Methode **Metainfo** angefordert werden. Die zurückgegebene Struktur ist vom Typ der SAP-Struktur **TOJTB**, welche die Grunddaten eines Geschäftsobjekts beschreibt.

[2]Die möglichen Konvertierungen sind nicht dokumentiert, müssen also durch Probieren gefunden werden.

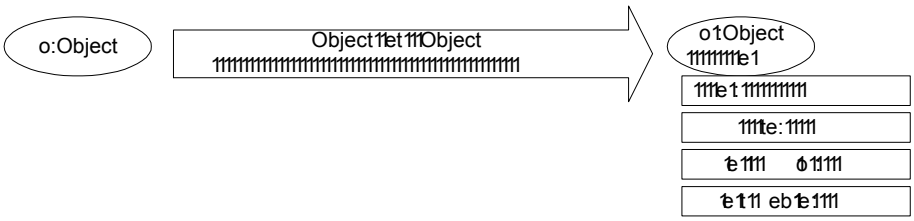

Abbildung 5.1: BAPI ActiveX: Erzeugung eines Geschäftsobjektes

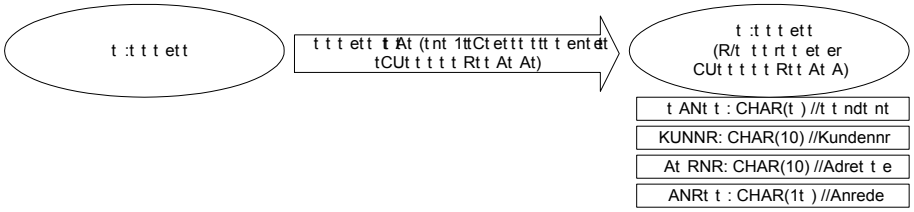

Abbildung 5.2: BAPI ActiveX: Erzeugung eines Funktionsparameters

Im Abbildung 5.3 ist eine beispielhafte Nutzung des *Controls* in *Visual Basic* dargestellt. Dabei wird mittels des BAPIs **C**ustomer.ExistenceCheck das Vorhandensein eines Kunden im R/3-System geprüft.

Auffällig bei diesem Quelltextbeispiel ist die Nutzung von Zeichenketten für eigentliche zahlorientierte Attribute wie die Kundennummer oder die Verkaufsorganisation. Dies wird jedoch durch das R/3-System so bestimmt, da die entsprechenden Felder in den Datenbanktabellen ebenfalls als Zeichenketten definiert sind.
Weiterhin fällt beim Methodenaufruf die Ähnlichkeit zum Aufruf eines Funktionsbausteins auf. Die strukturierten Exportparameter **oReturn** und **oData** werden nach dem BAPI-Aufruf durch das *Control* aufgebaut und stehen danach mit einer Feldsemantik zur Verfügung.

5.3 *Access Builder for SAP R/3*

Der *Access Builder for SAP R/3* der Firma IBM stellt eine Programmierschnittstelle zum Zugriff auf die R/3-Geschäftsobjekte auf Basis der Programmiersprache Java dar. Es ist sowohl direkt aus der Java-Entwicklungsumgebung (JDK) zu nutzen als auch in das IBM-Produkt *Visual Age for Java* integrierbar. Der Fokus liegt dabei auf den BAPIs. Ereignisse und Attribute der R/3-Geschäftsobjekte werden nicht unterstützt.

Von der SAP sind für den Zugriff von Java-Anwendungen auf das R/3-System die *R/3-Access Classes* entwickelt worden. Dies sind Java-Klassen, die in ihrem Aufbau den Klassen der RFC-C++-Bibliothek ähneln. Die semantischen Objekte der *R/3-Access Classes* sind also ebenfalls Funktionsbausteine. Die *R/3-Access Classes* greifen über den Mechanismus des *Java Native Interface (JNI)* auf die Funktionalität der plattformübergreifend vorhandenen

```
Dim customer As Object, oBapiControl As Object
Dim oReturn As Object, oData As Object
Dim oConnection As Object
Set oBapiControl = CreateObject("SAP.BAPI.1")
  Set oConnection = oBapiControl.Connection
  oConnection.MessageServer = ""
  oConnection.ApplicationServer = ""
  If Not oConnection.Logon(0, False) Then
    Set oConnection = Nothing
    Logon = False
    MsgBox "Logon failed.", , APPID
  End If
Set customer = oBAPICtrl.GetSAPObject_
("KNA1","0000000033","0001","01","01")
customer.CheckExistence Return:=oReturn, Customer_data:=oData
Set customer = Nothing
```

Abbildung 5.3: BAPI ActiveX: Programmierung mit Geschäftsobjekten

RFC-C-Bibliothek zurück, so daß die Benutzung von JNI keine Einschränkung der Portabilität bedeutet. Als Alternative zu JNI ist die Nutzung von CORBA als Middleware möglich, zum jetzigen Zeitpunkt aber noch nicht implementiert. Die Klassen des IBM-*Access Builders* nutzen die *R/3-Access Classes* und stellen darauf aufbauend die R/3-Funktionsbausteine als Methoden von Geschäftsobjekten dar. Die Architektur des *Access Builders* zeigt Abbildung 5.4. Der Access Builder besteht aus den Komponenten:

- *BORBrowser*, einem Werkzeug zur Untersuchung der im R/3 verfügbaren Geschäftsobjekte und ihrer Schnittstellen. Für die Nutzung des *BORBrowsers* muß nicht notwendigerweise ein R/3-System zur Verfügung stehen, da aus dem R/3-System gelesene Informationen in Dateien serialisiert werden und so auch ohne Verbindung zum R/3-System zur Verfügung stehen.

- *Access Builder* zum Generieren von Proxyklassen, gegen die javaseitig programmiert wird

- dem *Logon Java Bean*, einer grafischen Komponente, die den Anmeldevorgang am R/3 kapselt.

Der *Accesss Builder* kapselt die R/3-Geschäftsobjekte in Proxyklassen (*Business Object Proxy Beans*). Ein R/3-Geschäftsobjekt wird dabei durch genau eine Java-Proxyklasse repräsentiert, wobei die BAPIs die Methoden der Proxyklasse darstellen. Lediglich die Methoden (BAPIs) des Objektmodells finden sich in den Proxyklassen wieder, da nur diese bis jetzt relevante betriebswirtschaftliche Funktionalität zur Verfügung stellen. Attribute und Ereignisse sowie Assoziationen sind aus diesem Grund nicht modelliert. Das Ziel des *Access Builder* ist es dabei, die persistenten R/3-Objekte Java-Anwendungen zur Verfügung zu stellen. Die R/3-Geschäftsobjekte werden mittels eines oder mehrerer Schlüsselfelder identifiziert. Die

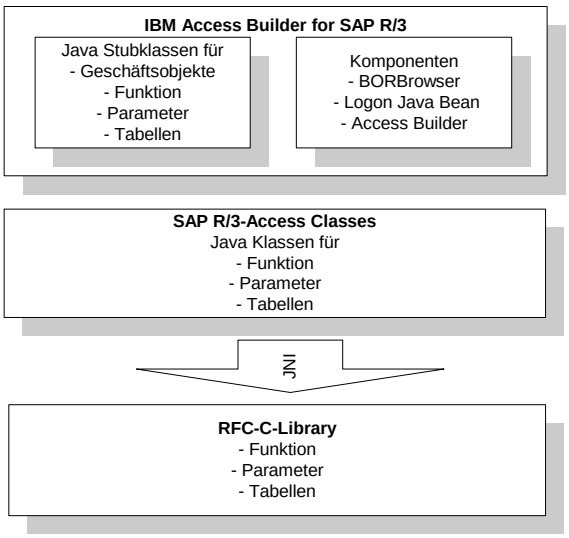

Abbildung 5.4: *Access Builder*: Architektur

Übergabe der Schlüsselfelder als Importparameter an den (exemplarabhängigen) BAPI wird durch den *Access Builder* automatisch vollzogen.

Java-Proxyklassen können aus einem vorhandenen R/3-System generiert werden. Es ist dabei möglich, sowohl für einzelne Funktionen (*Command Proxy bean for a RFC module*) als auch für Geschäftsobjekte und ihre Methoden Proxyklassen zu generieren. Es werden folgende Typen von Proxyklassen generiert:

- Geschäftsobjektproxyklassen
- Proxyklassen für strukturierte Parameter und Tabellenparameter. Proxyklassen besitzen einen Zeitstempel, der das Datum der Generierung, das zugehörige R/3-Release sowie die Version des erzeugenden Generators als Strings angibt. Die Kompatibilität zwischen den generierten Klassen und dem R/3-System ist aber manuell durch den Entwickler sicherzustellen und wird nicht abgeprüft.

Einfache Parameter eines BAPIs bildet der *Access Builder* auf die automorphe Javaklasse **ISimple** ab. Diese stellen damit die **RFC_PARAMETER**-Strukturen der darunterliegenden RFC-C-Bibliothek dar. **ISimple** bietet *Get*- und *Set*-Methoden zur Manipulation der Werte an und übernimmt so die Aufgabe der Typkonvertierung. Der Benutzer des *Access Builders* kommt so nur mit den Java-Typen in Kontakt.

Ein strukturierter Parameter eines Funktionsbausteins ist als Proxyklasse implementiert. Felder strukturierter Parameter sind skalare Werte und werden wiederum durch **ISimple** repräsentiert. Analog zum einfachen Parameter besitzt die Proxyklasse eine sogenannte *embeddedStructure* vom Typ **IStructure**, welches den automorphen **RFC_PARAMETER** der RFC-C-Bibliothek modelliert und **ISimple**-Objekte aggregiert.

Ein Tabellenparameter wird durch eine Proxyklasse für die Tabellenzeile sowie eine aggregierende Tabellenproxyklasse dargestellt, da Java keinen parametrischen Polymorphismus bietet. Die Gesamtzahl der Parameter eines BAPI wird wiederum in einer aggregierenden Proxyklasse abgebildet. Aus dieser Schilderung ist bereits ersichtlich, daß eine Vielzahl von Klassen generiert werden muß.

Schlüsselfelder des R/3-Systems werden durch die **ObjectId**-Exemplarvariable gekapselt, die alle Geschäftsobjektproxyklassen aus der gemeinsamen Basisklasse **SapObject** erben. Die Klasse **ObjectID** stellt ein Aggregat von **ISimple**-Objekten dar, die über den Namen des Schlüsselfeldes angesprochen werden. Vor dem Aufruf eines (exemplarabhängigen) BAPIs muß aus diesen Gründen die **ObjectId** korrekt gesetzt werden.

Die eigentliche Funktionalität zum Funktionsbausteinaufruf ist ebenfalls in **SapObject** verborgen. **SapObject** besitzt die Methoden **call** und **callClassMethod** zum Aufruf von exemplarabhängigen und klassenabhängigen BAPIs[3]. Klassenabhängige BAPIs erscheinen als statische Methoden der Geschäftsobjektklassen. Die Parameter der beiden **Call**-Methoden sind Datenfelder, die genau die Objekte der bei den Parametern erläuterten automorphen Objekte enthalten. In Abbildung 5.5 wird in Auszügen der erforderliche Programmquelltext für den Aufruf des BAPIs **Customer.existenceCheck** aufgeführt. Die Klassen **KNA1**, **KNA1CheckexistenceParams** und **Kna1Structure** stellen die generierten Proxyklassen für Geschäftsobjekt und BAPI-Parameter dar.

[3]Signatur: **call(String, IImpExpParam[], IImpExpParam[], ITable[])**

```
public class SampleCustomer
{
static private IRfcConnection establishConnection(MiddlewareInfo aMiddlewareInfo)
throws  Exception, com.sap.rfc.exception.RfcCommunicationException
{ ... ;  return aConnection ;}

public static void main (java.lang.String[] args)
{ MiddlewareInfo aMiddlewareInfo = new MiddlewareInfo(args) ;
FactoryManager aFactoryManager = FactoryManager.getSingleInstance() ;
aFactoryManager.setMiddlewareInfo(aMiddlewareInfo) ;
IRfcConnection aConnection = null ;
try { aConnection = establishConnection(aMiddlewareInfo) ;
} catch (Exception ex)
{System.out.println("ERROR create connection : " + ex); System.exit(-1);}
try
{ ObjectId objectId = KNA1.getEmptyObjectId() ;
objectId.getKeyField("CUSTOMERNO").setString("0000000033") ;
KNA1 customer = new KNA1(objectId) ;
doExistenceCheck(customer) ;
} catch (Exception ex)
{ System.out.println ("Unexpected exception occurred:");
System.out.println (ex); }
}

private static void doExistenceCheck(KNA1 customer)
{ try{
KNA1CheckexistenceParams aKNA1CheckexistenceParams=
new KNA1CheckexistenceParams();
aKNA1CheckexistenceParams.setSalesOrganization( "0001");
aKNA1CheckexistenceParams.setDistributionChannel("01");
aKNA1CheckexistenceParams.setDivision("01");
aKNA1CheckexistenceParams=
customer.checkexistence(aKNA1CheckexistenceParams);

Kna1Structure struct = aKNA1CheckexistenceParams.getCustomerData() ;
System.out.println ("anrede :"           + struct.getAnred()   );
System.out.println ("name1  :"           + struct.getName1()   );
System.out.println ("ernam  :"           + struct.getErnam()   );
} catch (Exception ex)
{ System.out.println("Exception in doExistenceCheck() : " + ex) ; }
return; }
}
```

Abbildung 5.5: *Access Builder*: Programmierung mit Geschäftsobjekten

Kapitel 6

TYCOON-2

Dieses Kapitel orientiert sich sehr stark an den Arbeiten [Wahlen 98] und [Wegner 98]. Es beschreibt die Eigenschaften und die Wurzeln des TYCOON-2-Systems, welches als Umgebung für die Erstellung der R/3-Schnittstelle diente. Daran anschließend wird TL-2, die objektorientierte Programmiersprache des TYCOON-Systems vorgestellt.

6.1 Das TYCOON-2-System

Das jetzige TYCOON-2-System ist unter Einbeziehung der Erfahrungen aus dem TYCOON-System entstanden. Im folgenden werden die Historie des TYCOON-2-Systems sowie markante Eigenschaften des Systems beschrieben.

6.1.1 Historie

Das TYCOON-System (Typed Communicating Objects in Open Environments) und seine Programmiersprache TL (Tycoon Language) wurden seit etwa 1993 am Arbeitsbereich Datenbanken und Informationssysteme (DBIS) des Fachbereichs Informatik der Universität Hamburg im Rahmen des europäischen Projekts FIDE entwickelt. Wesentliche Merkmale sind der funktional-imperative Stil von TL, parametrischer Polymorphismus und Subtyppolymorphismus, Funktionen höherer Ordnung als Sprachobjekte erster Klasse sowie die orthogonale Persistenz und Mobilität von Daten, Code und Prozessen ([Matthes 93], [Mathiske et al. 93], [Matthes et al. 94], [Matthes et al. 97], [Mathiske et al. 97],[Mathiske 96]).

Zwischen 1995 und 1996 wurde dann ein objektorientierter Dialekt namens TOOL (Tycoon object oriented language) geschaffen ([Gawecki, Matthes 95], [Gawecki, Matthes 96]), der zusammen mit dem alten TYCOON-System die Grundlage für das ab 1996 entwickelte TYCOON-2-System bildete. Letzteres entstand im wesentlichen im Rahmen der von Absolventen und Mitarbeitern von DBIS neugegründeten Firma HIGHER-ORDER ([Gawecki et al. 97], [HOX98 98]). Im Verlaufe der Entwicklung und des gleichzeitigen erfolgreichen kommerziellen Einsatzes wurde das TYCOON-2-System grundlegend revidiert, wobei eine eigene, vom ursprünglichen TYCOON-System unabhängige, neue virtuelle Maschine, ein Typprüfer und

ein Übersetzer entstanden ([Weikard 98], [Ernst 98], [Wienberg 97]). Eine breite Darstellung der Sprache TL-2 und der ihr zugrundeliegenden Entwurfsentscheidungen bietet [Wahlen 98]. Zur Zeit wird das TYCOON-2-System von der Firma HIGHER-ORDER zur Realisierung von anspruchsvollen Projekten insbesondere im Medienbereich und vom Arbeitsbereich Softwaresysteme (STS) der TU-Hamburg-Harburg zu Forschungszwecken im Bereich der persistenten Objektsysteme verwendet.

6.1.2 Eigenschaften

Das TYCOON-2-System besteht aus einer virtuellen Maschine und einem Objektspeicher. Der TYCOON-2-Objektspeicher (*store*) ist ein maschinenunabhängiges, persistentes Objektsystem mit virtuellem Bytecode für die einzelnen Methoden und Funktionen. Die virtuelle Maschine arbeitet auf diesen Objektstrukturen und führt den Bytecode aus (bzw. interpretiert den Bytecode). Neben dem Bytecodeinterpreter enthält die virtuelle Maschine noch andere Komponenten wie z.B. die Speicherbereinigung (*garbage collector*) (vgl. [Weikard 98]; [Schroe98]).

Das TYCOON-System zeichnet sich durch folgende Eigenschaften aus:

(Orthogonale) Persistenz Daten, Code und Threads (Code in Ausführung) können unabhängig von deren Definition über mehrere Programmabläufe persistent gespeichert und wiederhergestellt werden.

Portabilität Für eine Portierung des Systems auf eine andere Plattform ist „nur“ die relativ zum Objektspeicher kleine virtuelle Maschine zu portieren. Der Objektspeicher mit Daten, Code und Threads ist plattformunabhängig.

Uniforme Datenrepräsentation Alle Objekte werden als uniforme Datenstrukturen im Objektspeicher repräsentiert. Uniforme Datenrepräsentation stellt die systemseitige Grundlage für universellen Polymorphismus dar. Sie ist aber auch vorteilhaft für Systemkomponenten wie den Bytecodeinterpreter und die Speicherbereinigung [Weikard 98].

Interaktivität Klassen werden inkrementell übersetzt und dynamisch intregriert, so daß sich folgende Ausführung von Code bereits darauf bezieht.

Multithreading TYCOON erlaubt die effiziente nebenläufige Ausführung und Synchronisation mehrerer Berechnungen. Die einzelnen Ausführungspfade werden dabei direkt auf Betriebssystem-Threads abgebildet.

Reflektion Bis auf die virtuelle Maschine sind alle übrigen Teile des TYCOON-Systems, z.B. der *Compiler*, größenteils in TYCOON selbst implementiert und als langlebige Objekte im TYCOON-Objektspeicher abgelegt und damit von anderen Programmen als typisierte Objekte nutzbar. Die hierbei von der virtuellen Maschine benötigte Funktionalität beschränkt sich auf wenige eingebaute (*builtin*) Methoden. Eingebaute Methoden werden direkt von der virtuellen Maschine ausgeführt [Weikard 98]. Die Selbstimplementierung mit dem direkten reflektiven Zugriff im Objektspeicher erfordert bei der Weiterentwicklung des Systems einen Bootstrap [Wienberg 97].

(Orthogonale) Mobilität - vorgesehen Orthogonale Mobilität bedeutet, daß sowohl Daten, Code als auch Threads migrieren können (vgl. migrierende Threads in Tycoon-1 [Mathiske et al. 97], [Mathiske et al. 95]).

Das TYCOON-System weist eine große Ähnlichkeit zu Smalltalk-Systemen auf. Ein wesentlicher Unterschied ist jedoch die statische Typisierung der Sprache TL-2.

6.2 Die Sprache TL-2

TL-2 ist ein rein objektorientierte Sprache und orientiert sich syntaktisch und lexikalisch relativ stark an C und JAVA. TL-2 bietet/ist:

- (rein) objektorientiert
- klassenbasiert
- Mehrfachvererbung
- Metaklassen
- Ausnahmebehandlung
- parametrischen Polymorphismus
- (strukturelle) Subtypisierung
- explizit typisiert und typsicher
- modulare Übersetzung und Typüberprüfung
- Funktionen höherer Ordnung
- Offenheit

6.2.1 Klassen, Objekte und Nachrichten

TL-2 ist eine statisch typisierte Programmiersprache. Das bedeutet, daß jedem Wertausdruck statisch, d.h. zur Zeit der Übersetzung, durch Analyse des Programmtextes ein Typ zugeordnet werden kann. Dies garantiert, daß der Ausdruck zur Laufzeit auf jeden Fall zu einemWert dieses Typs evaluiert. Typfehler, in objektorientierten Sprachen also insbesondere der Fehler, daß ein Objekt eine Nachricht nicht versteht, sind somit ausgeschlossen.

Die Programmiersprache TL-2 gehorcht dem objektorientierten Paradigma, d.h. Objekte und Nachrichten zwischen ihnen bilden die alleinige Grundlage der Programmierung. Ausnahmslos jedes Objekt in TL-2 ist ein Exemplar einer Klasse. Es gibt keine primitiven Datentypen; auch Zeichenketten und Zahlen werden durch Objekte und Klassen dargestellt. Sie sind dadurch nahtlos in das Typsystem integriert. Man spricht in diesem Zusammenhangdann auch von „reiner“ Objektorientierung.

Die Objektorientierung geht einher mit einer strikten Referenzsemantik. Grundlage dafür ist das Vorhandensein der von Werten unabhängigen und unabänderbaren Objektidentität eines jeden Objekts. Alle Objekte sind auf einer Halde (hier einem persistenten Objektspeicher) angelegt, so daß alle Variablen tatsächlich nur Verweise auf die Objekte darstellen; die Dereferenzierung geschieht stets implizit. Dadurch entfallen fehleranfällige Zeigerkonstrukte und Probleme mit der Speicherverwaltung. Die automatische Freispeicherverwaltung beruht auf transitiver Erreichbarkeit von einem Wurzelobjekt aus [Weikard 98]. Die Referenzsemantik bedingt die Begriffe der Objektidentität, der flachen sowie tiefen Gleichheit von Objekten. Einher damit gehen analoge Begriffe der Kopiersemantik. Einer Einschränkung unterliegt die Objektidentität gewisser elementarer Klassen wie die der Zahlen, Wahrheitswerte etc. Deren Exemplare sind bereits bei Wertgleichheit objektidentisch und somit ununterscheidbar. Dies ist eine Folge der Implementierung, die allerdings auch sinnvoll ist: 14 ist immer gleich 14 ist immer gleich 14.

In einer Klasse sind Methoden und Variablen (*slots*) definiert. Sie bestimmen das Verhalten und den Zustand eines Objektes dieser Klasse. Klassen sind also Abstraktionen gleichartiger Objekte.

Das Senden einer Nachricht an ein Objekt im rein objektorientierten Sinne bewirkt die Ausführung des Codes einer Methode der Klasse des Objekts. Dabei können dieser Nachricht neben dem Methodenselektor (dem Namen der Methode) beliebig viele Parameter (weitere Objekte) mitgegeben werden, und es kann weiterhin ein Ergebnis zurückgegeben werden. Dies entspricht ziemlich genau dem Aufruf imperativer oder funktionaler Prozeduren oder Funktionen, nur daß jetzt ein Empfängerobjekt als impliziter Parameter mit im Spiel ist. Das Objekt, das die Nachricht empfängt, wird innerhalb der Methodendefinitionen seiner Klasse mit dem Schlüsselwort **self** referenziert. Sowohl Methoden als auch Variablen können als privat deklariert werden; dann haben nur die Methoden der Klasse Zugriff auf diese, und auch nur auf das aktuelle Objekt **self**.

6.2.2 Vererbung und Metaklassen

Klassen können voneinander erben. Die erbende Klasse, die Subklasse, übernimmt dabei alle Methoden- und Variablendefinitionen der Superklassen. Dabei können Methodendefinitionen durch Spezialisierungen überschrieben werden. Das Senden einer Nachricht an ein Objekt bewirkt die Suche nach einer Methodendefinition in der Klasse des aktuellen Empfängerobjektes. Wird keine gefunden, so wird in der oder den Superklassen weitergesucht. Es ist hierbei anzumerken, daß TL-2 kein Überladen von Methodennamen (*overloading*) kennt, so daß alle Methoden (auch die Konstruktormethoden) verschiedene Namen erhalten müssen. Wird auch dort keine gefunden, so wird ein Laufzeitfehler ausgelöst. Diese Art der Bindung von Nachrichten an Methodendefinitionen wird als späte oder dynamische Bindung bezeichnet. Weiterhin ist es möglich, mit Hilfe des Schlüsselwortes **super** als Empfängerobjekt alte, überschriebene Methodendefinitionen zu verwenden. Zusätzlich unterstützt TL-2 Mehrfachvererbung. Konflikte können dabei nicht auftreten, da die Superklassen einer Klasse durch ein der Breitensuche ähnliches Verfahren in eine lineare Ordnung gebracht werden (*class precedence list, CPL*). Dadurch ist stets genau definiert, auf welche Superklasse sich **super** bezieht, und außerdem ist gewährleistet, daß kaskadierende **super**-Aufrufe keine Klasse auslassen.

Vererbung in TL-2 bedeutet immer auch Vererbung von Implementierungen, nicht nur von Schnittstellen (anders als z.B. in JAVA, das nur einfache Vererbung (*subclassing*) und zusätzlich Schnittstellenvererbung durch eine implements-Klausel anbietet). Dennoch sind in TL-2 - neben beliebigen Mischformen - sowohl abstrakte Klassen, die niemals instantiiert werden dürfen, als auch reine Schnittstellendefinitionen als Klassen möglich. Erstere können bereits teilweise Methodendefinitionen besitzen, letztere ergeben sich einfach durch das vollständiges Weglassen der Methodenrümpfe und die Verwendung des Schlüsselwortes **deferred** an ihrer Stelle. Die Mehrfachvererbung unterstützt so einen Programmierstil, der von sehr fein granularen Abstraktionen und dem sogenannten *mixin-style* Gebrauch macht.

Die Klasse **Object** ist die allen anderen Klassen gemeinsame Superklasse. Sie definiert das für alle Objekte des Systems gemeinsame Protokoll. Tatsächlich sind alle auf den ersten Blick wie Konstanten etc. aussehende Konstrukte, z.B. die Wahrheitswerte true und false, lediglich Methoden oder Variablen von Object, die automatisch durch die dynamische Bindung verwendet werden, wenn sie in Ausdrücken auftauchen.

Ähnlich wie in SMALLTALK sind auch in TL-2 Klassen Objekte erster Klasse. Das bedeutet, daß sie Objekte sind, die einer Klasse angehören, die Methoden und evtl. sogar Variablen hat. Diese Klasse wird als Metaklasse einer Klasse bezeichnet. Die Methoden der Metaklasse sind für die Erzeugung von Exemplaren der Klasse zuständig. Dies wird normalerweise von einer Methode namens **new** geleistet. Diese Konstruktormethoden sind also anders als in vielen objektorientierten Sprachen (wie z.B. C++) keine besonderen Sprachkonstrukte. Die Möglichkeit, Metaklassen selbst zu definieren, läßt eine genaue Kontrolle und ggf. Verwaltung der Objekterzeugung zu. Ganz leicht läßt sich so zum Beispiel eine Klasse programmieren, die nur genau ein Exemplar besitzt (*singleton*). Es existiert die generische Metaklasse **SimpleConcreteClass**, die Objekte erzeugen und intialisieren kann.

Zu erläutern bleibt noch das Konzept des Klassenobjekts. Jede Klasse ist ein Objekt seiner Metaklasse. Das Objekt wird vom *Compiler* zum Zeitpunkt der Übersetzung der Klasse angelegt. Damit es aber auch verwendbar ist, existiert im TYCOON-2-System der sogenannte Pool, der alle Klassenobjekte enthält. Er ist stets letztes Glied in jeder Klassenpräzedenzliste (CPL). In diesem Zusammenhang ist noch die Methode **class** (oder synonym, aber einfacher zu schreiben, **clazz**) von **Object** zu nennen, die das Klassenobjekt eines Objektes liefert. Dadurch sind reflektive Mechanismen wie einfache Typtests zur Laufzeit leicht zu realisieren.

Es bleibt zu bemerken, daß gewisse elementare Klassen wie die der Zahlen oder Wahrheitswerte eine spezielle Metaklasse haben (**OddballClass**), die keinen expliziten Konstruktor besitzt. Die Konstruktion der Werte wird implizit durch Notation der Literale im Programmtext, bzw. durch eingebaute elementare Methoden geleistet.

6.2.3 Funktionen höherer Ordnung

Dem TYCOON-1-System und seinem funktional orientierten Paradigma entlehnt ist das Konzept der Funktionen höherer Ordnung als Objekte erster Klasse. Auch in TL-2 ist es möglich, ein evtl. sogar parametrisiertes Stück Programmcode als Wert erster Ordnung an Variablen zu binden, weiterzureichen, als Parameter zu übergeben oder als Wert zurückzugeben, um es erst später auszuführen. Die Bindungen an statisch im Programmtext sichtbare Variablen werden dabei in einem Funktionsabschluß (*closure*) für den Benutzer transparent zusammen mit der

Referenz gespeichert. Damit lassen sich elegante Muster wie z.B. Iterationsabstraktionen und andere Kontrollstrukturen realisieren.

Funktionen werden als Objekte verschiedener Klassen mit der Superklasse **Fun** repräsentiert. Insbesondere haben sie eine Methode [], die die Funktion auswertet.

6.2.4 Die Klassen Nil und Void

Die Klassen **Nil** und **Void** bilden die Eckpunkte des TL-2-Typsystems. Der Typ **Nil** hat genau ein Exemplar, den Wert **nil**. Diesen Wert können alle Variablen annehmen, um z.B. dadurch zu signalisieren, daß der Wert nicht definiert, nicht vorhanden oder nicht bekannt ist. Außerdem ist **nil** der Wert sämtlicher Variablen (*slots*) eines neuerzeugten Objekts. Die Methoden zur Initialisierung können diesen Wert natürlich danach verändern. Der Wert **nil** kann stets typsicher zugewiesen werden. Dies liegt daran, daß er gewissermaßen als speziellster möglicher Typ Subtyp aller Klassen ist. Die Klasse **Nil** verfügt dabei allerdings über keine Methoden, so daß jede Nachricht an **nil** mit einem Laufzeitfehler beantwortet wird.

Im Gegensatz dazu steht die Klasse **Void**. Sie ist die Superklasse von **Object**; es existiert aber im Gegensatz zu **Nil** kein einziger Wert dieser Klasse. Mit dem Typ **Void** läßt sich das Fehlen eines Wertes auf Typebene ausdrücken, zum Beispiel, daß eine Methode keinen Rückgabewert besitzt oder daß ein Typparameter nicht benötigt wird. Dieser allgemeinste aller Typen läßt sich so keinem Objekt zuweisen.

6.2.5 Polymorhpismus

Das Typsystem von TL-2 beruht im Gegensatz zu vielen anderen Sprachen (wie etwa C, C++ und MODULA) auf struktureller Äquivalenz und nicht auf Namensäquivalenz. Als Typ einer Klasse ist die Menge der Signaturen der Methoden und Variablen der Klasse definiert. Die Subtypisierungsrelation, die auf strukturellen Vergleichen beruht, ist zu unterscheiden von der Vererbungsrelation. In vielen objektorientierten Sprachen werden beide identifiziert (etwa in C++) und beruhen tatsächlich nur auf der Vererbungsrelation. Eine Klasse ist dort Subtyp einer anderen, wenn sie direkt oder indirekt von der zweiten erbt. In der Regel ist aber auch in TL-2 eine Subklasse gleichzeitig Subtyp seiner Superklasse. Einzelheiten dazu sind etwa in [Ernst 98] zu finden.

Die Subtypisierung ist die Grundlage für eines der wichtigsten Konzepte der polymorphen Sprachen: die Subsumption, die es gestattet, immer dann, wenn ein Objekt des Typs A erwartet wird, eines des Typs B einzusetzen, sofern B Subtyp von A ist. Dies wird auch als Subtyppolymorphismus bezeichnet.

Daneben unterstützt TL-2 noch den begrenzten parametrischen Polymorphismus (*bounded parametric polymorphism*). Hierdurch lassen sich sowohl Klassen als auch Methoden und Funktionen mit Typparametern versehen, wodurch allgemein formulierte Abstraktionen mit hoher Generizität möglich werden (dies entspricht soweit den *templates* in C++). Begrenzung des Typparameters bedeutet dabei die Angabe eines Typs als obere Schranke, der dann der Supertyp des Parameters ist.

6.2.6 Reflexivität

TL-2 stellt Mechanismen zum reflexiven Zugriff auf die Programmobjekte zur Laufzeit bereit. Dieses kann zur Implementierung generischer Algorithmen als auch zur Klassengenerierung benutzt werden. Diese reflexiven Mechanismen werden durch Methoden der Klasse **Object** implementiert. Dabei sind folgende Methoden zu nennen:

- **clazz :Class** für den Zugriff auf das Klassenobjekt zu einem gegeben Objekt
- **_doesNotUnderstand(selector :Symbol, args :Array(Object)) :Nil** zum Abfangen von erfolglosen Methodensuchen an einem Objekt
- **perform(selector :Symbol, args :Array(Object)) :Nil** zum dynamischen Ausführen von Methoden auf einem Objekt

6.2.7 Offenheit

TYCOON ist ein offenes System: über Bibliotheken werden z.B. Dienste wie Socketkommunikation und Anbindung an relationale Datenbanken in die Sprache TL-2 integriert. Diese Dienste werden durch Klassen in TL-2 gekapselt, so daß der Umgang mit diesen Klassen nicht von der in der virtuellen Tycoon-Masschine eingebauten Funktionalität zu unterscheiden ist.

Für diese Arbeit war die Anbindung an die RFC-C-Bibliothek der SAP relevant. Dazu stellt die Klasse **DLL** eine Schnittstelle zur Verfügung, die Methoden einer von **DLL** abgeleiteten Klasse auf die Funktionen der externen Bibliothek abbildet. Dabei werden ebenfalls die TL-2-Basistypen auf C-Basistypen abgebildet und umgekehrt. Der gemeninsame Nutzung von Speicherbereichen (*shared memory*) ist jedoch nicht möglich. Des weiteren ist es nicht möglich, TL-2-Funktionalität durch extern realisierte Funktionen zu rufen. Auch ist es für externe Funktionen nicht möglich, auf Inhalte des Tycoon-Objektspeichers zuzugreifen.

Durch die Implementierung der Klasse **DLL** als Ressource steht diese extern realisierte Funktionalität transparent für den TL-2-Programmierer zur Verfügung. Wird eine Ressource geöffnet, das Tycoon-System gespeichert und verlassen, so steht nach dem Wiederanfahren des Systems diese Ressource wieder geöffnet zur Verfügung.

6.2.8 Möglichkeiten der Klassengenerierung

Eine Klassengenerierung zur Laufzeit ermöglicht es, das Tycoon-System dynamisch durch die Anwendungslogik zu verändern. Für die Generierung gibt es prinzipiell zwei Möglichkeiten:

1. Bytecodeerstellung im Tycoon *Store*
   Durch eine Bytecodeerstellung wird direkt interpretierbarer Code erstellt. Die Tycoon-Werkzeuge TL-2-*Parser* und *Compiler* werden so umgangen. Dies stellt natürlich hohe Anforderungen an die syntaktische und semantische Richtigkeit des generierten Codes, da eine Fehlersuche so nur sehr schwer möglich ist. Ein weiteres Problem ist, daß Form und Umfang des durch die *Tycoon Virtual Machine (TVM)* interpretierten Bytecodes Änderungen unterworfen sind und so keine stabile Lösung ohne ständige Anpassung der

Generierungsfunktionalität möglich ist. Positiv ist der Performanzvorteil zu vermerken, da das zeitaufwendige *Parsen* und *Compilieren* von TL-2-Code umgangen wird.

2. TL-2-Codegenerierung
   Der generierte TL-2-Code wird in Dateien gespeichert. Nach erfolgter Generierung wird der Code durch *Parser* und *Compiler* eingelsen und kompiliert im Tycoon-Store als Bytecode abgelegt. Fehler im Code werden dabei erkannt und ermöglichen so eine gute Fehlersuche. Das Einlesen und *Compilieren* kann inkrementell und zur Laufzeit erfolgen.

In dieser Arbeit wird die Methode der TL-2-Codegenerierung verfolgt, da diese die generelle Methode darstellt und auch auf Umgebungen übertragbar ist, die keinen Zugriff auf die interne Klassenrepräsentation zulassen. Es wird nachvollziehbarer Code generiert, der auch manuell angepasst werden kann.

Kapitel 7

Die TYCOON-2-SAP-R/3 Schnittstelle

Nachdem die Architektur des R/3-Systems dargestellt, verschiedene Geschäftsobjektkonzepte und das Objektmodell des R/3-Systems vorgestellt, kommerzielle Lösungen der Integration aufgezeigt, sowie die Sprache TL-2 und das TYCOON-2-System behandelt wurden, wird der Entwurf und die Implementierung der TYCOON-2-R/3 Schnittstelle in diesem Kapitel beschrieben. Aus den bisherigen Kapiteln und aus [Lutz 97] geht hervor, daß über die RFC-Automation und den Zugriff auf die R/3-Funktionsbausteine ein Zugriff auf alle semantischen Objekte des R/3 möglich ist.

Dabei wird folgendermaßen vorgegangen: Nach der Festlegung der Kriterien und der Bestimmung des Umfanges der zu implementierenden Schnittstelle in Abschnitt 7.1 folgt eine Übersicht über die Gesamtarchitektur der Schnittstelle in Abschnitt 7.2. Abschnitt 7.3 stellt den eigentlichen Entwurf und die Implementierung der verschiedenen Teilsysteme dar. Dabei werden jeweils die Anforderungen, die verwendeten Techniken sowie die Ergebnisse detailliert angegeben. Anhand von Quelltextbeispielen wird die Benutzung der Schnittstelle beschrieben und dadurch Besonderheiten und Eigenschaften hervorgehoben bzw. verdeutlicht. Zum Abschluß dieses Kapitels werden die vorgestellten Eigenschaften der Schnittstelle zusammengefaßt und die vorgenommene Implementierung bewertet.

7.1 Zielsetzung

Primäres Ziel der Arbeit ist es, die im R/3 vorhanden Geschäftsobjekttypen dem TYCOON-2-System zur Verfügung zu stellen. Parallel zu dieser Arbeit wird in [Carlsen 99] ein Internet-Shop mit dem R/3-System als *Backend* entwickelt. Anwendungen dieser Art sollen durch die Schnittstelle in TYCOON-2 ermöglicht werden. Die Überprüfung des Geschäftsobjektkonzeptes der SAP in Kapitel 4 hat ergeben, daß dies auf Basis des entfernten Funktionsbausteinaufrufs erfolgen kann.

In [Lutz 97] wurde eine polymorph typisierte Schnittstelle zum R/3 in TYCOON-1 vorgestellt, die bereits für grundlegende Probleme, wie z.B. das *Marshalling* der Parameter und die

Ausnahmebehandlung adäquate Lösungen anbietet. Die Konzepte dieser Schnittstelle werden nach sorgfältiger Analyse und unter Beachtung der in Kapitel 6 vorgestellten objektorientierten Eigenschaften in die TYCOON-2 Schnittstelle übernommen.

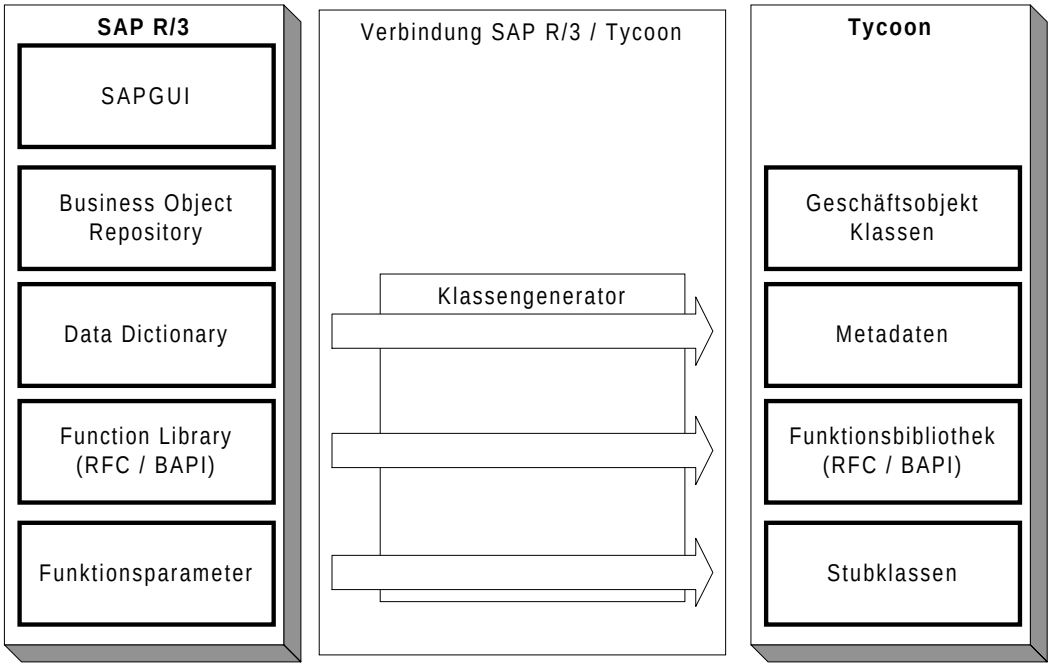

Abbildung 7.1: Umfang der Schnittstelle

Die richtige Behandlung der Parameter von Funktionsbausteinen ist entscheidend für einen erfolgreichen Funktionsaufruf im SAP R/3-System, daher ist eine wichtige Anforderung an die zu erstellenden Schnittstelle die statische Typisierung. Durch statische Typisierung lassen sich falsche, typverletzende Wertzuweisungen zur Übersetzungszeit erkennen, so daß Fehler zur Laufzeit vermieden werden [Watt 90]. Da den Parametern eine so große Bedeutung zukommt, wird außerdem bei Wertzuweisungen die Einhaltung von Länge und Wertebereich überprüft.

Die Benutzung der TYCOON-1-Schnittstelle aus [Lutz 97] erfordert vom Benutzer ein hohes Maß an Wissen über die RFC-Automation (siehe Abschnitt 2.6.1). Bevor er in der Lage ist, einen Aufruf eines Funktionsbausteins durchführen zu können, muß er die für den Aufruf nötigen Informationen aus dem R/3-System ermitteln und in TYCOON-1 in dafür bereitgestellte Strukturen übergeben. Der Aufwand dafür variiert je nach Komplexität eines Funktionsbausteins, kann sich aber schon bei nur 13 Funktionsbausteinen bzw. BAPIs auf über 168 Einzelinformationen ausdehnen (siehe Abbildung 1.1). Dieser Vorgang muß sehr sorgfältig erfolgen, da bereits ein falscher Wert für den Mißerfolg eines Funktionsbausteinaufrufs verantwortlich sein kann. Diese Fehlerquelle kann durch das automatische Herauslesen aller relevanten Informationen aus dem R/3-System beseitigt werden. Außerdem stehen diese herausgelesenen Informationen als Metainformationen zur Laufzeit für Validierungszwecke zur Verfügung.

7.2 Architektur

Eine Übersicht über die Klassenbeziehungen der Schnittstelle gibt das Klassendiagramm im Anhang A dieser Arbeit, welches explizit nicht weiter erläutert wird. Im nächsten Abschnitt 7.2.1 werden die einzelnen Schichten der Schnittstelle beschrieben, indem die zentralen Klassen den Schichten der Architektur zugeordnet werden. In Abschnitt 7.2.2 dieses Kapitels werden die semantischen Objekte des SAP R/3-Systems behandelt, für die es eine Entsprechung in der Schnittstelle gibt.

7.2.1 Softwareebenen

Aufgrund der Komplexität der Schnittstelle zum R/3 und zur besseren Strukturierung wird für die Implementierung eine Schichtenarchitektur gewählt. In Abbildung 7.2 ist die aus fünf Schichten bestehende Architektur dargestellt.

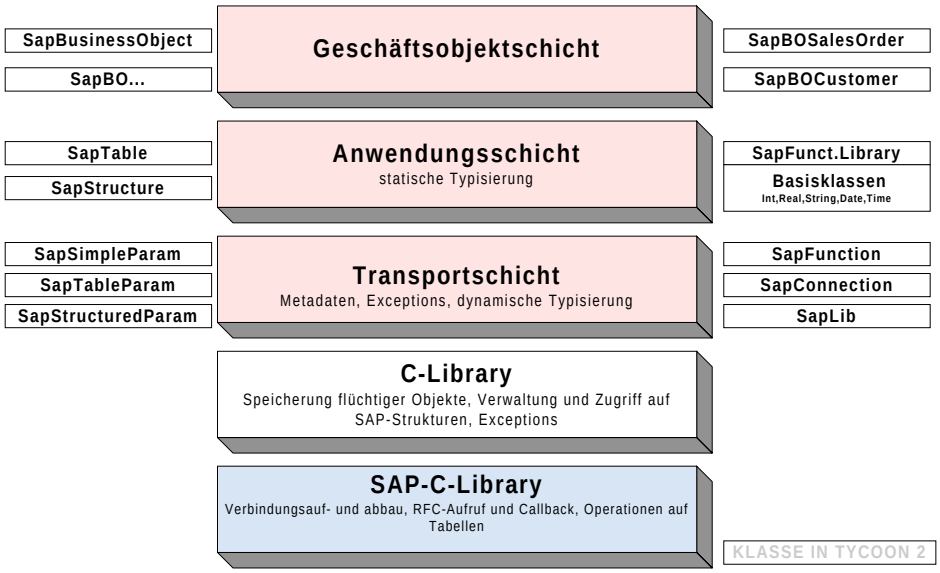

Abbildung 7.2: Schichtenmodell der TYCOON-2-SAP-Schnittstelle

Jede Schicht hat die Aufgabe, die von ihr angebotenen Dienste unter Nutzung der darunterliegenden Schichten möglichst transparent für die darüberliegenden Schichten zu erbringen.

Für die Beschreibung der einzelnen Softwarebenen wird zur näheren Erläuterung folgendes Beispiel benutzt:

Die Methode (BAPI) **Customer.CheckExistence** prüft im R/3-System anhand der übergebenen Schlüsselfelder die Existenz eines Kunden. Ist der Kunde vorhanden, liefert der BAPI Stammdaten des Kunden zurück. Dieser BAPI ist im R/3-System durch den Funktionsbaustein **bapi_customer_checkexistence** implementiert. Der Funktionsbaustein hat vier Importparameter und drei Exportparameter. Die Funktionssignatur des Funktionsbausteins befindet sich in Anhang C. Dort befindet sich auch ein Beispiel für einen Aufruf innerhalb des

R/3-Systems in ABAP.

Im folgenden sind die einzelnen Schichten mit ihren relevanten Klassen beschrieben:

SAP-C-LIBRARY Die unterste Schicht ist die plattformübergreifend verfügbare SAP-C-Library zum Aufruf von Funktionsbausteinen im R/3, die ausführlich in Kapitel 2.6.1 beschrieben ist. Die Funktionen dieser Bibliothek bilden eine Programmierschnittstelle, um synchrone, asynchrone und transaktionale Aufrufe von Funktionsbausteinen eines R/3-Applikationsservers durchzuführen. Desweiteren bietet sie die Möglichkeit, selbst Service-Funktionen der externen Anwendung zu installieren, so daß eine bidirektionale Kommunikation zwischen dem SAP R/3 und der externen Anwendung möglich ist. Beispiele für die Dienste dieser Schicht sind die Funktionen für das Öffnen (**RfcOpen()**) und Schließen (**RfcClose()**) einer Verbindung und für den synchronen (**RfcCallRecv()**) bzw. asynchronen Aufruf (**RfcCall()**, **RfcListen()**, **RfcReceive()**) eines Funktionsbausteins.

Folgendes Beispiel zeigt einen Aufruf des Funktionsbausteins **bapi_customer_checkexistence** innerhalb dieser Schicht von der Sprache C aus (Auszug):

```
RfcCallReceive(
handle,
"bapi_customer_checkexistence",
exporting,
importing,
tables,
&exception_ptr
);
```

Der **handle** stellt eine geöffnete Verbindung zum R/3-System dar. Die Variablen **exporting**, **importing** und **tables** stehen für Datenfelder, die mit den für den Funktionsbaustein nötigen Parametern gefüllt werden müssen z.B.:

```
importing[0].name = "SALES_ORGANIZATION";(*Parametername*)
importing[0].nlen = 18;                  (*Länge des Prameternamens*)
importing[0].addr = &sales_org;          (*Adresse des Parameters*)
importing[0].leng = sizeof(sales_org);   (*Länge des Parameters*)
importing[0].type = RFCTYPE_INT;         (* RFC-Type des Parameters*)
...
```

C-LIBRARY In der Programmierumgebung TYCOON-2 können mit Hilfe der Klasse **DLL** dynamisch externe Bibliotheken eingebunden werden. Die Funktionen der externen Bibliothek sind als Methoden der von der Klasse **DLL**[1] abgeleiteten Klasse **SapLib** abgebildet, die in TYCOON-2 die Dienste dieser Schicht anbietet.

Die Klasse **DLL** unterstützt ausschließlich atomare Werte für Parameter. Die Programmierschnittstelle der SAP-C-LIBRARY erfordern jedoch die Übergabe von Referenzen

[1]Die Klasse **DLL** stellt eine **Resource** dar. Die Klasse **Resource** verwaltet Objekte, die nicht unter der Kontrolle des persistenten TYCOON-2-*Stores* stehen wie z.B. Dateihandles. Abgeleitete Klassen der Klasse **Resource** können von der Flüchtigkeit dieser Objekte abstrahieren.

auf komplexe Strukturen an ihre Funktionen. Weil es nicht möglich ist, Referenzen zwischen TYCOON-2 und der Bibliothek auszutauschen und mit den referenzierten Speicherbereichen in TYCOON-2 zu arbeiten, ist diese Schicht zwischen TYCOON-2 und der SAP-C-LIBRARY eingefügt. Sie ist für die Verwaltung der entsprechenden Speicherbereiche verantwortlich und bietet Dienstfunktionen zur Erzeugung und Kontrolle der komplexen Parameterstrukturen für die Programmierschnittstelle der Schicht eins an.

Die C-LIBRARY wird aus [Lutz 97] übernommen und den Bedürfnissen dieser Schnittstelle und TYCOON-2 angepaßt. Das Funktionsmodell ist um Funktionen für die Behandlung von strukturierten Parametern erweitert, da diese nicht für TYCOON-1 implementiert sind. Ferner muß für TYCOON-2 eine andere Lösung für die Ausnahmebehandlung gefunden werden, da im Gegensatz zu TYCOON-1 Ausnahmen extern nicht ausgelöst werden können (siehe Abschnitt 7.3.7).

TRANSPORTSCHICHT Die TRANSPORTSCHICHT stellt das Pendant der C-LIBRARY in TYCOON-2 dar. Alle durch die C-LIBRARY alloziierten Strukturen sind flüchtigen Charakters. Für die geforderte Persistenz der Schnittstelle muß die TRANSPORTSCHICHT dafür sorgen, immer ein getreues Abbild der durch die C-LIBRARY erzeugten Strukturen zu halten, um diese gegebenenfalls mit Hilfe der Dienste der C-LIBRARY erneut anzulegen. Die Klassen **SapSimpleParam**, **SapStructuredParam** und **SapTableParam** enthalten die für einen Funktionsbausteinaufruf nötigen Informationen, die für den Aufbau der in der C-LIBRARY gehaltenen C-RFC-Speicherstrukturen benutzt werden. Die genannten Klassen haben ebenso wie die C-RFC-Speicherstrukturen automorphen Charakter.

SAP R/3 wird nur unidirektional an TYCOON-2 angebunden. Dienste der SAP-C-Library, die für den bidirektionalen Zugriff zuständig sind (z.B. für die Installation von Service-Funktionen einer externen Anwendung), werden in den höheren Schichten nicht benutzt.

Beispiel: Um den oben genannten Funktionsbaustein **bapi_customer_checkexistence** auf dieser Ebene aufrufen zu können, müssen folgende Schritte durchgeführt werden:

1. Öffnen einer Verbindung zum R/3-System (siehe auch Abschnitt 7.2.2.1):

```
conn:=SapConnection.newWith(
"010" , "user" , "password" ,'D',1,"sun03_NR1_01",
"sun03.sts.tu-harburg.de", 0,"sun03","sapgw00"),
conn.open
```

2. Definition der Formalparameter (hier am Beispiel der Importparameter):

```
(* Parametername, -länge und -typ *)
I1 := SapSimpleParam.new("SALES_ORGANIZATION", 4, 'i', 's'),
I2 := SapSimpleParam.new("CUSTOMERNO", 10,'i', 's'),
I3 := SapSimpleParam.new("DISTRIBUTION_CHANNEL ", 2, 'i', 's'),
I4 := SapSimpleParam.new("DIVISION " , 2 ,'i', 's'),
```

3. Erzeugen eines Funktionsobjektes entsprechend dem Funktionsbaustein **bapi_customer_checkexistence** mit einer zugehörigen Verbindung und Zuweisung der in Schritt zwei erzeugten Formalparameter:

```
sf:=SapFunction.new("bapi_customer_checkexistence", conn),
sf.importList.add(I1); ... ;sf.importList.add(I4)
```

4. Zuweisen der Aktualparameter: Die Einhaltung der Formalparametereigenschaften werden anhand der Informationen überwacht, die bei der Erzeugung eines **SapSimpleParam**-Objektes angegeben werden:

   ```
   I1.value:= "0001",
   I2.value:= "0000000010",
   ...
   ```

5. Funktionsaufruf

   ```
   sf.callReceive,
   ```

6. Schließen der Verbindung zum R/3-System

   ```
   conn.close,
   ...
   ```

Die Klasse **SapConnection** ist die einzige Klasse, die Zugriff auf das R/3 über die Klasse **SapLib** hat, da sie alle Informationen einer Verbindung verwaltet. Bei einem guten Klassendesign ist dafür zu sorgen, daß einzelne Klassen nicht mit Zuständigkeiten und Funktionen überladen werden. Die Klasse **SapFunction** entlastet einerseits die **SapConnection** in ihren Zuständigkeiten, andererseits stellt sie die fachliche Entsprechung zu einem Funktionsbaustein im R/3 dar.
Die TRANSPORTSCHICHT erkennt Ausnahmen, die durch einen Funktionsbausteinaufruf in der C-LIBRARY auftreten können, und löst entsprechende Ausnahmen in TYCOON-2 aus (siehe Abschnitt 7.3.7).

ANWENDUNGSSCHICHT Die auf die Transportschicht aufsetzende ANWENDUNGSSCHICHT ist zuständig für die statische Typisierung der Schnittstelle. Es werden abgeleitete Klassen von **SapFunctionLib** und **SapStructure** generiert, welche die für einen Funktionsbausteinaufruf benötigten Informationen enthalten. Konkrete generierte Ableitungen der **SapFunctionLib** entsprechen der Funktionsbibliothek eines R/3-Systems und ermöglichen den Zugriff auf die SAP R/3 Funktionsbausteine über deren Methoden. Abgeleitete generierte Klassen von **SapStructure** entsprechen den strukturierten Parametern eines Funktionsbausteins.

Die Klassen **SapFunctionLib** und **SapStructure** ermöglichen es, die automorphen Werte der unteren Schicht zu kapseln und den statisch typisierten Zugriff auf das R/3-System zu gewährleisten. In den erzeugten Klassen werden dazu die TYCOON-2-Basisklassen (**Int**, **String**, **Date**, **Time** usw.) für die einfachen Funktionsbausteinparameter oder generierte abgeleitete Klassen von **SapStructure** für die strukturierten Parameter eines Funktionsbausteins benutzt. Die abstrakten Klassen **SapFunctionLib** und **SapStructure** enthalten darüberhinaus Methoden für den Zugriff auf die in den generierten Klassen enthaltenen Metainformationen zur Laufzeit (siehe auch Abschnitt 7.3.5.3).

Die Klasse **SapTable** ermöglicht die Erzeugung von Objekten, die den Tabellenparametern eines Funktionsbausteins entsprechen. Die Klasse **SapTable** bietet Methoden an, welche die Verwendung einer Tabelle unterstützen (z.B. für das Einfügen und

Löschen von Zeilen). Die Klasse **SapTable** ist eine parametrisierbare Klasse. **SapTable** wird mit einer abgeleiteten Klasse von **SapStructure** parametrisiert und stellt in TYCOON-2 eine Tabelle zur Verfügung, deren Spalten durch die übergebene Struktur definiert werden. Die Spaltennamen der so erzeugten Tabelle entsprechen also den Feldern der übergebenen Struktur. Auf die einzelnen Felder einer Tabellenzeile kann über den Feldnamen zugegriffen werden.

Beispiel: Die Funktionsbibliothek **funcLib** enthält u.a. eine Methode **bapi_customer_checkexistence**. Ein Aufruf der Methode gestaltet sich folgendermaßen:

```
sales_org := "0001", (* vom Typ String *)
customer_no := "11", (* vom Typ String *)
d_channel := "01",   (* vom Typ String *)
division :=  "01,    (* vom Typ String *)
return:=funcLib.bapi_customer_checkexistence(
                                      sales_org,
                                      customer_no,
                                      d_channel,
                                      division
                                     )
(* return ist ein multipler Ergebnisparameter *)

(*Benutzung einer Tabelle, Klasse SapRFCSI bildet ABAP-Struktur RFCSI ab*)
table_system_info := SapTable.new(SapRFCSI),(*Erzeugung der Tabelle*)
tableLine := table_system_info.append(),
(*Zeile anfügen,tableLine vom Typ SapRFCSI *)
tableLine.RFCHOST := "134.28.70.8"
(*Zugriff auf das Feld RFCHOST der Tabellenzeile*)
```

GESCHÄFTSOBJEKTSCHICHT Die oberste Schicht der Geschäftsobjekte bietet den Zugriff auf R/3 Geschäftsobjekte über deren BAPIs. Eine von der Klasse **SapObject** abgeleitete Geschäftsobjektklasse bildet ein R/3-Geschäftsobjekttyp ab. Die Methoden der Geschäftsobjektklasse stellen die BAPIs dar. Diese Klasse ist für die Ausnahmenbehandlung der Geschäftsobjekte in TYCOON-2 zuständig (siehe Abschnitt 7.3.6).

Beispiel: Die Geschäftsobjektklasse **SapBOCustomer** bietet die Klassenmethode **checkexistence** an. Ein Aufruf sieht folgendermaßen aus:

```
sales_org := "0001",
customer_no := "11",
d_channel := "01",
division :=  "01,
return := SapBOCustomer.checkexistence(sales_org, customer_no,
                                       d_channel, division
                                      ),
(* return vom Typ SapBAPI_CUSTOMER_CHECKEXISTENCEexport *)

customer:=SapBOCustomer.new(sales_org,
```

```
                                customer_no,
                                d_channel,
                                division)
(* Erstellt in TYCOON-2 eine Referenz auf ein existierendes R/3-Objekt *)
```

Der Rückgabetyp der Methode **checkexistence** stellt eine Aggregation aller Exportparameter des BAPIs dar. Um auf einen Exportparameter dieses Aufrufs zugreifen zu können, genügt es den Namen des Parameters anzugeben:

```
kna1 := return.CUSTOMER_DATA (* Exportparameter CUSTOMER_DATA *)
```

Aus der Struktur vom Typ **SapKNA1** läßt sich z.B. der Kundenname herauslesen mit:

```
kna1.NAME
```

7.2.2 Semantische Objekte in TYCOON-2

Im folgenden werden die semantischen Objekte des R/3 beschrieben, an denen sich die gewählte Implementierung orientiert. Die semantischen Objekte werden dabei durch Klassen im TYCOON-2-System abgebildet. Bei der Beschreibung dieser TL-2-Klassen werden in der Art eines *bottom-up*-Ansatzes zunächst grundlegende Objekte zur Kommunikation mit dem R/3 beschrieben, um darauf aufbauend, höherwertige Abstraktionen zu finden.

Grundvoraussetzung einer Kommunikation mit dem R/3 stellt die Verbindung zum R/3 über die RFC-Automation dar. Mit dieser Verbindung lassen sich Funktionsbausteine aufrufen. Die Komplexität des Funktionsbausteinaufrufs wird durch die Parameter und die erforderlichen Metainformationen bestimmt. Für die statische Typisierung der Schnittstelle und zur Komplexitätsreduzierung werden diese durch generierte TL-2-Stubklassen modelliert. Funktionsbausteine werden durch eine generierte, von der abstrakten Klasse **FunctionLib** abgeleiteten, Klasse aggregiert, welche die R/3-Funktionsbibliothek darstellt. In der dazugehörigen Metaklasse befinden sich sämtliche Metainformationen zu den in der Klasse enthaltenen Funktionsbausteinen.

Die höchste Abstraktionsebene beim R/3-Zugriff und das primäre Ziel dieser Arbeit stellen Geschäftsobjekte und ihre Methoden dar, deren Abbildung auf TL-2-Klassen im letzten Abschnitt 7.3.6 beschrieben werden. Die Methoden der Geschäftsobjektklassen sind die sogenannten BAPIs, die beim Zugriff auf das R/3 betrachtet werden.

7.2.2.1 Verbindung zum SAP R/3-System

Die Kommunikation mit dem R/3 über die Technik der RFC-Automation ist verbindungsorientiert. Das heißt, zum Aufruf von R/3-Funktionsbausteinen muß eine Verbindung zum R/3 geöffnet werden und nach Beendigung der (unter Umständen langandauernden) Interaktion wieder geschlossen werden.

Diese Verbindung zum R/3 wird durch die TL-2-Klasse **SapConnection** modelliert. Die Verbindung ist dabei mit den Angaben zum auszuwählende R/3-System und mit den erforderlichen Benutzerinformationen für eine Anmeldung zum R/3-System zu parametrisieren (siehe Abbildung 7.3).

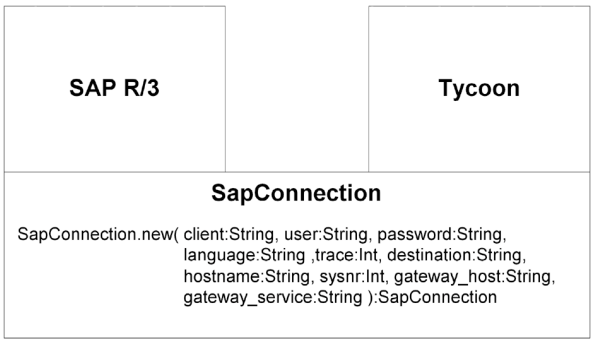

Abbildung 7.3: Verbindung zum R/3 über die Klasse **SapConnection**

Auf einer geöffneten Verbindung lassen sich Funktionsbausteinaufrufe ausführen. Aus diesem Grund ist die Funktionalität zur Durchführung von R/3-Aufrufen in der Klasse **SapConnection** konzentriert. Ein Objekt dieser Klasse hält dazu den Verbindungshandle[2] als auch über die Klasse **SapLib** die Verbindung zur SAP-C-Library.

Vor dem eigentlichen Aufruf werden die an den R/3-Funktionsbaustein zu übergebenen Parameter, die durch TYCOON-2-Objekte repräsentiert werden, durch die Klasse **SapConnection** in dynamisch typisierte Transferobjekte überführt, welche auch das *Marshalling* der Parameter durchführen.

Um einen R/3-Aufruf durchführen zu können, müssen durch die Klasse **SapConnection** C-RFC-Speicherstrukturen aufgebaut werden, welche die Laufzeitparameter eines Funktionsbausteinaufrufes darstellen. Aus Gründen der Performanz werden diese für einen Aufruf benötigten C-Strukturen nach dem R/3-Aufruf nicht wieder abgebaut, sondern aufrechterhalten, um so ohne Neuerzeugung für wiederholte Aufrufe bereitzustehen. Die **SapConnection** verwaltet die C-RFC-Strukturen, auf die im Detail im Kapitel 7.3.2 eingegangen wird.

SapConnection-Objekte sind Ressourcen, d.h. geöffnete Verbindungen zum R/3-System werden bei einem Neustart des TYCOON-2-Systems wieder geöffnet.

7.2.2.2 Funktionsbaustein

Das semantische Objekt Funktionsbaustein (repräsentiert durch die TL-2-Klasse **SapFunction**) stellt die grundlegende logische Einheit zur Interaktion mit dem R/3 dar. Mit diesem Element steht prinzipiell die gesamte Funktionalität des R/3 zur Verfügung.

Der Funktionsbaustein in TYCOON-2 wird durch seinen Namen beschrieben und aggregiert seine Parameter. Die Parameter des Funktionsbausteins werden dabei durch Listen von dynamisch typisierten Parameterobjekten beschrieben. Bei den Parametern wird zwischen einfachen und strukturierten Parametern sowie Tabellenparametern unterschieden. Die Unterscheidung in Import- und Exportparameter ist an dieser Stelle noch nicht relevant und wird

[2]Die Verbindungsinformationen werden der Funktion **RfcOpen()** übergeben, die damit eine Verbindung zum entsprechenden Applikationsserver herstellt. Die Funktion liefert einen *Handle* zurück, der die Verbindung bei folgenden Aufrufen von Funktionsbausteinen identifiziert. Es können mehrere Verbindungen nebenläufig aufgebaut werden.

erst bei der Erstellung der C-RFC-Speicherstrukturen für den Aufruf durch die Klasse **SapConnection** berücksichtigt. Die Parameterobjekte sind in Anlehnung an die für den Aufruf erforderlichen C-RFC-Strukturen als automorphe Werte modelliert. Das heißt, ein Parameter trägt zusätzlich zu seinem Wert auch seine Metainformationen (Name, Länge und Typ) mit sich.

Die Klasse **SapFunction** und die Klassen der Parameter **SapSimpleParam**, **SapStructuredParam** und **SapTableParam** sind aufgrund ihrer niedrigen Abstraktion nicht für den Benutzer der Schnittstelle gedacht, sondern werden nur für interne Zwecke benutzt. Der Benutzer der TYCOON-2-R/3-Schnittstelle verwendet stattdessen Objektexemplare einer generierten TYCOON-2-Klasse Funktionsbibliothek, deren Methoden die Funktionsbausteine darstellen.

7.2.2.3 Funktionsbibliothek

Das *Workbenchwerkzeug* Funktionsbibliothek verwaltet die Funktionsbausteine eines R/3-Systems und stellt aufgrund der Integration mit dem *ABAP Dictionary* auch die erforderlichen Metainformationen wie Funktionssignaturen sowie Informationen zu Strukturen der Parameter zur Verfügung.

In Anlehnung an die R/3-Funktionsbibliothek bietet die TYCOON-2-Klasse **SapFunctionLib** die R/3-Funktionsbausteine als Methoden an. Metainformationen zu Funktionsbausteinen und Parametersignaturen sind in der Metaklasse abgelegt. Diese Metainformationen werden dabei für den Aufruf selber benutzt, stehen aber auch den Benutzern der Schnittstelle für die Konsistenzprüfung zur Verfügung.

Da das *Workbenchwerkzeug* Funktionsbibliothek über 30000 Funktionsbausteine enthält, von denen eine Vielzahl für eine konkrete Anwendung nicht gebraucht werden, wird nur die wirklich benötigte Teilmenge an R/3-Funktionsbausteinen als Methoden der TYCOON-2-Klasse abgebildet. Die benötigte Menge von Funktionsbausteinen wird durch den Schnittstellennutzer sinnvollerweise zu Beginn der Entwicklung spezifiziert. Nach erfolgter Spezifizierung können die Stubklassen für die Funktionsbibliothek und die Parameter durch den in dieser Schnittstelle enthaltenen Generator erzeugt werden. Der Generator und seine generierten Konstrukte werden im Detail in Abschnitt 7.3.5 vorgestellt.

Um einen Aufruf an das R/3 auszuführen, ist ein Objekt der generierten Funktionsbibliotheksklasse **SapFunctionLib** zu erzeugen und an eine geöffnete Verbindung (**SapConnection**) zu binden. Der Aufruf einer Methode dieser Klasse führt den Aufruf des entsprechenden Funktionsbausteins im R/3 durch. Innerhalb eines Methodenkörpers werden die Parameterstubobjekte in Transferobjekte umgewandelt, der Aufruf mit den Funktionen der SAP-C-LIBRARY durchgeführt, und die vom R/3-System erhaltenen Werte wieder in die Parameterstubobjekte geschrieben.

Innerhalb des Methodenkörpers wird ein synchroner Aufruf des R/3 realisiert, um so die Semantik der synchronen Ausführung einer Objektmethode zu erhalten. Der synchrone Aufruf eines Funktionsbausteins bzw. BAPIs ist der von der SAP propagierte Weg. Der synchrone Aufruf ist allerdings nicht transaktional. Ein transaktionaler Aufruf[3] würde den Schnittstellen-

[3] Ein transaktionaler Aufruf ist immer ein asynchroner Aufruf.

benutzer mit der Programmierung des Wartens auf die Ausführung des Aufrufs belasten und so die angestrebte Komplexitätsreduzierung bei der Benutzung einer R/3-Schnittstelle wieder aufheben. Zudem hat der transaktionale Aufruf eines Funktionsbausteins Einschränkungen hinsichtlich der möglichen Parameter, da er keine Rückgabe von Exportparametern an das rufende Programm gestattet.

Das Konzept der **SapFunctionLib** bietet die Möglichkeit, flexibel auf verschiedene Releasestände des R/3 reagieren zu können. Das Problem ist, daß die in der Funktionsbibliothek vorhandenen Funktionsbausteine und ihre Signaturen über SAP-Releases hinweg Änderungen unterworfen sind. Daraus ergeben sich Problematiken für externe Schnittstellen, die auf hoher Abstraktionsebene mit Hilfe von Stubklassen auf das R/3 zugreifen wollen. Die im Rahmen dieser Arbeit entwickelte Schnittstelle versucht diese Problematik aufzufangen, indem die Stubklassen durch die Schnittstelle generierbar sind und so Änderungen an Funktionsbausteinen und Signaturen durch eine Neugenerierung nachgeführt werden können (siehe Abschnitt 7.3.5.1).

Ist der gleichzeitige Zugriff auf R/3-Systeme mit verschiedenen Releaseständen erforderlich, löst die Generierung jedoch nicht das Problem, daß jeder Klassenname innerhalb des globalen TYCOON-2-Namensraums eindeutig sein muß. In diesem Fall müßten manuell die generierten Stubklassen und die Funktionsbibliothek umbenannt werden. Die Problematik der nicht stabilen Schnittstellen wird aber im Kapitel 7.3.5.1 noch gesondert behandelt.

7.2.2.4 TYCOON-2-Basis- und Stubklassen zur Abbildung von Parametern

R/3-Funktionsbausteine sind - wie bereits erwähnt - als Methoden einer generierten Klasse **SapFunctionLib** abgebildet. Die Parameter des Funktionsbausteins sind als Argumente der Objektmethode modelliert. Bei den Parametern unterscheidet die Schnittstelle zwischen einfachen Parametern, strukturierten Parametern und Tabellenparametern. Strukturierte Parameter und Tabellenparameter besitzen im Gegensatz zu einfachen Parametern eine Feldstruktur.

Einfache Parameter werden gemäß der im Kapitel 7.3.2 definierten Abbildung auf TYCOON-2-Basistypen wie **Integer**, **String** und **Date** abgebildet. Dies bedeutet, daß ein TYCOON-2-Entwickler bei der Benutzung der R/3-Schnittstelle die unterschiedliche Semantik der ABAP-Basistypen nicht berücksichtigen muß.

Strukturierte Parameter werden durch TYCOON-2-Stubklassen modelliert. Den Feldern der strukturierten Parameter entsprechen die *Slots* der entsprechenden TYCOON-2-Klasse. Die Typen dieser *Slots* sind wiederum die TYCOON-2-Basistypen. Hierarchisch oder rekursiv strukturierte Parameter sind also nicht möglich. Dies modelliert die Beschränkung im R/3, daß solche Art von komplexen Parametern nur in Form von internen Tabellen und Feldleisten innerhalb von ABAP-Programmen erstellbar, aber nicht in der Signatur von Funktionsbausteinen zugelassen sind. Die *Slots* sind durch Zugriffsmethoden vor dem schreibenden Zugriff geschützt, da bei einem schreibendem Zugriff zusätzliche durch das R/3-System vorgegebene Integritätsbedingungen (hier insbesondere die Länge des zugewiesenen Wertes als auch der Wertebereich) abgeprüft werden. Die Namensgebung der TYCOON-2-Stubklasse orientiert sich dabei am Namen der ABAP

Dictionary-Struktur, z.B. repräsentiert die Klasse **SapKNA1** die ABAP *Dictionary*-Struktur KNA1.

Tabellenparameter stellen in der gewählten Implementierung eine Aggregation von TYCOON-2-Stubobjekten dar, welche die Zeilen des R/3-Tabellenparameters repräsentieren. Die Klasse **SapTable(T:<SapStructure)** nutzt hier die Möglichkeiten des parametrischen Polymorphismus in TYCOON-2. Sie bietet elementare Operationen wie das Erzeugen und Löschen auf den Tabellenzeilen an. Zugriffe auf Tabelleninhalte (Zellen) werden an die jeweilige Stubklasse delegiert.

Der Vorteil dieser gewählten Implementierung ist die Realisierung einer statischen Typprüfung, da sämtliche Typen der Funktionsbausteinparameter durch statisch getypte TYCOON-2-Basisklassen oder TYCOON-2-Stubklassen repräsentiert werden. Durch diese Übertragung des R/3-Typkonzeptes in die TYCOON-2-Umgebung ist erst eine Arbeit mit der Schnittstelle ohne tiefere Kenntnisse des R/3-Systems möglich. Die im Normalfall nicht erforderlichen Metainformationen zu einem Parameter (Name, Typ und Länge) werden in der Metaklasse der Stubklasse verborgen, stehen aber dem Schnittstellenbenutzer lesend zur Verfügung.

Eine gewisse Problematik birgt die erforderliche Anzahl von TYCOON-2-Stubklassen, die für den Aufruf verschiedener Funktionsbausteine nötig sind. Die für die Definition von strukturierten Parametern benutzten *ABAP Dictionary*-Strukturen werden nur in Ausnahmefällen durch mehrere Funktionsbausteine genutzt. Es ist vielmehr üblich, daß jeder Funktionsbaustein seine eigenen speziellen Strukturen benutzt. Bei einer gegebenen Menge anzubindender R/3-Funktionsbausteine ist ungefähr mit der doppelten oder dreifachen Menge von Stubklassen zu rechnen, die durch die Schnittstelle zu generieren sind. Dies kann zu einer nicht unerheblichen Belastung des TYCOON-2-Systems führen.

7.2.2.5 Geschäftsobjekt und BAPI

Das Hauptziel dieser Arbeit stellt die Anbindung der R/3-Geschäftsobjekte an das TYCOON-2-System dar. Ein R/3-Geschäftsobjekt stellt seine Funktionalität externen Anwendungen über Methoden (BAPIs) zur Verfügung. Die Nutzung von Makro-ABAP und damit zusätzlich der Zugriff auf Schlüsselfelder und Attribute eines Geschäftsobjekts ist prinzipiell durch den Aufruf der Funktionsbausteine möglich, auf die sich Makro-ABAP abstützt. Dieses stellt jedoch keine stabile Schnittstelle zum R/3 dar, da sie von der SAP nicht freigegeben ist, jederzeit Änderungen unterworfen sein kann und zudem keine Typisierung zuläßt.

Dies läßt externe Anwendungen bei der Nutzung der R/3-Geschäftsobjekte auf die BAPIs fokussieren, wie es die Vorstellung der kommerziellen R/3-Schnittstellen in Kapitel 5 bereits gezeigt hat. Aus dem R/3-System läßt sich zu einem BAPI der implementierende Funktionsbaustein und dessen Signatur ermitteln, der dann über die RFC-Automation angesprochen werden kann. Aus diesen Gründen wird die TYCOON-2-R/3-Schnittstelle auf der Grundlage des Aufrufs von Funktionsbausteinen entwickelt. In TYCOON-2 wird ein Geschäftsobjekt gemäß der im *Business Object Repository* enthaltenen Definition angeboten.

TYCOON-2-Geschäftsobjekte übertragen den R/3-Identitätbegriff in die TYCOON-2-Welt. R/3-Geschäftsobjekte werden durch Schlüsselfelder identifiziert, während TYCOON-2-Objekte durch eine vom TYCOON-2-System vergebene für den Objektbenutzer nicht zugreifbare

Objektnummer gekennzeichnet werden. Die TYCOON-2-Geschäftsobjekte stellen persistente R/3-Geschäftsobjekte dar, an denen die BAPIs aufgerufen werden können. Die Erzeugung von TYCOON-2-Geschäftsobjekten, die keine Entsprechung im R/3-System haben, ist nicht vorgesehen, da diese keine BAPI-Funktionalität besitzen. R/3-Geschäftsobjekte besitzen eine erweiterte Ausnahmebehandlung (siehe Abschnitt 7.3.7), die auf der Exportstruktur **BAPIRETURN** der R/3-Geschäftsobjektmethoden aufsetzt.

Die Implementierung von TYCOON-2-Geschäftsobjekten läßt sich nicht wie eine Funktionsbibliothek (**SapFunctionLib**) oder die Stubklassen generieren. Das Problem ist hierbei die Unvollständigkeit des fachlichen Objektmodells, wie es im R/3 im *Business Object Repository* (BOR) abgelegt ist. Konkret sind es fehlende Informationen zu den Schlüsselfelder von Geschäftsobjekten sowie die Abbildung der Schlüsselfelder auf die Parameter der BAPIs.

Beispielsweise wird der Debitor im BOR durch sein Schlüsselfeld **DEBITORID** identifiziert. Sowohl beim Aufruf der BAPIs als auch über die SAP-Transaktion wird jedoch zusätzlich der Buchungskreis als Schlüssel benötigt. Beim R/3-Geschäftsobjekt Verkaufsauftrag ist ebenso nur eines der vier notwendigen Schlüsselfelder modelliert.

Mit den erwähnten Einschränkungen wird so das Modell der R/3-Geschäftsobjekte an das TYCOON-2-System angebunden und eine Interaktion auf der Grundlage von BAPIs ermöglicht. Auf die Problematiken bei der Nutzung von Funktionsbausteinen als Implementierung von BAPIs soll im Detail in 7.3.6 eingegangen werden.

Durch den Mechanismus der RFC-Automation ist prinzipiell ein bidirektionale Anbindung an das R/3 möglich. Diese ist jedoch für das Ziel einer Anbindung der R/3-Geschäftsobjekte an das TYCOON-2-System nicht relevant. Im Rahmen dieser Schnittstelle werden Aufrufe im R/3 initiiert und nicht umgekehrt. Diese Einschränkung ist aus folgenden Gründen sinnvoll:

1. Die Geschäftslogik befindet sich im R/3-System. Ziel der Schnittstelle ist es, die Funktionalität des R/3-Systems in TYCOON-2 zur Verfügung zu stellen.

2. TYCOON-2 unterstützt keinen Funktionsrückruf, d.h. es ist nicht möglich, von einer externen Anwendung in TYCOON-2 eine Funktion aufzurufen. Der Funktionsrückruf könnte alternativ z.B. mit Hilfe eines parallel laufenden Beobachterprozeßes realisiert werden, der für die Verwaltung von rückrufbaren Objekten bzw. deren Methoden zuständig ist, auf entsprechenden Anforderungen seitens der Schicht C-LIBRARY reagiert und alle nötigen Aktionen ausführt[4]. Die Umsetzung einer solchen Lösung wird aber aufgrund der Aussage in Punkt eins nicht in Betracht gezogen.

3. Es existieren keine BAPIs, die den Funktionsrückruf nutzen. Funktionsrückrufe sind innerhalb von BAPIs nicht sinnvoll, da ein Rückruf die transaktionale Semantik eines BAPIs gefährdet[5].

[4] In [Schneider 98] wird beispielsweise eine bidirektionale typisierte Kommunkikation zwischen zwei reflexiven, objektorientierten Systemen vorgestellt, wobei TYCOON-2 eins der Systeme ist.

[5] Durch einen Funktionsrückruf wird eine zweite SAP-LUW geöffnet. Eine transaktionale Semantik erfordert die Ausführung des BAPIs in genau einer SAP-LUW.

7.3 Implementierung

In den bisherigen Abschnitten dieses Kapitels wurden grundlegende Eigenschaften der implementierten Schnittstelle erläutert. Über das Schichtenmodell wurde eine Zuordnung der einzelnen Klassen der Schnittstelle zur jeweiligen Schicht vorgenommen, wobei die Aufgaben jeder Schicht kurz skizziert wurden. Im vorherigen Abschnitt wurden diejenigen semantischen Objekte des SAP R/3 identifiziert und beschrieben, die bei der programmiertechnischen Umsetzung zu berücksichtigen sind.

In den folgenden Abschnitten wird die Implementierung im Detail erläutert. Dabei wird mit der Beschreibung der Typabbildung in Abschnitt 7.3.1 und des *Marshallings* in Abschnitt 7.3.2 begonnen, welche sich grundlegend auf die weiteren Eigenschaften der Schnittstelle auswirken. Aufbauend auf der gewählten Typabbildung wird in Abschnitt 7.3.3 die Interaktion von TYCOON-2 mit dem R/3-System auf der Ebene von Funktionen erläutert. Den für den Zugriff benötigten Metainformationen, die für die erweiterte Konsistenzprüfung benötigt werden und auch zur Laufzeit zur Verfügung stehen, wird Abschnitt 7.3.4 gewidmet. Die Abbildung der Metainformationen ermöglicht es, im darauffolgenden Abschnitt 7.3.5 die automatische Generierung von Stubklassen zu beschreiben. Im Abschnitt 7.3.6 „Geschäftsobjekte“ wird die gewählte Abbildung der Geschäftsobjekte erläutert. Zum Abschluß dieses Abschnittes wird die Ausnahmebehandlung beschrieben, indem die möglichen Ausnahmearten den einzelnen Schichten zugeordnet und erklärt werden.

7.3.1 Typabbildung

ABAP-Typ	**TYCOON-2-Klasse**
C (Zeichen)	Char
C(n) (Zeichenkette)	String
D (Datum)	Date
T (Zeit)	Time
N(n) (*Numeric*)	Int
X(n) (Hexadezimalzahl)	String
F (Gleitkommazahl)	Real
P (Gepackte Zahl)	Real
I (Ganzzahl)	Int
ABAP Dictionary-Struktur oder Tabelle	Stubklasse (Felder als Slots)

Tabelle 7.1: Typabbildung zwischen ABAP und TYCOON-2

Tabelle 7.1 zeigt die Abbildung eines ABAP-Typ auf die entsprechende TYCOON-2 Stubklasse bzw. TYCOON-2-Basisklasse. Die ABAP-Typen C (Zeichen), I (Ganzzahl) und F (Gleitkommazahl) besitzen dieselbe Semantik im TYCOON-2-System und werden als Parameter der Schicht **SAP-C-LIBRARY** an die Zwischenschicht zwei weitergereicht bzw. von dort nach TYCOON-2 durchgereicht. Anstatt der ABAP-Typen Feldleiste und interne Tabelle sind in der genannten Tabelle ABAP Dictionary-Struktur und -Tabelle aufgeführt, da diese in der benutzten RFC-Automationsschnittstelle die Eigenschaften von strukturierten Parametern bestimmen.

Für die längenbeschränkten ABAP-Typen **C(n)**, **N(n)** (*Numeric*) und **X(n)** (Hexadezimalzahl), die bei der RFC-Automation durch den Typ **char**[] repräsentiert wird, ist eine weitere Behandlung erforderlich. Da der ANWENDUNGSSCHICHT alle relevanten Typinformationen des ABAP-Typs (Metadaten) zur Verfügung stehen, ist es möglich beim Setzen bzw. Lesen des jeweiligen Wertes dieser Typen für eine Überprüfung bzw. korrekte Konvertierung des Werts zu sorgen. Wie die aus dem R/3-System herausgelesenen Typinformationen in TYCOON-2 zur Verfügung gestellt werden, ist in Abschnitt 7.3.4 beschrieben.

Der ABAP-Typ **X(n)** (Hexadezimalzahl) stellt ein gesondertes Problem dar, da er ein Feld von beliebigen Werten darstellt. Zwischen TYCOON-2 und der **SAP-C-LIBRARY** ist es nicht möglich, ungetypte Werte auszutauschen, so daß für den Typ **X(n)** in TYCOON-2 der Typ **String** benutzt wird. Die Klasse **String** für den Typ **X(n)** bereitet ein Problem bei einem Nullwert (0x00), da dieser damit den String terminiert. Folgende Informationen werden abgeschnitten.

In TYCOON-2 werden generierte Stubklassen zur Verfügung gestellt, die den Strukturen aus dem R/3 entsprechen. Mit Hilfe dieser Klassen lassen sich nicht nur die Strukturen selber darstellen, sondern auch R/3-Tabellen, und zwar werden diese in TYCOON-2 durch eine mit Stubklassen parametrisierte Tabellenklasse **SapTable(T<:SapStructure)** abgebildet. Komplexe Typen wie Strukturen und Tabellen werden nicht gesondert behandelt, da diese ihrerseits als linearisierte Ströme der ABAP-Basistypen durch die SAP-C-LIBRARY übertragen werden und die TYCOON-2-R/3-Schnittstelle die Semantik von komplexen Typen aus diesem Strom wiederherstellen muß.

7.3.2 Transportobjekte und *Marshalling*

Ein Transportobjekt entspricht genau einem für einen Funktionsbausteinaufruf benötigten Parameter. Ein Transportobjekt dient einerseits für den Aufbau der korrespondierenden C-RFC-Speicherstrukturen (Formalparameter) als auch für die Übertragung des Aktualparameters. In Abbildung 7.4 ist die C-RFC-Speicherstruktur für einen einfachen Parameter bereits erzeugt, so daß bei weiteren Aufrufen nur noch der Wert (Aktualparameter) übertragen werden muß. Ein Transportobjekt enthält den für das *Marshalling* benötigten ABAP-Typ sowie den Namen, Länge und Wert eines Parameters. Transportobjekte sind Objekte der Klasse **SapSimpleParam** bzw. der Klassen **SapStructuredParam** oder **SapTableParam**, mit denen die drei verschiedenen Parametertypen (einfache Parameter, strukturierte Parameter und Tabellenparameter) durch TYCOON-2-Klassen abgebildet sind.

Initialisierung der Transportobjekte: Transportobjekte werden bei ihrer Erzeugung mit typabhängigen Standardwerten initialisiert. Diese Initialisierung bewirkt, daß sich gültige Parameterstrukturen in TYCOON-2 befinden, was insbesondere bei strukturierten Parametern sinnvoll ist, da bei einem Funktionsbausteinaufruf oftmals nur eine geringe Teilmenge der Felder in den Parameterstrukturen gefüllt werden muß. Die Initialisierung der Parameter läßt sich bei strukturierten Parametern und Tabellen elegant über die Methode **perform**[6] der Klasse **Object** von TYCOON-2 lösen. Damit kann eine generische Initialisierungmethode

[6]Die Methode **perform** erlaubt es, eine durch ihren Namen spezifizierte Methode an einem Objekt auszuführen. Dadurch besteht die Möglichkeit, über die einzelnen *Slots* zu iterieren und anhand der zugehörigen Metadaten den entsprechenden Initialwert zu setzen.

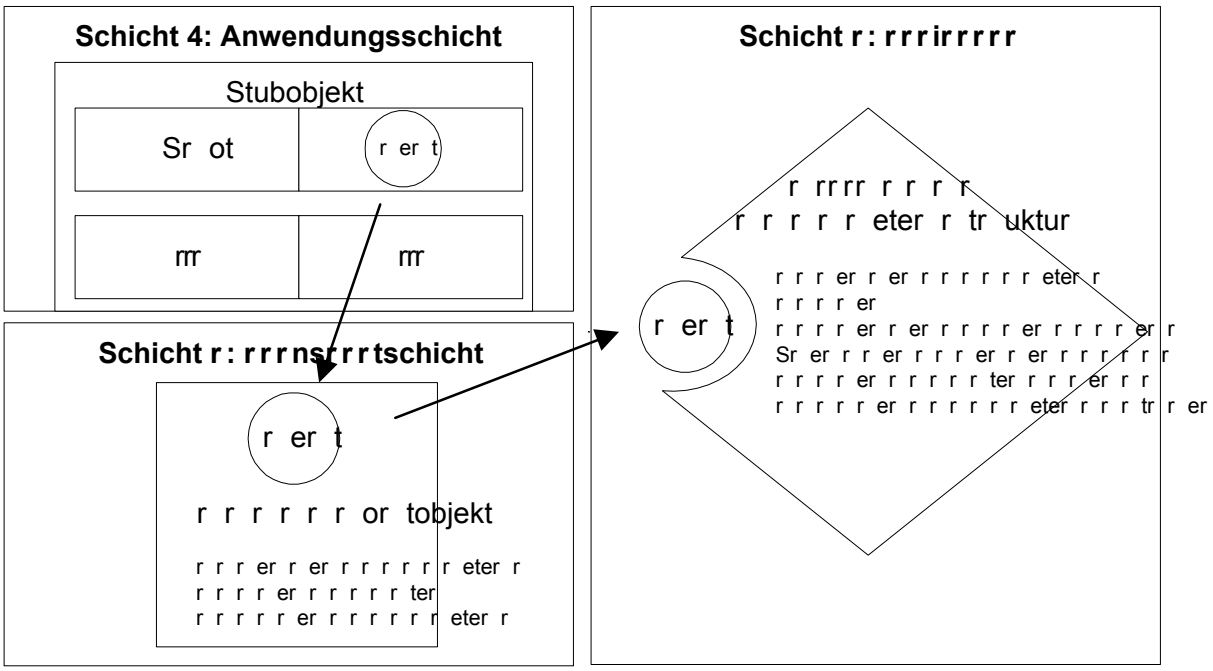

Abbildung 7.4: Transportobjekte für das *Marshalling*

implementiert werden.

Marshalling der Transportobjekte: Transportobjekte haben Zugriff auf ihre Metadaten. Bei Wertzuweisungen wird der Wert auf Übereinstimmung mit den Metadaten geprüft. Die TRANSPORTSCHICHT trägt dadurch zur Konsistenzprüfung und zur Sicherheit in der Benutzung der Schnittstelle bei.

Transportobjekte werden innerhalb eines Objektes der Klasse **SapFunction** aggregiert, das für den Aufruf eines bestimmten Funktionsbausteins innerhalb der TRANSPORTSCHICHT zuständig ist. **SapFunction**-Objekte müssen sich wiederum bei einem Objekt der Klasse **SapConnection** registrieren, um auf einer geöffneten R/3-Verbindung Funktionsaufrufe durchführen zu können. Nachdem ein **SapFunction**-Objekt registriert wurde, werden die Formalparameter aus einem Transportobjekt, also die Informationen, die sich in den Metadaten befinden, zur Schicht C-LIBRARY transportiert. Die Formalparameter befinden sich damit im flüchtigen Speicher. Dies geschieht einmalig entweder bei der Registrierung oder bei Neustart eines zuvor gesicherten TYCOON-2-Systems, da dann wieder die C-RFC-Speicherstrukturen für den Funktionsaufruf neu angelegt werden müssen. Bei einem dann folgenden R/3-Funktionsbausteinaufruf über die Methode **callReceive** werden nur noch die Aktualparameter zur zweiten Schicht C-LIBRARY durchgereicht und nach einem erfolgreichen Aufruf wieder ausgelesen. Das Schreiben der Formalparameter und die Behandlung der Aktualparameter wird folgendermaßen gelöst: Mit Hilfe des Besuchermusters [Gamma et al. 96] wird abhängig vom Typ und Art des Parameters eines Transportobjektes die entsprechende Methode aus der Klasse **SapLib** für das *Marshalling* eines Parameters aufgerufen. Dafür existieren jeweils Besucher für das Schreiben bzw. Lesen aller Parameterarten. Dies gilt nicht

ABAP-Typ	Initialisierung
C (Zeichen)	‘‘
C(n) (Zeichenkette)	” ”
D (Datum)	Date.new
T (Zeit)	Time.new
N(n) (Numeric)	” ”
X(n) (Hexadezimalzahl)	” ”
F (Gleitkommazahl)	0
P (Gepackte Zahl)	0
I (Ganzzahl)	0

Tabelle 7.2: Initialisierung der ABAP-Typen

für strukturierte Parameter:

Ein strukturierter Parameter muß gesondert behandelt werden, da die Felder des Parameters innerhalb der C-LIBRARY auf einen RFC-Parameter abgebildet werden. Der C-Typ **RFC_PARAMETER** wird sowohl für einfache, als auch für strukturierte Parameter benutzt. Dem **RFC_PARAMETER** fehlt aber die Feldsemantik. Der Schnittstelle sind die Typen und Längen der einzelnen Felder aber bekannt, so daß daraus die einzelnen Felder wieder gewonnen werden können. Die Parameterleisten der Methoden für das *Marshalling* von strukturierten Parametern müssen also im Gegensatz zu den typabhängigen Methoden für die einfachen Parameter um einen Anfangswert (*Offset*)[7] und eine Länge ergänzt werden.

7.3.3 TYCOON-2 - R/3 Interaktion auf Funktionsebene

Die Interaktion mit dem R/3-System findet mit Hilfe der entfernt aufrufbaren Funktionsbausteine statt. In Abbildung 7.5 stellt die generierte Klasse **FunctionLib** den Container der Funktionsbausteine dar, vergleichbar mit der Funktionsbibliothek des R/3-Systems. Bevor eine Methode der Funktionsbibliothek ausgeführt werden kann, ist eine Verbindung zum R/3-System zu öffnen und der Funktionsbibliothek bekannt zu geben. Der Funktionsbaustein **BAPI_CUSTOMER_SEARCH**[8] hat beispielweise als Methode der generierten Klasse **SapFunctionLib** folgende Signatur:

```
bapi_customer_search(
                     pi_salesorg:String,
                     pi_search_flag:Char,
                     pi_address:SapBAPIKNA101,
                     multiple:SapTable(SapBAPIKNA103)
                    ):SapBAPI_CUSTOMER_SEARCHexport
```

Folgender Programmausschnitt zeigt die Benutzung dieser Methode:

[7] Der *Offset* beschreibt den Beginn eines Feldes innerhalb des **RFC_PARAMETER**.

[8] Der Funktionsbaustein **BAPI_CUSTOMER_SEARCH** liefert nach der Eingabe verschiedener Suchkriterien eine Liste von Kundenschlüsseln zurück.

```
kna101 := SapBAPIKNA101.new, (* strukturierter Parameter BAPIKNA101 *)
table_kna103 :=SapTable(SapBAPIKNA103).new,
(* Tabellenparameter parametrisiert mit der Struktur BAPIKNA103 *)
exportStruct := (* vom Typ SapBAPI_CUSTOMER_SEARCHexport *)
functionLib.bapi_customer_search("0001",
                                 '*',
                                 kna101,
                                 table_kna103
                                 );
```

Der Methodenname in der Klasse **FunctionLib** entspricht dabei dem Namen des Funktionsbausteins. Die Menge der möglichen Typen der Parameter setzt sich aus TYCOON-2-Basisklassen oder aus generierten Stubklassen bzw. den mit einer Stubklasse parametrisierten Klasse **SapTable** zusammen. Ein Funktionsbaustein kann mehrere Ergebnisparameter haben. Die gewählte Lösung für die Behandlung der mehrfachen Ergebnisparameter, die in einer neuen Klasse zusammengefaßt werden, findet sich in Abschnitt 7.3.5.2. Dadurch hat eine Methode, die einen Funktionsbaustein abbildet, maximal einen Rückgabewert. Im Beispiel stellt **exportStruct** einen multiplen Ergebnisparameter dar.

Die Superklasse **SapFunctionLib** einer generierten Funktionsbibliothekklasse bietet Methoden für den Zugriff auf die Metadaten eines Funktionsbausteins an. Dadurch verfügt TYCOON-2 über alle aus dem R/3-System herausgelesenen Metadaten zur Laufzeit. Die für den Aufruf eines Funktionsbausteins und für die Konsistenzüberprüfung eines Parameters nötigen Metadaten (gespeichert in Objekten der Klasse **SapMethodMetadata**) werden zentral in der Metaklasse der generierten Funktionsbibliothek aufbewahrt und ermöglicht deren redundanzfreie Speicherung. Die Klasse **SapMethodMetadata** verwaltet die Informationen zu einem Funktionsbaustein, indem es mit Hilfe der Klassen **SapStructuredParamMetadata** und **SapSimpleParamMetadata** die zu einem Fuktionsbaustein zugehörigen strukturierten bzw. einfachen Parameterinformationen aggregiert. Auf die letztgenannten Klassen geht der nachfolgende Abschnitt genauer ein.

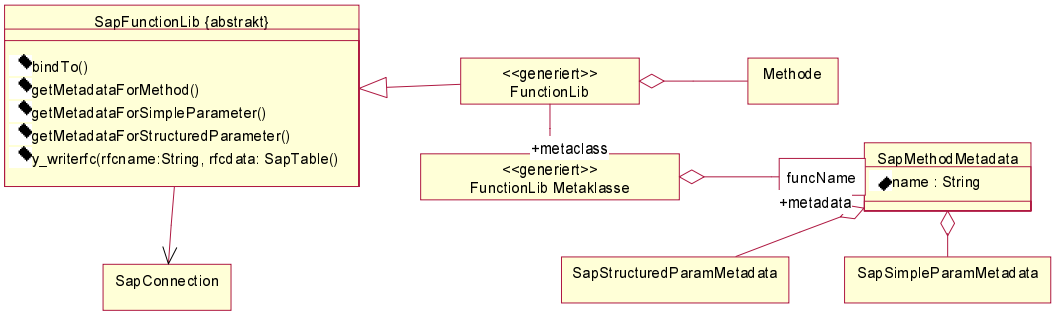

Abbildung 7.5: Die Klasse **SapFunctionLib** als Container der R/3-Funktionsbausteine

7.3.4 Metadaten zum Zugriff auf R/3

Ein Parameter wird auf dieser Ebene durch seinen Namen, seinen Typ und seine Länge beschrieben. Die Relevanz der Metadaten soll an dieser Stelle nochmals betont werden. Fehlerhafte Metadaten können zu Fehlern beim Aufruf des Funktionsbausteins führen. Dies gibt die Schnittstelle durch das Auslösen von Ausnahmen bekannt. Von größerer Tragweite ist es jedoch, wenn trotz fehlerhafter Metainformationen der R/3-Aufruf korrekt durchgeführt wird, die Daten vom R/3 aber falsch persistent gespeichert werden. Dieser Fehlertyp ist insbesondere bei Fehlern in Längen und Typen von Feldern innerhalb strukturierter Parameter möglich, da strukturierte Parameter als linearer Strom von Informationen übertragen werden und Fehler in diesem Strom unter Umständen nicht durch das R/3 erkannt werden. Deshalb ist es notwendig, bei der Zuweisung von Werten an Parameter diese auf die Einhaltung der Maximallängen hin zu überprüfen und die Informationen über Name und Typ des Parameters dem Anwendungsprogrammierer zur Verfügung zu stellen. Die ebenfalls notwendige Typprüfung wird durch statische Typisierung erreicht. Wie im vorherigen Abschnitt beschrieben, bietet die **SapFunctionLib** Methoden an, die den Zugriff auf die nötigen Informationen gewährleisten. Mit Hilfe dieser Informationen könnten z.B. automatisch Benutzerschnittstellen generiert werden, in denen bereits Längen und Typinformationen verarbeitet sind. Abbildung 7.6 illustriert die dafür benutzten Klassen.

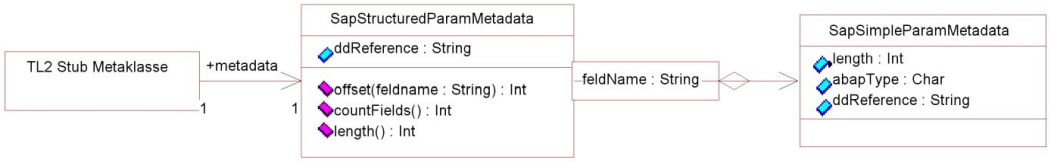

Abbildung 7.6: Metadaten für den Zugriff auf R/3

Die Klasse **SapSimpleParamMetadata** stellt genau die benötigten Informationen für einen einfachen Parameter zusammen, die wie oben beschrieben aus dem Namen, Typ und Länge des Parameters bestehen.

Eine Struktur oder Tabelle besteht aus mehreren einfachen Feldern. Die Metadaten dieser Felder aggregiert die Klasse **SapStructuredParamMetadata**. Auf die darin enthaltenen Informationen wird mit Hilfe des Feldnamens zugegriffen. Sie besitzt außerdem ein Attribut mit dem Namen der dargestellten Struktur, die dem im R/3 entspricht (**ddReference**). Metadaten für Strukturen und Tabellen müssen nicht differenziert behandelt werden, da sowohl Strukturen als auch Tabellen aus einfachen Feldern bestehen.

7.3.5 Automatische Generierung von Klassen

Die Generierung von Klassen für den Zugriff auf das System SAP R/3 löst mehrere Probleme: Neben der bereits schon erwähnten statischen Typisierung, die durch die Klassengenereirung ermöglicht wird, kann die Versionsproblematik beim externen Zugriff auf das R/3-System (siehe Abschnitt 7.3.5.1) durch speziell auf ein bestimmtes System abgestimmte Klassen vermieden werden. In Abschnitt 7.3.5.2 bzw. in Abschnitt 7.3.5.3 wird gezeigt, wie durch gene-

rierte Klassen die multiplen Ergebnisparameter eines Funktionsbausteins aggregiert werden. Zum Abschluß dieses Abschnittes wird der Vorgang des Generierens mit den dazugehörigen Quelltextvorlagen in allen Einzelheiten beschrieben.

7.3.5.1 Stubgenerierung und Versionsproblematik

Die SAP bietet keine wirklich stabile Schnittstelle für den Zugriff auf das R/3-System an. Trotz der Zusicherung der SAP, daß die Signaturen der Funktionbausteine langfristig stabil sind, zeigt sich in der Realität, daß durchaus Änderungen vorgenommen werden.

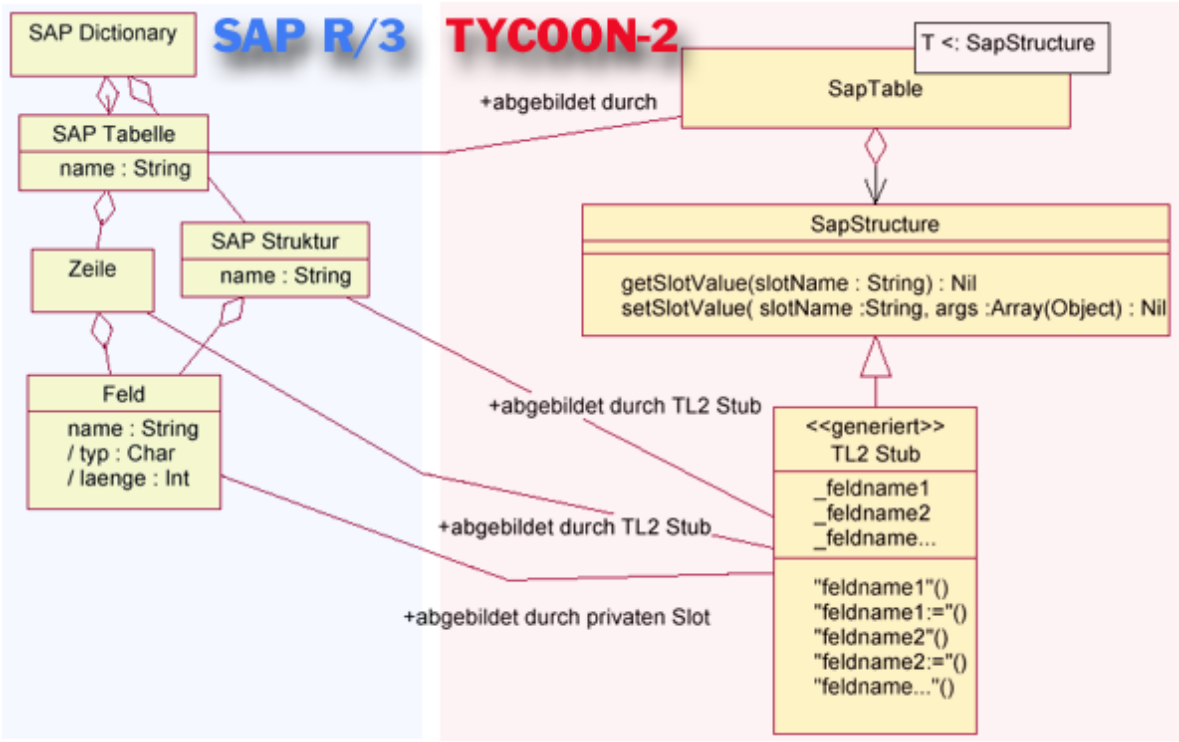

Abbildung 7.7: Abbildung der ABAP *Dictionary*-Strukturen auf Stubklassen

Neben den Änderungen von Funktionsbausteinnamen[9] und Parameternamen werden auch Feldnamen und Feldlängen in strukturierten Parametern[10] verändert. Änderungen in Funktionsbausteinnamen führen zu Fehlern der C-LIBRARY und erfordern eine Ausnahmebehandlung, das Gleiche gilt für die Umbenennung von Parameternamen. Schwerwiegender sind Änderungen von Feldnamen und Feldlängen bei strukturierten Parametern, da diese nicht immer zum Abruch führen, sondern unerkannt bleiben und inkonsistente Daten im R/3-System hinterlassen können. Es gibt also keine Garantie, daß nach einem R/3-Releasewechsel ein Funktionsbausteinaufruf durch eine externe Anwendung korrekt durchgeführt wird[11].

[9] So ist z.B. der Funktionsbaustein **RFC_DIRECTORY** (Release 3.0d) in Release 4 umbenannt in **RFC_FUNCTION_SEARCH**.

[10] z.B. bei ABAP Dictionary-Struktur BAPI1007_4 in Release 3.1i

[11] Bei R/3-internen Funktionsbausteinaufrufen ist dies nicht so kritisch, wenn auf die Parameter mit einer

Die SAP erreicht die versprochene Stabilität auch nicht durch eine dynamische Schnittstelle im Sinne von *BAPI Active X* (siehe Abschnitt 5.2). Hier wird die Schnittstelle vor jedem Aufruf mit dem R/3 neu verhandelt. Wenn sich aber nun die Namen von Parametern oder von Feldern in Strukturen ändern, stellt das auch keine adäquate Lösung dar, da in diesem Fall der Quelltext eines Programmes geändert werden muß. Vermieden werden dadurch nur Fehler, die aufgrund von Änderungen bei Feldlängen und Funktionsbausteinnamen auftreten können.

Ein dynamischen Aushandeln bei einer generierten Schnittstelle ist nur in Grenzen möglich, da Änderungen von Funktionsbausteinnamen im R/3-System nicht dokumentiert sind. Möglich wäre es, Feldnamen und -längen bei strukturierten Parametern mit dem R/3-System auszuhandeln. Bei erkannten Unstimmigkeiten kann die Anwendung zumindest entsprechend darauf reagieren: Die Anwendung kann entweder eine Ausnahme auslösen, falls ein Feldname geändert wurde oder intern ihre Metadaten auf den neuesten Stand bringen, wenn sich die Länge eines Feldes verändert hat. Vor einem Aufruf ist es dann notwendig, die Feldlänge eines Feldes zu überprüfen und gegebenenfalls eine Ausnahme auszulösen

Eine Generierung auf Basis eines Versionsstempels wäre daher die beste Lösung. Sie besteht darin, durch einen Versionsvergleich vor einem Aufruf die jeweils passenden, für ein bestimmtes System generierten, Parameterstrukturen auszuwählen. Dies scheitert daran, daß *ABAP-Dictionary*-Strukturen einen solchen Versionsstempel nicht haben. Eine alternative Möglichkeit wäre der Vergleich auf Basis von R/3-Releaseständen. Durch die anstehende Komponentisierung des R/3 würde auch solch ein Versionsstempel nicht die Sicherheit der Schnittstelle erhöhen können, da es dann durchaus Module mit verschiedenen Releaseständen geben kann. Eine Generierung auf Ebene von R/3-Versionsständen ist damit auch nicht ausreichend.

Die von uns gewählte Lösung abstrahiert von den verschiedenen Releaseständen und geht davon aus, ein einziges System über die Schnittstelle anzubinden. Es werden also Stubklassen für Funktionsbausteine und Parameter aus genau einem System generiert (siehe Abbildung 7.7). Dadurch ist sichergestellt, daß die Anbindung fehlerfrei funktioniert. Der Zugriff auf genau ein System ist nicht befriedigend, sei es, daß global agierende Unternehmen R/3-Systeme mit unterschiedlichen Releaseständen installiert haben, oder auch nur, daß ein neues Release eingesetzt wird. In beiden Fällen kann ein Zugriff mit Stubklassen, die für ein System generiert sind, fehlschlagen. Hier besteht die Möglichkeit, die benötigten Klassen neu zu generieren.

7.3.5.2 Multiple Ergebnisparameter

Die Exportparameter eines Funktionsbausteins stellen die Ergebnisparameter dar. Funktionsbausteine können nach ihrer Definition (siehe Kapitel 2.5.2.2) multiple Ergebnisparameter haben, da die Anzahl der Exportparameter nicht festgelegt ist. Ergebnisparameter sind entweder einfache Parameter oder strukturierte Parameter.

Bei den BAPIs tritt dies sehr häufig auf, da diese generell als Exportparameter die Struktur **BAPI_RETURN** zurückliefern, die einen Status und eine zugehörige Beschreibung enthält. Schon ein zusätzlicher Rückgabewert bedeutet, daß ein BAPI multiple Ergebnisparameter hat.

Feldsemantik zugegriffen wird.

Die multiplen Ergebnisparameter eines Funktionsbausteins erschweren die Benutzbarkeit der Schnittstelle, da der Benutzer sämtliche Ergebnissparameter eines Funktionsbausteins vor dem Aufruf erzeugen und diesem übergeben muß. Bei der TYCOON-2-SAP Schnittstelle soll dies durch Abstraktion von den multiplen Ergebnisparametern vermieden werden:

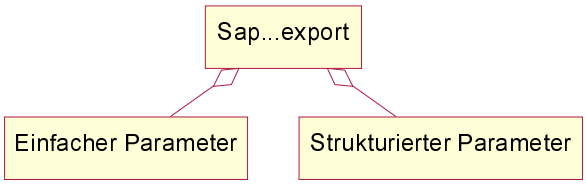

Abbildung 7.8: Behandlung multipler Ergebnisparameter

In TYCOON-2 werden Funktionen zur Verfügung gestellt, die maximal einen Rückgabewert besitzen. Um dies zu realisieren, werden bei der Generierung von Stubklassen multiple Ergebnisparameter eines Funktionsbausteins in einer neuen Klasse aggregiert[12] (siehe Abbildung 7.8). Diese generierte Klasse wird als einziger Rückgabetyp eines in TYCOON-2 aufrufbaren Funktionsbausteins zurückgegeben. Die Erzeugung der Ergebnisparameter wird durch die Schnittstelle übernommen. Als Beispiel wird der bereits in Abschnitt 7.3.3 beschriebene Funktionsbaustein benutzt:

```
bapi_customer_search(pi_salesorg:String,
                     pi_search_flag:Char, pi_address:SapBAPIKNA101,
                     multiple:SapTable(SapBAPIKNA103)
                    ):SapBAPI_CUSTOMER_SEARCHexport
```

Als Rückgabewert erhält das aufrufende Programm ein Exemplar der Klasse **SapBAPI_CUSTOMER_SEARCHexport**. Die Klasse hat einen *Slot* vom Typ **SapBAPI_RETURN** und einen *Slot* vom Typ **String**.

7.3.5.3 Der Stubgenerator

Der Stubgenerator ermöglicht den Übergang von dynamischer zur statischen Typisierung, da Quelltext generiert wird, der die Transformation zwischen den statisch getypten Objekten und den dynamisch getypten Transportobjekten vornimmt. Er benutzt die bisher beschriebenen Klassen und erleichtert die Anbindung verschiedener R/3-Systeme. Er bezieht alle Informationen für die Erzeugung der Klassen aus dem R/3-System und trägt damit zur Vermeidung von Fehlern bei, da diese sonst vom Benutzer herauszulesen sind.

Generiert werden Klassen für die strukturierten Parameter, Klassen für die multiplen Ergebnisparameter und für eine Funktionsbibliothek. Die R/3-Funktionsbibliotek enthält Tausende von Funktionsbausteinen, von denen üblicherweise nur eine kleine Teilmenge benutzt wird. Außerdem kann das benutzte TYCOON-2-System nur eine bestimmte Menge von Klassen verwalten, so daß es erforderlich ist, die Anzahl der erzeugten Klassen durch eine Auswahl von Funktionsbausteinen zu begrenzen. Mit dem Generator wird ein Werkzeug

[12] Der Zugriff auf einen Exportparameter erfolgt über einen *Slot*, der den Namen des Exportparameters trägt.

zur Verfügung gestellt, mit dem sich aus allen entfernt aufrufbaren Funktionsbausteinen die benötigten Funktionsbausteine auswählen lassen. Das Werkzeug nutzt den Funktionsbaustein **RFC_FUNCTION_SEARCH**, um eine Liste aller entfernt aufrufbaren Funktionsbausteine nach Angabe eines Suchparameters für eine Benutzerauswahl präsentieren zu können.

Die für die Generierung von TYCOON-2-Quelltext erforderlichen Informationen werden durch den Aufruf des im Rahmen dieser Arbeit entwickelten Funktionsbausteins **Y_WRITERFC** im zu rufenden R/3-System bereitgestellt. Folgender Auschnitt aus dem zugehörigen ABAP-Programm illustriert die Schnittstelle des Funktionsbausteins:

```
FUNCTION Y_WRITERFC.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"       IMPORTING
*"             VALUE(RFCNAME) LIKE  TFDIR-FUNCNAME OPTIONAL
*"       TABLES
*"              RFCDATA STRUCTURE  YTYCSAPGWY
*"----------------------------------------------------------------------
...
```

Dir Funktionsbaustein **Y_WRITERFC** erwartet als Eingabe den Namen des zu rufenden Funktionsbausteins und erzeugt daraufhin eine Tabelle, in der sämtliche benötigten Informationen zu den Parametern dieses Funktionsbausteins enthalten sind. Dieses sind v.a. die Informationen zu Typ und Länge eines Parameters. Er stützt sich dabei wiederum auf die Funktionsbausteine **RFC_GET_FUNCTION_INTERFACE** und **RFC_GET_STRUCTURE_DEFINITION** ab (siehe auch Kapitel 5):

RFC_GET_FUNCTION_INTERFACE Dieser Funktionsbaustein liefert die Schnittstellenbeschreibung eines Funktionsbausteins, die aus den Namen, Längen und Typen der Parameter besteht.

RFC_GET_STRUCTURE_DEFINITION Der Funktionsbaustein ermöglicht das Lesen von Feldinformationen zu einer im ABAP *Dictionary* definierten Struktur oder Tabelle.

Die Tabelle, die der Funktionsbaustein **Y_WRITERFC** zurückliefert, wird in TYCOON-2 ausgewertet. Mit Hilfe der für den Generator erstellten Quelltextvorlagen, die nach einzelnen Textblöcken unterteilt sind, werden die Klassen generiert. Die Anzahl der so erzeugten Klassen variiert je nach Art des Funktionsbausteins und ist abhängig von der Anzahl der im Funktionsbaustein benutzten Strukturen. Für jede Struktur wird eine Stubklasse in TYCOON-2 erzeugt.

Es existieren insgesamt vier Vorlagen (siehe Anhang B), wobei jeweils zwei einer Klasse und der zugehörigen Metaklasse zuzuordnen sind. Wie schon in den vorherigen Abschnitten beschrieben, wird aus den Quelltextvorlagen zum einen eine Funktionsbibliothekklasse erstellt, zum anderen werden für die strukturierten Parameter Stubklassen erzeugt. Die Quelltextvorlage für die Stubklassen kann außerdem für die Erzeugung der Klassen für die multiplen Ergebnisparameter benutzt werden.

Funktionsbausteine als Methoden einer Funktionsbibliothekklasse: Auf Basis der Vorlagen **SapFunctionLibrary** und **SapFunctionLibraryClass** wird eine Funktionsbibliothek erzeugt, in die Methodensignaturen und Methodenkörper eingefügt werden. Innerhalb eines solchen Methodenkörpers, der genau einen Funktionsbausteinaufruf enthält, sind folgende Einzelschritte für einen Aufruf durchzuführen (eine gültige Verbindung zum R/3-Applikationsserver wird dabei vorausgesetzt):

1. Aufbau der einfachen Parameterstrukturen
2. Aufbau der Tabellenparameter über die Tabellen-API
3. Aufbau der Strukturparameter über die Struktur-API
4. Aufruf des Funktionsbausteins mit den erzeugten Parametern
5. Überprüfen des Aufrufergebnisses auf Ausnahmen
6. Auslesen der Ergebnisparameter und der Tabellenparameter

In der Metaklasse der auf Basis der Quelltextvorlage **SapFunctionLibrary** generierten Klasse werden die Metadaten für den Funktionsbausteinaufruf zusammengefaßt. Für jede Methode (R/3 Funktionsbaustein) wird ein Objekt von der Klasse **SapMethodMetadata** erzeugt. Im folgenden wird die Zuordnung von generierten Quelltext zu den jeweils zuständigen Vorlagen vorgenommen:

SapFunctionLibrary Anhand der aus dem SAP R/3-System herausgelesenen Typinformationen kann bei der Generierung die in Tabelle 7.1 beschriebene Typabbildung umgesetzt werden, so daß als erstes die Schnittstelle einer Methode erzeugt wird:

```
bapi_customer_getdetail(
          customerno:String, pi_distr_chan:String, pi_division:String,
          pi_salesorg:String):SapBAPI_CUSTOMER_GETDETAILexport
```

Im Rückgabeparameter werden die multiplen Ergebniswerte zusammengefaßt. Darauf werden die Metadaten des Funktionsbausteins aus der Metaklasse für die weitere Generierung gelesen und ein für den Aufruf benötigtes Exemplar der Klasse **SapFunction** erzeugt:

```
...
let functionMetadata=metadata["BAPI_CUSTOMER_GETDETAIL"],
let simpleMetadata=functionMetadata.simpleParam,
let structuredMetadata=functionMetadata.structuredParam,
let sf=SapFunction.new(functionMetadata.name,_connection),
...
```

Mit den Metadaten können die vier einfachen Importparameter des Funktionsbausteins erzeugt, die Aktualparameter zugewiesen und der **SapFunction** zugeordnet werden. Bei der Zuweisung der Aktualparamter findet in der Klasse **SapSimpleParam** die erweiterte Konsistenzprüfung statt:

```
...
importParam:=
simpleMetadata["CUSTOMERNO"].asSapSimpleParam.value:=customerno,
importParam.name:="CUSTOMERNO",
sf.importList.add(importParam),
importParam:=
simpleMetadata["PI_DISTR_CHAN"].asSapSimpleParam.value:=pi_distr_chan,
importParam.name:="PI_DISTR_CHAN",
sf.importList.add(importParam),
importParam:=
simpleMetadata["PI_DIVISION"].asSapSimpleParam.value:=pi_division,
importParam.name:="PI_DIVISION",
sf.importList.add(importParam),
importParam:=
simpleMetadata["PI_SALESORG"].asSapSimpleParam.value:=pi_salesorg,
importParam.name:="PI_SALESORG",
sf.importList.add(importParam), ...
```

Dasselbe gilt für die Exportparameter, die erzeugt und dem Objekt der Klasse **SapFunction** zugeordnet werden:

```
...
export:= SapBAPI_CUSTOMER_GETDETAILexport.new,
export.PE_ADDRESS := SapBAPIKNA101.new,
sf.structList.add(SapStructuredParam.new(export.PE_ADDRESS,
                                         SapBAPIKNA101,
                                         "PE_ADDRESS",'e')),
export.RETURN := SapBAPIRETURN.new,
sf.structList.add(SapStructuredParam.new(export.RETURN,
                                         SapBAPIRETURN,
                                         "RETURN",'e')), ...
```

Im letzten Schritt registriert sich die Funktion bei der Verbindung zum R/3-Applikationsserver und ein synchroner Aufruf wird durchgeführt. Nachdem der Aufruf erfolgt ist, werden die Exportparameter als Ergebnis des Aufrufs zurückgegeben:

```
...
sf.register, (* einmalige Registrierung bei einer Verbindung*)
sf.callReceive,
export} (* SapBAPI_CUSTOMER_GETDETAILexport*)
```

SapFunctionLibraryClass In der Metaklasse der Funktionsbibliothek befinden sich die Parameterinformationen. Sie aggregiert dazu die für alle erzeugten Funktionsbausteine nötigen Metainformationen in einem **Dictionary**-Objekt von der Klasse **SapMethodMetadata**. Die abstrakte Oberklasse **SapFunctionLib** bietet dem Benutzer Methoden für den Zugriff auf diese Metadaten an. Dies wird unter anderem dazu benutzt, innerhalb der generierten Funktionsbibliothek auf die Metadaten zuzugreifen und diese für den Aufbau der nötigen formalen Parameterstrukuren zu verwenden (siehe oben):

```
...
let m=SapMethodMetadata.new,
m.name:="BAPI_CUSTOMER_GETDETAIL",
m.structuredParam["PE_ADDRESS"]:=SapBAPIKNA101.metadata,
m.structuredParam["RETURN"]:=SapBAPIRETURN.metadata,
m.simpleParam["CUSTOMERNO"]:=
SapSimpleParamMetadata.new( "CUSTOMERNO" ,10,'I','S'),
(* Importparameter vom Typ String der Länge 10 *)
m.simpleParam["PI_DISTR_CHAN"]:=
SapSimpleParamMetadata.new( "PI_DISTR_CHAN" ,2,'I','S'),
m.simpleParam["PI_DIVISION"]:=
SapSimpleParamMetadata.new( "PI_DIVISION" ,2,'I','S'),
m.simpleParam["PI_SALESORG"]:=
SapSimpleParamMetadata.new( "PI_SALESORG" ,4,'I','S'),
m.exportStructuredParam:=SapBAPI_CUSTOMER_GETDETAILexport.metadata,
...
```

Strukturen und multiple Ergebnisparameter: Neben der Generierung der Funktionsbibliothek sind Klassen für die strukturierten Parameter und Klassen für die Behandlung der multiplen Ergebnisparameter zu erzeugen. Für jede Struktur, wobei Tabellen auch als Strukturen behandelt werden (siehe Abschnitt 7.2.2.4), wird eine Klasse **SapStructure** mit zugehöriger Metaklasse **SapStructureClass** angelegt. In der Metaklasse zu einer Struktur befinden sich die Metainformationen der einzelnen Felder.

SapStructure Bis auf eine Methode für die Ausgabe der Struktur auf den Bildschirm werden mit dieser Quelltextvorlage Klassen für Strukturen generiert, deren Felder private *Slots* sind. Zu jedem solchen privaten *Slot* werden Schreib- und Lesemethoden erzeugt. Die Lesemethoden geben unverändert direkt den Inhalt des geschützten *Slots* wieder. Bei den schreibenden Zugriffen findet eine Überprüfung anhand der in der Metaklasse enthaltenen Informationen statt. Erfüllt der übergebene Wert nicht die darin enthaltenen Bedingungen wird eine Ausnahme ausgelöst (siehe Abschnitt 7.3.7). Folgendes Beispiel zeigt die generierte Methode für den schreibenden Zugriff auf einen *Slot*, wobei vor der Wertzuweisung die erweiterte Konsistenzprüfung stattfindet:

```
...
"FIRST_NAME:="( value:String)
{
checkMetadataConstraintsForParam("FIRST_NAME",value),
_FIRST_NAME:=value
}
...
```

SapStructureClass In der durch die Vorlage **SapStructureClass** erzeugten Klassen werden die Metadaten der einzelnen Felder einer Struktur gespeichert, die als *Slots* einer konkreten **SapStructure** abgebildet sind. Für jeden *Slot* der Klasse wird in der Metaklasse ein Objekt der Klasse **SimpleParamMetadata** erzeugt und unter dem Namen des *Slots* in einem **Dictionary**-Objekt abgelegt:

```
simpleDict["FIRST_NAME"]:=
SapSimpleParamMetadata.new( "FIRST_NAME",35,'T' ,'S'),
(* String der Länge 35 *)
...
simpleDict["DATE_BIRTH"]:=
       SapSimpleParamMetadata.new("DATE_BIRTH",8,'T' ,'D'),
...
```

Bei der Erzeugung von Klassen für die Behandlung von multiplen Ergebnisparametern kann es vorkommen, daß diese Klasse wieder Strukturen aggregiert (siehe Abschnitt 7.3.5.2). Für diese Strukturen existieren bereits generierte Klassen, d.h. die dazugehörigen Metadaten werden nicht noch einmal erzeugt.

7.3.6 Geschäftsobjekte

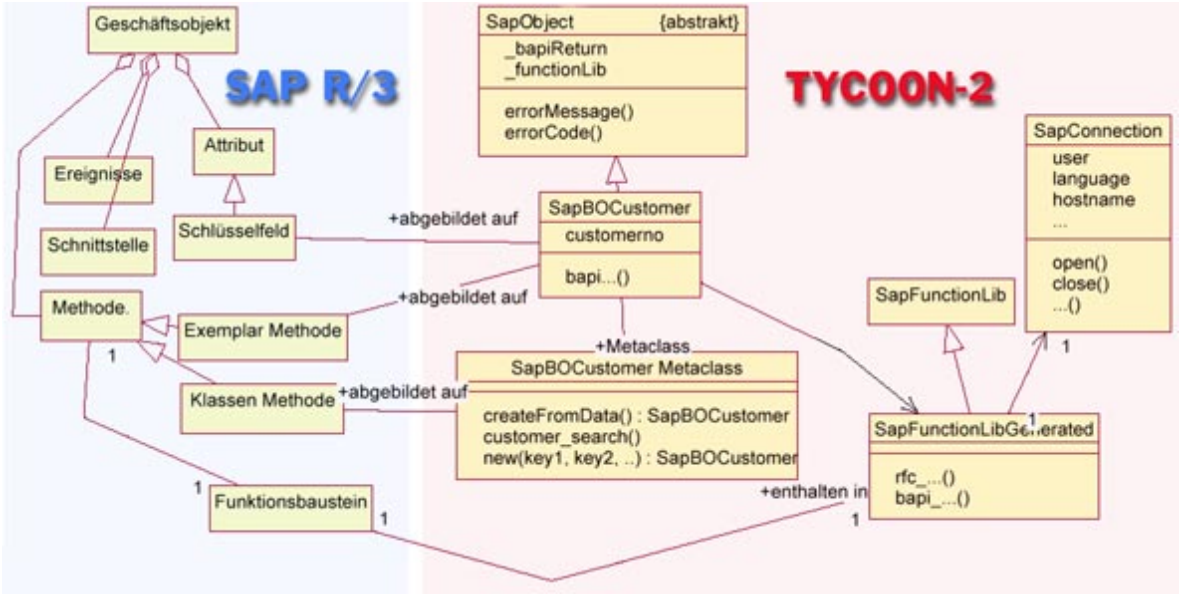

Abbildung 7.9: Abbildung der R/3 Geschäftsobjekte nach TYCOON-2

Die R/3-Geschäftsobjekte stellen das Ziel der objektorientierten R/3-Integration dieser Arbeit dar. Dazu sind die R/3-Geschäftsobjekttypen durch entsprechende Stubklassen im TYCOON-2-System repräsentiert. Ein TYCOON-2-Geschäftsobjekt stellt ein Verweis auf ein im R/3-System existierendes Geschäftsobjekt dar. Dieses kann bereits vor der Erzeugung des TYCOON-2-Objekts existieren, aber auch erst durch die Erzeugung des TYCOON-2-Objekts im R/3 angelegt werden. Ein existierendes R/3-Geschäftsobjekt wird also durch genau ein TYCOON-2-Objekt abgebildet und umgekehrt.

Ein R/3-Geschäftsobjekttyp besteht gemäß der Definition im *Business Object Repository* aus Grunddaten, Schnittstellen, Schlüsselfeldern, Attributen, Ereignissen und Methoden (siehe Abschnitt 4.2). Eine TYCOON-2-Stubklasse bildet einen R/3-Geschäftsobjekttyp ab. Durch

die Stubklasse und ihre Metaklasse werden die Konzepte von Grunddaten, Schlüsselfeldern und Methoden modelliert. Schnittstellen, Attribute und Ereignisse sind nicht modelliert, da auf diese Informationen durch die Beschränkungen des R/3-Objektmodells nicht zugegriffen werden kann.

Schlüsselfelder:
Schlüsselfelder identifizieren ein R/3-Geschäftsobjekt. Gemäß der relationalen Semantik sind sie als Schlüssel zu betrachten, die einen Datensatz innerhalb des R/3-Datenmodells identifizieren.
Schlüsselfelder sind als *Slots* der Stubklasse modelliert. Die Schlüsselfelder sind bei der Erzeugung des TYCOON-2-Objekts relevant und vor dem schreibenden Zugriff geschützt, wenn sie ein existierendes R/3-Geschäftsobjekt identifizieren. Auf die Details der Objekterzeugung wird weiter unten eingegangen.

Methoden (BAPIs):
Über den Aufruf von Methoden (BAPIs) läßt sich der Objektzustand eines R/3-Geschäftsobjektes ändern. Dies ist die einzige Möglichkeit, den Objektzustand zu verändern. Über die sukzessive Definition und Implementierung von BAPIs stellt die SAP die Funktionalität eines Geschäftsobjekts externen Anwendungen zur Verfügung.
Die BAPIs des R/3-Objektmodells entsprechen den Methoden der Stubklasse. Wie bereits erwähnt sind BAPIs durch entfernt aufrufbare Funktionsbausteine modelliert. Funktionsbausteine, die exemplarabhängige BAPIs implementieren, erfordern in ihrer Signatur die Angabe der Schlüsselfelder des R/3-Geschäftsobjekts, welches sie bearbeiten sollen. Die Einbettung der BAPI-Funktionsbausteine in eine Geschäftsobjektstubklasse ermöglicht eine Abstraktion von dieser funktionsorientierten Sichtweise. Die Schlüsselfelder werden transparent für den Schnittstelllenbenutzer an den Funktionsbaustein übergeben. Das folgende Beispiel zeigt den Übergang von der funktionsorientierten Sichtweise der Klasse **SapFunctionLib** zur objektorientierten Sichtweise einer Klasse **SapBOCustomer**, welche den R/3-Geschäftsobjekttyp Kunde im TYCOON-2-System darstellt, anhand der Signaturen der jeweiligen Objektmethode:

```
(* BAPI-Methode Customer.getDetail *)
(* als Methode einer generierten Funktionsbibilthek *)
bapi_customer_getdetail(customerno:String, pi_salesorg:String,
                        pi_distr_chan:String, pi_division:String )
                        :SapBAPI_CUSTOMER_GETDETAILexport

(* als Methode der Geschäftsobjektklasse SapBOCustomer *)
customer_getdetail():SapBAPI_CUSTOMER_GETDETAILexport
```

Klassenmethoden wie **CheckExistence**, **GetList** oder **CreateFromData** sind als Methoden der Metaklasse abgebildet. Eine der Aufgaben einer Metaklasse im TYCOON-2-System ist die Objekterzeugung, auf die im folgenden eingegangen werden soll.

Identität und Objekterzeugung:
Das R/3-System formuliert mit der Identifikation eines Geschäftsobjekts über dessen Schlüsselfelder einen gänzlich anderen Weg als objektorientierte Systeme. Da ein R/3-Geschäftsobjekt mit (mindestens) einem Datensatz einer Tabelle im R/3 korrespondiert, ist ein R/3-Ge-

schäftsobjekt immer persistent. In objektorientierten Systemen wird die Identität durch eine vom Laufzeitsystem vergebene eindeutige OID (*Object Identifier*) definiert. Ein TYCOON-2-Stubobjekt referenziert[13] über seine *Slots*, die den R/3-Schlüsselfeldern entsprechen, ein R/3-Geschäftsobjekt. Bei der Objekterzeugung sind die Schlüsselfelder als Parameter dem Konstruktor zu übergeben. Es existieren in der Metaklasse der Stubklasse zwei Konstruktoren:

1. Der Konstruktor **CREATE_FROM_DATA** erzeugt ein R/3-Geschäftsobjekt, d.h. es wird ein neues Geschäftsobjekt angelegt, welches vorher im R/3 nicht vorhanden war. Schlägt die Objekterzeugung im R/3 fehl, wird eine Ausnahme ausgelöst. Die Schlüsselfelder des neu erzeugten R/3-Geschäftsobjekts müssen nicht zwingend durch das R/3-System vorgegeben werden. Aufgrund der weitreichenden Einstellungen im *Customizing* ist es möglich, die Schlüsselfelder durch die externe Anwendung vorgeben zu lassen. Dies bedeutet wiederum für die Signatur der Methode **CREATE_FROM_DATA**, daß die Schlüsselfelder in Form von Importparametern definiert sein müssen. Der BAPI **Customer.CreateFromData** ist ein Beispiel für ein BAPI, in dem dies nicht erfüllt ist. Für die Nutzung dieses BAPIs muß die Vergabe der Schlüsselfelder zwingend durch das R/3 vorgenommen werden. Diese Tatsache macht bewußt, daß die BAPIs der SAP oftmals nur für eng umrissende Szenarien zu gebrauchen sind[14]. In der Praxis ist es oftmals notwendig, BAPIs zu kopieren und gemäß den eigenen Anforderungen zu modifizieren.

2. Der Konstruktor **NEW** prüft anhand der übergebenen Schlüsselfelder die Existenz des R/3-Geschäftsobjekts und erzeugt bei Vorhandensein des Objektes in SAP R/3 ein entsprechendes TYCOON-2-Geschäftsobjekt. Ist das Objekt nicht im R/3-System vorhanden, wird eine Ausnahme ausgelöst.

Mit dieser Konstruktion erhält die Schnittstelle die Eigenschaft, daß ein TYCOON-2-Geschäftsobjekt einem R/3-Geschäftsobjekt gegenübersteht. Dies ist die zwingende Voraussetzung für den Aufruf von Objektmethoden (BAPIs) an diesem Objekt. Diese Objekte besitzen den Status **valid**. Die Schlüsselfelder eines gültigen TYCOON-2-Geschäftsobjekt lassen sich nicht mehr verändern. Eine Veränderung würde die Beziehung zum referenzierten R/3-Geschäftsobjekt zerstören. TYCOON-2-Geschäftsobjekte mit dem Status **invalid** können durch die Löschung von Geschäftsobjekten im R/3 entstehen. Eine Freigabe des Geschäftsobjekts im TYCOON-System[15] ist unproblematisch und löst keine Aktion der Schnittstelle aus. Ein Problem ergibt sich bei der Löschung eines R/3-Geschäftsobjekts, da dieses Objekt im TYCOON-2-System weiterhin als gültig betrachtet wird. Der Fehler wird erst beim versuchten Aufruf einer Objektmethode erkannt. Dabei wird das Objekt als ungültig gekennzeichnet und eine Ausnahme ausgelöst. Es ist vorstellbar, daß die Metaklasse eines TYCOON-2-Geschäftsobjekts einen Dienst anbietet, der zu definierten Zeitpunkten[16] alle vorhandenen TYCOON-

[13] Der Begriff Referenzierung ist an dieser Stelle konteptuell zu verstehen. Es existiert keine Referenzierung im Sinne eines Zeigers einer Programmiersprache.

[14] Die angesprochenen Szenarien werden durch die *Internet Application Components (IAC)* definiert. IACs sind R/3-Anwendungen, die über das Internet ausführbar sind. Dabei wird ausgiebig von den BAPIs Gebrauch gemacht.

[15] z.B. durch Gleichsetzung mit **nil** oder durch eine Zuweisung

[16] Denkbare Zeitpunkte wären beispielsweise das Hochfahren des TYCOON-2-Systems oder das Speichern des Systemzustands (**tycoon.saveSystem**).

2-Geschäftsobjekte auf ihr Vorhandensein im R/3 überprüft und entsprechend den Zustand der Objekte setzt. Ungültige Objekte könnten in einer Poolvariablen zur weiteren Bearbeitung durch das TYCOON-2-Anwendungsprogramm, welches die R/3-Schnittstelle benutzt, abgelegt werden. Im jetzigen Entwicklungsstand der Schnittstelle wird bei der Objekterstellung nicht überprüft, ob bereits ein TYCOON-2-Geschäftsobjekt existiert, welches dasselbe R/3-Geschäftsobjekt referenziert.

TYCOON-2-Geschäftsobjekte und Funktionsbibliothek:
TYCOON-2-Geschäftsobjekte erhalten ihre Funktionalität aus dem Aufruf von BAPI-Methoden. BAPI-Methoden sind als Methoden einer generierten Klasse **SapFunctionLib** realisiert, welche die R/3-Funktionsbibliothek abbildet. Aus diesem Grund muß ein TYCOON-2-Geschäftsobjekt an ein **SapFunctionLib-Objekt** gebunden werden und dieses wiederum an eine geöffnete R/3-Verbindung, auf der die BAPI-Aufrufe ausgeführt werden. Folgende Implementierung der BAPI-Methode **Customer.getDetail** der Klasse **SapBOCustomer** zeigt die Beziehung zwischen einer Geschäftsobjektstubklasse und einer Funktionsbibliothek. Das Objekt **_functionLib** stellt die erwähnte Funktionsbibliothek dar; die Parameter des BAPI-Aufrufs (**customerno** usw.) sind *Slots* der Stubklasse.

```
getdetail():SapBAPI_CUSTOMER_GETDETAILexport
{
let export = _functionLib.bapi_customer_getdetail
       ( customerno, pi_distr_chan, pi_division, pi_salesorg),
_bapiReturn := export.RETURN,
export
}
```

Generierung von TYCOON-2-Stubklassen:
R/3-Geschäftsobjekttypen werden im BOR definiert. Die Definitionen lassen sich wie die Signaturen von Funkionsbausteinen durch externe Anwendungen auslesen (siehe Abschnitt 4.4). Die Generierung von Stubklassen für R/3-Geschäftsobjekte wäre somit möglich. In der Praxis verhindern jedoch die Ungenauigkeiten im BOR eine automatische Generierung. Dabei sind vor allem fehlende Definitionen von Schlüsselfeldern als auch Probleme bei der Zuordnung der Schlüsselfelder eines R/3-Geschäftsobjektes auf die Parameter des implementierenden Funktionsbausteins zu nennen. Als konkretes Beispiel kann hier das Geschäftsobjekt Kunde angeführt werden: Gemäß der Definition im BOR wird es durch das Feld **CUSTOMERNO** identifiziert. Wenn man den oben dargestellten Funktionsbaustein **Customer.GetDetail** betrachtet, wird offensichtlich, daß für diesen BAPI-Aufruf zusätzlich die Felder **PI_SALESORG**, **PI_DISTR_CHAN** und **PI_DIVISION** zur Identifizierung des Kunden benötigt werden. Der BAPI **Customer.GetSalesAreas** wiederum benötigt tatsächlich nur den Teilschlüssel **CUSTOMERNO** gemäß der BOR-Definition. Die Signaturen der BAPI-Funktionsbausteine orientieren sich also nicht an der fachlichen Definition im BOR, sondern an den Schlüsselfeldern der Tabellen, die für den jeweiligen BAPI gerade benötigt werden.

Die Grafik 7.9 zeigt die Abbildung des R/3-Geschäftsobjektmodells auf Geschäftsobjektstubklassen in TYCOON-2 am Beispiel der Klasse **SapBOCustomer**. Alle Stubklassen erben von der gemeinsamen Klasse **SapObject**, welche vor allem für die Ausnahmebehandlung

zuständig ist. Das Konzept der Ausnahmebehandlung wird im folgendem Abschnitt beschrieben.

7.3.7 Ausnahmebehandlung

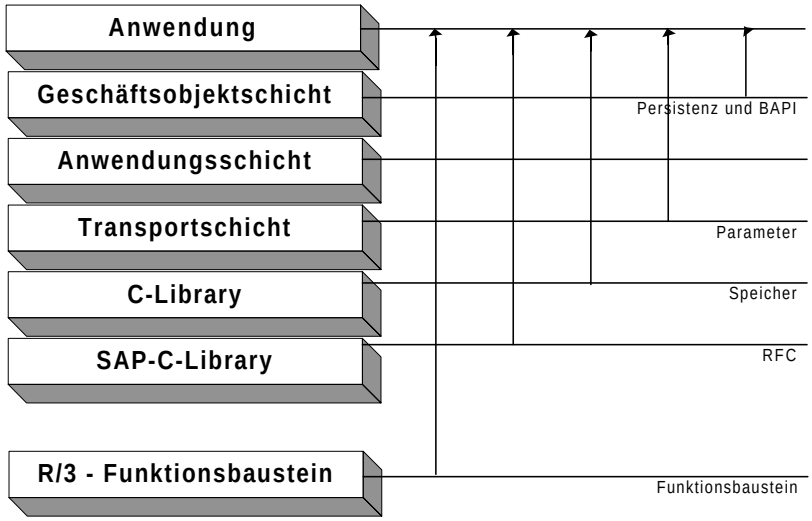

Abbildung 7.10: Ausnahmebehandlung in den einzelnen Schichten

Ausnahmen stellen ein Konstrukt objektorientierter Sprachen zur Behandlung unerwarteter Programmsituationen dar. Den Entwickler entlasten sie insofern, daß nicht der Erfolg jeder Programmaktion einzeln geprüft werden muß. ABAP kennt ebenfalls ein Ausnahmenkonzept. Es ist jedoch auf Funktionsbausteine begrenzt und nicht in herkömlichen Programmen benutzbar. Das ABAP-Ausnahmenkonzept ist jedoch nicht mit Ausnahmen objektorientierter Sprachen vergleichbar. Ausnahmen werden in der Schnittstelle des Funktionsbausteins definiert und durch das Schlüsselwort **RAISE** im Funktionsbaustein ausgelöst und dem rufendem Programm mitgeteilt. Dieses muß jedoch nach jedem Aufruf manuell auf das Eintreten einer Bedingung testen. Dieses Ausnahmenkonzept wurde vom Ansatz in das BAPI-Programmiermodell übernommen.

Jeder BAPI besitzt als Bestandteil seiner Schnittstelle einen strukturierten Exportparameter vom Typ **BAPIRETURN** (siehe Abbildung 7.11), der den Erfolg oder Mißerfolg eines BAPI-Aufrufs darstellt. Meldungen des Typs E (*Error*) und A (*Abort*) signalisieren einen Mißerfolg des BAPI-Aufrufs. Die weiteren Felder der **BAPIRETURN**-Struktur enthalten eine Fehlernummer sowie erläuternden Text. Nach einem erfolgtem BAPI-Aufruf werden Meldungen des Typs E und A auf TYCOON-2-Ausnahmen umgesetzt. Dies wird durch die Klasse **SapObject** realisiert. Die Klasse **SapObject** definiert Zugriffsmethoden für die Felder der **BAPIRETURN**-Struktur.

Ausnahmen werden der TYCOON-2-Anwendung über Objekte der Klasse **SapError** mitgeteilt. Abbildung 7.10 zeigt die Zuordnung der Ausnahmetypen zu ihrem Ursprung, der jeweiligen Schicht der Schnittstelle bzw. dem SAP-Applikationsserver. Die Ausnahmen lassen

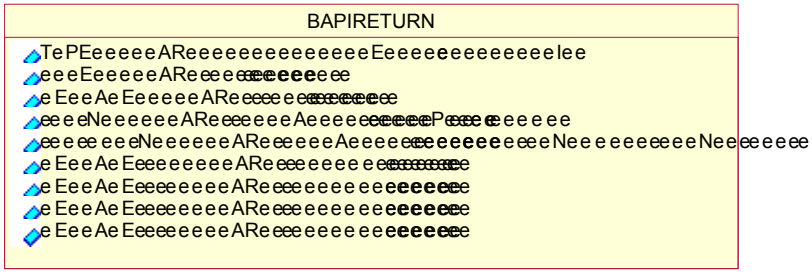

Abbildung 7.11: BAPI-Ausnahmebehandlung: ABAP *Dictionary*-Struktur **BAPIRETURN**

sich dabei wie folgt klassifizieren:

Persistenz Auf einem ungültigen TYCOON-2-Geschäftsobjekt wurde eine Objektmethode aufgerufen. Ein TYCOON-2-Geschäftsobjekt ist ungültig, wenn das zugehörige R/3-Geschäftsobjekt gelöscht wurde.

BAPI Wie am Anfang dieses Abschnitts beschrieben, signalisieren BAPIs Ausnahmesituationen über die **BAPIRETURN**-Struktur. Die Ausnahme **BAPI** wird nach der mißglückten Ausführung einer Objektmethode ausgelöst (Meldungstyp A oder E in Parameter **BAPIRETURN**).

Parameter Dieser Ausnahmentyp wird beim Fehlschlagen der erweiterten Konsistenzprüfung auf Funktionsbausteinparametern ausgelöst[17].

RFC Diese Klasse von Ausnahmen signalisieren Probleme im System oder im Netzwerk. Sie werden durch die SAP-C-Library erkannt und an das TYCOON-2-System weitergereicht. Die beiden vorhandenen Ausprägungen dieses Typs sind die im Abschnitt 2.5.2.2 angesprochenen Ausnahmen **SYSTEM_FAILURE** und **COMMUNICATION_FAILURE**.

Funktionsbaustein Ausnahmen dieser Klasse werden im R/3-Funktionsbaustein (Schlüsselwort **RAISE**) ausgelöst. Im Gegensatz zu den BAPI-Funktionsbausteinen, die eine Ausnahme über den Parameter **BAPIRETURN** signalisieren, ist dies der von konventionellen Funktionsbausteinen beschrittene Weg. In BAPI-Funktionsbausteinen ist solch eine Auslösung von Ausnahmen untersagt.

Speicher Eine Ausnahme des Typs Speicher wird beim Fehlschlagen der Alloziierung von Speicher ausgelöst. Durch die Schnittstelle wird beim Aufbau der Parameterstrukturen für die SAP-C-Library dynamisch Speicher alloziiert.

[17] Beispiel: ABAP-Zeichenketten sind längenbeschränkt und werden auf die Klasse **String** abgebildet. Vor der Zuweisung wird die Länge des Strings mit der in den Metainformationen gespeicherten Maximallänge überprüft. Bei einer Überschreitung wird eine Ausnahme ausgelöst.

7.4 Bewertung der Schnittstelle

Mit der realisierten objektorientierten, generischen, statisch typisierten Schnittstelle ist es gelungen, die R/3-Geschäftsobjekte dem TYCOON-2-System zur Verfügung zu stellen. Zusammengefaßt hat die Schnittstelle folgende Eigenschaften:

- Abbildung von R/3-Objektmodell und R/3-Funktionsbausteinen in das TYCOON-2-System
- Statische Typisierung
- Automatische Generierung von Stubklassen
- Erweiterte Konsistenzprüfung von Funktionsbausteinparametern
- Laufzeitinformationen zu Funktionen und Parametern (Metainformationen)
- R/3-Verbindungen und Geschäftsobjekte als TYCOON-2-Ressourcen
- Metadatenbrowser zur Entwicklungsunterstützung

R/3-Geschäftsobjekte und ihre BAPIs sind als Stubklassen im TYCOON-2-Systems abgebildet. Aufgrund der Beschränkungen des R/3-Objektmodells besitzen TYCOON-2-Geschäftsobjekte gegenüber gängigen objektorientierten Systemen eine andere Semantik. Die Schnittstelle portiert aus diesem Grund die Begriffe der wertbasierten Objektidentität über Schlüsselfelder und die BAPIs nach TYCOON-2. Weitere Bestandteile des R/3-Objektmodells wie Schnittstellen, Attribute und vor allem die für kommerzielle Anwendungen sehr interessanten Ereignisse sind aufgrund der Beschränkung der R/3-Schnittstellen zu externen Anwendungen[18] für das TYCOON-2-System nicht zugreifbar.
Deshalb reduziert sich die Nutzung von Geschäftsobjekten auf die BAPIs, über die allein ein Zugriff auf die R/3-Geschäftsobjekte möglich ist. Durch die Fokussierung der SAP auf die BAPIs besitzt ein TYCOON-2-Geschäftsobjekt keine Attribute. Attribute eines Objekts sind implizit über die Parameter eines BAPIs modelliert.
Für kommerzielle Anwendungen ist aber bereits diese eingeschränkte Sicht ein großer Fortschritt. Die BAPI-Schnittstelle ermöglicht es, R/3 als betriebswirtschaftliches *Backend* zu nutzen.

Neben den Schnittstellen zu den R/3-Geschäftsobjekten sind jedoch auch die Definitionen der R/3-Geschäftsobjekte im *Business Object Repository* (BOR) kritisch zu beurteilen. Hier sind vor allem die unvollständige Definition von Schlüsselfeldern als auch Probleme bei der Zuordnung der Schlüsselfelder eines R/3-Geschäftsobjektes auf die Parameter des implementierenden BAPI-Funktionsbausteins zu nennen. Bei der Analyse der BAPI-Signaturen wird deutlich, daß diese sich nicht am fachlichen Modell im BOR orientieren, sondern durch das relationale Datenmodell des R/3 bestimmt sind. Das ist der Grund, warum die Schlüsselfelder, die für die Identifizierung eines R/3-Geschäftsobjekts benötigt werden, bei den BAPIs eines R/3-Geschäftsobjekts unterschiedlich sind können.
Diese Ungenauigkeiten verhindern ein Generierung von TYCOON-2-Geschäftsobjektklassen,

[18] Hier ist vor allem die Schnittstelle der RFC-Automation gemeint.

wie es für die Funktionsbausteine der R/3-Funktionsbibliothek vorgenommen wird. Die BOR-Schnittstellen zum Auslesen der Informationen sind gleichwohl vorhanden.

Die Generierung von Stubklassen für Funktionsbausteine und deren Parameter besitzt zwei positive Aspekte:
Die Stubklassen besitzen eine statische Typsierung und verhindern so Fehler der Anwendung zur Laufzeit. Fehler werden auch durch eine erweiterte Konsistenzprüfung auf den Funktionsbausteinparametern vermieden[19]. Dies erhöht die Sicherheit und Zuverlässigkeit der Schnittstelle. Weiterhin löst die Generierung von Quellcode das Problem, daß die BAPI-Schnittstelle sich trotz der Zusicherung durch die SAP nicht als stabil herausgestellt hat. Zwischen den verschiedenen R/3-Releases werden Änderungen an den Signaturen der Funktionsbausteine vorgenommen, so daß ein Programm ohne Berücksichtigung dieses dynamischen Verhaltens in den verschiedenen R/3-Releases nicht fehlerfrei funktioniert.

Durch die Implementierung von R/3-Verbindungen und Geschäftsobjekten als TYCOON-2-Ressourcen kann von der expliziten Verbindungsaufnahme zum R/3-System bzw. vom Prüfen, ob ein TYCOON-2-Geschäftsobjekt im R/3 noch gültig ist, abstrahiert werden.

Die Entwicklung von Anwendungen basierend auf der TYCOON-2-R/3-Schnittstelle wird durch zwei prototypisch implementierte Werkzeuge unterstützt, die bereits bei der Entwicklung der Schnittstelle eingesetzt wurden. Es handelt sich hierbei um das bereits beschriebene Werkzeug für die Generierung von Funktionsbausteinen und einem Metadatenbrowser, der die nach TYCOON-2 abgebildeten Funktionsbausteine mit ihren Parametern und Methodensignaturen anzeigt. Einerseits lassen sich durch den Generator die für einen Zugriff benötigten Funktionsbausteine komfortabel erzeugen, anderseits stehen mit Hilfe des Metadatenbrowsers die Metadaten für einen Funktionsaufruf während der Programmierung zur Verfügung.

[19] Die erweiterte Konsistenzprüfung validiert vor allem Längen und Wertebereiche der Aktualparameter

Kapitel 8

Ausblick

Mit der realisierten Schnittstelle ist eine Integration zwischen dem objektorientierten TYCOON-2-System und dem SAP R/3 geschaffen worden. Mit den beschriebenen Einschränkungen sind die R/3-Geschäftsobjekte im TYCOON-2-System abgebildet. Diese Geschäftsobjekte können Ansatzpunkt zur Entwicklung von Anwendungen im TYCOON-2-System sein, die das R/3 als *Backend* zur Speicherung der Anwendungsdaten nutzen. Umgekehrt können TYCOON-2-Anwendungen betriebswirtschaftliche Daten aus dem R/3 lesen. Eine konkretes Szenario, in dem so eine Funktionalität benötigt wird, wäre z.B. eine Internet-E-*Commerce*-Anwendung (*Internet-Shop*). Ein *Internet-Shop* könnte das R/3 zur Speicherung sowohl von Produkt- und Benutzerdaten als auch von Verkaufsaufträgen nutzen. Der Vorteil einer solchen Lösung wäre, daß der *Internet-Shop* nahtlos in die firmeninternen durch das R/3 unterstützten Geschäftsprozesse integriert ist. Weiterhin können die umfangreichen *Reporting*-Möglichkeiten im R/3 benutzt werden, um z.B. Auswertungen über Verkaufsverhalten und Umsätze zu erhalten.

Integrationslösungen mit dem R/3 sind allgemein von sehr großem kommerziellen Interesse, da sich das System R/3 zum dominierenden Marktführer innerhalb des Marktsegments betriebswirtschaftlicher Anwendungssoftware entwickelt hat. Mit der Öffnung des bisher monolithischen R/3 über die Geschäftsobjekte und BAPIs ergeben sich sehr viele Anwendungsmöglichkeiten, wobei der erwähnte *Internet-Shop* nur eine darstellt.
Das Ziel von Standardisierungsgremien wie der OMG und der OAG ist die Ermöglichung der Konstruktion komplexer Softwaresysteme mit Hilfe objektorientierter Komponenten. Die Interaktion zwischen diesen Komponenten vollzieht sich auf der Grundlage von Geschäftsobjekten. R/3 stellt in diesem Szenario eine Komponente dar, die im Vergleich zu heutigen gängigen Komponenten um ein Vielfaches größer und komplexer ist. Inwieweit dieses Ziel aber realistisch ist, muß sich erst noch zeigen. Eine beliebige Kombination und Austauschbarkeit von Softwarekomponenten würde die Bedeutsamkeit einer einzelnen Software wie dem R/3 drastisch vermindern und damit den kommerziellen Erfolg des R/3 gefährden. Aus diesem Grund wird die Öffnung des R/3 gegenüber externen Anwendungen wahrscheinlich niemals so weitreichend sein. Dennoch ist zu erwarten, daß die über BAPIs verfügbare R/3-Funktionalität in den nächsten R/3-Releases weiter drastisch zunehmen wird. Allein die Anzahl der BAPIs hat sich vom Release 3.1g bis zum Release 4.0b mit 450 BAPIs mehr als verdoppelt. Neben der SAP werden auch andere Softwarehersteller BAPIs erstellen und anbieten. Dennoch muß

betont werden, daß auch weiterhin ein gute Kenntnis des R/3 vonnöten ist, um BAPIs zu benutzen.

Problematisch und aufgrund des oben Beschriebenen unter Umständen auch gar nicht gewollt, ist eine wirklich objektorientierte Integration mit dem R/3. Das objektorientierte Modell der R/3-Geschäftsobjekte basiert bis jetzt (Releasestand 4.0b) ausschließlich auf den Funktionsbausteinen, einer Technologie, die über ihre Schnittstelle ausschließlich eine Datenkapselung zuläßt. Die Semantik von Geschäftsobjekten ist ohne weitere Arbeit an einer objektorientierten Implementierung der Geschäftsobjekte durch die SAP nicht zu erreichen. Mit der Einführung von OO-ABAP existiert der Ansatz einer objektorientierten Implementierung. Konzeptionell besitzt OO-ABAP alle Eigenschaften einer objektorientierten Programmiersprache. Im Release 4.0b sind jedoch eine Vielzahl von Eigenschaften noch nicht implementiert. Ob sich durch OO-ABAP verbesserte Möglichkeiten zur objektorientierten R/3-Integration bieten, ist noch nicht zu beurteilen.

Um eine objektorientierte Integration mit dem R/3 zu realisieren, muß jedoch auch das fachliche Modell des *Business Object Repository* überarbeitet werden. Insbesondere die Schlüsselfelder von Geschäftsobjekten sind exakt zu formulieren, um eine generische Schnittstelle mit dem R/3 konstruieren zu können. Die Generizität einer Schnittstelle leidet momentan daran, daß die BAPI-Schnittstellen (also die Signaturen von Funktionsbausteinen) nicht über R/3-Releasestände stabil sind. Mit einer Kapselung dieser BAPI-Schnittstellen in OO-ABAP-Klassen sind diese Probleme jedoch zu lösen. Abschließend betrachtet ist man für eine weitergehende objektorientierte R/3-Integration auf eine Änderung des R/3-Systems im fachlichen Konzept als auch in der Implementierung durch die SAP angewiesen.

Literaturverzeichnis

Arnold, Gosling 96: Arnold, K. und Gosling, J. *Java - Die Programmiersprache.* Addison-Wesley, 1996.

BODTF 98a: BODTF, Business Object Domain Task Force. *Combined Business Object Facility - Interoperability Specification.* Technical report, Object Management Group, 4 1998. http://www.omg.org.

BODTF 98b: BODTF, Business Object Domain Task Force. *RFP Status - Task and Services.* Technical report, Object Management Group, 4 1998. http://www.omg.org.

BODTF 98c: BODTF, OMG Business Object Domain Task Force. *BODTF-RFP 1 Submission Combined Business Object Facility Business Object Component Architecture Proposal.* Technical report, OMG Document: bom/98-01-07, 1 1998. http://www.omg.org.

Buck-Emden, Galimow 96: Buck-Emden, R. und Galimow, J. *Die Client/Server-Architektur des Systems R/3.* Addison Wesley, 3 edition, 1996.

Burt 95: Burt, C. *OMG BOMSIG.* Technical report, OMG document 95-02-04, 2 1995. http://www.omg.org.

Carlsen 99: Carlsen, A. *Ein generisches Online-Verkaufssystem: Anforderungsanalyse, objektorientierter Entwurf, Realisierung mit Lotus Notes und dem SAP R/3.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1999.

Casanave 98: Casanave, C. *Business-Object Architectures and Standards.* Technical report, Data Access Corporation, 1 1998.

Dadam 96: Dadam, P. *Verteilte Datenbanken und Client/Server-Systeme : Grundlagen, Konzepte und Realisierungsformen.* Springer, 1996.

Deimel 98: Deimel, A. *The SAP R/3 Business Framework.* Technical report, SAP AG, Walldorf, Germany, 7 1998. Software-Concepts and Tools (1998) vol.19.

Dömer 97: Dömer, F. *Erfahrungen mit SAP Migration von R/2 auf R/3.* HMD - Theorie und Praxis der Wirtschaftsinformatik 34 (1997) 198, 1997, S. S. 94–106.

Douglas et al. 98: Douglas, L., Little, M., und Zuliani, F. *San Francisco Evaluation Kit (V1R2) Getting Started.* Technical report, IBM, 7 1998. http://www.redbooks.ibm.com.

EE 97: Enterprise Engines Inc.: Convergent Engineering . http://www.engines.com/company/cei.html, 1997.

Eeles, Sims 98: Eeles, P. und Sims, O. *Building Business Objects.* Wiley Computer Publishing, 1998.

Erler 98: Erler, T. *Business Objects - Eine objektorientierte Gestaltungsphilosophie für das Business Engineering.* Technical report, Ruhr-Universität Bochum Lehrstuhl für Wirtschaftsinformatik Fakultät für Wirtschaftswissenschaft, 5 1998. http://www.winf.ruhr-uni-bochum.de.

Ernst 98: Ernst, M. *Typüberprüfung in einer polymorphen objektorientierten Programmiersprache. Analyse, Design und Implementierung eines Typprüfers für Tycoon-2.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1998.

Gamma et al. 96: Gamma, E., Helm, R., Johnson, R., und Vlissides, J. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software - Deutsche Übersetzung von Dirk Riehle.* Addison-Wesley (Deutschland) GmbH, Bonn, 1 edition, 1996.

Gawecki et al. 97: Gawecki, A., Matthes, F., Schmidt, J.W., und Stamer, S. *Persistent Object Systems: From Technology to Market.* In: Jarke, M. (Hrsg.). *27. Jahrestagung der Gesellschaft für Informatik.* Springer-Verlag, September 1997.

Gawecki, Matthes 95: Gawecki, A. und Matthes, F. *TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification.* FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

Gawecki, Matthes 96: Gawecki, A. und Matthes, F. *Integrating Subtyping, Matching and Type Quantification: A Practical Perspective.* In: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, Linz, Austria, Juli 1996, S. 26–47. Springer-Verlag.

Gray, Reuter 93: Gray, J. und Reuter, A. *Transaction Processing – Concepts and Techniques from The Morgan Kaufmann Series in Data Management Systems.* Morgan Kaufmann Publishers, 1993.

Group 96: Group, Object Management. *Common Facilities RFP-4 Common Business Objects and Business Object Facility.* Technical report, OMG TC Document CF/96-01-04, 1 1996. http://www.omg.org.

Heiderich 98: Heiderich, J. *Representation of Business Semantics in an OMG Business Object Facility.* Technical report, Interactive Objects Software GmbH, 1 1998.

HOX98 98: Higher-Order Informations- und Kommunikationssysteme GmbH . http://www.higher-order.de, 1998.

IBM, NIIIP 98: IBM und NIIIP. *Joint Common Business Objects Revised Final Submission.* Technical report, IBM Cooperation, NIIIP Consortium, OMG Document Number: bom/98-03-02, 1 1998.

IBM 98: IBM. *San Francisco Concepts and Facilities.* Technical report, IBM, 7 1998. http://www.redbooks.ibm.com.

INPRISE 99: Inprise Corporation . http://www.inprise.com, 1999.

König, Buxmann 97: König, W. und Buxmann, P. *Empirische Ergebnisse zum Einsatz der betrieblichen Standardsoftware SAP R/3.* Wirtschaftsinformatik 39 (1997) 4, 1997, S. S. 331–338.

Lockemann, Schmidt 87: Lockemann, P.C. und Schmidt, J.W. (Hrsg.). *Datenbank-Handbuch.* Springer, 1987.

Lutz 97: Lutz, S. *Eine polymorph typisierte Schnittstelle der Sprache Tycoon zum System SAP R/3.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1997.

Mathiske et al. 93: Mathiske, B., Matthes, F., und Müßig, S. *The Tycoon System and Library Manual.* DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993. (Revised 19-jan-1996).

Mathiske et al. 95: Mathiske, B., Matthes, F., und Schmidt, J.W. *Scaling Database Languages to Higher-Order Distributed Programming.* In: *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995. Also appeared as TR FIDE/95/137, FIDE Technical Report Series, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.

Mathiske et al. 97: Mathiske, B., Matthes, F., und Schmidt, J.W. *On Migrating Threads.* Journal of Intelligent Information Systems, Jg. 8, 1997, Nr. 2, S. 167–191.

Mathiske 96: Mathiske, B. *Mobilität in persistenten Objektsystemen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, Oktober 1996.

Matthes et al. 94: Matthes, F., Müßig, S., und Schmidt, J.W. *Persistent Polymorphic Programming in Tycoon: An Introduction.* FIDE Technical Report Series FIDE/94/106, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.

Matthes et al. 97: Matthes, F., Schröder, G., und Schmidt, J.W. *Tycoon: A Scalable and Interoperable Persistent System Environment.* In: ATKINSON, M.P. (Hrsg.): Fully Integrated Data Environments. Springer-Verlag, 1997.

Matthes, Ziemer 98: Matthes, F. und Ziemer, S. *Understanding SAP R/3:A Tutorial for Computer Scientists.* Technical report, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Germany, Maerz 1998.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

Matzke 98: Matzke, B. *ABAP/4 - Die Programmiersprache des SAP-Systems R/3.* Addison-Wesley, 2. edition, 1998.

Mende 98: Mende, U. *Softwareentwicklung für R/3 : Data Dictionary, ABAP/4, Schnittstellen.* Berlin [u.a.]: Springer, 1998.

Niemann 95: Niemann, K.D. *Client/Server-Architektur, Organisation und Methodik der Anwendungsentwicklung.* Vieweg, 1995.

OAG 98: OAG. *WHITE PAPER Open Applications Integration.* Technical report, Open Application Group, 1 1998.

OAG 99: Open Application Group . http://www.openapplications.org, 1999.

OMG97 : What are the OMG Task Forces ? . http://www.omg.org/about/task.htm.

OMG 99: Object Management Group . http://www.omg.org, 1999.

Pour 97: Pour, N. V. *Entwicklung von Client/Server-Anwendungen mit 4GL-Technologie: Tycoon, NATURAL und ABAP/4 im Vergleich.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Sepetember 1997.

Pressmar, Scheer 98: Pressmar, D.B. und Scheer, A.-W. (Hrsg.). *SAP R/3 in der Praxis : Neuere Entwicklungen und Anwendungen.* Wiesbaden: Gabler, 1998.

SAP 96a: SAP. *Das Business Framework.* SAP, 1996.

SAP 96b: SAP. *Funktionen im Detail: System R/3: SAP Business Workflow.* Walldorf: SAP AG, 1996.

SAP 96c: SAP. *Funktionen im Detail: System R/3: Technologie-Infrastruktur.* Walldorf: SAP AG, 1996.

SAP 97a: SAP. *ABAP Objects: Objektorientierung in ABAP.* SAP, 1997.

SAP 97b: SAP. *SAP Business-Objekte.* SAP, 1997.

SAP 98a: SAP. *Funktionen im Detail: SAP R/3 System Architektur.* SAP, 1998.

SAP 98b: SAP. *SAP Homepage* . http://www.sap.com, 1998.

SAP 98c: SAP. *SAP Online Hilfe 4.0b, BC-ABAP Dictionary.* SAP, 1998.

SAP 98d: SAP. *SAP Online Hilfe 4.0b, BC-ABAP Programmierung.* SAP, 1998.

SAP 98e: SAP. *SAP Online Hilfe 4.0b, BC-Business Engineer.* SAP, 1998.

SAP 98f: SAP. *SAP Online Hilfe 4.0b, BC-Customizing.* SAP, 1998.

SAP 98g: SAP. *SAP Online Hilfe 4.0b, BC-Desktop Office Integration.* SAP, 1998.

SAP 98h: SAP. *SAP Online Hilfe 4.0b, BC-Erweiterungen des SAP-Standards.* SAP, 1998.

SAP 98i: SAP. *SAP Online Hilfe 4.0b, BC-Remote Communications.* SAP, 1998.

SAP 98j: SAP. *SAP Online Hilfe 4.0b, BC-SAP Business Workflow.* SAP, 1998.

SAP 98k: SAP. *SAP Online Hilfe 4.0b, CA-Application Link Enabling (ALE).* SAP, 1998.

SAP 98l: SAP. *SAP Online Hilfe 4.0b, CA-BAPI Programmierleitfaden.* SAP, 1998.

SAP 98m: SAP. *SAP Online Hilfe 4.0b, CA-BAPIs Einführung und Überblick.* SAP, 1998.

Schneider 98: Schneider, D. *Eine bidirektionale typisierte Kommunikation zwischen zwei reflexiven, objektorientierten Sprachen.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Januar 1998.

SF 98: IBM San Francisco . http://www.ibm.com/java/SanFrancisco, 1998.

Shelton 97: Shelton, R. *Business Objects.* Technical report, Open Engineering Inc., 1 1997. http://www.openeng.com/busobj.html.

Siemens2 97: Siemens R/3 LIVE - LIVE Method and Tools. Vol 2 . CD-ROM, München, 1997.

Siemens 97: Siemens. *R/3 Modellfirma, LIVE Productions- und Vertriebs AG, Dokumentation.* Technical report, Simens Informationssysteme AG, 7 1997. auf CD.

Sims 94: Sims, O. *Business Objects.* New York: McGrawHill, 1994.

Sinha 92: Sinha, A. *Client/Server Computing.* Comm. of the ACM, Jg. 35, 1992, Nr. 7, S. 77–98.

Wahlen 98: Wahlen, J. *Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juni 1998.

Watt 90: Watt, D.A. *Programming language concepts and paradigms.* Prentice Hall international series in computer science. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Wegner 98: Wegner, H. *Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Mai 1998.

Weikard 98: Weikard, M. *Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1998.

Wenzel 96: Wenzel, P. (Hrsg.). *Betriebswirtschaftliche Anwendungen des integrierten Systems SAP R/3: Projektstudien, Grundlagen und Anregungen fuer eine erfolgreiche Praxis.* Braunschweig [u.a.]: Vieweg, 1996.

Wenzel 99: Wenzel, P. (Hrsg.). *Betriebswirtschaftliche Anwendungen des integrierten Systems SAP/R3.* Braunschweig [u.a.]: Vieweg, 1999.

Wienberg 97: Wienberg, A. *Bootstrap einer persistenten objektorientierten Programmierumgebung.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1997.

Will et al. 96: Will, L., Hienger, Ch., Strassenburg, F., und Himmer, R. *R/3-Administration.* Addison-Wesley, 1996.

Wolff 98: Wolff, E. *San Francisco: Framwork für Geschäftsanwendungen, Im Rahmen.* IX - Magazin für professionelle Informationstechnik, Jg. 7, 1998, Nr. 7, S. 116–121.

Anhang A

Klassendiagramm der Schnittstelle

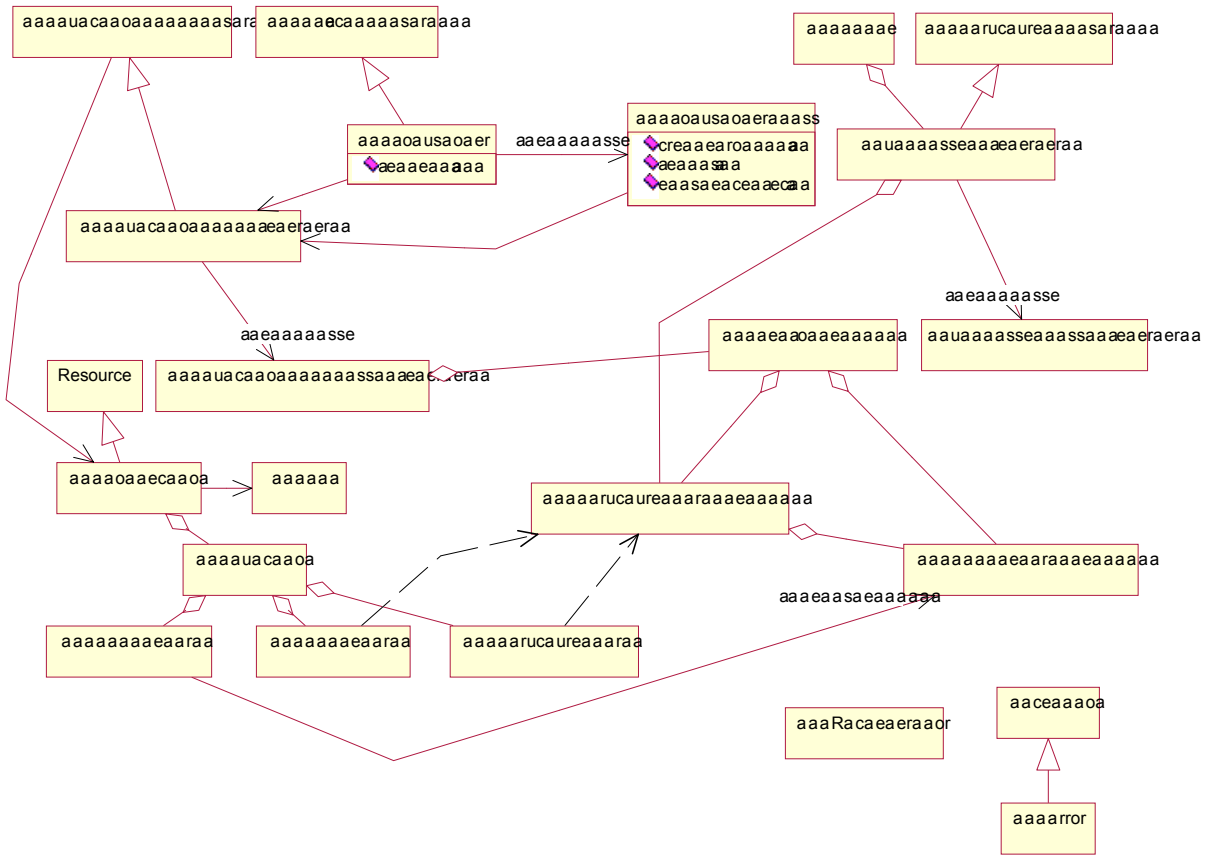

Abbildung A.1: Klassendiagramm der TYCOON-SAP-Schnittstelle

Anhang B

Quelltextvorlagen für die Klassengenerierung

Die hier gezeigten vier Quelltextvorlagen werden für alle automatisch generierten Klassen benutzt. Sie geben jeweils Klasse und Metaklasse von Ableitungen von einer **SubStructure** und einer **SapFunctionLib** wieder. Die Vorlagen sind jeweils nach Blöcken <**Block=Blockname**> unterteilt, die einzeln aus den Vorlagen vom Generator eingelesen werden. Die Blöcke die nicht mit „Block“ anfangen, dienen als Variable und werden vom Generator durch Tycoon-Schlüsselwörter oder R/3-Daten ersetzt.

Quelltextvorlage für Ableitungen von SapStructure

```
<block=ssg_header>
class Sap<class-struct-name>
super SapStructure
metaclass Sap<class-struct-name>Class
public
methods

readFrom( s:SapStructure)
{    let struct  = _typeCast( s, :Sap<class-struct-name>),
<block=ssg_readFromParam>
     <param-name>:= struct.<param-name>,
<block=ssg_show>
     nil
}
mclass:SapStructureMetaclass
{
     Sap<class-struct-name>
}
<block=ssg_get>
     "<param-name>":<param-tycoon/type>
{
```

```
     _<param-name>
}
<block=ssg_set>
     "<param-name>:="( value:<param-tycoon/type>)
{
     checkMetadataConstraintsForParam("<param-name>",value),
     _<param-name>:=value
}
<block=ssg_setStruct>
     "<param-name>:="( value:<param-tycoon/type>)
{
     _<param-name>:=value
}
<block=ssg_privat>
     private
<block=ssg_slots>
     _<param-name>:<param-tycoon/type>, (* <param-abap/type> <param-length> *)
<block=ssg_tail>
;
```

Quelltextvorlage für Metaklassen von Ableitungen von SapStructure

```
<block=sscg_header>
class  Sap<class-struct-name>Class
super SapStructureMetaclass
metaclass MetaClass
public
methods
new :Sap<class-struct-name>
{
     let instance = _new,
<block=sscg_perform>instance.<param-name>,
     instance.<param-name>:=nil,
<block=sscg_metadata>
     instance._init, (* call init in class SapStructure *)
     instance
}
     metadata :SapStructuredParamMetadata
{
     _metadata.isNotNil ? { _metadata } :
{
let mt = SapStructuredParamMetadata.new,
    let simpleDict = OrderedDictionary.new,
    let structDict = OrderedDictionary.new,
<block=sscg_params>
    simpleDict["<param-name>"]:= SapSimpleParamMetadata.new(
     "<param-name>",<param-length>,'<param-role>' ,'<param-tycoon/type>'),
```

```
<block=sscg_paramsStructure>
    structDict["<param-name>"]:= SapSimpleParamMetadata.new( "<param-name>",
    <param-length>,'<param-role>' ,'<param-tycoon/type>'),
<block=sscg_tail>
   (*mt.role := '<structure-role>',*)
    mt.ddReference := "<structure-name>",
    mt.fieldsMetadata := simpleDict,
    mt.structMetadata := structDict,
    _metadata := mt,
    _metadata
    }
}
;
```

Quelltextvorlage für Tycoon-Funktionsbibliotheken

```
<block=sf_header>
class <class-fclib-name>
super SapFunctionLib
metaclass <class-fclib-name>Class
public
(*   M E T H O D E N *)
methods
metadata :Dictionary(String, SapMethodMetadata)
{
    <class-fclib-name>.metadata
}
<block=sf_functionBegin>
<method-name-lowercase>(
<block=sf_functionImport>
<(i)import-para-name-lowercase>:<(i)import-para-type/class>,
<block=sf_functionTable>
<(i)table-para-name-lowercase>: SapTable(Sap<(k)structure-name>),
<block=sf_functionExport>)<export-param-class>
{
    let export<export-param-class-name> = nil,
    let functionMetadata=metadata["<function-name>"],
    let simpleMetadata=functionMetadata.simpleParam,
    let structuredMetadata=functionMetadata.structuredParam,
    let sf=SapFunction.new(functionMetadata.name,_connection),
    let importParam:SapSimpleParam=nil,
<block=sf_import>
    importParam:=simpleMetadata["<import-para-name>"].asSapSimpleParam.value:=
    <import-para-name-lowercase>,
    importParam.name:="<import-para-name>",
    sf.importList.add(importParam),
<block=sf_importStructure>
```

```
    sf.structList.add(SapStructuredParam.new(<param-name>,
    <structure-type>,"<param-name>",'i')),
<block=sf_exportSimple>
(* 1 Export :*)
 export := SapSimpleParam.new("<export-para-name>"," ",<export-para-length>,
     'e','<export-para-type>'),
    sf.exportList.add(export),
<block=sf_exportRealStructure>
    (* 1 export:*)
    export:= Sap<structure-type>.new,
    sf.structList.add(SapStructuredParam.new(export,Sap<structure-type>,
    "<param-name>",'e')),
<block=sf_exportStructured>
    (*>1 export:*)
    export:= Sap<structure-type>.new,
<block=sf_exportStructuredSlot>
    export.<struct-slot-name> := Sap<struct-slot-type>.new,
    sf.structList.add(SapStructuredParam.new(export.<struct-slot-name>,
    Sap<struct-slot-type>,"<struct-slot-name>",'e')),
<block=sf_exportSimpleSlot>
    let i<number>=SapSimpleParam.new("<slot_name>","<slot_type>",
    <slot_length>,'e','<slot_Saptype>'),
    sf.exportList.add(i<number>),
<block=sf_tables>
    (*TABLES START*)
    sf.tableList.add(SapTableParam.new(<(i)table-para-name-lowercase>,
    Sap<structure-name>,"<structure-realname>")),
    (*TABLES END*)
<block=sf_funcEnd>
    sf.register,
    (*conn.open; *)
    sf.callReceive,
    export.value
}
<block=sf_funcStructuredEnd>
    sf.register,
    (*conn.open; *)
    sf.callReceive,
    export
}
<block=sf_structuredSlots>
    export.<slot_name>:= i<number>.value,
<block=sf_noExports>
    sf.register,
    sf.callReceive
}
<block=sf_structRegisterCall>
```

```
    sf.register,
    sf.callReceive,
<block=sf_structEnd>
    export
}
<block=sf_tail>
;
```

Quelltextvorlage für Tycoon-Funktionsbibliotheken

```
<block=sfc_header>
class <class-fclib-name>Class
super SapFunctionLibMetaclass
metaclass MetaClass
public methods
new:<class-fclib-name>
{
super._new
}
host:String
{
"<host-name>"
}
sysNr:String
{
"<host-sysNr>"
}
metadata:Dictionary(String,SapMethodMetadata)
{
    _metadata.isNotNil ? {_metadata} : {
    let tmpDict :Dictionary(String,SapMethodMetadata)= Dictionary.new,
<block=sfc_funcBegin>
(* FUNCTION BEGIN *)
    let m=SapMethodMetadata.new,
    m.name:="<function-name>",
<block=sfc_funcStructured>
    m.structuredParam["<structure-name>"]:=Sap<structure-type>.metadata,
<block=sfc_funcSimple>
    m.simpleParam["<param-name>"]:=
    SapSimpleParamMetadata.new( "<param-name>" ,<param-length>,
    '<param-role>','<param-tycoon/type>'),
<block=sfc_exportStructuredParam>
    m.exportStructuredParam:=Sap<function-nameUpperCase>export.metadata,
<block=sfc_funcEnd>
    tmpDict[m.name]:=m,
(*FUNCTION END*)
<block=sfc_tail>
```

```
    _metadata:=tmpDict,
    _metadata
    }
}
    private
    _metadata:Dictionary(String,SapMethodMetadata)
;
```

Anhang C

Funktionsbaustein: BAPI_CUSTOMER_CHECK-EXISTENCE

Parameter des Funktionsbausteins

Die Parameter des Funktionsbausteins **BAPI_CUSTOMER_CHECKEXISTENCE** sind im folgenden dargestellt, dabei ist jeweils der Name, die Länge und der Typ eines Parameters angegeben (bei den strukturierten Parametern sind zuzsätzlich die Felder zur besseren Übersichtlichkeit durchnummeriert).

Exportparameter (gekürzt)

Nr	Name	Länge	Typ	Nr	Name	Länge	Typ
Customer_Data vom Typ KNA1							
1	CFOPC	2	S	136	KUNNR	10	S
2	BBSNR	5	N	137	XXIPI	1	C
3	INSPBYDEBI	1	C	138	KNURL	132	S
4	PERIV	2	S				
5	STRAS	35	S	Return vom Typ BAPIRETURN			
6	PFORT	35	S	1	MESSAGE	220	S
7	TXLW1	3	S	2	MESSAGE_V3	50	S
8	TXLW2	3	S	3	MESSAGE_V4	50	S
10	LAND1	3	S	4	CODE	5	S
...				5	LOG_NO	20	S
132	BRAN4	10	S	6	LOG_MSG_NO	6	N
133	PFACH	10	S	7	MESSAGE_V1	50	S
134	BRAN5	10	S	8	TYPE	1	C
135	VBUND	6	S	9	MESSAGE_V2	50	S

```
Importparameter

Name                    Länge    Typ
CUSTOMERNO               10       S
DISTRIBUTION_CHANNEL      2       S
DIVISION                  2       S
SALES_ORGANIZATION        4       S
```

Aufruf des Funktionsbausteines innerhalb des SAP R/3 Systems (ABAP)

```
REPORT  ZCHECKEXISTENCE.
DATA:
CUSTOMERNO LIKE  BAPI1007-CUSTOMER VALUE '11',
SALES_ORGANIZATION LIKE  BAPIORDERS-SALES_ORG VALUE '0001',
DISTRIBUTION_CHANNEL LIKE  BAPIORDERS-DISTR_CHAN VALUE '01',
DIVISION LIKE  BAPIORDERS-DIVISION VALUE '01',
CUSTOMER_DATA LIKE  KNA1,
CUSTOMER_NUMBER_OUT LIKE  BAPI1007-CUSTOMER,
RETURN LIKE  BAPIRETURN.

CALL FUNCTION 'BAPI_CUSTOMER_CHECKEXISTENCE'
     EXPORTING
          CUSTOMERNO           = CUSTOMERNO
          SALES_ORGANIZATION   = SALES_ORGANIZATION
          DISTRIBUTION_CHANNEL = DISTRIBUTION_CHANNEL
          DIVISION             = DIVISION
    IMPORTING
         CUSTOMER_DATA        = CUSTOMER_DATA
         RETURN               = RETURN
     EXCEPTIONS
          OTHERS               = 1.

END REPORT.
```

Anhang D

Makro-ABAP-Programm zum Geschäftsobjekt Kunde

```
INCLUDE <OBJECT>.
BEGIN_DATA OBJECT. " Do not change.. DATA is generated
* only private members may be inserted into structure private
DATA:
" begin of private,
"   to declare private attributes remove comments and
"   insert private attributes here ...
" end of private,
  BEGIN OF KEY,
      CUSTOMERNO LIKE KNA1-KUNNR,
  END OF KEY,
      _KNA1 LIKE KNA1.
END_DATA OBJECT. " Do not change.. DATA is generated

DEFINE RETURN.
  EXIT_RETURN &1 SPACE SPACE SPACE SPACE.
END-OF-DEFINITION.

TABLES KNA1 .

BEGIN_METHOD DISPLAY CHANGING CONTAINER.
  MOVE OBJECT-KEY-CUSTOMERNO TO KNA1-KUNNR.
  SET PARAMETER ID 'KUN' FIELD KNA1-KUNNR .
  SET PARAMETER ID 'DDY' FIELD '110/120/125/340/360/370'.
  CALL TRANSACTION 'VD03'.          " AND SKIP FIRST SCREEN.
END_METHOD.

BEGIN_METHOD CREATE CHANGING CONTAINER.
  SET PARAMETER ID 'KUN' FIELD OBJECT-KEY-CUSTOMERNO.
  CALL TRANSACTION 'VD01'.
```

```
END_METHOD.

BEGIN_METHOD EDIT CHANGING CONTAINER.
  SET PARAMETER ID 'KUN' FIELD OBJECT-KEY-CUSTOMERNO.
  SET PARAMETER ID 'DDY' FIELD '110/120/125/340/360/370'.
  CALL TRANSACTION 'VD02'.            " AND SKIP FIRST SCREEN.
END_METHOD.

*
GET_TABLE_PROPERTY KNA1.
DATA SUBRC LIKE SY-SUBRC.
* Fill TABLES KNA1 to enable Object Manager Access to Table Properties
  PERFORM SELECT_TABLE_KNA1 USING SUBRC.
  IF SY-SUBRC NE 0.
    EXIT_OBJECT_NOT_FOUND.
  ENDIF.
END_PROPERTY.
*
* Use Form also for other(virtual) Properties to fill TABLES KNA1
FORM SELECT_TABLE_KNA1 USING SUBRC LIKE SY-SUBRC.
* Select single * from KNA1, if OBJECT-_KNA1 is initial
  IF OBJECT-_KNA1-MANDT IS INITIAL
  AND OBJECT-_KNA1-KUNNR IS INITIAL.
    SELECT SINGLE * FROM KNA1 CLIENT SPECIFIED
        WHERE MANDT = SY-MANDT
        AND KUNNR = OBJECT-KEY-CUSTOMERNO.
    SUBRC = SY-SUBRC.
    IF SUBRC NE 0. EXIT. ENDIF.
    OBJECT-_KNA1 = KNA1.
  ELSE.
    SUBRC = 0.
    KNA1 = OBJECT-_KNA1.
  ENDIF.
ENDFORM.

BEGIN_METHOD CHANGEPASSWORD CHANGING CONTAINER.
DATA:
      PASSWORD LIKE BAPIUID-PASSWORD,
      NEWPASSWORD LIKE BAPIUID-PASSWORD,
      VERIFYPASSWORD LIKE BAPIUID-PASSWORD,
      RETURN LIKE BAPIRETURN.
  SWC_GET_ELEMENT CONTAINER 'Password' PASSWORD.
  SWC_GET_ELEMENT CONTAINER 'NewPassword' NEWPASSWORD.
  SWC_GET_ELEMENT CONTAINER 'VerifyPassword' VERIFYPASSWORD.
  CALL FUNCTION 'BAPI_CUSTOMER_CHANGEPASSWORD'
    EXPORTING
      VERIFY_PASSWORD = VERIFYPASSWORD
```

```
      NEW_PASSWORD = NEWPASSWORD
      PASSWORD = PASSWORD
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD CHECKPASSWORD CHANGING CONTAINER.
DATA:
      PASSWORD LIKE BAPIUID-PASSWORD,
      SALESORGANIZATION LIKE VBAK-VKORG,
      DISTRIBUTIONCHANNEL LIKE VBAK-VTWEG,
      DIVISION LIKE VBAK-SPART,
      RETURN LIKE BAPIRETURN,
      CUSTOMERNUMBEROUT LIKE KNA1-KUNNR,
      CUSTOMERDATA LIKE KNA1.
  SWC_GET_ELEMENT CONTAINER 'Password' PASSWORD.
  SWC_GET_ELEMENT CONTAINER 'SalesOrganization' SALESORGANIZATION.
  SWC_GET_ELEMENT CONTAINER 'DistributionChannel' DISTRIBUTIONCHANNEL.
  SWC_GET_ELEMENT CONTAINER 'Division' DIVISION.
  CALL FUNCTION 'BAPI_CUSTOMER_CHECKPASSWORD'
    EXPORTING
      DIVISION = DIVISION
      DISTRIBUTION_CHANNEL = DISTRIBUTIONCHANNEL
      SALES_ORGANIZATION = SALESORGANIZATION
      PASSWORD = PASSWORD
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      CUSTOMER_DATA = CUSTOMERDATA
      CUSTOMER_NUMBER_OUT = CUSTOMERNUMBEROUT
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
  SWC_SET_ELEMENT CONTAINER 'CustomerNumberOut' CUSTOMERNUMBEROUT.
  SWC_SET_ELEMENT CONTAINER 'CustomerData' CUSTOMERDATA.
```

```
END_METHOD.

BEGIN_METHOD INITPASSWORD CHANGING CONTAINER.
DATA:
      PASSWORD LIKE BAPIUID-PASSWORD,
      RETURN LIKE BAPIRETURN.
  CALL FUNCTION 'BAPI_CUSTOMER_INITPASSWORD'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      PASSWORD = PASSWORD
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Password' PASSWORD.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD CREATEPASSWORD CHANGING CONTAINER.
DATA:
      RETURN LIKE BAPIRETURN.
  CALL FUNCTION 'BAPI_CUSTOMER_CREATEPWREG'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD DELETEPASSWORD CHANGING CONTAINER.
DATA:
      RETURN LIKE BAPIRETURN.
  CALL FUNCTION 'BAPI_CUSTOMER_DELETEPWREG'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
```

```
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD GETPASSWORD CHANGING CONTAINER.
DATA:
      RETURN LIKE BAPIRETURN,
      STATUSINFO LIKE BAPIUSWSTA OCCURS 0.
  SWC_GET_TABLE CONTAINER 'Statusinfo' STATUSINFO.
  CALL FUNCTION 'BAPI_CUSTOMER_GETPWREG'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
    TABLES
      STATUSINFO = STATUSINFO
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
  SWC_SET_TABLE CONTAINER 'Statusinfo' STATUSINFO.
END_METHOD.

BEGIN_METHOD CHECKEXISTENCE CHANGING CONTAINER.
DATA:
      SALESORGANIZATION LIKE BAPIORDERS-SALES_ORG,
      DISTRIBUTIONCHANNEL LIKE BAPIORDERS-DISTR_CHAN,
      DIVISION LIKE BAPIORDERS-DIVISION,
      CUSTOMERDATA LIKE KNA1,
      CUSTOMERNUMBEROUT LIKE BAPI1007-CUSTOMER,
      RETURN LIKE BAPIRETURN.
  SWC_GET_ELEMENT CONTAINER 'SalesOrganization' SALESORGANIZATION.
  SWC_GET_ELEMENT CONTAINER 'DistributionChannel' DISTRIBUTIONCHANNEL.
  SWC_GET_ELEMENT CONTAINER 'Division' DIVISION.
  CALL FUNCTION 'BAPI_CUSTOMER_CHECKEXISTENCE'
    EXPORTING
      DIVISION = DIVISION
      DISTRIBUTION_CHANNEL = DISTRIBUTIONCHANNEL
      SALES_ORGANIZATION = SALESORGANIZATION
```

```
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
      CUSTOMER_NUMBER_OUT = CUSTOMERNUMBEROUT
      CUSTOMER_DATA = CUSTOMERDATA
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'CustomerData' CUSTOMERDATA.
  SWC_SET_ELEMENT CONTAINER 'CustomerNumberOut' CUSTOMERNUMBEROUT.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD GETSALESAREAS CHANGING CONTAINER.
DATA:
      RETURN LIKE BAPIRETURN,
      SALESAREAS LIKE BAPIKNVVKY OCCURS 0.
  CALL FUNCTION 'BAPI_CUSTOMER_GETSALESAREAS'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      RETURN = RETURN
    TABLES
      SALESAREAS = SALESAREAS
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
  SWC_SET_TABLE CONTAINER 'SalesAreas' SALESAREAS.
END_METHOD.

BEGIN_METHOD CHANGEFROMDATA CHANGING CONTAINER.
DATA:
      PIADDRESS LIKE BAPIKNA101,
      PISALESORG LIKE BAPIKNA102-SALESORG,
      PIDISTRCHAN LIKE BAPIKNA102-DISTR_CHAN,
      PIDIVISION LIKE BAPIKNA102-DIVISION,
      PEADDRESS LIKE BAPIKNA101,
      RETURN LIKE BAPIRETURN.
  SWC_GET_ELEMENT CONTAINER 'PiAddress' PIADDRESS.
  SWC_GET_ELEMENT CONTAINER 'PiSalesorg' PISALESORG.
```

```
  SWC_GET_ELEMENT CONTAINER 'PiDistrChan' PIDISTRCHAN.
  SWC_GET_ELEMENT CONTAINER 'PiDivision' PIDIVISION.
  CALL FUNCTION 'BAPI_CUSTOMER_CHANGEFROMDATA'
    EXPORTING
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
      PI_DIVISION = PIDIVISION
      PI_DISTR_CHAN = PIDISTRCHAN
      PI_SALESORG = PISALESORG
      PI_ADDRESS = PIADDRESS
    IMPORTING
      PE_ADDRESS = PEADDRESS
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'PeAddress' PEADDRESS.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD SEARCH CHANGING CONTAINER.
DATA:
      PIADDRESS LIKE BAPIKNA101,
      PISALESORG LIKE BAPIKNA102-SALESORG,
      PISEARCHFLAG LIKE BAPIKNA104-SEARCHFLAG,
      PECUSTOMER LIKE BAPIKNA103-CUSTOMER,
      RETURN LIKE BAPIRETURN,
      MULTIPLE LIKE BAPIKNA103 OCCURS 0.
  SWC_GET_ELEMENT CONTAINER 'PiAddress' PIADDRESS.
  SWC_GET_ELEMENT CONTAINER 'PiSalesorg' PISALESORG.
  SWC_GET_ELEMENT CONTAINER 'PiSearchFlag' PISEARCHFLAG.
  SWC_GET_TABLE CONTAINER 'Multiple' MULTIPLE.
  CALL FUNCTION 'BAPI_CUSTOMER_SEARCH'
    EXPORTING
      PI_SEARCH_FLAG = PISEARCHFLAG
      PI_SALESORG = PISALESORG
      PI_ADDRESS = PIADDRESS
    IMPORTING
      PE_CUSTOMER = PECUSTOMER
      RETURN = RETURN
    TABLES
      MULTIPLE = MULTIPLE
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
```

```
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'PeCustomer' PECUSTOMER.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
  SWC_SET_TABLE CONTAINER 'Multiple' MULTIPLE.
END_METHOD.

BEGIN_METHOD CREATEFROMDATA CHANGING CONTAINER.
DATA:
      PIADDRESS LIKE BAPIKNA101,
      PICOPYREFERENCE LIKE BAPIKNA102,
      PECUSTOMER LIKE BAPIKNA103-CUSTOMER,
      RETURN LIKE BAPIRETURN.
  SWC_GET_ELEMENT CONTAINER 'PiAddress' PIADDRESS.
  SWC_GET_ELEMENT CONTAINER 'PiCopyreference' PICOPYREFERENCE.
  CALL FUNCTION 'BAPI_CUSTOMER_CREATEFROMDATA'
    EXPORTING
      PI_ADDRESS = PIADDRESS
      PI_COPYREFERENCE = PICOPYREFERENCE
    IMPORTING
      RETURN = RETURN
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
      PE_CUSTOMER = PECUSTOMER
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'PeCustomer' PECUSTOMER.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.

BEGIN_METHOD GETDETAIL CHANGING CONTAINER.
DATA:
      PISALESORG LIKE BAPIKNA102-SALESORG,
      PEADDRESS LIKE BAPIKNA101,
      RETURN LIKE BAPIRETURN,
      DISTRIBUTIONCHANNEL LIKE BAPIKNA102-DISTR_CHAN,
      DIVISION LIKE BAPIKNA102-DIVISION.
  SWC_GET_ELEMENT CONTAINER 'PiSalesorg' PISALESORG.
  SWC_GET_ELEMENT CONTAINER 'DistributionChannel' DISTRIBUTIONCHANNEL.
  SWC_GET_ELEMENT CONTAINER 'Division' DIVISION.
  CALL FUNCTION 'BAPI_CUSTOMER_GETDETAIL'
    EXPORTING
      PI_DIVISION = DIVISION
```

```
      PI_DISTR_CHAN = DISTRIBUTIONCHANNEL
      PI_SALESORG = PISALESORG
      CUSTOMERNO = OBJECT-KEY-CUSTOMERNO
    IMPORTING
      PE_ADDRESS = PEADDRESS
      RETURN = RETURN
    EXCEPTIONS
      OTHERS = 01.
  CASE SY-SUBRC.
    WHEN 0.            " OK
    WHEN OTHERS.       " to be implemented
  ENDCASE.
  SWC_SET_ELEMENT CONTAINER 'PeAddress' PEADDRESS.
  SWC_SET_ELEMENT CONTAINER 'Return' RETURN.
END_METHOD.
```

Eidesstattliche Erklärung

„Wir versichern, daß wir die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und uns nicht anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel bedient haben. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.“

Hamburg, den 16. Februar 1999

`--------------------`	`--------------------`
`(Jens Latza)`	`(Rüdiger Lühr)`