Universität–Gesamthochschule Siegen **Technische Informatik**
Prof. Dr.–Ing. H. Wojtkowiak Dr.–Ing. K. Quibeldey–Cirkel

Diplomarbeit

Entwurf und Implementierung einer Schnittstelle zwischen dem SAP–R/3 *Business Object Repository* und der *Open Scripting Architecture* (OSA)

Georg Odenthal
1.7.1996

Niedwiesenstraße 21
60431 Frankfurt

Frankfurt,

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der in der Informatik gern verwendete Begriff *Paradigmawechsel* zur Kennzeichnung des Übergangs hin zu einer neuen Denkweise ist für die gesellschaftlichen Auswirkungen, die durch die Entwicklungen im Bereich verteilte Systeme und Objekttechnologien entstehen, angebracht.

Die gesamte industrialisierte Welt stürzt sich zur Zeit auf eine Verheißung namens Internet, die in den Medien bis hin zur kleinsten Regionalzeitung mehr oder meist weniger fundiert ausposaunt wird. Das Ergebnis ist, daß die meisten Menschen der vernetzten Zukunft positiv oder doch zumindest offen gegenüberstehen, und daß bald auch jeder Handwerker seine *Homepage* hat, die von einem der sich allesamt auf den neuen Markt stürzenden EDV–Dienstleistern in der Rolle eines *Internet–Providers* administriert wird. Die Anzahl der "Surfer" steigt rasant (potentiell mehr als eine Millionen sind es schon in Deutschland, Motto: "Schulen ans Netz") und die gesamte Geschäftswelt steht in den Startlöchern, nur noch die Durchsetzung eines Standards für *eCash* abwartend. Derweil werden Erfahrungen gesammelt, was das Internet als Werbeplattform hergibt (die Internet–Adresse von "Ariel Ultra" ist http://www.procter.de).
Aber es passiert noch mehr: Suchmaschinen (*Web Crawler* – die derzeit schnellste ist wohl http://www.altavista.digital.com) durchsuchen alle Seiten, die auf permanent an das Internet angeschlossenen WWW–Servern liegen und erzeugen einen Volltextindex. Dieser wird ständig aktualisiert und läßt sich durch Anfragen in Sekundenschnelle nach beliebigen Stichworten durchsuchen. Wenn jemand Informationen auf einer Seite im Netz veröffentlicht, sind sie Tage später weltweit verfügbar. Der globale Informationsraum ist bereits Wirklichkeit, mit allen Konsequenzen.

Wie spannt sich nun der Bogen zu der vorliegenden Arbeit? Dazu ein Zitat:

> *"Today our industry stands at a new threshold brought on by: 1) the exponential increase of low–cost bandwidth on Wide Area Networks – for example, the Internet and CompuServe; and 2) a new generation of network–enabled, multithreaded desktop operating systems – for example, OS/2 and Windows 95. This new threshold marks the beginning of a transition from Ethernet client/server to intergalactic client/server that will result in the irrelevance of proximity. The center of gravity is shifting from single–server, LAN–based departmental client/server to a post–scarcity form of client/server where every machine on the global "information highway" can be both a client and a server."* [Orfali et al., 19976, S.3f]

Das Buch, aus dem das Zitat stammt, stellt die momentan marktfähigen Infrastrukturen für verteilte Objekte (*Distributed Objects*) vergleichend dar. Eines dieser Konzepte ist *OpenDoc* und eine Komponente von *OpenDoc*, die *Open Scripting Architecture* (OSA) ist Teil meiner Arbeit. Diese Infrastrukturen werden in Zukunft große Bedeutung erlangen, was es angeht das Internet mit Leben zu füllen und es transparent benutzbar zu machen (weiteres hierzu in KapiteB).

Die Firma SAP als einer der Global Player im Software–Sektor hat die Verbreitung der Client–Server–Technologie ganz wesentlich mitbefördert. Durch die Größe ihrer Kunden war SAP schon immer mit der Anforderung der weltweiten Vernetzung konfrontiert und ist momentan dabei, auch das Internet und die tangierenden Technologien in ihre Systeme zu integrieren. Im Rahmen einer Evaluation habe ich einen Mittler zwischen der angesprochenen *Open Scripting Architecture* und einer objektorientierten Kommunikationskomponente von SAP–R/3, dem *Business Object Broker* entwickelt. Ziel war es, die Machbarkeit dieser Verbindung zu zeigen und Erfahrungen zu sammeln (weiteres hierzu in Kapitel2).

Die Infrastruktur der Verteilten Systeme und die ihnen das Leben einhauchende Objekttechnologie gehorchen dem Schlüssel–Schloß–Prinzip: Die Objekte haben den endgültigen Durchbruch ihrer Ähnlichkeit zum aus Knoten und Assoziationen bestehenden Netz zu verdanken; die den Netzen inhärente Komplexität erfordert neue Methoden der Softwareentwicklung.
Hier liegt dann auch der Schwerpunkt der Arbeit: die Software–Dokumentation mit Entwurfsmustern und die Anforderungsdefinition an entsprechende Entwicklungsumgebungen. Die Entwicklung und Implementation des OSA–Servers wurde aus dieser Sicht zum Vehikel für die Verifikation der Dokumentationskonzepte am praktischen Beispiel (einleitend hierzu meine im Anhang D angefügte Ausarbeitung zum Entwurfsmusterseminar und weiterführend in Kapitel 4).

Zurück zum Paradigmawechsel: Objekte und Netze werden das Leben verändern – mehr als es die gesamte Industrialisierung bisher vermocht hat. Das ist die eigentliche Auswirkung der neuen Denkweise. Sie gibt ihr den entsprechenden Stellenwert.

Eine Anleitung zum Querlesen: Die Abschnitte
2, 2.2, 2.2.2 und 2.3.7 für SAP–R/3,
3, 3.1 und 3.2 für OSA und
4, 4.1, 4.3 und 5 für die Entwurfsmusterdokumentation
geben jeweils Überblicke über die Themen.

2 Überblick SAP–R/3

Die erst 20 Jahre alte deutsche Firma SAP AG (einst: Systemanalyse und Programmentwicklung, heute: *Systems, Applications, Products in Data Processing*), das größte unabhängige europäische Softwarehaus, hat mit dem 1992 erstmals angebotenen System R/3 (das 'R' steht für *Realtime*) die Weltmarktführerschaft im Bereich betriebswirtschaftlicher Standardanwendungen erlangt. Der Erfolg ist wohl damit zu begründen, daß neue Entwicklungen in diesem Bereich rechtzeitig erkannt wurden und den Kunden ein zukunftsweisendes System angeboten wird.
Die Kunden von SAP sind bisher überwiegend Großkonzerne, der Erfolg von R/3 in mittelständischen Firmen ist bisher nicht durchschlagend. Die Gründe hierfür sind vielschichtig aber vor allem wohl in den hohen Systemanforderungen und dem Administrationsaufwand zu suchen.

Die wichtigsten Stichworte bezüglich R/3 sind:

- **Vollständige Infrastruktur** für die betriebliche Informationsverarbeitung. Ausgereifte betriebliche Standardanwendungen (z.B. Finanzbuchhaltung, Personalwirtschaft, Logistik), Werkzeuge für die Einführung (*Customizing*, Referenzmodelle), Steuerung und Überwachung des Systems.
- **Client/Server–Prinzip** und ***Three Tier Architecture***. R/3 wird meist in einer dreistufigen Konfiguration betrieben, die aus zentralen Datenbankservern, Applikationsservern und Präsentationsrechnern besteht. Der Benutzer greift über die Abarbeitung von Dialogen, die auf einem Präsentationsrechner angezeigt werden auf ggf. mehrere Applikationsserver zu, die jeweils wieder auf mehrere Datenbankserver zugreifen können. Die Systembelastung läßt sich statisch (Zuordnung von Anwendungen auf bestimmte Applikationsserver) oder dynamisch verteilen.
- **SAP–GUI**. (SAP–*Graphical User Front End*) Mit dem so benannten graphischen Front End wird die Ein–/Ausgabe auf den Präsentationsrechnern realisiert. Durch den SAP–GUI ist eine einheitliche Bedienung von R/3 auf den verschiedenen Hard–/Software–Plattformen gegeben.
- **Heterogene Systemplattformen**. R/3 ist auf unterschiedlichen Hard– und Software–Plattformen verfügbar. Das ist auch ein Grund für die gute Skalierbarkeit des Gesamtsystems. Bezüglich der Datenbank ist R/3 auf relationale Systeme beschränkt.
- **Offenheit**. Durch die Unterstützung wichtiger Standards ist die leichte Integration in vorhandene Systeme und die Erweiterbarkeit um kundenspezifische Lösungen gegeben. Wichtige Standards hierbei sind: TCP/IP, CPI–C, SQL, ODBC, OLE, X.400/X.500, MAPI, EDI.
- **Modellierung von Geschäftsprozessen** (***Business Process Modeling***). Durch die Bereitstellung von Softwarekomponenten, die sich an betriebswirtschaftlichen Abläufen (*Workflows*) und nicht an festen Funktions– oder Aufgabenzuweisungen orientieren, beschreitet SAP mit R/3 den Weg, den heute viele Unternehmen gehen bzw. gehen wollen, um auf die Anforderungen nach immer kostengünstigeren Produktionswegen mit immer kürzeren Produktzyklen reagieren zu können.
- **Internationalität**. R/3 ist durchgängig mehrsprachig. Bei der Anmeldung am System kann der Anwender die von ihm gewünschte Sprache wählen und erhält in der anschließenden Sitzung alle textuellen Ausgaben in dieser Sprache.
- **Verteilung**. Durch die Client–/Server–Architektur ist auch eine Verteilung in *Wide Area Networks* (WAN) möglich. Mit der ab R/3 3.0 eingeführten SAP–eigenen ALE*–Technologie (*Application Link Enabling*) können verteilte R/3–Systeme untereinander kommunizieren. Für weltweit operierende Firmen ist so der Zugriff auf eine konsistente Informationsbasis möglich.

Einen aktuellen Überblick über R/3 bietet ein Artikel in der Zeitschrift c't 2/96 [Born, 1996].

2.1 Die Systemarchitektur von R/3

Um eine Einordnung des B*usiness Object Repositories* (BOR, Abschnitt 2.3) als der für meine Arbeit wesentlichen Komponente des R/3–Systems zu ermöglichen, möchte ich eine kurze Übersicht über die Systemarchitektur von R/3 geben. Diese Beschreibung bezieht sich im Wesentlichen nur auf die unmittelbar für das BOR wichtigen Komponenten und erhebt nicht den Anspruch auf Vollständigkeit.
Die Forderung nach weitestgehender Plattformunabhängigkeit für R/3 machte es notwendig, eine gemeinsame Basis zu schaffen, auf der die nötige Infrastruktur zur Realisierung betriebswirtschaftlicher Applikationen bereitgestellt werden kann. Plattformeigenschaften beziehen sich in diesem Zusammenhang auf spezifische Hardware–Eigenschaften, die unterstützten Betriebssysteme und Datenbanken.
Diese Basis muß wiederum mit den Diensten aus der gemeinsamen Schnittmenge aller Plattformen auskommen, bzw. fehlende Dienste einer Plattform eigenständig bereitstellen. Die Basis ist vollständig in C implementiert und stellt der darüberliegenden Schicht die notwendigen Dienste (z.B. verallgemeinerter Datenbankzugriff und Präsentation) zur Verfügung.
Auf die Basis setzt die Sprachkomponente ABAP/4* auf. ABAP/4 ist eine sog. 4GL, die sich durch eine höhere Abstraktion und mächtige Funktionalität von konventionellen Programmiersprachen abhebt. Alle Applikationen des Systems R/3 sind vollständig in ABAP/4 kodiert und bedienen sich über die Sprachelemente von ABAP/4 der Dienste der Basis. Die Anwendungsentwicklung findet typischerweise in der *ABAP/4 Development Workbench* statt, einer integrierten Umgebung, die alle Werkzeuge für die Entwicklung bereitstellt. Wesentlich wäre die Datenmodellierung mit erweiterten ER–Diagrammen, daß Repository zur Verwaltung aller Entwicklungsdaten, Debugger, Testtools, Wartung und Tuning zu nennen.
Durch die beschriebene Architektur ist es möglich, daß R/3 auf einer großen Anzahl sehr unterschiedlicher Plattformen abläuft, daß das System extrem skalierbar ist und die Applikationsentwicklung und die Anpassung schnell und kostengünstig ablaufen kann.

2.1.1 Das Transaktionskonzept

Die bekannte Definition der Transaktion als der Überführung eines Systems von einem konsistenten Zustand in einen anderen, ebenfalls konsistenten Zustand (ACID–Prinzip, siehe [Vossen, 1994, S.449f]) wird in R/3 erweitert.
Eine wichtige Eigenschaft des R/3–Systems ist die Kommunikation mit dem Anwender über Dialoge. Anhand der Abarbeitung einer Dialogfolge definiert der Anwender die für eine Transaktion nötigen Daten.
Folgende Entsprechungen bestehen zwischen den allgemeinen Begriffen und der SAP–Terminologie:

- Die Zeit, in der eine Applikation aktiv ist, entspricht einer SAP–Transaktion.
- Eine SAP–Transaktion besteht aus einer Sequenz von sogenannten *Logical Unit of Work* (LUW). Als Beispiel für eine LUW sei die Erfassung eines Auftrags genannt.
- Eine LUW besteht aus einer Sequenz von Dialogschritten; alle Aktionen einer LUW werden zwischengespeichert. Variablen können für die Dauer einer LUW auf der Datenbank gesperrt werden.
- Die abschließende Bestätigung einer LUW (*Commit*) löst eine Folge von Transaktionen auf der Datenbank aus.

Die wohl wichtigste Eigenschaft dieses Konzepts ist, daß die Transaktion auf der Datenbank am Ende einer Folge von Dialogschritten steht, die sich über einen prinzipiell beliebigen Zeitraum erstrecken können. Je länger eine LUW dauert, desto größer wird die Wahrscheinlichkeit, daß angezeigte Informationen nicht mehr aktuell sind, da sie mittlerweile durch andere Benutzer

verändert wurden. Die vollständige Pufferung einer LUW ist also ein Kompromiß zwischen Aktualität der angezeigten Informationen und der schlichten Voraussetzung, komplexe Transaktionen mit Dialogen überhaupt realisieren zu können. In den Bereichen, in denen R/3 eingesetzt wird, ist der Aktualitätsaskpekt somit dem Wunsch und der Notwendigkeit nach der Bereitstellung komplexer Transaktionen untergeordnet.
Die Steuerung und Pufferung der LUWs und die Abarbeitung der durch eine bestätigte LUW ausgelösten Transaktionen auf der Datenbank werden von Dialog– und Updateprozessen, die in der Gesamtheit als *Workprozesse* bezeichnet werden, erledigt. Die *Workprozesse* laufen auf den Applikationsservern.

2.2 Der SAP *Business Workflow*

An den Anfang möchte ich eine zusammenfassende Aussage des für den Workflow–Bereich zuständigen Entwicklungsleiters Dr. Fritz [Fritz, 1995] stellen:

> „Workflow–Management ist ein wichtiges Werkzeug zur Gestaltung und Optimierung von Geschäftsprozessen im Rahmen des Einsatzes betriebswirtschaftlicher DV–Anwendungen und geeignet, das Business Process Reengineering in der praktischen Umsetzung zu unterstützen. Kernleistungen von Workflow–Management sind die **Automatisierung des Informations– und Prozeßflusses**, die aktive **Verknüpfung von Arbeitsschritten** und die flexible **Implementierung organisatorischer Strukturen**. Im Zusammenwirken mit Standard–Anwendungssoftware kommt es darauf an, deren Nutzen– und Integrationspotential zu verstärken und zu ergänzen. Besonders augenfällig wird das im Bereich des elektronischen Datenaustauschs (EDI) und der intelligenten Bearbeitung von Ausnahmesituationen.“

In dieser einleitenden Zusammenfassung werden viele wichtige Stichworte, die für den Work-flow auszeichnend sind genannt.
„Gestaltung und Optimierung von Geschäftsprozessen“ spricht ein Manko an, das nicht nur für den Bereich der Software–Entwicklung gilt: Die organisatorischen Abläufe in Betrieben und in der Verwaltung sind meist nicht dokumentiert und entziehen sich somit einer analytischen Betrachtung. Die Entscheidung, Standard–Software zur Steuerung der Abläufe in eine Firma oder eine Verwaltung einzuführen, geht meist damit einher, die Arbeitsabläufe grundlegend neu zu strukturieren (*Business Process Reengineering* [SAP_4, 1995]).
Durch die Verwendung von Workflow–Systemen wird zunächst eine Bestandsaufnahme der bestehenden Strukturen gemacht. Anhand der so vorliegenden Visualisierung des Ist–Zustands und der im Zitat nicht erwähnten Bibliothek vordefinierter Workflow–Muster ist die Formulierung eines Soll–Zustands möglich. Diese Tätigkeiten sind Teil des Vorgehensmodells zur R/3–Einführung. Allgemein formulierte Ziele nach einer erfolgreichen Einführung sind die oben fett gedruckten Aussagen.
Zu den oben beschriebenen allgemeinen Eigenschaften möchte ich noch die Offenheit und Integrationsfähigkeit des Workflow–Systems hinzufügen, die in Zukunft sicher eine große Rolle spielen wird. Es stehen Schnittstellen zur *Erzeugung und Bearbeitung von Workflows* zur Verfügung, so daß externe Workflow–Manager angekoppelt werden können. Die Schnittstelle für einen *externen Eingangskorb* ermöglicht die Ankopplung von Fremdsystemen für die Übertragung von Nutz–Daten, die zur Bearbeitung durch Workflows bereitgestellt werden. Auch die Weiterleitung im System generierter Ereignisse (siehe 2.2.1) und der Empfang von externen Ereignissen ist über eine Schnittstelle möglich, so daß systemübergreifende Workflows definiert werden können.

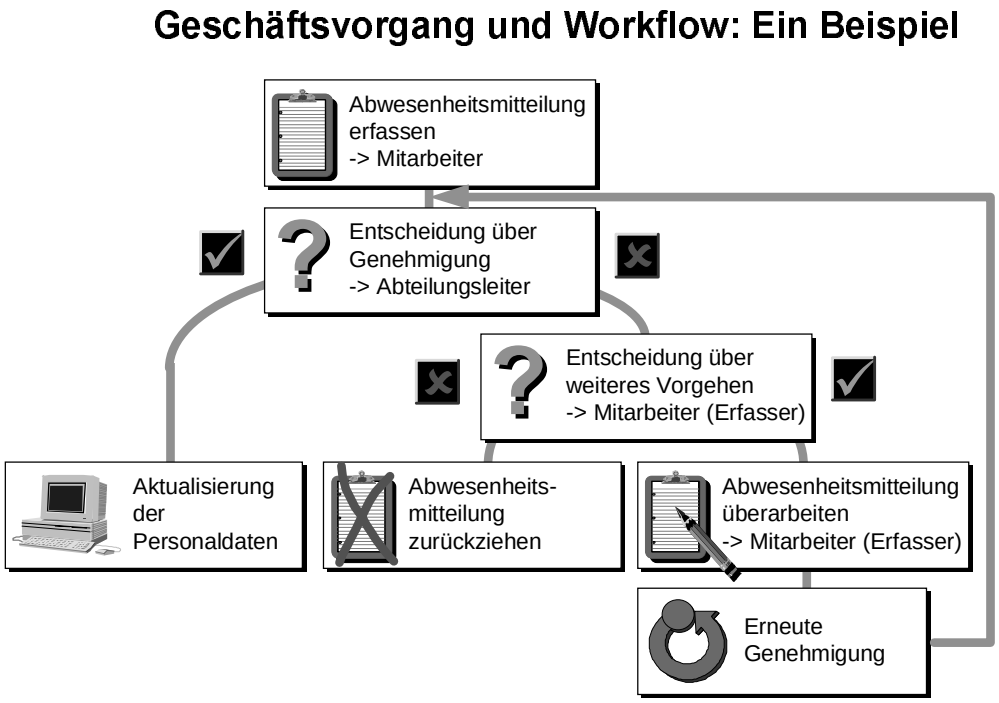


Abbildung 1: Der Workflow einer Abwesenheitsmitteilung [SAP_3, 1995]

Als Beispiel habe ich einem Tutorial über Workflow–Definition [SAP_3, 1995] die als Grafik in Abbildung 1 dargestellte Abwesenheitsmitteilung entnommen.
Der dargestellte Ablauf hat folgende Bedeutung: Das zugrundeliegende Szenario beginnt mit dem Ausfüllen eines Urlaubsantrags durch einen Sachbearbeiter. Im Anschluß wird das ausgefüllte Formular automatisch an den Vorgesetzten des Sachbearbeiters weitergereicht. Genehmigt dieser den Antrag, so erhält der Sachbearbeiter eine Benachrichtigung und der Workflow kann beendet werden (Auf die Einbeziehung einer Verbuchung wird verzichtet). Wird der Antrag vom Vorgesetzten abgelehnt, kann der Sachbearbeiter zunächst entscheiden, ob er den Antrag überarbeiten oder zurückziehen möchte. Entscheidet sich der Sachbearbeiter für eine Überarbeitung, so wird das Antragsformular automatisch in seinen Eingangskorb weitergeleitet und der Vorgang beginnt erneut.

2.2.1 Internas

Zunächst möchte ich die unter dem Begriff *SAP Business Workflow* versammelten Komponenten noch etwas näher beleuchten. Hierzu ein Zitat aus einem anderen Dokument [Berthold, 1995]:

> Der **SAP Business Workflow** umfaßt Technologien und Werkzeuge zur automatisierten Steuerung und Bearbeitung von anwendungsübergreifenden Abläufen.
> Im Vordergrund stehen dabei die Koordination
>
> - der beteiligten Personen
> - der anfallenden Arbeitsschritte
> - der dabei zu bearbeitenden Daten (Geschäftsobjekte)
>
> **Wesentliche Ziele sind die Verringerung der Durchlaufzeiten und der Kosten der Geschäftsprozeß–Abwicklung sowie eine Steigerung der Transparenz und Qualität.**

Auch hier sind wieder wichtige Stichworte enthalten: „automatisierte Steuerung und Bearbeitung von anwendungsübergreifenden Abläufen“. In Abschnitt 2.1.1 habe ich das Transaktionskonzept von R/3 beschrieben. Der Anwender initiiert durch die Bedienung einer Applikation eine Folge

von LUWs. Die Ausführung einer Workflow–Definition im Workflow–Manager produziert auch eine Sequenz von LUWs und ist so mit einem Makro vergleichbar.

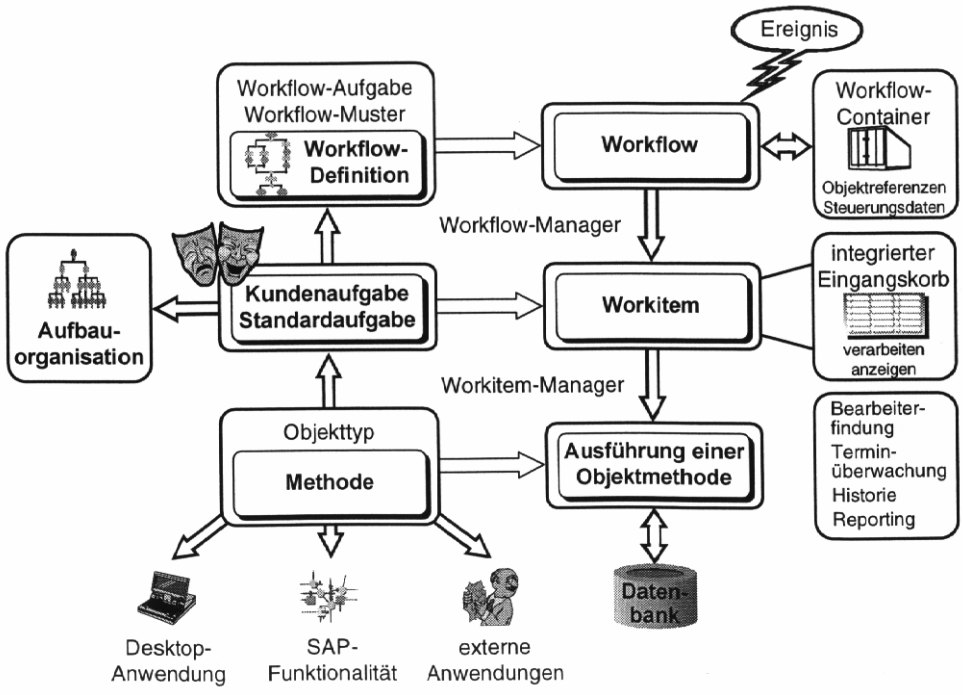


Abbildung 2: Konzeptionelle Grundlagen des SAP *Business Workflow* [SAP_3, 1995]

Aber die Workflow–Sicht ermöglicht noch mehr: „Koordination der beteiligten Personen, der anfallenden Arbeitsschritte und der dabei zu bearbeitenden Daten“: An einem durch eine Workflow–Definition spezifizierten Geschäftsprozeß können mehrere Personen und somit Arbeitsschritte und Daten nacheinander oder auch konkurrierend beteiligt sein (Beispiel: Ein Genehmigungsvorgang, der über mehrere Instanzen läuft).

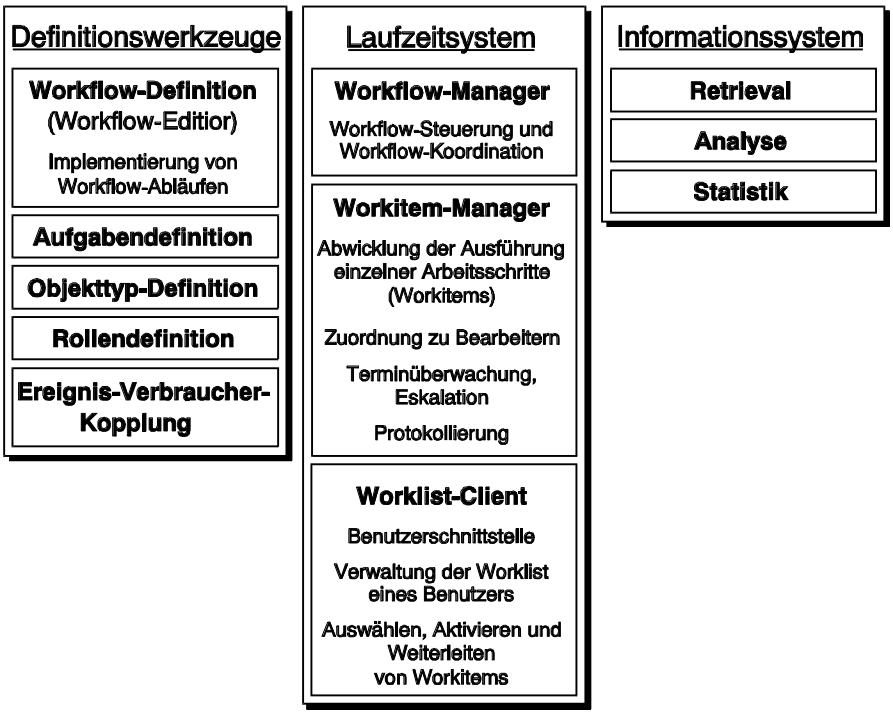


Abbildung 3: Die Komponenten des *SAP Business Workflow* [SAP_3, 1995]

Der SAP *Business Workflow* ist eine Repository–Basierte Entwicklungsumgebung, in der *Aufgaben*, *Rollen, Objekttypen* und darauf aufbauend *Workflows* graphisch–interaktiv definiert werden können. Zur Ausführung der Workflows steht ein Laufzeitsystem zur Verfügung, das durch ein Informationssystem zur Überwachung und Auswertung ergänzt wird.
Im Einzelnen ist das:

- Ein **Laufzeitsystem** mit den Ausführungskomponenten
  - zur Steuerung und Koordination des Workflow–Ablaufes (Workflow–Manager),
  - zur Abwicklung der Ausführung einzelner Arbeitsschritte (Workitems) einschließlich der Zuordnung zu Bearbeitern und der Terminüberwachung (Workitem–Manager),
  - zur Anzeige und Verwaltung der Worklist eines Benutzers (integrierter Eingangskorb).
- Die **Definitionswerkzeuge** mit den Komponenten
  - zur Implementierung einer Workflow–Definition (graph. Workflow–Editor),
  - zur Beschreibung von Aufgaben mit der Möglichkeit, Arbeitsschritte an Personen oder Gruppen zu adressieren (Aufgabendefinition im Aufgabenkatalog des Organisationsmodells),
  - zur Definition von Objekttypen mit Methoden, Attributen und Ereignissen, auf denen die Aufgaben operieren (Objekttyp–Definition im Workflow–Objekttypkatalog).
- Das **Informationssystem** mit seinen Möglichkeiten eines leistungsfähigen *Reportings* über den Workflow.
- Ein **Repository**, das alle anfallenden Daten verwaltet und die Konsistenz zwischen den verschiedenen Werkzeugen und Komponenten gewährleistet.

Wichtige inhaltliche Einheiten:

- **Informationen**. Werden bei der Abarbeitung eines Workflows zwischen den Aktionen transportiert (Datenfluß). Hierfür wir eine als *Container* bezeichnete Datenstruktur verwendet, die durch eine homogene dynamische Liste realisiert ist. Der Eingangskorb einer Person, die Arbeitsaufträge aus ablaufenden Workflows erhalten kann, ist ebenfalls ein Container.
- **Akteure**. Aus der Sicht der Workflow–Definition stellen die Mitarbeiter einer Firma Akteure dar, die mit unterschiedlichen Rechten und Fähigkeiten ausgestattet sind und an der Abarbeitung eines Workflows durch eine (oder ggf. mehrere) **Rollen** beteiligt sind. (Eine allgemeinere Sicht könnte alle zur definierten Reaktion auf Anforderungen fähigen „Komponenten eines Systems“ als Akteure betrachten...).
  Beispiele für Akteure sind: Sachbearbeiter, Abteilungsleiter. Die Rollen, die ein Akteur einnehmen kann, hängen von den Aufgaben ab: So ist der Abteilungsleiter z.B. in der Rolle des Gehaltsempfängers, wenn es um die Aufgabe der Gehaltsabrechnung geht und in der Rolle des Personalverantwortlichen, wenn es um die Gehaltserhöhung eines Sachbearbeiters geht.
- **Objekttypen**. Die Objektmethoden eines Objekttyps kapseln SAP–Transaktionen, Funktions– und Dialogbausteine, also ggf. vollständige Applikationsaufrufe mit Benutzerdialogen. Objekttypen werden im Objekttyp–Katalog definiert. Außer der SAP–Funktionalität werden dort auch externe Objekttypen, wie z.B. OLE–Objekte und MSWord– oder MSExcel–Dokumente beschrieben. Beispiele für Objekttypen aus dem SAP–Umfeld sind: Auftrag, Rechnungsbeleg, Material.
  Die Definition eines Objekttyps umfaßt Attribute (Objekteigenschaften) – sie dienen zur Formulierung von Bedingungen und somit zur Steuerung des Workflows, der Methoden (Operationen auf Objekten) und Ereignisse (Signalisierung von Zustandsänderungen eines Objekts). Ereignisse können Daten transportieren (Container) und dienen zur Auslösung von Folgeaktionen (i. A. Start eines weiteren Workflows) oder zur Benachrichtigung anderer Prozesse.

- **Aufgaben**. Eine Aufgabe (auch als Schritt oder Aktion in der Workflow–Definition bezeichnet) stellt die Beziehung zwischen einer Tätigkeit und einem Bearbeiter her.

Exkurs – Workflow–Aufgabe und Methodenaufruf:

Man muß zwischen der Situation zur Laufzeit und zur Definitionszeit unterscheiden:
Die Definition einer Aufgabe umfaßt die Zuordnung eines Objekttyps und genau einer Methode aus der Schnittstelle als *Tätigkeit*. Desweiteren wird ein *Bearbeiterkreis* zugeordnet. Dabei kann entweder jeder Mitarbeiter als potentieller Bearbeiter eine bestimmte Gruppe von Bearbeitern oder genau eine Person für die Bearbeitung zugelassen werden.
Zur Laufzeit (d.h. bei der Abarbeitung eine Workflows durch den Workflow–Manager) wird ein Bearbeiter aus der Menge der möglichen Bearbeiter ausgewählt (*Rollenauflösung*) und eine Instanz des Objekttyps erzeugt. Die Objektreferenz zusammen mit dem Namen der auszuführenden Methode und ggf. schon feststehenden Parametern wird als *Workitem* bezeichnet und in den Eingangskorb des ausgewählten Bearbeiters gestellt. Die Bearbeitung durch den Bearbeiter besteht in der Vervollständigung der evtl. noch fehlenden Übergabeparameter und dem anschließenden Ausführen der Methode auf dem durch die Objektreferenz übergebenen Objekt.

Eine Aufgabe beschreibt eine Aktion, wie sie real in einem betrachteten System auftritt und als Teil eines Workflows modelliert werden soll. Ein vollständiger Workflow, wie z.B. die in Abbildung 1 dargestellte Abwesenheitsmitteilung wird als Mehrschrittaufgabe bezeichnet.
Die Festlegung eines bestimmten Bearbeitertyps (z.B. Abteilungsleiter) wirkt sich bei der Abarbeitung eines Workflows aus. Die durch die Aufgabe initiierte Aktivität wird nur Bearbeitern mit dem festgelegten Typ zugeordnet.
Eine Mehrschrittaufgabe kann wiederum Teil einer vollständigen Workflow–Definition sein (Sub–Workflow).
Beispiele für Aufgaben sind: Antrag stellen, über Antrag entscheiden.

- **Ereignisse** werden als Bestandteile von Objekttypen im Objektrepository definiert und beschreiben die Zustandsänderungen von Objekten. Zum derzeitigen Stand findet die Laufzeitverwaltung im Workflow–Ereignismanager statt.
  Ereignisse sind ein wichtiger Bestandteil der Workflow–Definition. Meist wird der Start eines Workflows durch ein eintreffendes Ereignis ausgelöst und das Ende ist durch die Erzeugung eines Ereignisses gekennzeichnet.
  Beispiele für Ereignisse sind: Rechnung erfaßt, Urlaubsantrag genehmigt, Material eingetroffen.
- **Workflow–Definitionen**, bestehen aus Aufgaben, Ereignissen, und Kontrollstrukturen (siehe Abbildung 4).
  Die Workflow–Definition ist vergleichbar mit einem Programmablaufplan [Duden, 1989, S.458], der um einige Konstrukte erweitert wurde. Als wesentlich wäre hierbei zu nennen: *Nebenläufigkeit* von Prozessen, *Ereignissteuerung*, Steuerung durch die Auslösung von *Eskalationsaktivitäten*, Definition von *Rollen*.

Eine vertiefende Beschreibung der Komponenten und Inhalte würde im Kontext der Arbeit zu weit führen; ich verweise auf die entsprechenden Stellen im Literaturverzeichnis (Seite 71).
Abschließend die Bildschirmhardcopy des grafischen Workflow–Editors mit einem nachträglich kommentierten Ausschnitt aus dem Workflow Abwesenheitsmitteilung.

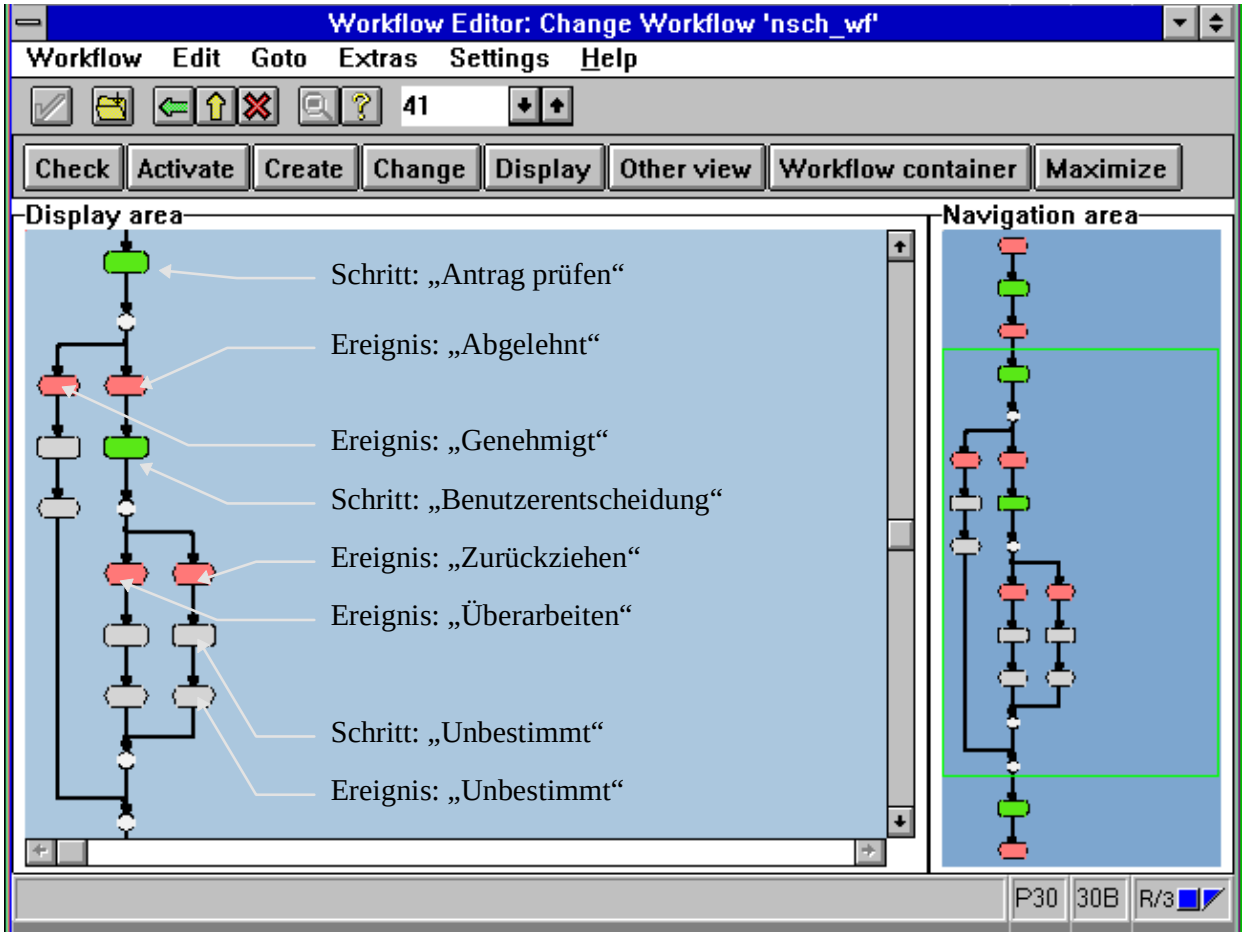


Abbildung 4: Bildschirm–Hardcopy des Workflow–Editors

2.2.2 Die Rolle des *Business Object Repositories*

Wichtig im Zusammenhang mit dem Inhalt meiner Arbeit ist, daß durch das Workflow–Konzept eine *Metasicht* auf die funktionalen Einheiten und die Daten, die im R/3–System (und durch die Schnittstellen nach außen sogar systemübergreifend) verwaltet werden, möglich wird. So kann der Weg eines modellgestützten *Customizings* (Anpassung) bei der Einführung von R/3 beschritten werden. Voraussetzung hierfür ist die Denkwelt der *Objekte*, was Zusammengehörigkeit von Funktionalität und Daten sowie statische und dynamische Beziehungen angeht. Durch das *Business Object Repository* wurde dem auf relationaler Datenbanktechnologie und strukturierten Methoden basierenden System der Objektmantel umgehängt und in zweiter Linie so die Voraussetzung für die Kopplung von R/3 mit anderen Interoperabilitäts–Technologien geschaffen.
Aus einer allgemeinen Sicht gehen die auf den ersten Blick so unterschiedlichen Konzepte Workflow und OLE/CORBA/OpenDoc in sehr ähnliche Richtungen: Es geht um die Definition von Prozessen, die sich an den zu bearbeitenden Daten und den beteiligten Akteuren orientieren (und nicht mehr an funktionalen oder administrativen Einheiten).
Die Definition muß durch geeignete Werkzeuge, die in die Klasse der Software–Entwicklungsumgebungen fallen, unterstützt werden. Der SAP *Business Workflow* stellt eine solche Umgebung dar; ein anderes Beispiel ist das *ARIS–Toolset* [Scherhuhn, 1996]. Allgemein werden solche Werkzeuge als *Upper–Case–Tools* (ebd.) bezeichnet.
Desweiteren ist ein geeignetes Laufzeitsystem, das eine einheitliche Integration in Bedieneroberflächen realisiert und die Überwachung des Prozeßablaufs ermöglicht nötig. Derzeit bietet einzig SAP mit dem Business Workflow eine Umgebung an, die die Definition von Workflows *und* die Laufzeitumgebung in einem einheitlichen Konzept verbindet.

Um die SAP R/3–Funktionalität aus Workflow–Sicht mit der Objektbrille sehen zu können, war die Definition von *Wrappern* (vgl. [Gamma, 1992, S.120]) nötig, die jeweils einen semantisch und/oder kontextbezogen zusammengehörigen Teil der Funktionen und Daten aus R/3 als Objekt(typ) einkapseln. Der Begriff Wrapper verdeutlicht an dieser Stelle, daß die Objekte (nur) eine weitere Abstraktionsebene einführen und keine neue Funktionalität oder neue Attribute, die nicht auch ohne die Objekte bereits vorhanden wären.
Das Wrapper–Konzept ist ein pragmatisches Vorgehen zur Integration vorhandener Nicht–Objektorientierter Komponenten in Umgebungen, die von der Struktur und vom Zugriff her objektorientiert sind.
Die Definition, Verwaltung und der Zugriff auf Objekte wird durch das *Business Object Repository* geregelt. Wie aus obiger Aufstellung schon deutlich wurde ist das BOR als Teil des SAP *Business Workflow* zu sehen. Unter die auf Seite 5 beschriebene Offenheit und Integrationsfähigkeit von R/3 ist die ORB–Schnittstelle des *Business Object Repositories* einzuordnen, auf der dann Interoperabilitätskonzepte wie OLE–*Automation* oder OSA aufgesetzt werden können.
Objekte sind in der Workflow–Definition als Einheit nicht sichtbar. Die Dienstleistung des Repositories an den Workflow besteht lediglich in der Bereitstellung der Referenz eines Objekts und der Adresse jeweils genau einer Objektmethode (siehe Exkurs auf Seite 12), die zur Abarbeitung des Workflows aufgerufen wird.
Die weiteren Möglichkeiten, die das Objekt Repository bietet, werden zum derzeitigen Stand des Workflows nicht genutzt. Unter anderem sind das:

- Navigation durch die Laufzeitstruktur mittels der Beziehungen zwischen den Objekten (Abschnitt 2.3.2),
- Modellierung von Protokollen in Objekttyp–Schnittstellen, die aus mehr als einem Methodenaufruf bestehen.

Der OLE–Server und der in dem Rahmen dieser Arbeit entstandene OSA–Server machen das volle Potential des *Business Object Repositories* verfügbar.

2.3 Das *Business Object Repository*

Das *Business Object Repository* (kurz BOR) ist die Verwaltungsinstanz für *Business Objekte* (Geschäftsobjekte).
Business Objekte sind zum momentanen Entwicklungsstand des Repositories ausschließlich *Wrapper* (vgl. [Gamma, 1992, S.120]) um vorhandene Funktionalität (SAP–Transaktionen (Abschnitt 2.1.1), Funktionsbausteine*) und Daten aus Tabellen der relationalen Datenbankschemata. Der Zugriff auf die unterliegende Funktionalität und Daten ist durch die Objektschnittstelle deutlich vereinfacht und generalisiert; ein Migrationsweg „von oben“ hin zu einem vollständig objektorientierten System ist möglich; die Integration in andere objektorientierte Umgebungen, wie z.B. OLE von Microsoft [Brockschmidt, 1995] ist Realität.
Teilt man das BOR in Funktionskomponenten auf, lassen sich folgende Einheiten identifizieren:

- Datenverwaltung der *Business Objekte* im R/3.
- Definition von Objekttypen.
- Navigation durch die statische Struktur (Vererbungsrelation siehe Abschnitt2.3.2).
- *Business Object Broker* als aktive Komponente, die die Erzeugung und den Zugriff auf die Instanzen erlaubt.

Das BOR ist durch die Anforderungen aus dem *SAP Business Workflow* (Abschnitt 2.2) entstanden. Während der weiteren Entwicklung erkannte man, daß es sich auch als Basis einer Interoperabilitätsschnittstelle (Abschnitt 2.3.4) für R/3 nutzen läßt. Die hierfür notwendige aktive Schnittstellenkomponente des BOR, der *Business Object Broker* (kurz BOB genannt), wurde nach den Prinzipien eines *Dynamic Invocation Interface** (DII), wie es von OLE und CORBA her bekannt ist ([Brockschmidt, 1995], [Orfali et al., 1996]), entwickelt.

Im folgenden wurde dann ein OLE–Server implementiert, der auf der Funktionalität der Broker–Schnittstelle des *Repositories* aufsetzt und den Zugriff von OLE–Fähigen Applikationen auf R/3 ermöglicht. Als Beispiel in Abbildung 5 ein Visual Basic–Programm, das sich der Funktionalität des OLE–Servers bedient.
Die vorliegende Arbeit ist aus der Idee heraus entstanden, außer OLE auch den Interoperabilitäts–Standard OSA zu unterstützen.

```
Dim R3 As Object
Dim AuftragFactory As Object
Dim Auftrag As Object

Set R3 = CreateObject('SAP.BusinessObjectFactory')
R3.Logon()

Set AuftragFactory = R3.CreateSAPObject('Kundenauftrag')
Set Auftrag = AuftragFactory.Insert('4711','17.01.96','Müller', ...)
Set Position1 = Auftrag.AddPosition('Nähmaschine', 5, ...)

R3.Logoff()
```

Abbildung 5: Visual Basic–Script zur Ansteuerung des OLE–Servers [SAP_1, 1996]

Das Objektmodell des *Business Object Repositories* orientiert sich ausdrücklich an den Standards COM/OLE 2.0 ([Brockschmidt, 1995]) von Microsoft und der herstellerunabhängigen Definition CORBA [OMG, 1991] der OMG (*Object Management Group**). Im folgenden eine Beschreibung der Objekttyp–Definition und der zwischen den Objekten definierten Beziehungen.

2.3.1 Die Objekttyp–Definition

Die vier Elemente einer Objekttyp–Definition sind **Identifikation, Attribute, Methoden** und **Ereignisse**.
Alle Elemente einer Objekttyp–Definition unterliegen einer strengen Versionskontrolle, die eine aufwärtskompatible Entwicklung ermöglichen. Die mit einem *Release** von R/3 freigegebenen Elemente können in nachfolgenden *Releases* weder gelöscht noch umbenannt werden.
Im folgenden eine Beschreibung der Komponenten, die sich eng an die Darstellung in [Krane_2, 1995] hält:

- **Attribute** beschreiben Eigenschaften von Objekten, wie z.B. Bezeichnung, Datum der Erfassung bzw. Änderung, Status, usw. Als Datentypen werden hier alle elementaren Datentypen des *ABAP/4 Dictionary** unterstützt, indem auf entsprechende Tabellenfelder als Referenzen verwiesen wird. Ebenfalls werden Sequenzen elementarer Datentypen unterstützt, z.B. eine Aufzählung von Werten.
  Attribute können aber auch den Typ einer beliebigen Objekt–Referenz annehmen, wie z.B. bei einem Kundenauftrag, der Erfasser (SAP Benutzer), der Auftraggeber (Kunde), die Verkaufsorganisation oder die Auftragspositionen (Sequenz von Auftragspositionen). Durch diese wichtige Eigenschaft des Repositories lassen sich Laufzeitbeziehungen zwischen Objekten abbilden.
  Es ist derzeit nicht möglich, heterogene Sequenzen oder strukturierte Datentypen als Typ eines Attributs zu definieren.
  Alle Attribute sind direkt lesend zugreifbar. Das Setzen von Attributen erfolgt über Methoden, in denen die nötigen Prüfungen vorgenommen werden. Im BOR erfolgt die Implementierung lesender Zugriffe über ABAP/4–Datenbankzugriffe bzw. Funktionsbausteine.

Auf Attribute und Schlüsselfelder als besondere Attribute eines Objekts kann man über den *Business Object Broker* direkt zugreifen, sofern sie in der Objekttypbeschreibung als exportierbar verzeichnet sind.

- Die **Identifikation** beschreibt, wie die Objekte eines Objekttyps persistent identifiziert werden. Die Identifikation wird über ggf. mehrere ausgezeichnete Attribute realisiert.
  Die persistente Identifikation eines Objektes ist dreidimensional und umfaßt:
  - Logisches System,
  - Objekttypname,
  - Typspezifische Schlüssel.

  Das *Logische System* identifiziert ein System innerhalb der Systemlandschaft eines Kunden. Ist das logische System ein SAP–System, so identifiziert es einen Mandanten einer SAP Datenbank. Das logische System eines Objekts identifiziert sein originäres System (Originalsystem), von dem z.B. aus Replikationen über*ALE* in andere Systeme gelangen können.
  Der *Objekttypname* weist die Zugehörigkeit eines Objektes zu seinem Objekttyp (Klasse) aus. Beispiele sind hier Kunde oder Material.
  Der Objekttyp bestimmt den *typspezifischen Schlüssel*, der die einzelnen Instanzen des Typs voneinander unterscheidet. Beispiele sind hier Kundennummer oder Materialnummer. Ein typspezifischer Schlüssel kann auch aus mehreren Teilen bestehen. Die Schlüsselfelder eines Objekts sind zugleich invariante Attribute eines Objektes.
  Die Identifikation eines Objekts über typspezifische Schlüssel trägt dem Umstand Rechnung, daß das SAP System auf einer relationalen Datenbank aufbaut.
  Außer dem persistenten Schlüssel werden Objekte innerhalb des BOR–Laufzeitsystems und auch an der DII–Schnittstelle (siehe Abschnitt 2.3.4) über einen Laufzeitschlüssel (*handle*) eindeutig identifiziert.
  Bezüglich des Identifikators lassen zwei Arten von Objekten in R/3 unterscheiden: Objekte, die auf Daten referenzieren haben immer einen Identifikator. Es ist aber auch möglich, Objekttypen zu definieren, die kein Identifikator–Attribut haben. Die von diesen Objekttypen erzeugten Objekte sind Fabrik–Objekte (siehe [Gamma et al., 1995, S.87ff]). Sie dienen zur Instanziierung von Objekten eines bestimmten Typs*mit* Identifikator.
- **Methoden** sind Operationen auf Objekten, die diese in der Regel in einen neuen Zustand überführen. Durch den Zustandsübergang können Ereignisse ausgelöst werden (siehe Abschnitt 2.2.1).
  Die Definition einer Methode umfaßt:
  - einen Ergebnisparameter (optional).
  - Übergabeparameter (in der Objekttyp–Definition ist spezifiziert, ob ein Parameter optional ist oder zwingend). Ferner wird unterschieden in:
    - Export (*call by value*) und
    - Import (*call by reference*).
  - Ausnahmen (Fehlermeldungen der Laufzeitumgebung des BOR).

  Die Datentypen von Parametern entsprechen denen von Attributen (siehe oben). Alle Übergabeparameter sind *call by name*, so daß die Reihenfolge bei der Übergabe keine Rolle spielt.
  Das Methoden–Konzept ermöglicht die gleichartige Kapselung von so unterschiedlichen ABAP/4–Funktionsaufrufen wie:
  - Funktionsbausteinen
  - Dialogbausteinen
  - SAP–Transaktionen
  - Reports
  - OLE 2.0 Automation Befehlen

  in einer einheitlichen Weise.

Die Implementierung der Methoden und Attributzugriffe erfolgt in Prozeduren des dem Objekttyp zugeordneten ABAP/4 Programms. Der Aufruf der Methoden eines Objektes erfolgt über den *Business Object Broker*.

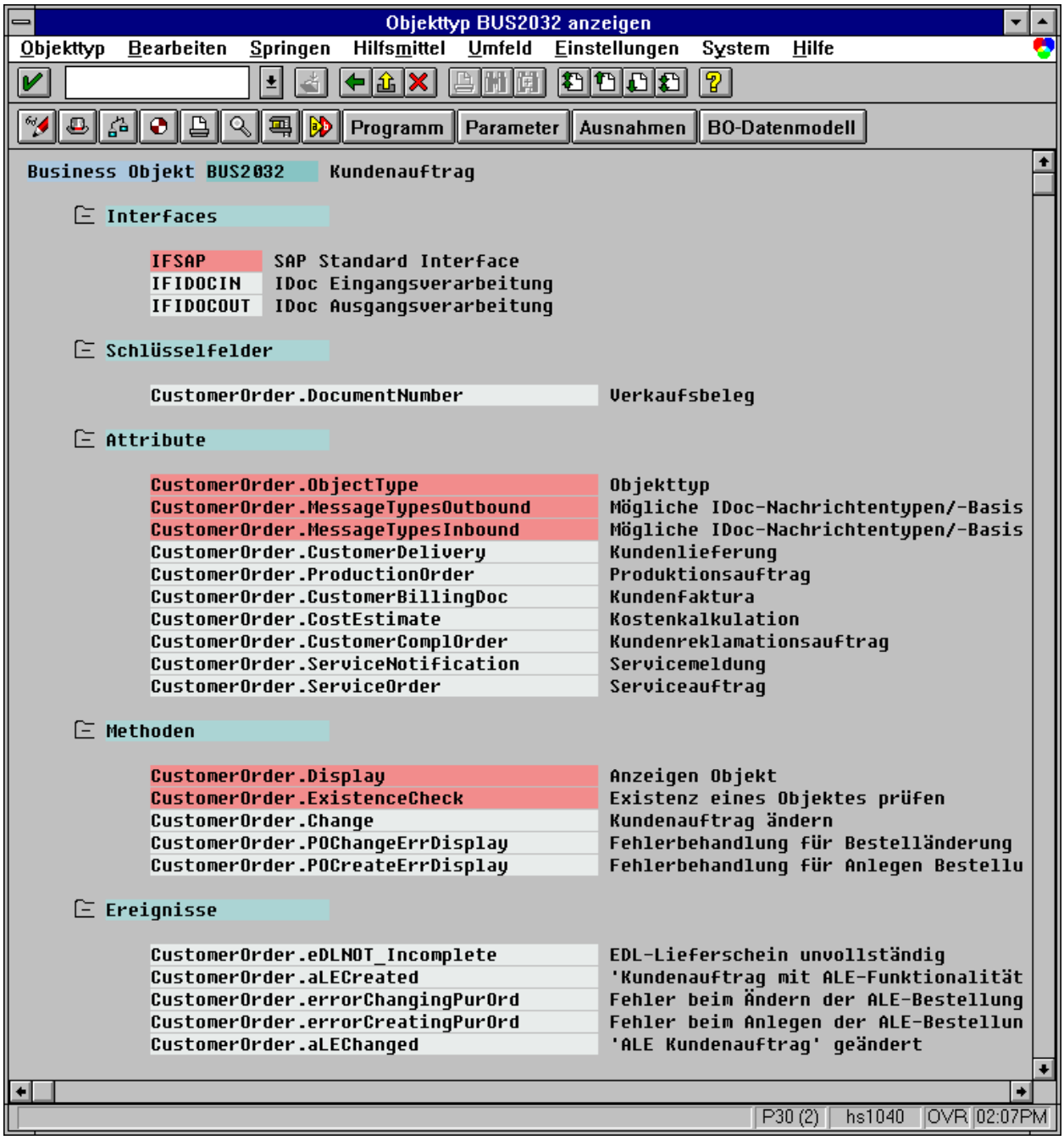


Abbildung 6: Der Objekttyp Kundenauftrag (Bildschirm–Hardcopy des BOR-Browsers)

- **Ereignisse** signalisieren das Eingetretensein von ausgewählten Zustandsänderungen eines Objektes.
  Die Definition eines Ereignisses umfaßt die optionale Angabe von Parametern (vgl. Methoden). Das *Business Object Repository* beschränkt sich derzeit lediglich auf die Definition und Dokumentation von Ereignissen, die von Objekten eines Objekttyps ausgelöst werden können. Das Auslösen von Ereignissen (*publish*) und die Kopplung von Verbrauchern (*subscribe*) erfolgt über den Ereignismanager des SAP *Business Workflow*. Daher ist die Weiterleitung und der Empfang von Ereignissen an/von Klienten der Broker–Schnittstelle derzeit nicht möglich.

2.3.2 Beziehungen zwischen Objekttypen

Für die Objekte des *Business Object Repositories* sind folgende Beziehungstypen definiert:

- **Spezialisierung** (*is a*)
- **Teilbeziehung** (*is part of*)
- **Verknüpfung** (*has link to*)

Ein Objekttyp (z.B. Terminauftrag) als **Spezialisierung** eines anderen Objekttyps (z.B. Auftrag) erbt alle dort definierten Attribute, Methoden und Ereignisse einschließlich ihrer Implementierung. Dem abgeleiteten Typ können Attribute, Methoden und Ereignisse hinzugefügt werden; die Definition der geerbten Elemente darf nicht verändert werden, mit Ausnahme der Erweiterung der Parameterliste von geerbten Methoden und Ereignissen. Geerbte Implementierungen von Methoden und Ereignissen können durch eine neue Implementierung überschreiben werden.
Attribute vom Typ Objekt–Referenz bilden eine unidirektionale **Verknüpfung** zwischen Objekten. So ist z.B. ein Objekt vom Typ Auftrag verknüpft mit seinem Auftraggeber (Kunde), einer Verkaufsorganisation und gegebenenfalls mit einem Angebot.
Eine spezielle Verknüpfung stellt die **Teilbeziehung** dar. Diese Beziehung dient zur Modellierung der Lebensdauerabhängigkeit zwischen Objekten. Objekttypen deren Objekte durch die Teilbeziehung von Objekten eines bestimmten Typs aggregierbar sind, müssen das Interface IFAGGREGATE implementieren. Das so geerbte Attribut *Aggregate* hat als Wert den Namen des aggregierenden Objekttyps. Dieser wiederum muß Auskunft geben, über die Objekttypen, deren Objekte aggregiert werden können.
Zur Laufzeit bewirkt das Löschen des Aggregatobjekts auch die Zerstörung aller aggregierten Objekte. Das ist bei der Verknüpfungsbeziehung nicht der Fall.
Desweiteren gilt: Die Teilbeziehung ist bidirektional. Die Komponenten–Objekte verfügen über die Objektreferenz des Aggregatobjekts.

2.3.3 Interfacetypen

Die beschriebene Spezialisierungsrelation läßt nur einfache Vererbung zu. Um dennoch die Mix–In–Vererbung von Protokollen (vgl. [Gamma et al., 1995, S.16]) zu ermöglichen, können Interfacetypen analog den oben beschriebenen Objekttypen definiert werden. Die Methoden und Ereignisse erhalten eine Default–Implementierung, sie müssen also nicht überschrieben werden.
Als Beispiel sei der Interfacetyp Drucken genannt, der lediglich die Methode *Print* mit den für das SAP System üblichen Druckparametern wie z.B. Ausgabegerät, Drucken sofort, Halten im Spool, Benutzerdialog für Parameter etc. enthält. Objekttypen, die dieses Interface unterstützen sollen, müssen davon erben und die Methode *Print* und die damit verbundenen Parameter durch eine eigene Implementierung überdefinieren.
Alle Objekte von Objekttypen, die das Interface Drucken unterstützen, können nun in der gleichen Weise zum Drucken veranlaßt werden.
Die Elemente von Interfacetypen bilden einen gemeinsamen globalen Namensraum. Die Methode *Print* kann z.B. nur in einem Interfacetyp definiert werden. Interfacetypen können nicht spezialisiert, aber aggregiert werden. Ein Objekttyp kann so jedes beliebige Interface unterstützen, ohne Namenskonflikte zu erzeugen.
Die Unterscheidung von Objekttypen und Interfacetypen ermöglicht eine verteilte Entwicklung, in der Interfacetypen in zentralen Integrationssystemen bereitgestellt werden und Objekttypen in unterschiedlichen Entwicklungssystemen ausgestaltet werden können. Aufwärtskompatibilität in einem sich zeitlich über Releasewechsel entwickelndem System wird ermöglicht durch Hinzufügen weiterer Interfaces, die bereits ausgelieferte umfassen können.

2.3.4 Der *Business Object Broker*

Der *Business Object Broker* (BOB) wurde in Anlehnung an die in [OMG, 1991] beschriebene Spezifikation eines *Dynamic Invocation Interfaces* (DII) entwickelt, um die Kompatibilität zu anderen ORB–Realisierungen zu gewährleisten. Folgend eine Prinzip–Beschreibung der API des BOBs, die von der konkreten API insoweit abstrahiert, als das die Datenstrukturen einige für eine erste Betrachtung unwichtige Parameter und Prozeduren sowie die Parameter für die Ausnahmebehandlung nicht beschrieben werden. Für eine ausführliche Beschreibung verweise ich auf [Krane_1, 1995].

`create`(In:***ObjectType***, OptIn:***ObjectKey***, Out:***ObjectHandle***)

Mit `create` wird durch die Angabe eines Objekttyps und optional dem Wert des Identifikatorattributs ein Objekt im BOR angelegt. Wird *ObjectKey* angegeben, referenziert das Objekt auf existierende Daten (siehe Abschnitt2.3.1).

Der Aufrufer erhält mit *ObjectHandle* den Laufzeitschlüssel des Objekts in Form einer positiven Ganzzahl zurück. Alle nachfolgenden Zugriffe auf das Objekt erfolgen unter Angabe dieses Handles.

`get_type_info`(In:***ObjektType***, Out:***MethodAndAttributeList***)

Diese Prozedur gibt die Schnittstellendefinition eines Objekttyps zurück (Beispiel siehe Anhang C, Seite 68 – Der Objekttyp APPOINTMNT).

`invoke`(In:***ObjectHandle***, In:***Method***, In/Out: ***Parameter***)

Mit `invoke` wird die durch *Method* angegebene Objektmethode des durch *ObjectHandle* bezeichneten Objekts aufgerufen. Die Übergabeparameter werden in einer Containerstruktur transportiert, in der anschließend auch Rückgabewerte retourniert werden.

`free`(In:***ObjectHandle***)

Erzeugte Objekte werden durch die Angabe des Laufzeitschlüssels mit `fee` gelöscht.

`get_object_id`(In:***ObjectHandle***, Out:***LogSys***, Out:***ObjectType***, Out:***ObjectKey***)

Durch diese Prozedur kann das zum Laufzeitschlüssel korrespondierende Tripel des persistenten Schlüssels zu einem bereits existierenden Objekt abgefragt werden.

2.3.5 Wichtige Eigenschaften des *Business Object Brokers*

2.3.5.1 Generizität

Die wichtigste Eigenschaft eines DIIs ist die *Generizität der Schnittstelle*: Objekterzeugung und der Methodenaufruf erfolgen unabhängig von konkreten Objekttypen sowie Datentypen der Aufruf– und Rückgabeparameter (falls wie im BOR integrale Typen Bestandteil des Objektmodells sind). Im Beispiel des *Business Object Brokers* wird die Generizität durch zwei Eigenschaften erreicht:

- Objekte werden durch die Angabe des Laufzeitschlüssels bzw. bei der Objekterzeugung durch die Angabe des persistenten Schlüssels einheitlich referenziert.
- Der Datentransport beim Aufruf einer Methode sowie zur Rückgabe des Ergebnisses erfolgt durch eine Container–Datenstruktur, die die Daten ohne Typinformation in stringifizierter Form enthält.

2.3.5.2 Datenidentifikation

Die zweite wichtige Eigenschaft eines DIIs ist die Art, wie die Datenidentifikation erfolgt. Die Daten sind entweder Objektreferenzen oder Nutzdaten eines bestimmten Typs. Das DII muß die nötige Funktionalität zur Bestimmung des Typs eines Datums beinhalten. Der BOB bietet durch

die Prozedur `swo_get_type_info` die Möglichkeit, die Typ–Definition eines Objekts zu erhalten. Dort ist für jedes Attribut und jeden Parameter einer Methode die Typinformation enthalten.
Als Beispiel ist im Anhang C auf Seite 68 die Schnittstelle des Objekttyps APPOINTMNT abgedruckt.
Anhand dieser Typ–Definition hat der BOB sowie ein Klient der Schnittstelle die Informationen, um die zu übertragenden Daten korrekt zu konvertieren.

Exkurs – CORBA *TypeCodes*:

Der CORBA–Standard [Orfali et al., 1996, S.97ff] bietet einen vergleichbaren Mechanismus an: *TypeCodes*. Sämtliche Attribute, Parameter und Rückgabewerte werden in der CORBA–IDL (*Interface Definition Language*) durch TypeCodes beschrieben. TypeCodes sind spezielle Objekttypen mit einer Schnittstelle zur Abfrage von Informationen über den Datentyp (Abbildung 7).

```
interface TypeCode {
       boolean equal (in TypeCode tc);
       TCKind kind ();
       RepositoryId id ();
       Identifier name ();
       unsigned long member_count ();
       Identifier member_name (in unsigned long index);
       TypeCode member_type (in unsigned long index);
       any member_label (in unsigned long index);
       TypeCode discriminator_type ();
       long default_index ();
       unsigned long length ();
       TypeCode content_type ();
       long param_count ();
       any parameter (in long index);
   };
```

```
enum TCKind {
     tk_null, tk_void,
     tk_short, tk_long,
     tk_ushort, tk_ulong,
     tk_float, tk_double,
     tk_boolean, tk_char,
     tk_octet, tk_any,
     tk_TypeCode,
     tk_Principal, tk_objref,
     tk_struct, tk_union,
     tk_enum, tk_string,
     tk_sequence, tk_array,
     tk_alias, tk_except
    };
```

Abbildung 7: Die *Type–Codes* aus dem CORBA–Standard

Als Erweiterung zum TypeInfo–Konzept des BORs bieten die TypeCodes den Datentyp `any` an, der wiederum als Struktur aus TypeCode und Wert definiert ist [OMG, 1991]:

```
typedef struct any { TypeCode _type; void* _value; } any;
```

So ist es möglich, generische Datenstrukturen (z.B. heterogene Listen) zu handhaben, da sich erst zur Laufzeit entscheidet, welche Datentypen konkret transportiert werden (siehe auch Abschnitt 2.3.6).

2.3.5.3 Integration in R/3*:* Die RFC–Schnittstelle

Die Aufrufe der Funktionen der API des BOB werden über die RFC–Schnittstelle (*Remote Function Call*) von R/3 [SAP_2, 1995] geleitet. Die Schnittstelle hat prinzipiell folgendes Aussehen:

```
Connection RfcOpen(RfcOptions)
Boolean    RfcClose(Connection)
Boolean    RfcCall(Connection, MethodName, ParamContainer)
Boolean    RfcReceive(Connection, ReturnContainer, Errors)
```

Mit `RfcOpen` und `RfcClose` wird eine RFC–Verbindung gestartet (Logon) und beendet (Logoff). `RfcCall` bewirkt einen Funktionsaufruf auf dem Zielsystem. Hierzu werden sämtliche Parameter in Zeichenketten konvertiert und anschließend in eine Container–Struktur (`ParamContainer`) geschrieben. Danach erfolgt der Empfang der Rückgabeparameter mit `RfcReceive` in gleicher Weise.

2.3.5.4 Netzwerktransparenz

In [OMG_2, 1995] wird das einem *Object Request Broker* unterliegende Prinzip aufgrund der Eigenschaft, vom unterliegenden Netzwerk zu abstrahieren und so den netzwerkweiten transparenten Zugriff auf Objekte über deren Identifikator (*Handle*) zu ermöglichen, in den Status eines Entwurfsmusters erhoben.
In der Tat ist die durch den Einsatz von ORBs erreichte Vereinfachung und Verallgemeinerung für den Zugriff auf verteilte Daten eine ganz wichtige Eigenschaft, da so die Entwicklung von Anwendungen, die diese Dienste benötigen sehr viel einfacher wird. Die Durchsetzung dieses Musters wird durch dessen Einfachheit und den zu erwartenden Nutzen sehr schnell und umfassend erfolgen.

2.3.6 Anmerkungen zum SAP–Objektmodell und zum *Business Object Repository*

Die bisher vorgestellten Komponenten und Eigenschaften von Objektmodell und *Business Object Repository* sollen zu allgemeinen Anforderungen des Objekt–Paradigmas in Bezug gebracht und die Konzepte gleichzeitig auch in einigen Aspekten vertieft dargestellt werden. Desweiteren wird in manchen Punkten auch der CORBA–Standard [OMG, 1991] und die OLE/COM–Spezifikation [Microsoft, 1995] zum Vergleich herangezogen.
Da dieser Abschnitt schon als Dokument innerhalb der SAP veröffentlicht wurde, habe ich es unverändert als Anhang A beigefügt.

2.3.7 Die Objekttyp–Bibliothek

In der Navigationskomponente des BORs werden die Objekttypen anhand einer Baumdarstellung visualisiert (Abbildung 8): Ausgehend von einer gemeinsamen Wurzel wird zunächst nach Problembereichen eingeteilt (z.B. Finanzwesen, Controlling, ...) und ab einer bestimmten Hierarchietiefe dann anhand der Vererbungsrelation.
In Objekttypen werden auch die Metadaten abgelegt – ein Beispiel hierfür sind die Workitems (Abschnitt 2.2). In Abbildung 8 ist die vollständige Vererbungsstruktur der Workitems dargestellt. Wie sich erkennen läßt, ist an dieser Stelle die Vererbungsrelation extensiv angewendet worden.
Objekttypen sind derzeit vor allem im Bereich Finanzwesen (FI) definiert worden; bisher sind rund 10 % der R/3–Funktionalität in Objekte verpackt.
Mit der verfügbaren Funktionalität der Werkzeuge ist es nicht einfach, einen Überblick über die umfangreiche Struktur der Objekttyp–Bibliothek zu bekommen. Die selektive Expansion eines Teilbaums ist nur für Objekttypen, nicht aber für die Einteilung nach Problembereichen möglich.
Vererbung wird i.A. selten angewandt, die am häufigsten eingesetzte Relation ist die Verknüpfung (siehe Abschnitt 2.3.2) mit anderen Objekten zur Laufzeit. Die Objekttypen haben meist nur eine kleine Schnittstelle; häufig besteht sie ausschließlich aus Methoden von geerbten Interfaces. Als Beispiel ist in Abbildung 6 der Objekttyp Kundenauftrag als Bildschirmhardcopy abgebildet.
Das größte Hindernis für die schnelle Vervollständigung der Objekttypbibliothek liegt in der Komplexität der in Methoden zu kapselnden Transaktionen. Die Komponenten sind einerseits durch vielfältige Benutzt–Beziehungen voneinander abhängig. Andererseits ist oftmals keine saubere Trennung von Dialog– und Funktionskomponenten realisiert. So ist das Wrapping der Funktionen nicht ohne weiteres möglich, sondern erfordert eine Restrukturierung und Neuimplementierung in größerem Umfang. Ob sich unter diesen Gesichtspunkten das Wrapping lohnt, ist eine der aktuell drängenden Fragen.

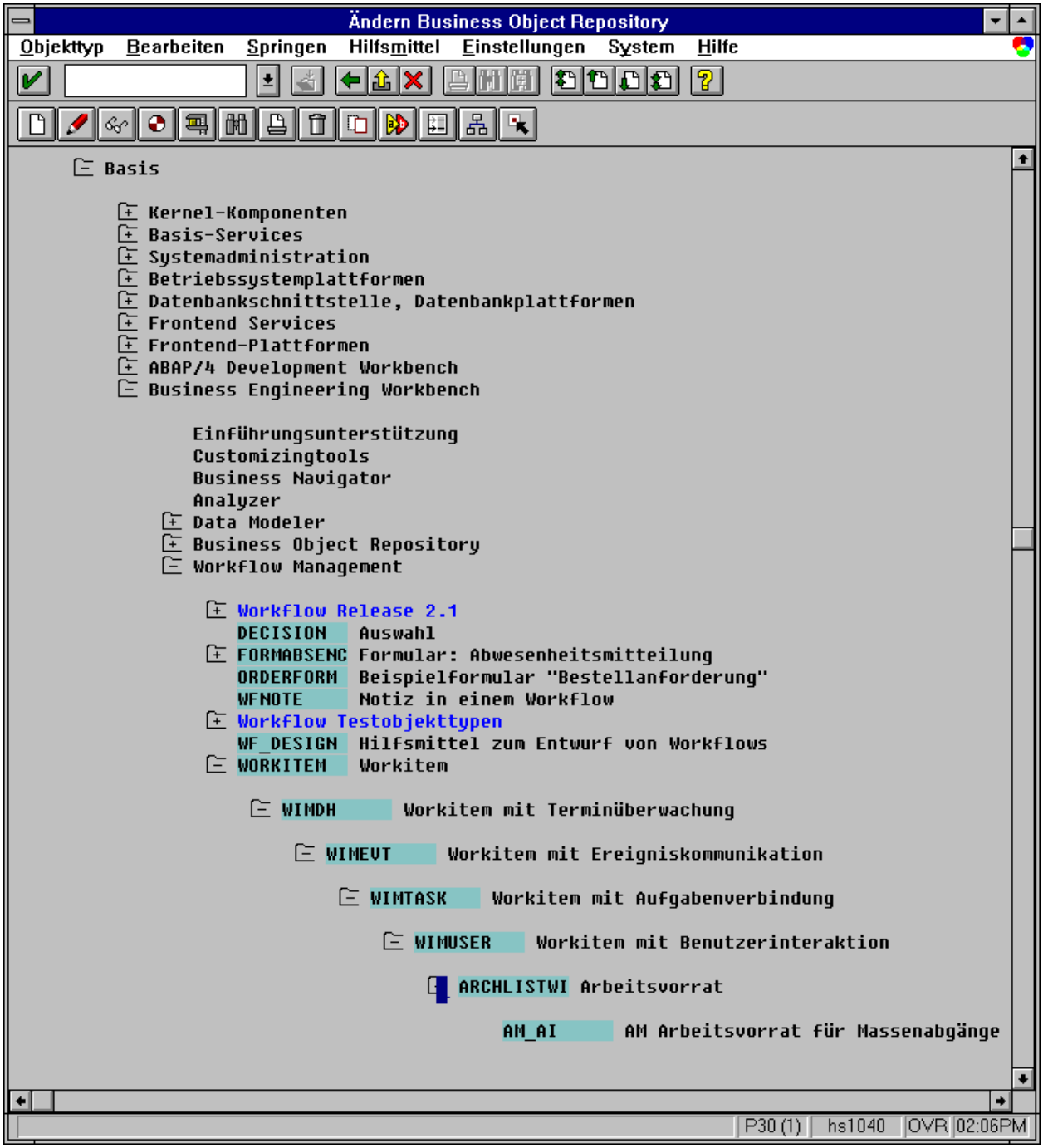


Abbildung 8: Die Objekttyp–Hierarchie (Bildschirm–Hardcopy des BOR–Browsers)

3 Die *Open Scripting Architecture* (OSA)

Seit Anfang der 90er Jahre zeichnet sich ein neuer Lösungsweg für die Software–Krise ab: Die Komplexität wird durch die Aufspaltung in überschaubare Einheiten, genannt *Components,* reduziert.
Die Aufteilung als ein Grundprinzip der Komplexitätsbewältigung ist soweit nichts Neues, was die Modularisierung von Entwürfen angeht. Mit den Components wird dieses Prinzip aber auf ablauffähige Teilapplikationen übertragen: Eine Applikation ist aus einer Menge von Components zusammengesetzt, die sich dynamisch ändern kann. Als Beispiel sei die Textverarbeitungskomponente genannt, aus der mit einer Internet–Zugriffskomponente ein Email–System wird.
Seit dem Aufkommen der graphischen Benutzeroberflächen ist durch DLLs (*Dynamic Link Library*) bzw. *Shared Libs* die mehrfache Verwendung von identischen Applikationsteilen zum Alltag geworden.
Der eigentliche Clou der Components ist aber das, was sie verbindet.

3.1 Überblick – *Automation*

Unter den Begriffen *Automation* (im OLE–Umfeld entstanden) oder *Scripting* versammeln sich die Bestrebungen verschiedener Firmen und Konsortien aus der IT–Branche zur Bereitstellung von Technologien für die Kommunikation zwischen Applikationen.
Automation ist in der einfachsten Form das, was man in früheren Zeiten als Batch–Programmierung bezeichnet hat. In den letzten Jahren hat sich dieser Bereich aber dramatisch weiterentwickelt und wird in Zukunft als integraler Bestandteil der Objektverteilung (*distributed objects*) einen hohen Stellenwert als*enabling technology* haben [Orfali et al., 1996, S. 399].
Die wichtigsten marktrelevanten Entwicklungen, die produktiv angewendet werden oder kurz vor der Verfügbarkeit stehen sind:

- ***OLE–Automation*** von Microsoft für die Windows–Plattformen [Microsoft, 1995],
- die ***Open Scripting Architecture (OSA)*** [IBM_1, 1996] von Apple für die Apple–Plattform, IBM OS/2 und weitere IBM–Plattformen sowie langfristig auch für Windows und
- die ***CORBA User Interface Common Facilities*** [Orfali et al., 1996, S.247f].
- Als neuer Kandidat im Rennen ist JAVA von Sun mit***JAVAScript*** (http://www.sun.de).

Das Prinzip, mittels Batch–Programmierung Abläufe zu automatisieren und Daten durch ein oder ggf. mehrere darauf angewandte Programme ohne Benutzereingriff zu verarbeiten, ist auch weiterhin das Ziel bei der Entwicklung neuerer Konzepte.
Diesen neuen Konzepten gemeinsam ist folgende Eigenschaft: Eine Applikation präsentiert sich an der *Automation*–Schnittstelle mit einer *Objekthierarchie* (auch als *Objektmodell* bezeichnet) ihrer Daten und Benutzerfunktionen.
Als Beispiel ein Ausschnitt aus der Objekthierarchie der Tabellenkalkulation Excel 5.0 von Microsoft (Abbildung 9). Die Objekte stehen über die Aggregationsrelation in Beziehung. Jedes Objekt hat Attribute und Methoden, über die abfragend und modifizierend auf die Daten zugegriffen werden kann. Der entstehende Graph muß kein Baum sein, wie die Abbildung evtl. suggeriert, da auch Querverbindungen möglich sein können – beispielsweise wird das Objekt "Schriftart" sicherlich von vielen anderen Objekten aggregiert.
An dieser *Automation*–Schnittstelle sollte die vollständige Benutzerfunktionalität verfügbar sein, so daß alle möglichen Benutzereingaben auch via Scripting erfolgen können. Sie kann Klienten mit einfachen Aufgabenstellungen, wie z.B. Formatieren und Ausdrucken eines ASCII–Dokuments, in einheitlicher Weise genauso bedienen, wie komplexen Anforderungen gerecht werden: An eine OLE–Automation–fähige Datenbank (z.B. POET 3.0) läßt sich so eine Datenbankappli-

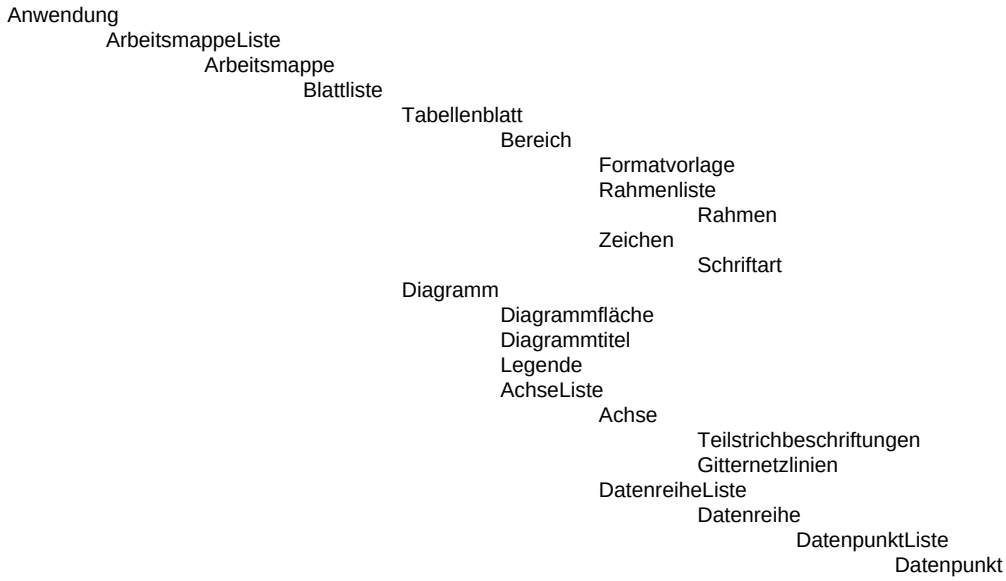


Abbildung 9: Ausschnitt aus der Objekthierarchie von MS Excel 5.0

kation koppeln oder die Benutzerschnittstelle von Excel vollkommen durch eine neuentwickelte ersetzen (3D–Visualisierung . . .).
Mittels einer Scriptsprache ist es auch möglich, *composite applications* aus mehreren Komponenten zusammenzubauen, die evtl. nur für die Zeit ihrer Aufgabenerfüllung existent sind (*disposable applications,* [Orfali et al., 1996, S.405]).
Im OSA–Konzept spielt *recording* eine wichtige Rolle: Anwender können Scripte ggf. auch über mehrere Applikationen hinweg durch simples Vormachen der gewünschten Aktionen erzeugen.

3.2 Visionen

Im nächsten Schritt, der, was die Microsoft–Welt angeht, mit Windows NT 4.0 wohl endlich in diesem Jahr Wirklichkeit werden wird, ist Scripting unter Windows auch über Rechnergrenzen hinweg möglich.
Doch die Entwicklungen in diesem Umfeld sind noch wesentlich umfassender und grundlegender, als das nur durch Scripting erreichbar wäre. Scripting–Komponenten sind meist in Umgebungen eingebettet, die noch weitergehende Dienste bieten. So ist OSA ein Teil von OpenDoc; OLE–Automation ist nur eine Teilkomponente von OLE/COM, und das *User Interface Service Facility* nur ein Dienst unter anderen des CORBA–Standards. Allen diesen *compound–document–*Konzepten gemein ist, daß sie angetreten sind, den Wechsel vom applikationszentrierten hin zum dokumenten– und prozeßorientierten Arbeiten herbeizuführen; monolithische Applikationen werden in Komponenten aufgespalten, die im Idealfall durch andere Komponenten gleicher oder ähnlicher Funktionalität austauschbar sind.
Hier lassen sich Parallelen ziehen zwischen den Veränderungen im Bereich der Softwareentwicklung (verstärkte Konzentration auf *reuse* mit Klassenbibliotheken und Frameworks) und dem Markt der *ready–to–use–*components: Zukünftig wird es sich an dem Maß der Übereinstimmung von den Anforderungen und dem am Markt vorhandenen Komponenten entscheiden, ob

- eine **Komponente** mittels Scripting in eine vorhandene Laufzeitinfrastruktur (z.B. ein Workflow–Mananger) eingebunden wird (Skalierbarkeit und Funktionsumfang sowie die Plattform entsprechen den Anforderungen exakt),
- ein **Framework** mit der nötigen spezifischen Funktionalität erweitert und ggf. in die Umgebung noch integriert werden muß (da Skalierbarkeit, Funktionsumfang oder die unterstützte Plattform nicht den Anforderungen entspricht) oder

- eine vollständige **Eigenentwicklung** nötig ist (da keine Komponente und auch kein Framework am Markt vorhanden ist, das die Anforderungen erfüllt).

Nicht nur die Applikationen werden durch die allseits beschworenen *Components* aufgespalten (die Graphik in Abbildung 10 zeigt diese Notwendigkeit anschaulich) und damit der Softwaremarkt grundlegend verändert, sondern auch die Arbeitsbedingungen für SoftwareentwicklerInnen.

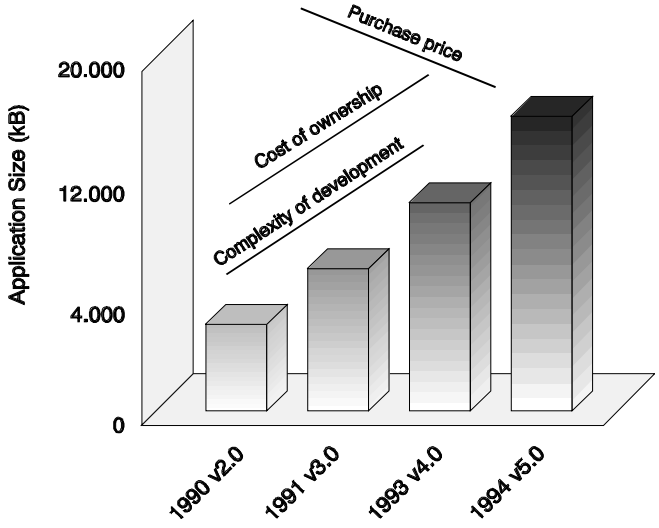


Abbildung 10: Entwicklung des Umfangs von MS Excel [CILabs, 1995]

Ein weiteres vollkommen neues Feld, das durch Scripting aufgetan wird, ist das der *Agenten* ([Orfali et al., 1996, S.401f]). Agenten sind Components, die ausschließlich durch Scripting angesprochen werden, keine Benutzerschnittstelle besitzen und auf bestimmte Aufgaben spezialisiert sind. Die Aufgaben der Agenten hängen meist mit Datenbeschaffung zusammen: Beispielsweise die Suche nach Informationen im WWW. Wird die JAVA–Vision der *Applets* (siehe z.B. [Römer, 1996]) im Zusammenhang mit der globalen Vernetzung Wirklichkeit, dann transportieren sich zukünftig *Roaming Agents* [Orfali et al., 1996, S.401f] oder *Mobile Agents* [Baryla, 1995] zur Erfüllung ihres Auftrags samt ihrer nötigen Umgebung über das Netz, stoßen auf anderen Rechnern via Scripting neue lokale Prozesse an und verlassen den Rechner wieder, nachdem sie die nötigen Informationen erhalten haben.

Abbildung 11: "*Where do you want to go today ?*" [Orfali et al., 1996, S.406]

3.3 Software–Architektur für Scripting–fähige Applikationen

Die geeignete Software–Architektur für eine Applikation, die mit geringem Aufwand scripting–fähig gemacht werden kann, sollte folgende Voraussetzungen erfüllen:

- Die Benutzerfunktionalität in Verbindung mit dem Datenmodell sollte in der Architektur so abgebildet sein, daß eine weitere Ein–/Ausgabeschnittstelle integriert werden kann. Die Strukturierung einer Applikation nach der MVC–Architektur [Gamma et al., 1995, S.293ff] eignet sich hierfür besonders gut, da die Austauschbarkeit der Komponenten eine ganz wesentliche Eigenschaft dieses Musters ist. Aus dieser Sichtweise würde die Scripting–Komponente aus einem neuen View (Abbildung der Funktionen und Daten in das Objektmodell) und einem neuen Controller (Bearbeitung der eingehenden Anforderungen) bestehen.
- Um ggf. verschiedene *Automation*–Konzepte unterstützen zu können und den Entwurf möglichst flexibel zu gestalten, sollte eine zweistufige Architektur gewählt werden, die aus einer allgemeinen Scripting–Komponente und der davon getrennten Implementation der spezifischen Konzepte besteht.

Für bereits bestehende Applikationen, die scripting–fähig gemacht werden sollen, gilt folgendes:

- Wahrscheinlich sind Programme, die nicht nach den Prinzipien des invertierten Kontrollflusses [Weinand, 1992] arbeiten, auch nicht geeignet für die Erweiterung um eine Scripting–Komponente. Typische UNIX–Tools, die per Kommandozeile gesteuert werden, sind hingegen gut geeignet.
- Bei Applikationen, die auf Sourcecode–Ebene schon über eine API verfügen, die die Benutzerfunktionalität abbildet, da z.B. eine eigene Makrosprache damit arbeitet, müssen lediglich ein Wrapper für diese API [Baryla, 1995] und Komponenten zur evtl. nötigen Typkonvertierung für die transportierten Daten implementiert werden.

3.4 OSA–Prinzipien

Ein gravierender Nachteil von OLE gegenüber OSA besteht darin, daß Microsoft es versäumt hat bzw. aus marktpolitischen Gründen vorzog, keinen Standard für eine einheitliche Sprache zur Kommunikation zwischen OLE–fähigen Applikationen zu definieren. So exportiert jede Applikation ihr spezifisches Objektmodell mit mehr oder weniger starken strukturellen Unterschieden und mit lokaler Namensgebung.

Im oben angeführten Beispiel der Nutzung von OLE als Kommunikationsweg zwischen Datenbankapplikationen und unterliegender Datenbank ist so nicht der einfache Austausch der Datenbank gegen eine andere möglich. Auch die schöne neue Welt der Components, aus Dokumentensicht die freie Wahl der am Besten für eine Aufgabe geeigneten Komponente zu haben oder bei Nichtvorhandensein einer spezifischen Komponente eine andere zu verwenden, ist so nicht realisierbar: Wenn Excel nicht installiert ist (Lotus 1–2–3 aber schon), kann das Dokument weiterhin *nicht* angezeigt werden.

3.4.1 *Events* und *Object Classes*

Apple ist mit OSA angetreten, mit diesen Unzulänglichkeiten Schluß zu machen. OSA soll eine Infrastruktur für austauschbare *scripting components* darstellen. Die Infrastruktur wird durch

- eine Bibliothek von zulässigen *Events*, *Object Classes* und weiteren Elementen, genannt *OSA Event Registry: Standard Suites* [Apple, 1992] und
- eine API mit Funktionalität zum Empfangen und Senden von *Events* sowie zum Abfragen und Erweitern der *Event*–Bibliothek [IBM_1, 1996]

gebildet. *Scripting Components* können scripting–fähige Applikationen sein, die Kommandos durch *Events* empfangen oder nur senden können oder auch beides. Eine ausgezeichnete Klasse von scripting–fähigen Applikationen stellen Scriptsprachen dar. Auf der Apple Plattform ist das

AppleScript, eine sehr „redselige“, für das Recording gut geeignete Makrosprache [IBM_1, 1996], unter OS/2 wird es zunächst das durchgängig objektorientierte *ObjectREXX* [IBM_2, 1996] sein und längerfristig auch ein VisualBasic–Pendant (*Visual Age Basic*). Soweit besteht noch kein Unterschied zu OLE.

Die Besonderheit von OSA ist das *Event Registry* [Apple, 1992]. Die wesentlichen dort definierten Elemente sind die *Events* und die *Object Classes*:

- Ein **Event** besteht aus einem Namen, einer Liste aus teils optionalen Parametern und evtl. Rückgabewerten. Im *Event Registry* ist die von Apple für einen Event gedachte *Semantik* beschrieben. Die Intention besteht nun darin, daß bei der Entwicklung einer scripting–fähigen Applikation anhand des *Registries* eine Abbildung der vordefinierten Events auf die Semantik der spezifischen Funktionalität der Applikation gefunden wird. Beispielsweise wäre es sinnvoll, den Create–Event bei einer Applikation, die Wetterberichte aus dem WWW verwaltet, der Aktion zuzuordnen, die den aktuellen Bericht liest und als lokales Dokument sichert.
- Der zweite, mindestens ebenso wichtige Teil der Spezifikation sind die **Objektklassen** (*Object Classes*).

  Die zur Laufzeit ansprechbare Objekt–Aggregationsstruktur von MS–Excel (siehe Abbildung 9) ist nicht veränderbar. Für eine Scripting–Infrastruktur, wie sie OSA darstellt, muß hingegen die Möglichkeit zur flexiblen *Definition* einer solchen Aggregationsstruktur bestehen. Hierzu dienen die Objektklassen.

  Objektklassen lassen sich zur Modellierung der Daten und Benutzerfunktionen einer Applikation einsetzen. Eine bestehende Klassenhierarchie der Applikation aus der Analysephase läßt sich an dieser Stelle sicherlich gut wiederverwenden, da dieses Modell noch frei ist von technischen Klassen.

  Als Beispiel ein Ausschnitt aus der OSA–Objektklassenhierarchie in Abbildung 12.

  Das Konzept sieht die abstrakte Superklasse cObject vor, von der alle anderen Klassen direkt oder indirekt erben müssen. Klassen können einfache Attribute (*Properties* genannt), wie z.B. den Namen eines Dokuments und Objektreferenzen (*Element Classes* genannt) enthalten. Durch die Elementklassen wird die zur Laufzeit sichtbare Aggregationsstruktur realisiert. Für die in Abbildung 12 dargestellte Hierarchie besteht beispielsweise eine Aggregationsbeziehung zwischen cWindow und cDocument: Ein Fenster enthält ein Dokument, um es anzuzeigen.

  Die bei der Modellierung festgelegte Aggregationsstruktur stellt, wie oben bereits beschrieben, dann das zentrale Element für die Navigation in den von einer Applikation verwalteten Daten dar.

  Die begründete Vermutung, daß die Events Methoden der Objektklassen darstellen, erfüllt sich nicht. Das Konzept ist an dieser Stelle ein anderes: Events *unterstützen* eine Objektklasse, das heißt, Parameter dieser Events können vom Typ der Objektklasse **und davon abgeleiteter Klassen sein**. Beispielsweise unterstützt der Open–Event die Klasse cWindow.

  Durch Vererbung werden die Attribute und die unterstützten Events an die Subklasse weitergegeben und es ist so eine flexible Erweiterung der Objektklassen–Hierarchie für spezifische Anforderungen möglich.

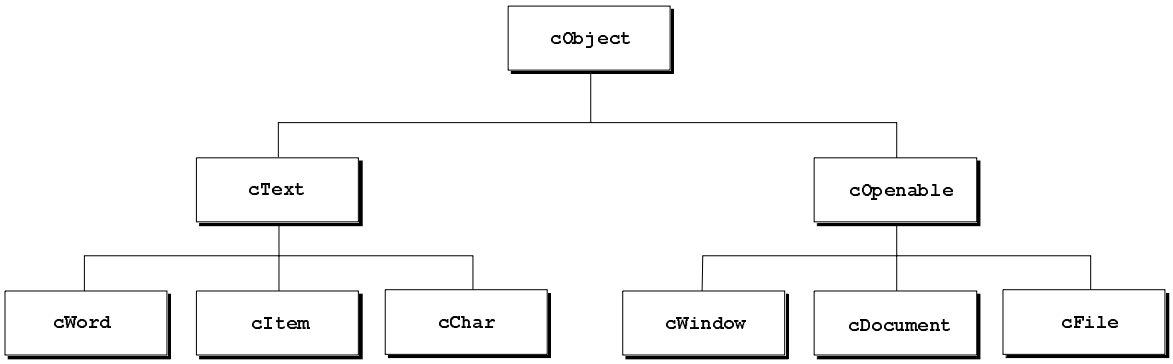


Abbildung 12: Ausschnitt aus der OSA–Objektklassenhierarchie [Apple, 1995]

3.4.2 *Event*–Suiten

Alle Elemente des *Event Registries* sind jeweils genau einer Suite zugeordnet. Die so erzeugte Strukturierung teilt die Elemente nach Applikationsklassen und Basisdiensten ein. Jede Applikation, die an der Kommunikation über Events teilnehmen will, sollte zumindest die *Required*–Suite [Apple, 1992] unterstützen, d.h. auf die *Events* aus dieser Suite reagieren können. In der *Core* Suite sind weitere grundlegende Events beschrieben. Diese Suite muß von einer Applikation nicht vollständig unterstützt werden.
Für die Applikationsklassen Textverarbeitung, Graphikbearbeitung und Tabellenkalkulation sind vordefinierte Suiten vorhanden (*Text, Graphics,* und *Table* Suite), die momentan noch sehr auf die Apple–Bedürfnisse zugeschnitten sind. Da die Verwaltung des Registries aber im Rahmen von OpenDoc* an die herstellerunabhängigen CILabs* übertragen wurde, steht der Definition von plattformunabhängigen Standards für die genannten und andere Applikationsklassen nichts im Wege.
Um die Kommunikation zwischen Applikationen mittels Events erst möglich zu machen, muß der von einer Applikation unterstützte Teil des *Event Registries* (sinnvollerweise immer nur vollständige Suiten) von anderen Applikationen, die mit ersterer kommunizieren wollen, abgefragt werden können. Für diesen Zweck ist die *Registry*–Information in der zur Laufzeit ansprechbaren ‘aete’–Resource abgelegt (*Apple Event Terminology Extension,* gesprochen: E.T.).
Da das Mapping des Objektmodells einer spezifischen Applikation auf die im *Event Registry* vorhandenen Klassen und *Events* nicht immer vollständig möglich sein wird, was das in der Objektklassen–Hierarchie ausgedrückte Modell der Applikation und die durch die *Events* ausgedrückte Semantik angeht, sieht das OSA–Konzept zwei Wege zur Erweiterung der ‘aete’–Resource für spezifische Zwecke vor. Diese Erweiterungen sind im nächsten Abschnitt beschrieben.

3.4.3 *Suite Extension* und *Custom*–Suiten

Das, was in der OLE–Terminologie eine TypeInfo ist [Microsoft, 1995], wird bei OSA als *Apple Event Terminology Extension* oder kurz ‘aete’–Resource bezeichnet. Resourcen beinhalten Informationen über die von der Applikation unterstützten Standard–Suiten, Erweiterungen (*extensions*) der Standard–Suiten sowie neue (*custom*) Suiten. *Extensions* und *Custom*–Suiten gelten nur lokal für die Applikation. Ein OSA–Client, der die Applikation per Script ansteuern möchte, kann deren ‘aete’ –Resource abfragen.
Das wichtigste Mittel der Suite–*Extension* ist die Erweiterung der Parameterliste von *Events.* Zum Beispiel kann der PrintDocument–*Event* um einen Parameter für die Druckqualität erweitert werden. Die Erweiterung wirkt sich nur lokal aus, d.h. eine Applikation, die zwar den PrintDocument–*Event* unterstützt, nicht aber den weiteren Parameter, wird ihn schlicht ignorieren. Somit ist die Kompatibilität gesichert.
Im Gegensatz zur Suite–*Extension* bieten *Custom*–Suiten die Möglichkeit der **strukturellen** Erweiterung der ‘aete’–Resource und somit des Befehlssatzes, der dem Klient zur Verfügung steht. Der Nachteil dieser Methode besteht in der daraus resultierenden Inkompatibilität, da nur ein spezieller Klient diese *Events* ansprechen kann. Wird die Definition von *Custom*–Suiten allerdings von den CILabs koordiniert und veröffentlicht, steckt darin ein großes Potential, da sich die über OSA–*Events* ansprechbare Bandbreite an Applikationen für spezifische Problemlösungen so in kontrollierbarer Weise verbreitern kann.
Alles zur *Extension* und zur Definition von *Custom*–Suites Gesagte, gilt auch für die Objektklassen.
Die oben besprochene Erweiterung der Objekthierarchie mittels Vererbung stellt sicherlich den flexibelsten und konsistentesten Weg zur Anpassung des *Registries* an spezifische Anforderungen dar.

3.5 Bemerkungen zur Implementation

Das in den vorhergehenden Kapiteln dargestellte Konzept der *Open Scripting Architecture* konnte leider nur unvollständig in den OSA–Server, der im Rahmen der Diplomarbeit entwickelt wurde, umgesetzt werden.
Der Hauptgrund liegt in der noch sehr instabilen Laufzeitumgebung der OpenDoc–Implementierung auf der OS/2–Plattform. (Für die Arbeit wurde die seit Mitte Februar vorliegende Version 1.0 [*IBM DevCon CD–ROM Vol.9, Special Edition*] verwendet). Der Test einer OSA–Verbindung zwischen der Script–Sprache ObjectREXX und dem OSA–Server erforderte den Neustart des Rechners nach jedem Versuch, da das Laufzeitsystem dann meist in einem inkonsistenten Zustand war, der keine Aussage mehr über Fehlerursachen zuließ. Komplexere Anweisungen endeten in einem Absturz der Scripting–Komponente. So konnte der OSA–Server nur von einem selbstentwickelten Test–Client angesprochen werden, der natürlich nicht den Funktionsumfang einer Script–Sprache besaß.
Desweitern sind in der vorliegenden OpenDoc–Version keine wirklich exemplarischen Beispiele enthalten, die ein Verständnis der komplexen Materie erleichtern.
Die OSA–Dokumentation ist soweit gut, kann allein aber höchstens die theoretischen Konzepte vermitteln.

3.6 Literatur

Folgend Literaturangaben zu Artikeln und Büchern zum Thema Interoperabilität:

[Pürckhauer, 1995] Pürckhauer, Christoph: *OpenDoc–Basis einer Komponenten Architektur*. 4 (1995) S.49ff. In: Objektspektrum.
Ein guter Übersichtsartikel. Das Thema OSA wird durch den Leiter des Entwicklersupports der IBM in Deutschland kompakt und verständlich darstellt.

[Orfali et al., 1996] Orfali, Robert. Harkey, Dan. Edwars, Jeri: *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons Inc. 1996.
Eine von begabten technischen Autoren verfaßte vergleichende Übersicht über moderne Informatik–Konzepte. Verteilte Systeme und die Umsetzung in CORBA, OLE und OSA werden spannend und verständlich beschrieben.

[Borghoff et al., 1995] Borghoff Uwe M.. Schlichter Johann H.: *Rechnergestützte Gruppenarbeit. Eine Einführung in Verteilte Anwendungen*. Berlin, Heidelberg: Springer–Verlag 1995.
Computer Supported Cooperative Work (CSCW) ist der Schwerpunkt dieses Buches. Interoperabilität als Basiskomponente wird in diesen Kontext eingeordnet.

4 Dokumentation mit Entwurfsmustern

Zum Thema Software–Dokumentation habe ich im Rahmen der Ausarbeitung zu meinem Hauptseminarvortrag einige Vorüberlegungen angestellt, sowie die Literatur aufgearbeitet. Die dort angestellten Überlegungen werden hier wieder aufgegriffen und weiterentwickelt. Zur weiterführenden Information habe ich die Ausarbeitung im Anhang D angefügt.
Mit der vorliegenden Dokumentation möchte ich diese Arbeit inhaltlich fortführen durch die praktische Erprobung der Konzepte und die anschließende Auswertung der Ergebnisse.
Den Entwurf und die Implementierung des OSA–Servers habe ich hierzu mit einem Hypertextsystem dokumentiert, um die in der Praxis entstehenden Probleme (Abschnitt 4.2.5) an einer vom Umfang her realistischen Problemstellung (die Implementierung umfaßt ca. 5.000 LOC) kennenzulernen.
Das Dokument ist, wie in Abbildung 13 dargestellt, in vier Abstraktionsebenen unterteilt (zu den Pfeilen rechts und links siehe Abschnitt 4.3). Ausgehend von der Übersicht auf der obersten Ebene ergibt sich eine Baumstruktur der Dokumente.

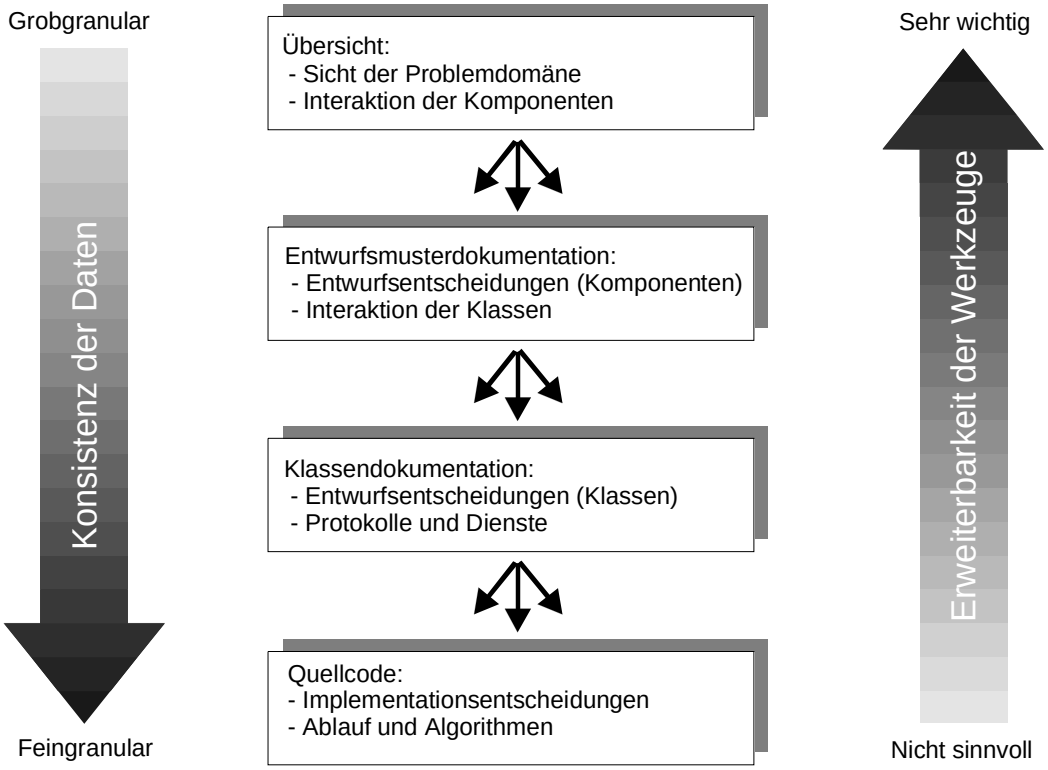


Abbildung 13: Abstraktionsebenen der Entwurfsmusterdokumentation

4.1 Strukturiert graphische und textuelle Notationen

Es gibt sehr viele verschiedene Notationen, die in der Softwareentwicklung zur Dokumentation des Entwurfs auf unterschiedlichen Abstraktionsniveaus eingesetzt werden. In den folgenden Abschnitten will ich die von mir verwendeten Notationen darstellen.

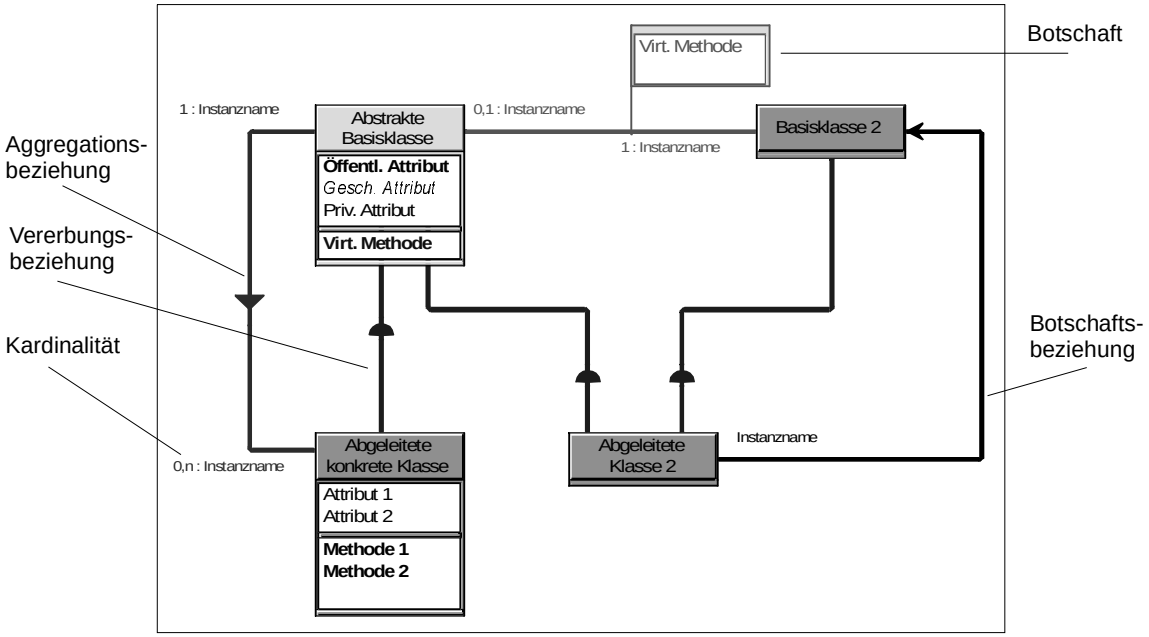
4.1.1 Klassendiagramme nach Coad/Yourdon

Das Klassendiagramm dient zur Darstellung der statischen Struktur einer in der Regel aus mehreren Klassen bestehenden Komponente (Beispiel auf Seite 40, Abbildung 16).

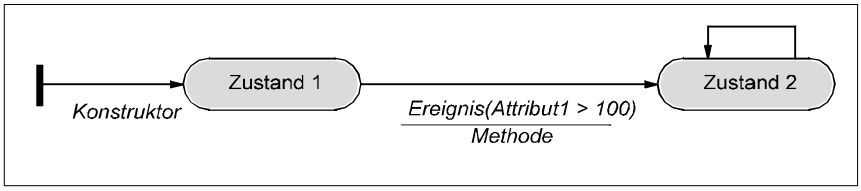
In Abbildung 14 ist die erweiterte Coad/Yourdon–Notation, die von der Entwicklungsumgebung objectiF der Firma microTOOL, Berlin angeboten wird, dargestellt. Die Erweiterungen gegenüber der Coad/Yourdon–Notation [Coad et al., 1991] sind:

- die Annotation von Botschaften an Aggregations– und Assoziationsbeziehung sowie an die Botschaftsbeziehung selbst,
- die Anpassung an den C++–Sprachstandard (öffentliche, geschützte und private Methoden),
- die Aufnahme des Interaktionsdiagramms in die Notation.

Klassendiagramm:



Objektlebenszyklus:



Interaktionsdiagramm:

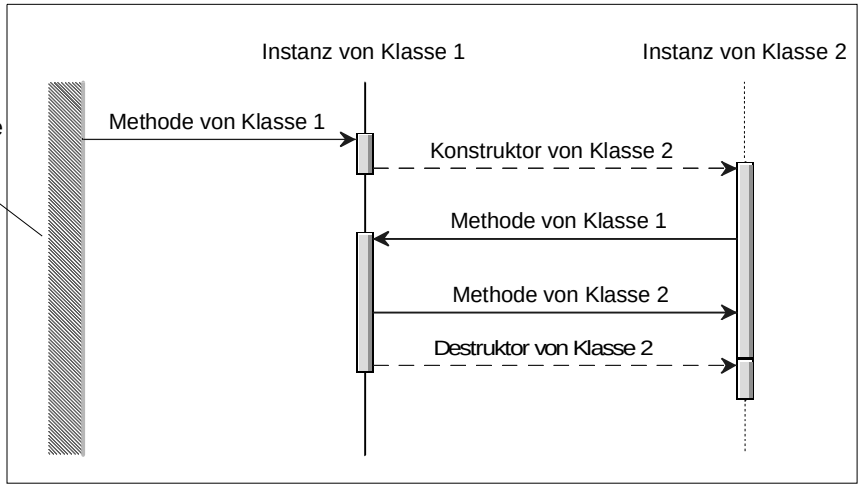


Abbildung 14: Die (erweiterte) Coad/Yourdon–Notation von objectiF

4.1.2 Interaktionsdiagramme, Objektlebenszyklen und Objektdynamik

Bei dieser Gruppe von Notationen geht es um die Darstellung der Zusammenhänge zur Laufzeit. Standen beim Klassendiagramm die Beziehungen zwischen Klassen im Mittelpunkt, so ist es hier die Kausalität der Abläufe.
Im Interaktionsdiagramm wird der Botschaftsfluß zwischen mehreren Objekten visualisiert (Beispiel auf Seite 58, Abbildung 17). Der Objektlebenszyklus ist ein konventioneller Zustandsautomat, der den Zustand eines Objekts über dessen Lebensdauer abbildet (Seite58, Abbildung 18).
Die Objektdynamik soll komplexere Objektstrukturen, die sich zur Laufzeit durch die Verschachtelung von Aggregationen und Assoziationen ergeben kompakt graphisch darstellen (Seite 59, Abbildung 19). Die Notation ist eine Eigenentwicklung.

4.1.3 Entwurfsmusterinstanzen

Die Idee, Entwurfsmuster zur Dokumentation zu verwenden, ist naheliegend. Ausgehend von einem Musterkatalog, findet der Entwerfer eine Entsprechung zwischen einem Muster und seinem Entwurf oder benutzt ein Muster gezielt, um eine Komponente danach zu entwickeln (eine Instanz eines Musters zu bilden).
Zur Dokumentation dieser Entwurfsentscheidung kann er nun die Mustergliederung [Gamma et al., 1995] verwenden, mit einer Zuordnung wie folgt:

- **Übersicht**: Klassendiagramm und evtl. weitere Visualisierungen, die zur Verdeutlichung geeignet sind.
- **Intention**: Knappe Beschreibung der wesentlichen Intention. Der Grund, warum das Muster verwendet wurde, sollte deutlich werden.
- **Motivation**: Ausführliche Beschreibung der Zusammenhänge. Übersicht über die Komponente.
- **Rollen**: In Klammern wird hinter die Rolle einer Klasse aus dem Entwurfsmuster die konkrete Klasse der Implementierung geschrieben (Zuordnung).
- **Zusammenarbeit (*Collaboration*)** Die Interaktion der Klassen für den konkreten Fall wird beschrieben. Evtl. Verweis auf ein Interaktionsdiagramm.
- **Konsequenzen**: Die Konsequenzen aus der konkreten Implementierung (z.B. Erweiterbarkeit, Vergleich mit anderen Realisierungen).
- **Implementation**: (Optional) Besonderheiten der Implementierung.
- ***Hot Spots*** und ***Frozen Spots***: Für die Dokumentation von Frameworks ist die gesonderte Beschreibung der festgelegten Eigenschaften des Frameworks und der durch den Benutzer anzupassenden bzw. zu erweiternden Komponenten sinnvoll (vgl. [Pree, 1995, S.105ff]).

4.1.4 Kochrezept

Mit der von [Johnson, 1992] beschriebenen Technik des Kochrezepts (*cookbook*) werden Anleitungen erstellt, die den Einsatz einer Komponente anhand eines schrittweisen, durch Beispiele ergänzten Vorgehens verdeutlichen.
Kochrezepte können auf unterschiedlichen Abstraktionsebenen eingesetzt werden. So kann ein Kochrezept als Ergänzung zu einer Entwurfsmusterbeschreibung sinnvoll sein, ebenso aber auch Implementationsdetails verständlich machen.

4.1.5 Klassendokumentation

Für die Dokumentation von Klassen wurde folgende Gliederung verwendet:

- **Aufgabe**: Das wird meist die Beschreibung der in der Klasse angelegten Abstraktion sein; daher könnte dieser Punkt auch "Abstraktion" heißen.
- **Besonderheiten**, **Bemerkungen** und/oder **Einschränkungen** (optional).

- **Protokolle**: (optional) Hierzu auch den **Zustandsautomaten** der Klasse oder einen Link auf ein **Interaktionsdiagramm**.

- **Objektdynamik** bzw. **Laufzeitstruktur**: (optional) Bei komplexeren Laufzeitstrukturen könnte eine textuelle Beschreibung oder eine Graphik sinnvoll sein.
- **Kardinalität**: (optional) Wieviel Objekte entstehen zur Laufzeit? Besonderheiten.
- **Lebensdauer**: (optional) Wann und wie lange existiert ein/das Objekt? Besonderheiten.
- **Elternklasse**: (optional) Beschreibung, welche Klassen geerbt werden und warum.
- **Muster**: (optional) Verweis zum Entwurfsmuster, dem die Klasse angehört.
- **Ausnahmebehandlung**: Falls Methoden der Klasse Ausnahmen erzeugen (*Exceptions*), das Konzept der Ausnahmebehandlung erläutern. Desweiteren einen Link auf die Exceptionklasse.
- **Used by**: Links zu den Klassen, die die aktuelle Klasse verwenden.
- **Uses**: Links zu den Klassen, die von der aktuellen Klasse verwendet wird.
- **Implementation**: Link auf die Definition und/oder Implementation der Klasse.
- **Testumgebung**: Ausführung eines Testrahmens, der das Verhalten der Klasse möglichst interaktiv erfahrbar macht.

Die Dokumentation einer Methode erfolgt anhand dieser Gliederung:
- **Aufgabe**: Beschreibung des prinzipiellen Ablaufs.
- **Bemerkungen**: (optional).
- **Überladen**: (optional) Wenn die Methode aus einer Elternklasse redefiniert wird. Oder, wenn die Methode Teil eines geerbten Protokolls (abstraktes Interface) ist.
- **Implementation**: Link auf die Implementation der Methode.
- **Ausnahmebehandlung**: (optional) Durch die Methode eventuell ausgelöste Exceptions. Hierzu dann auch evtl.: Was passiert, wenn die Exception nicht gefangen wird.
- Optional: Link auf ein **Interaktionsdiagramm** oder den **Zustandsautomaten** der Klasse, falls der Aufruf der Methode in direktem Zusammenhang mit anderen Methodenaufrufen der selben Klasse (Zustandsautomat) oder anderer Klassen (Interaktionsdiagramm) steht.
- **Übergabeparameter**
- **Rückgabeparameter**

4.1.6 Implementationsdokumentation

Die konventionelle Implementationsdokumentation, bei der im Quellcode dessen Ablauf in Kommentarblöcken beschrieben wird, kann durch das Medium Hypertext ergänzt und verbessert werden. Aus dem Quellcode kann entlang der Verweise (*Links*) hin zu anderen Implementations– oder Klassendokumentationen verzweigt werden. Im Vergleich mit dem manuellen Vorgehen beim Durchforsten eines Quellcodes in einem Texteditor ist so eine starke Vereinfachung gegeben. Gleichzeitig wird der Quellcode in die Dokumentation mit eingebunden.

4.2 Praktische Umsetzung

Anhand von Ausschnitten aus der zum OSA–Server erstellten Hypertextdokumentation möchte ich die Benutzung der im vorangegangenen Abschnitt beschriebenen Notationen zeigen. Nebenbei ergibt sich für den interessierten Leser auch einen Einblick in den Entwurf des OSA–Servers.
Zunächst aber noch eine kurze Darstellung der Navigationstechniken, die das von mir verwendete Hypertext–System (WinHelp von Microsoft) anbietet.

Exkurs – Navigationstechniken in WinHelp:

- **Index und Inhaltsverzeichnisse als *Guided Tours***: Die Überschriften (*topics*) werden in einem Index zusammengefaßt, den der Benutzer durchblättern kann. Weiterhin kann man Überschriften auch in einem Inhaltsverzeichnis hierarchisch gruppieren. Es besteht die Möglichkeit, mehrere Inhaltsverzeichnisse mit jeweils verschiedenen Untermengen der Überschriften zu bilden. So können *Guided Tours* durch den Informations-

raum gelegt werden, um dem Benutzer damit verschiedene Zugänge zu den Inhalten zu offerieren (Abbildung 15).

- **Stichwortsuche**: Die Stichwortsuche findet im Index durch Blättern oder Eingabe der ersten Buchstaben des gesuchten Wortes statt.
- **Volltextsuche**: WinHelp legt einen Volltextindex an, indem nach beliebigen Worten oder mittels regulärer Ausdrücke gesucht werden kann.
- **Textuelle Verweise (*Links*)**: Von jedem beliebigen Zeichen aus kann ein Link zu einer Überschrift definiert werden. Der Benutzer gelangt durch Klicken mit der Maus auf diesen Link zu dem referenzierten Thema.
- **Hotspot–Graphiken**: In WinHelp eingebundene Graphiken können mit rechteckigen Bereichen (*hot spots*) versehen werden, die dieselbe Funktionalität aufweisen, wie die textuellen Verweise. Aus einer Graphik kann der Benutzer somit auch zu anderen Themen verzweigen.
- **Aufruf von Applikationen**: Außer der Auslösung eines Sprungs hin zu einem anderen Thema kann durch einen Verweis auch ein Makro ausgeführt werden. Technisch gesehen sind die Makros Funktionsaufrufe mit Parameterübergabe/–rückgabe in einer DLL. So lassen sich beliebige Programme ausführen.

Folgend Beispiele aus der Hypertextdokumentation zur Implementation des OSA–Servers. In Klammern steht die Einordnung in die Abstraktionsebenen aus Abbildung 13.
Die Sequentialisierung eines Hypertextes hat natürlich zur Folge, daß die Verweise (textuelle und *hot spots*) nicht mehr "funktionieren". Des besseren Eindrucks wegen habe ich die Hervorhebung eines Verweises durch doppelte Unterstreichung belassen.
Zur Unterscheidung zum übrigen Text der Dokumentation sind die Hypertexte durch horizontale Balken begrenzt.

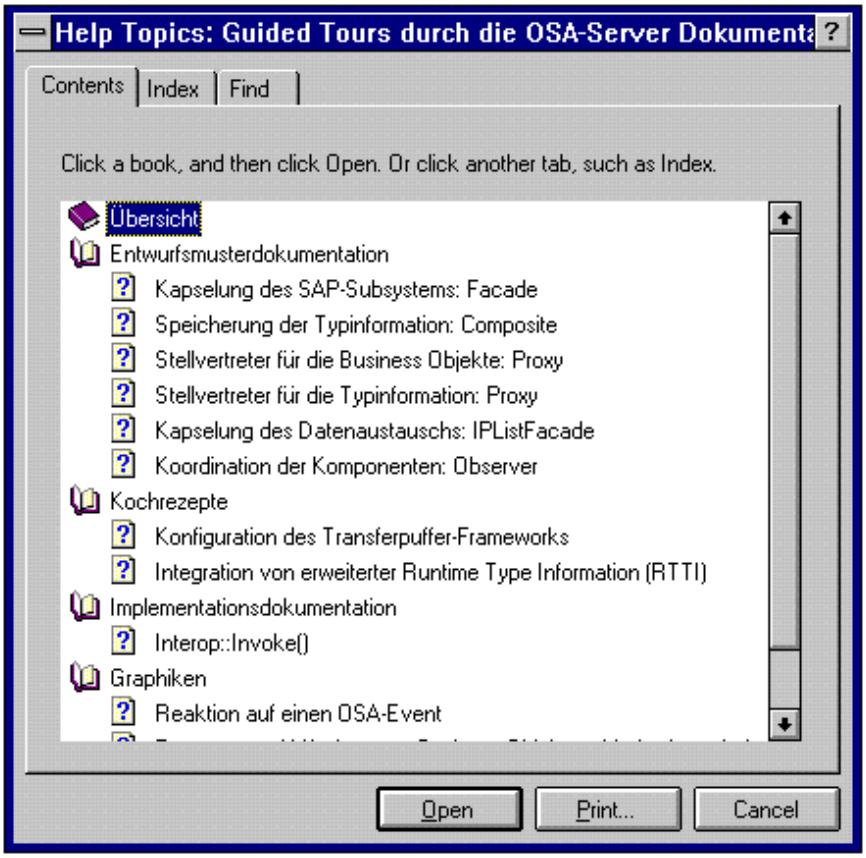


Abbildung 15: Inhaltsverzeichnis mit *Guided Tours*

Abbildung 16: Klassendiagramm – Gesamtübersicht

4.2.1 Klassendiagramm als Gesamtübersicht (Ebene 1)

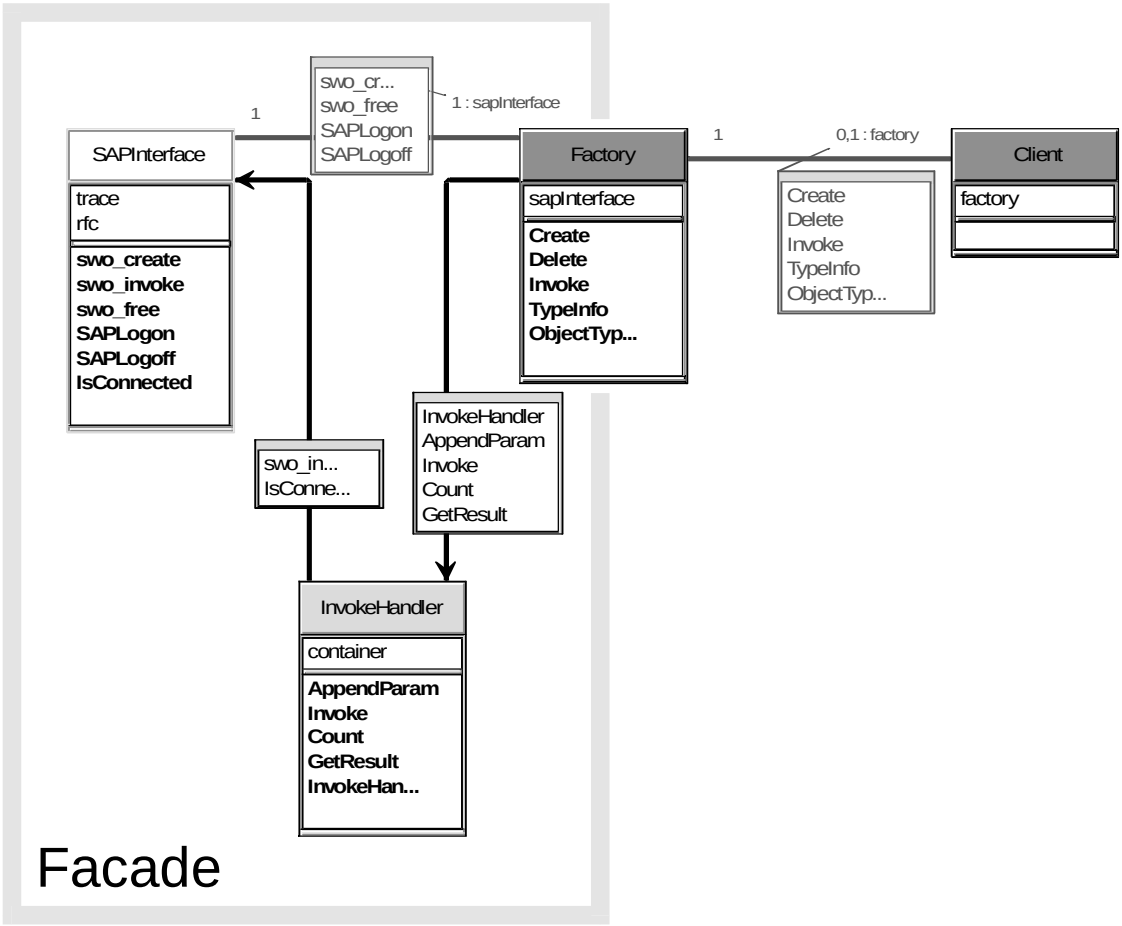
Die Gesamtübersicht (Abbildung 16) trägt solange nicht zur Übersicht bei, wie nicht ein gewisses Verständnis der Komponenten gegeben ist. Dann jedoch ist der Methodenfluß zwischen den Komponenten dort nachvollziehbar. Anfänglich gelangt der Benutzer durch *Hot Spots* von den Musterbezeichnungen (Facade, etc.) der Gesamtübersicht hin zu den Entwurfsmuster-dokumentationen.

4.2.2 Entwurfsmusterdokumentation der SAP–Komponente (Ebene 2)

Als erstes Beispiel für eine Entwurfsmusterdokumentation habe ich die Beschreibung der SAP–Komponente gewählt. In dieser Komponente ist die für die Kommunikation mit einem SAP–System nötige Funktionalität vereint.

Kapselung des SAP–Subsystems: *Facade*

Übersicht



Intention

Um die Internas der RFC–Kommunikation vor dem Klienten zu verbergen und so auch die Austauschbarkeit der SAP–Anbindung zu gewährleisten, wurde die RFC–Schnittstelle gekapselt.

Motivation

Die Kapselung ist auf mehrere Klassen verteilt, um die Übersicht zu erhöhen und nicht mehrere Abstraktionen in einer Klasse anzulegen.
Ein wichtiger Anspruch war, dem Klienten eine von SAP–Datentypen freie Schnittstelle anzubieten (siehe Factory und InvokeHandler). Somit sind alle SAP–Spezifika in dem von den drei Klassen SAPInterface, Factory und InvokeHandler gebildeten Subsystem isoliert.
Eine weitere wichtige Forderung im Zusammenhang mit der Änderbarkeit der Komponenten war die signifikante Reduzierung der Schnittstellenbreite. Die im Durchschnitt 4 Parameter pro Methode an der Schnittstelle von SAPInterface konnten so auf durchschnittlich 1,5 Parameter pro Methode bei Factory und InvokeHandler verringert werden.
Das Protokoll der RFC–Schnittstelle wurde beibehalten und an die Schnittstellenklasse Factory weitergegeben. Der Klient erzeugt also weiterhin Objekte (Factory::Create(), Factory::TypeInfo()), ruft Methoden auf den Objekten auf (Factory::Invoke()) und löscht Objekte (Factory::Delete()). Die An– und Abmeldung am SAP–System (SAPInterface::SAPLogon(), SAPInterface::SAPLogoff()) erfolgt nicht mehr explizit, sondern wird bei der Erzeugung bzw. dem Zerstören einer Instanz von Factory bewerkstelligt.
Da der Methodenaufruf (SAPInterface::swo_invoke()) einen komplexen Ablauf mit Vorbereitungsphase (Aufbau der Containerdatenstruktur) und Auswertung (Ergebnisdaten abfragen) erfordert, wurde eine eigene Klasse InvokeHandler für diese Aktion definiert. Eine Instanz von InvokeHandler wird durch einen Aufruf vonFactory::Invoke() erzeugt und dem Klienten zurückgegeben. Dieser kopiert die Aufrufparameter (InvokeHandler::AppendParam()), löst den Aufruf aus (InvokeHandler::Invoke()) und fragt anschließend die Ergebnisparameter ab (InvokeHandler::GetResult()).

Rollen

Facade (SAPInterface)

- Kapselt die RFC–Schnittstelle: Initialisierung der Containerstrukturen; Auswertung.
- Leitet Statusinformationen durch Methoden der Klasse Trace in eine Log–Datei.

Subsystem class (Factory)

- Realisiert die Objekterzeugung, die Abfrage der Objekttyp–Schnittstelle und das Löschen von Objekten.
- Die Statusinformationen der RFC–Schnittstelle werden im Fehlerfalle in Ausnahmen (*Exceptions*) umgesetzt.

Subsystem class (InvokeHandler)

- Realisiert die Vor– und Nachbereitung eines Methodenaufrufs.
- Die Statusinformationen der RFC–Schnittstelle werden im Fehlerfalle in Ausnahmen (*Exceptions*) umgesetzt.

Konsequenzen

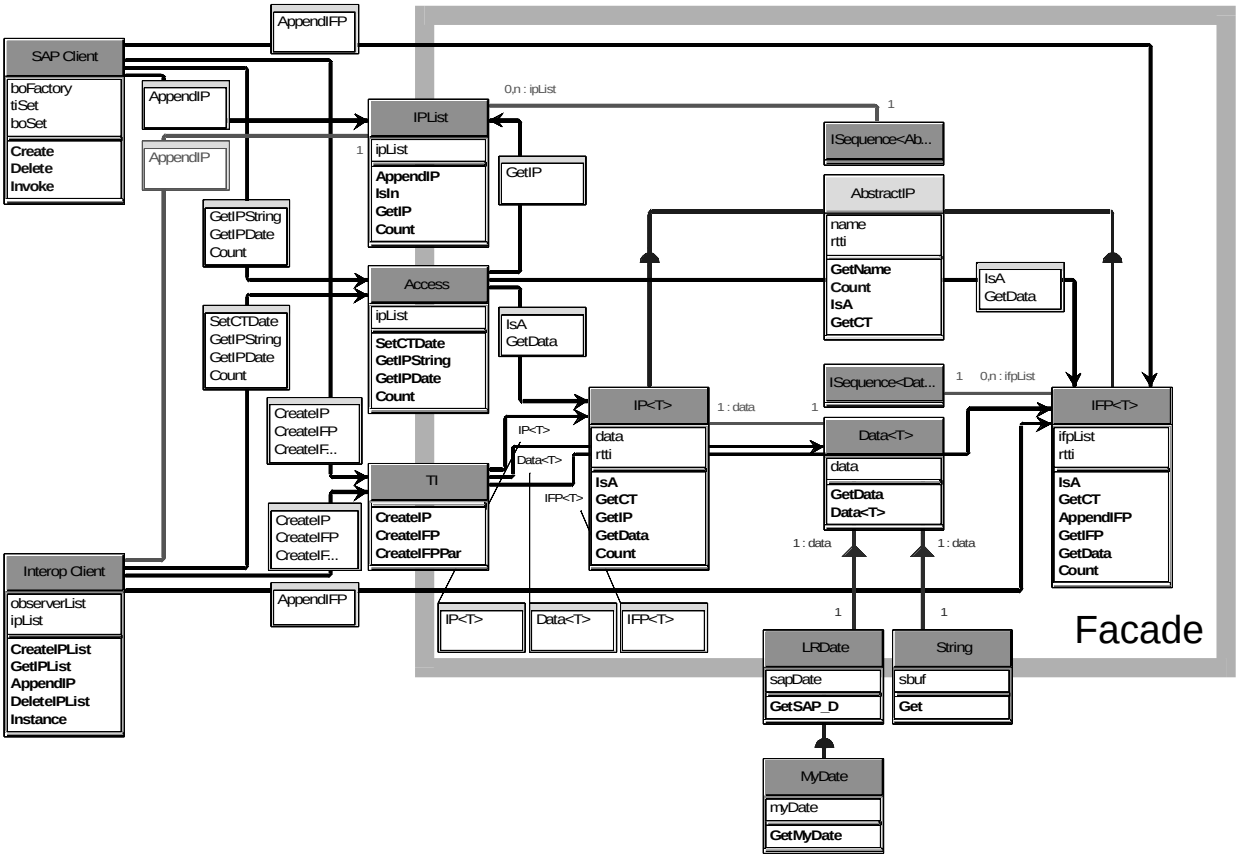
- Die unhandliche und generalisierte Bedienung der RFC–Schnittstelle wird auf das Nötigste zur Kommunikation mit dem*Business Object Broker* reduziert resp. spezialisiert.
- Die C–API der RFC–Schnittstelle ist in Klassen "verpackt" worden.
- Da auf der Ebene von Factory und InvokeHandler nur noch in Termen der Problemdomäne agiert wird, ist ein relativ leichter Austausch der unterliegenden Kommunikationsschnittstelle bezüglich der Klienten von Factory und InvokeHandler gegeben.

Als zweites Beispiel für eine Entwurfsmusterdokumentation habe ich das Framework Transferpuffer ausgewählt. Das instanziierte Entwurfsmuster ist wiederum *Facade* [Gamma et al., 1995, S.185]. Wie im Abschnitt "Bemerkung" auf Seite 45 beschrieben, beinhaltet die Facade–Instanz

eine Instanz des Composite–Musters in einer erweiterten Form. Durch diese Erweiterung und eine Anpassung an die Problemdomäne (siehe Seite 45) ist das Composite–Muster auch strukturell stark verändert worden. Um trotzdem die Zuordnung zwischen Muster und dessen Instanz herzustellen, ist eine textuelle Beschreibung der Entwurfsentscheidungen unerläßlich.
Die Konfiguration des Frameworks ist im Beispiel zum Kochrezept (Abschnitt 4.2.3) beschrieben.

Kapselung des Datenaustauschs: *Facade*

Übersicht



Intention

Der Datenaustausch zwischen der Interoperabilitätskomponente und der SAP–Komponente soll möglichst flexibel und erweiterbar sein, um einen hohen Grad der Entkopplung zwischen diesen Komponenten zu erreichen und den Datenfluß zu zentralisieren.
Stichworte sind: Implizite Typkonvertierung der Daten (SAP– in OSA–Typen und umgekehrt), Integration benutzerdefinierter Typen, Laufzeittypüberprüfung (RTTI), leichte Benutzbarkeit durch schmale Schnittstellen.
Das entstandene Framework Transferpuffer (in der Implementation auch als IPList bezeichnet) kann ohne Modifikation generisch erweitert werden durch Vererbung von den spezifischen Datentypklassen (siehe Kochrezept). Dieses Framework wird in der vorliegenden Entwurfsmusterdokumentation beschrieben.

Bemerkung

Das Entwurfsmuster Facade stellt den "allgemeinsten Nenner" zur Klassifizierung des Frameworks dar. Da die verwalteten Daten analog der Typinformation (Composite) auch eine Baumanordnung darstellen, ist auch hier das Muster Composite instanziiert, allerdings in einer erweiterten Form.
Das von Gamma ([Gamma et al., 1995, S.163ff]) beschriebene Muster *Composite* paßt sehr gut auf eine Systemkomponente, die zwar Daten (in seinem Beispiel graphische Objekte) in einer Baumstruktur verwaltet, aber nur über eine *steuernde* Schnittstelle verfügt (im Beispiel die Methode void Draw() zum Neuzeichnen der graphischen Objekte). Sobald die Abfrage von Daten nötig wird (auf Gammas Beispiel übertragen z.B. eine Methode Get(), die ein graphisches Objekt zurückliefert) verkompliziert sich die Klassenstruktur erheblich, da generische Klassen (in C++ als Templates bezeichnet) für die Blattklassen verwendet werden müssen, um eine Methode der Art T Get() zur Verfügung zu haben. Diese kompliziertere Struktur wirkt sich auch auf die Benutzung der Klassenschnittstellen aus. Um die Benutzung so einfach wie möglich zu halten, habe ich die Parametererzeugung und die Abfrage in den zwei Schnittstellenklassen TI und Access zusammengefaßt. Das Kochrezept zeigt die Verwendung dieser Klassen.
Zwei Eigenschaften der Problemdomäne sind in die Struktur des Entwurfs abgebildet:

- Parameterfelder sind homogen: Die Methode eines *Business Objekts* erhält eine Liste von heterogenen Datenobjekten. Ein solcher Übergabeparameter kann wiederum aus einer Liste von Datenobjekten bestehen; diese müssen allerdings homogen, also von einem Typ sein.
  Die Konsequenz für den Entwurf: Die Klasse IFP ist generisch. Dadurch können immer nur Datenobjekte eines einzigen Typs angefügt werden (AppendIFP()).
- Ein Übergabeparameter kann ein Parameterfeld sein; der Parameter eines Parameterfelds nicht. Daraus ergibt sich eine Rekursionstiefe von maximal eins für die Datenobjekte.
  Die Konsequenz für den Entwurf: Statt einer Assoziation zwischen der Klasse IFP<T> und AbstractIP, wie sie üblicherweise besteht (*1:N Recursive Connection Metapattern* [Pree, 1995, S.126]) wurde die Blattklasse aufgeteilt (in die Klassen IP<T> und Data<T>) und eine Assoziation von IFP<T> zu Data<T> gelegt. IFP<T> verwaltet somit eine Liste von Instanzen der Klasse Data<T> und nicht, wie üblich eine Liste von Instanzen der Klasse AbstractIP (und abgeleiteter Klassen). Letzteres hätte zur Folge, daß die von IFP<T> verwalteten Parameterfelder wiederum Parameterfelder enthalten könnten, was nicht erwünscht ist.

Die im Transferpuffer verwendete Laufzeittypidentifikation (RTTI) mit Beachtung der Vererbungshierarchie ist eventuell als Muster einzuordnen.

Motivation

Der Sinn und Zweck des OSA–Servers ist der Datenaustausch zwischen einem SAP–System und einer durch ein OSA–Script repräsentierten Abfrage an dieses System. Die Daten sind hierbei in einfache Parameter und Feldparameter (Vektoren aus einfachen Parametern) zu unterscheiden. Da die Daten getypt sind, muß eine Konvertierung stattfinden. Was die Datentypen angeht, wurde der Transferpuffer so ausgelegt, daß die SAP–Datentypen als Vorgabe die Menge der möglichen Datentypen festlegt. Neue Datentypen lassen sich also nur als Spezialisierung der vorhandenen SAP–Datentypen integrieren.
Alle Methodenparameter der *Business Objekte* sind *call by name*. Daher werden auch die Daten im Transferpuffer durch eindeutige Namen angesprochen (siehe Interop).
Die Klassen, mit denen dieser Transferpuffer realisiert wurde, sind nicht für die direkte Verwendung gedacht. Statt dessen dienen die Schnittstellenklassen Access und TI (Template Instanziierer) zur Steuerung der Vorgänge.

Rollen

Facade (TI)

- Dient zur Erzeugung der getypten Container, in denen die Nutzdaten transportiert werden.
- Kapselt den aufwendigen Konstruktionsprozeß der Container.

Facade (Access)

- Stellt Methoden für den Zugriff auf die im Transferpuffer gehaltenen Daten bereit.
- Fängt alle Ausnahmen (*Exceptions*) der unterliegenden Klassen.

Subsystem class (AbstractIP)

- Definiert abstrakte Parameter. Alle Parametercontainer werden so einheitlich angesprochen.
- Verwaltet die Namen der Parameter.
- Implementiert *Runtime Type Identification* (RTTI) für die Parameter.

Subsystem class (IP<T>)

- Der generische Container für normale Parameter.
- Implementiert RTTI für die Parametercontainer.

Subsystem class (IFP<T>)

- Der generische Container für Parameterfelder.
- Implementiert RTTI für die Feldparametercontainer.

Subsystem class (Data<T>)

- Der generische Container für die verschiedenen Datentypen.

Konsequenzen

- Durch die Kapselung der Typkonvertierung und durch die schmale Schnittstelle ist eine sehr einfache Bedienung (sprich Integration) gegeben.
- Der Austausch der Interoperabilitätskomponente gestaltet sich einfacher durch diese Kapselung, da nur für die spezifischen Datentypen Klassen implementiert werden müssen.

Frameworkeigenschaften

Das Framework Transferpuffer eignet sich für die Datenkopplung von unterschiedlichen Systemkomponenten. Hierbei wird in eine feste Komponente und eine austauschbare unterschieden. Die Menge der verfügbaren Datentypen ist hierbei durch das Framework festgelegt (sichtbar an den Zugriffsmethoden der Klasse Access) und auf die feste Komponente zugeschnitten. Diese Komponente verwendet nur die durch die Klasse Access angebotenen Methoden zum Zugriff auf die Daten. Die andere variable Komponente kann die Datentypen des Frameworks auf eigene spezifische Datentypen anpassen.
So können austauschbare Komponenten mit einer festen Komponente über einen semantisch vorbestimmten aber syntaktisch variablen Satz von Datentypen bidirektional kommunizieren.
Das Framework kann ein- und mehrelementige Daten (Vektoren) und heterogene Sequenzen aus Elementen transportieren.
Im Kochrezept ist die Konfiguration des Frameworks beschrieben.

Hot Spots

Die Hotspots des Frameworks Transferpuffer sind die Datentypklassen, wie sie exemplarisch im Klassendiagramm für die Datentypen String und LRDate dargestellt sind.

Frozen Spots

Alle Klassen, bis auf die unter *Hot Spots* beschriebenen sind fest.

4.2.3 Kochrezept für die Konfiguration des Transferpuffer–Frameworks (Ebene 2–4)

Im folgenden Kochrezept wird die Konfiguration des Frameworks Transferpuffer (siehe auch die Entwurfsmusterdokumentation auf Seite 44) beschrieben. Um eigene Datentypen durch das Framework zu verwalten, werden die vorgegebenen Datentypen spezialisiert. Im Kochrezept wird die Spezialisierung des Datum–Typs LRDate beschrieben.

Kochrezept: Erweiterung des Transferpuffers

Vorbemerkung

Das Klassendiagramm zum Transferpuffer zeigt unten die als Klassen repräsentierten Datentypen String und LRDate (*Local Representation Date*). LRDate (Verwaltung des SAP–Datum–Typs) wurde spezialisiert in die Klasse MyDate (Verwaltung des OSA–Datum–Typs). Die Definition und Implementierung sowie die Verwendung der Klasse MyDate wird im Folgenden beschrieben. Hierdurch wird exemplarisch die Erweiterung des Transferpuffers um einen spezialisierten Datentyp nachvollzogen.

1. Definition der Basisklasse LRDate

Zunächst die Definition der Basisklasse. Die Basisklasse (hier LRDate) und die abgeleitete Klasse (MyDate) teilen sich die Arbeit: LRDate implementiert den Datentyp vollständig, was die SAP–Komponente angeht. Mit dem Konstruktor in Zeile 1 wird ein Datenobjekt durch die SAP–Komponente erzeugt. Die Methode GetSAP_D() aus Zeile 4 dient zur Abfrage eines Datenobjekts im SAP–Format (YYYYMMDD).
Der geschützte Konstruktor aus Zeile 5 und die Methode SetSAP_D() aus Zeile 6 dienen zur Konstruktion eines leeren Datenobjekts und anschließender Initialisierung mit einem Datumswert. Sie werden von der abgeleiteten Klasse verwendet.

```
        class CLRDate
        {
          public:
1               CLRDate(const TSAP_D& sapDate);
2               CLRDate(const CLRDate&);
3               ~CLRDate();

4               virtual PTSAP_D GetSAP_D() const;

         protected:
5               CLRDate();
6               void SetSAP_D(const TSAP_D& sapDate);

        private:
                String _sapDate;
      };
```

2. Definition der abgeleiteten Klasse MyDate

Die abgeleitete Klasse implementiert die für den Zugriff durch die Interoperabilitätskomponente nötige Funktionalität und die Konvertierung aus dem spezifischen Format in das SAP–Format. Der Konstruktor in Zeile 7 erwartet das Datum im spezifischen Format (hier DD.MM.YYYY), konvertiert es in das SAP–Format und initialisiert den geerbten Teil des Datenobjekts damit. Der Konstruktor in Zeile 8 wird zur umgekehrten Konvertierung eingesetzt.

Der Konstruktor aus Zeile 7 wird aufgerufen, um ein Datum im Fremdformat an die SAP–Komponente zu übergeben; der Konstruktor aus Zeile 8, um ein im SAP–Format vorliegendes Datum

in das Fremdformat zu konvertieren. Die Methode GetMyDate() aus Zeile 9 dient zur Abfrage des Datums im spezifischen Format.
Das Makro aus Zeile 10 schließlich deklariert die nötigen Datentypen in einer vorgegebenen Konvention, initialisiert die systemweit eindeutigen RTTI–Typnamen und deklariert eine statische Instanz der generischen Klasse CTI, die mit dem neuen Datentyp parametrisiert wird. Über diesen *Template–Instanziierer* werden die korrekt getypten Container erzeugt (s.u.).
Das Makro erhält den technischen Namen der Klasse und einen frei wählbaren Namen aus dem dann die Typnamen erzeugt werden. Zum besseren Verständnis darunter das Ergebnis des Makros.

```
        class CMyDate : public CLRDate
        {
          public:
7               CMyDate(const char* date);
8               CMyDate(const CLRDate& lrDate);
                CMyDate(const CMyDate&);
                ~CMyDate();

9               const char* GetMyDate() const;

          private:
                String _myDate;
        };

10      DECLARE_IP(CMyDate, MyDate)
        /***    Der vom Makro produzierte Code:

                typedef CIP<CMyDate> CIPMyDate;
                typedef CIPMyDate* PCIPMyDate;
                const char* CTCIPMyDate = typeid(CIP<CMyDate>).name();

                typedef CIFP<CMyDate> CIFPMyDate;
                typedef CIFPMyDate* PCIFPMyDate;
                const char* CTCIFPMyDate = typeid(CIFP<CMyDate>).name();

                typedef CData<CMyDate> CIFPMyDateParam;
                typedef CIFPMyDateParam* PCIFPMyDateParam;

                CTI<CMyDate> ctiMyDate;  // Der Template–Instanziierer
        ***/
```

Um eine Klasse zu implementieren, die ein spezifisches Datumsformat (oder ein spezifisches Zeit–, String–, etc. Format) verarbeitet, ist somit minimal die in MyDate deklarierte Schnittstelle nötig. Ferner der Aufruf des Makros DECALRE_IP.

3. Die Anwendung der Klasse MyDate

Die Datentypklassen werden zum Einen bei der Übergabe von Daten in einem Fremdformat an die SAP–Komponente eingesetzt; zum Anderen bei der Abfrage des Transferpuffers nach dem Aufruf der Methode eines *Business Objekts*. Die dann im SAP–Format vorliegenden Daten werden mit dem Konstruktor aus Zeile 8 in das Fremdformat konvertiert.
Folgend ein Code–Ausriß, der aus dem OSA–EventCallback stammen könnte:

```
        CIPList aParamList;
11      CAccess acc = CAccess(&aParamList);
12      acc.SetCTDate (CTCIPMyDate, CTCIFPMyDate);

        const char* paramName = "Date";
        aParamList.AppendIP(ctiMyDate.CreateIP (CMyDate(osaDate), paramName));
```

In der Methode Interop::Invoke() würde dieses Datum anschließend abgefragt (Zeile 13):

```
13      TSAP_D sapDate = acc.GetIPDate ("Date").GetSAP_D(),
14      aParamList.ClearList();

        // Der Invoke laeuft ab

15      aParamList.AppendIP(ctiLRDate.CreateIP (returnValue, "_RETURN"));
```

Der Aufruf in Zeile 12 macht den neu definierten Datentyp MyDate bei der Instanz von CAccess bekannt. Nach dem Abfragen der Parameterliste, wird deren Inhalt gelöscht (Zeile 14) und der Invoke ausgelöst. Anschließend werden eventuelle Rückgabeparameter wieder in den Transfer puffer geschrieben (Zeile 15). Interop gibt das Datum im SAP–Format weiter (der Template–Instanziierer ctiLRDate wird verwendet).
Im EventCallback wird das Datum im Fremdformat benötigt:

```
16      const char* myDate =  CMyDate(acc.GetIPDate("_RETURN")).GetMyDate());
```

Hierzu wird ein Objekt der Klasse MyDate mit dem zurückgegebenen Objekt der Klasse LRDate erzeugt (Zeile 16). Bei der Konstruktion wird das Datum entsprechend konvertiert und kann mit GetMyDate() abgefragt werden.

4. Beispielprogramm und interaktiver Test

Um die in den vorangegangenen Abschnitten dargestellten Sachverhalte besser zu verstehen, haben Sie die Möglichkeit ein Testprogramm ablaufen zu lassen und auch interaktiv mit dem "Turbo Debugger" der Firma Borland zu testen:
Starten Sie ein Beispielprogramm: Beispielprogramm
Starten Sie den Debugger: Interaktiver Test

4.2.4 Klassendokumentation der Klasse Interop (Ebene 3)

Zunächst die Beschreibung der Klasse Interop, der zentralen Steuerungsinstanz des Systems.
Anschließend auf Seite 58 ein Interaktionsdiagramm (Abbildung 17), das die Abläufe zur Reaktion auf einen eintreffenden OSA–Event darstellt. Die Klasse InteropOSAAdapter ist eine Spezialisierung der Klasse Interop (siehe Gesamtübersicht in Abbildung 16). Somit ist diese Klasse in den Botschaftsfluß indirekt involviert (Delegation der Aufrufe).
Als Beispiel für ein Objektlebenszyklus–Diagramm (Seite 58, Abbildung 18) dient die Klasse SAPInterface (siehe die Entwurfsmusterdokumentation auf Seite40).
Auf Seite 59 dann noch ein Objektdynamikdiagramm (Abbildung 19) der Datenstruktur zur Speicherung der Objekttyp–Information (Entwurfsmuster Composite inAbbildung 16).

Interop

Klassenbeschreibung: Interop

- **Aufgabe**: Die Klasse Interop ist die zentrale Klasse des OSA–Servers. Sie kontrolliert alle Anfragen, die von der OSA–Schnittstelle kommen, übersetzt die Anfragen, steuert den Aufruf an der RFC–Schnittstelle und die Konvertierung der Ergebnisdaten.
  Das durch die Musterinstanz beschriebene Subsystem Facade und der Transferpuffer werden in den Methoden dieser Klasse synchronisiert.
- **Bemerkungen**: Interop fungiert auf einem von den konkreten SAP– und Scripting–Terminologien abstrahierten Ebene. Dieser Sachverhalt ist an der Klassen–Schnittstelle direkt sichtbar:

Die durch die Methoden Create(), Delete() und Invoke() gebildete dynamische Aufrufschnittstelle (*dynamic invocation interface*) ist frei von Datentypen der Problemdomänen. (Einzig der semantische Sachverhalt, daß persistente *Business Objekte* über einen Schlüssel (*objKey*) verfügen, schlägt sich in der Parameterliste von Create() nieder).
Im Übrigen sind Objekte vor ihrer Erzeugung durch den Objekttyp (*objType*) benannt und werden, nachdem sie erzeugt sind, über den Objekttyp und einen ganzzahligen Index (*objHandle*), der bei der Erzeugung vom *Business Object Broker* vergeben wurde, eindeutig angesprochen. Attribute, Methoden und deren Parameter (*verb*) sind ebenfalls benannt (die Parameterübergabe an die Methoden der *Business Objekte* erfolgt analog den ABAP/4–Konventionen generell *call by name*).
Der Datentransfer findet über das Transferpuffer–Framework IPList (*Invoke Parameter List*) statt.
Der Vorteil dieses Vorgehens besteht in der leichten Austauschbarkeit der Scripting–Komponente und der Kommunikationskomponente zum SAP–System.
Die Implementation der konkreten Scripting–Komponente wird in einer von Interop abgeleiteten Klasse realisiert. Die Interop–Methodenaufrufe erfolgen aus den überladenen Methoden der abgeleiteten Klasse (hier: InteropOSAAdapter) unter Ausnutzung der Polymorphie.
Die Scripting–Komponente muß eindeutige Namen für Objekttypen verwalten können sowie Namen für die Attribute, Methoden und Parameter. Um die erzeugten Objekte eindeutig ansprechen zu können ist die Verwaltung des Laufzeitidentifikators nötig. Für persistente Objekte muß schließlich auch noch der Schlüssel verwaltet werden. Für die Datenkommunikation muß die Datenstruktur IPList des Transferpuffers unterstützt werden.
Auch die durch das Muster Facade dokumentierte Kommunikationskomponente hin zum SAP–System ist leicht austauschbar, da die Klasse Factory, die die Schnittstelle der Komponente darstellt auch schon über eine genügend abstrakte und schmale Schnittstelle verfügt.

- **Protokolle**: Interop definiert auf einer abstrakten Ebene das Protokoll Create, Delete, Invoke zur Erzeugung und Löschung von *Business Objekten* sowie zum Methodenaufruf. Im **Interaktionsdiagramm** wird die Bearbeitung eines eintreffenden OSA–Events dargestellt.
- **Kardinalität**: Von Interop wird zur Laufzeit nur genau eine Instanz erzeugt. Die Erzeugung erfolgt implizit während der Erzeugung der Instanz von InteropOSAAdapter.
- **Lebensdauer**: Systemlaufzeit.
- **Muster**: Observer.
- **Ausnahmebehandlung**: Alle von falschen Aufrufparametern resultierenden Fehler werden durch Ausnahmen beantwortet. Die von der Komponente Factory erzeugten Ausnahmen werden in Ausnahmen vom Typ InteropException übersetzt.
- **Used by**: InteropOSAAdapter.
- **Uses**: Factory, BOSet, BOProxy, TISet, TypeInfo
- **Implementation**: Die Implementierung der Methode Invoke() ist kommentiert verfügbar.
- **Testumgebung**

```
interface Interop
{
  //-------- Konstruktor etc.:
  Interop();
  virtual ~Interop();

  //-------- Die dynamische Aufrufschnittstelle (DII):
  virtual MyBoolean Create(const char* objType, const char* objKey, IPList* ipList);
  virtual MyBoolean Delete(unsigned objHandle);
  virtual MyBoolean Invoke(unsigned objHandle, const char* verb, IPList* ipList);

  //-------- Deklaration der Ausnahmebehandlung:
```

```
  interface InteropException : public Exception
}  // Interop
```

Interop::Interop()

Interop()

- **Aufgabe**: Erzeugt die lokal verwalteten Instanzen von Factory, BOSet und TISet.
- **Implementation**
- **Ausnahmebehandlung**: Schlägt die Anmeldung (Logon) am SAP–System fehl, wird die durch Factory ausgelöste Exception hier übersetzt und weitergegeben.

Interop::~Interop()

virtual ~Interop()

- **Aufgabe**: Zerstört die lokal verwalteten Instanzen von Factory, BOSet und TISet.
- **Implementation**

Interop::Create()

virtual MyBoolean Create(const char* objType, const char* objKey, IPList* ipList)

- **Aufgabe**: Erzeugung eines neuen *Business Objekts* im SAP–System; Erzeugung des Stellvertreterobjekts im OSA–Server und Eintrag von Objekttyp und Laufzeitidentifikator (Handle) in die übergebene Referenz der Aufrufparameterliste (IPList).
- **Übergabeparameter**:
  objType – Der Typ des *Business Objekts*, das erzeugt werden soll.
  objKey – Der Schlüssel des persistenten Objekts, das erzeugt werden soll (optional).
  ipList – Referenz auf den Transferpuffer vom Typ IPList.
- **Rückgabeparameter**:
  true – Die Objekterzeugung verlief (auch lokal im OSA-Server) erfolgreich.
  false – Ein Speicherfehler ist aufgetreten.
- **Implementation**
- **Ausnahmebehandlung**: Schlägt die Erzeugung des *Business Objekts* fehl, wird eine Exception ausgelöst.

Interop::Delete()

virtual MyBoolean Delete(unsigned objHandle)

- **Aufgabe**: Löschen eines *Business Objekts* im SAP–System; Löschen des Stellvertreterobjekts im OSA–Server.
- **Übergabeparameter**:
  objHandle – Der eindeutige Laufzeitidentifikator des *Business Objekts*.
- **Rückgabeparameter**:
  Momentan immer true.
- **Implementation**
- **Ausnahmebehandlung**: Ist das *Business Objekt* nicht vorhanden oder schlägt die Löschung fehl, wird eine Exception ausgelöst.

Interop::Invoke()

virtual MyBoolean Invoke(unsigned objHandle, const char* verb, IPList* ipList)

- **Aufgabe**: Ausführen der durch verb bezeichneten Methode (bzw. der impliziten Abfragemethode des Attributs) des durch objHandle eindeutig identifizierten *Business Objekts*. Falls verb eine Methode mit Übergabeparametern bezeichnet, stehen im Transferpuffer ipList die

Übergabeparameter. Die Rückgabewerte werden ebenfalls in ipList retourniert, nachdem die Liste geleert wurde.

- **Übergabeparameter**:
  objHandle –Der eindeutige Laufzeitidentifikator des *Business Objekts*.
  verb – Der Bezeichner der auszuführenden Methode.
  ipList – Referenz auf den Transferpuffer (IPList), indem die Werte der Übergabeparameter stehen und in den nach dem Aufruf der Rückgabewert eingetragen wird.
- **Rückgabeparameter**:
  Momentan immer true.
- **Implementation** und ein Link zur Implementationsdokumentation.
- **Ausnahmebehandlung**: Ist das Verb nicht in der Signatur des Objekttyps oder ist die Zahl und der Typ der Übergabeparameter nicht korrekt oder ist einer der nicht–optionalen Parameter nicht angegeben, wird eine Exception ausgelöst. Ebenso, wenn ein Fehler beim Aufruf der Methode oder beim Abfragen der Rückgabewerte im SAP–System auftrat.

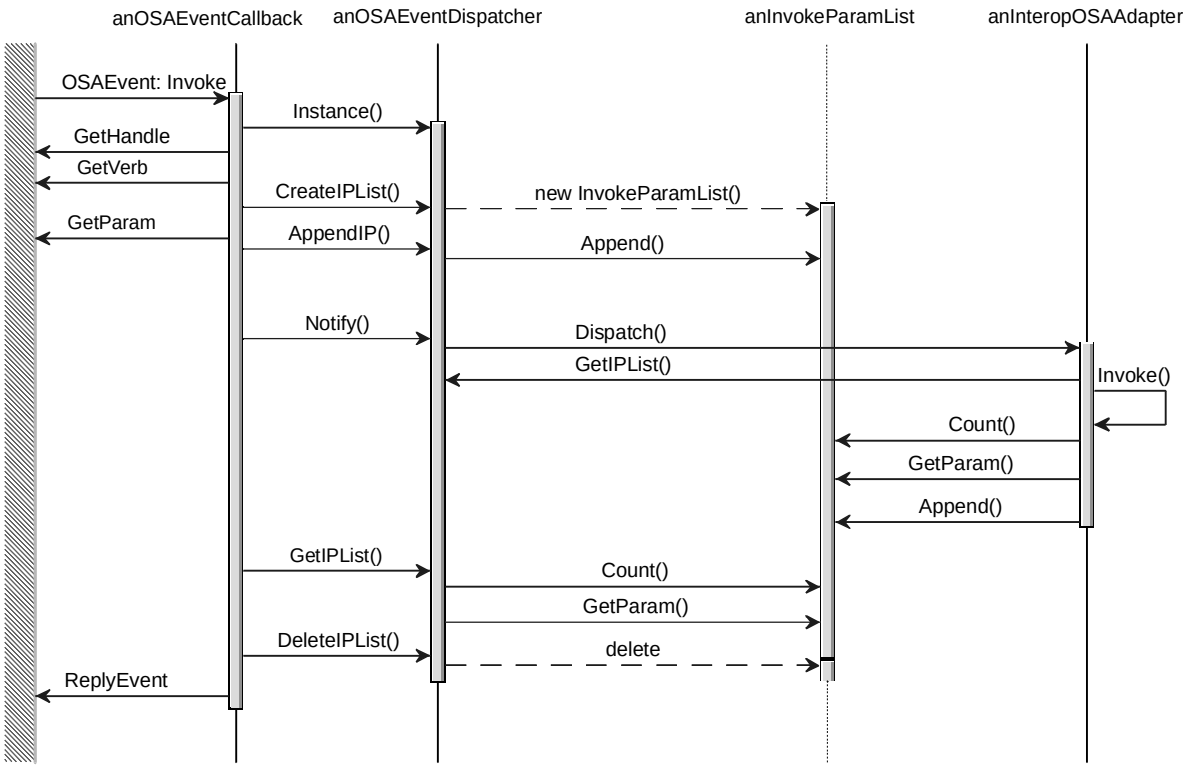


Abbildung 17: Interaktionsdiagramm: Reaktion auf einen OSA-Event

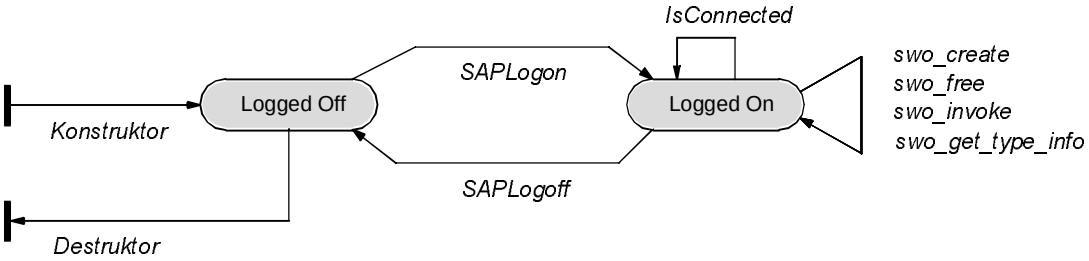


Abbildung 18: Objektlebenszyklus: Die Zustände der Klasse SAPInterface

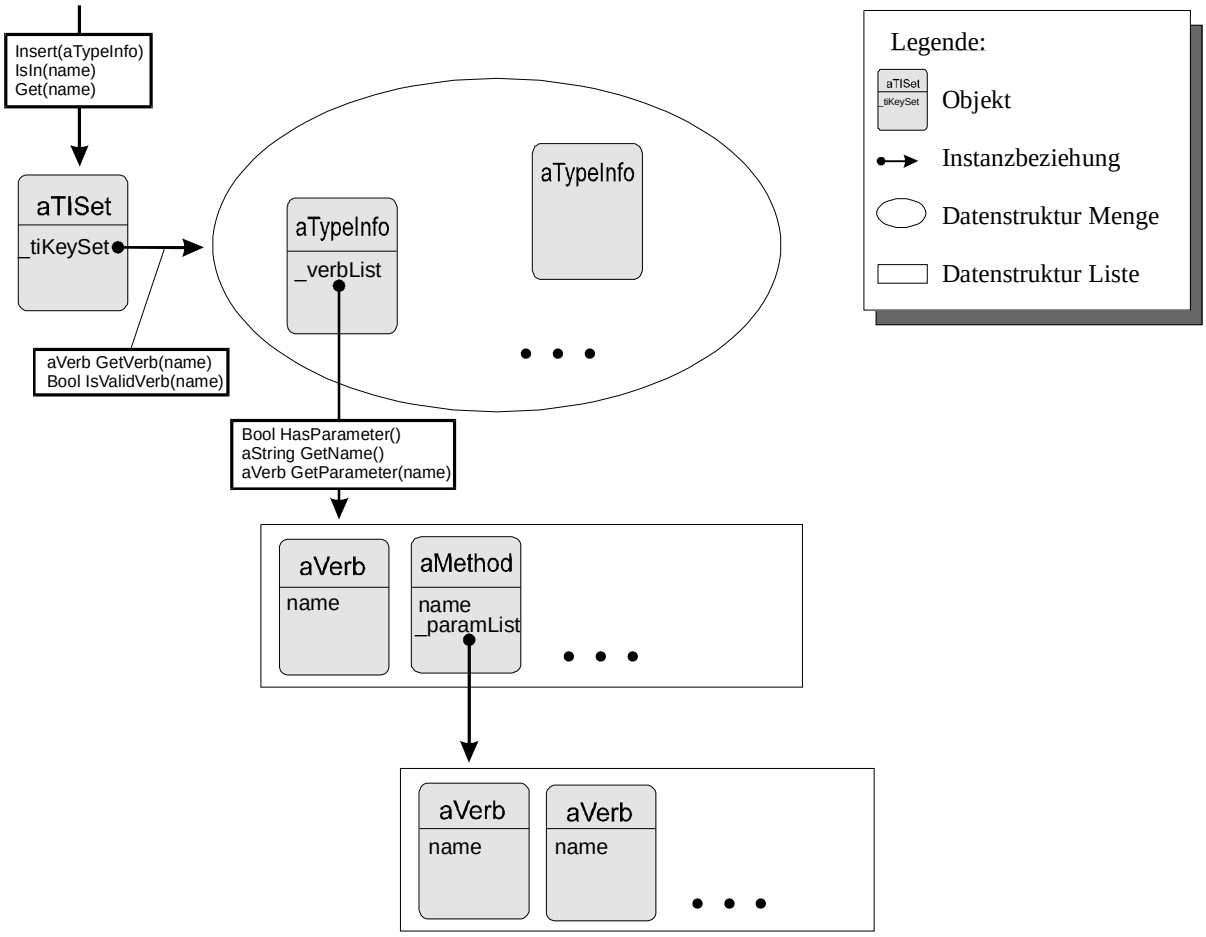


Abbildung 19: Objektdynamik: Datenstruktur zur Verwaltung der Typinformation

4.2.5 Implementationsdokumentation (Ebene 4)

Im folgenden Beispiel zur Implementationsdokumentation wird die Methode Invoke der Kasse Interop (siehe Seite 51) beschrieben. Da im Code dieser Methode mehrere Systemkomponenten zusammengreifen, ist sie von besonderem Interesse und wurde deshalb auf diese Art dokumentiert. Der Leser kann beim Durcharbeiten zu den tangierenden Themen verzweigen und erhält so einen Überblick "von unten" im Gegensatz zur Herangehensweise "von oben" durch die Entwurfsmusterdokumentationen (siehe die Abstraktionshierarchie in Abbildung 13).

Interop::Invoke() - Sourcecode

Vorbemerkung

Im Folgenden wird der Ablauf eines Methodenaufrufs auf einem Business Objekt anhand des Sourcecodes der Methode Interop::Invoke() dargestellt.
Wichtige Methodenaufrufe sind mit einem Link hin zu deren Beschreibung versehen.
Die Aufrufe der Methoden von IPList sind nicht näher dokumentiert, da die Schnittstelle hier in einer älteren Version verwendet wird. Die aktuelle Version von IPList konnte nicht mehr in den übrigen Code integriert werden (siehe Transferpuffer).
Der Aufruf erfolgt mit dem Laufzeitidentifikator als erstem Parameter (*objHandle*), dem eindeutigen Namen der Methode, die aufgerufen bzw. des Attributs, dessen Wert abgefragt werden soll (*verb*) sowie einem Zeiger auf einen Transferpuffer mit den Methodenparametern (*ipList*):

```
MyBoolean CInterop::Invoke(unsigned objHandle, const char* verb, IPList* ipList)
{
 MyString objType;
 char buf[300];
```

Zunächst wird der Typ des *Business Objekts* bestimmt. Eine evtl. entstehende Ausnahme wird in eine Ausnahme vom Typ InteropException umgewandelt.
Anschließend wird das übergebene Verb auf Zugehörigkeit zur Schnittstelle des Business Objekts hin überprüft. Wenn das Verb aus der Signatur des Objekttyps stammt, wird ein Zeiger auf dessen Datensatz in TISet angefordert.

```
 try
 {
  objType = _boset->get(objHandle)->GetType();
 }
 catch(CBOSet::CBOSetException x)
 {
  throw CInteropException(__FILE__, __LINE__, "Object is not in BOSet");
 }

 if(_tiset->get(objType)->IsValidVerb(verb) == true)
   {
    PCVerb v = _tiset->get(objType)->GetVerb(verb);

    CInvokeHandler* invoke = 0;
```

Die Instanz von Factory erzeugt eine Instanz von InvokeHandler. Diese Instanz wird im Folgenden mit weiteren Daten, die für den eigentlichen Aufruf im SAP–System nötig sind, parametrisiert.
Wenn das übergebene Verb Parameter besitzt, ist es eine Methode des Objekttyps.

```
    try
    {
     invoke = _factory->Invoke(objHandle, v);

     if(v->HasParameter()== true)
       {
            PCMethod m = (PCMethod)v;  // From now lets talk about verb as to be a method

            if(ipList->Count() > m->GetParamCount())
              {
               throw CInteropException(__FILE__, __LINE__, "More Parameters passed as expected");
              }
```

Es wird anhand der Typinformation über alle Parameter iteriert. Zunächst wird der Name des Parameters abgefragt. Dann wird ein Zeiger auf den entsprechenden Container angefordert. Wenn der Parameter nicht in der Liste enthalten ist und es ein nicht–optionaler Parameter ist, wurde der Aufruf falsch parametrisiert. Sonst wird der Wert abgefragt, in eine Zeichenkette konvertiert (die RFC–Schnittstelle transportiert nur stringifizierte Daten) und an die Instanz von InvokeHandler übergeben.

```
            CMethod::TParamIterator pi = m->GetParamIterator();
            for(pi.First() ; !pi.IsDone() ; pi.Next())
              {
               const char* name = pi.Item()->GetName().GetSZ();
               PCAbstractInvokeParam iParam = ipList->GetParam(name);

               if((iParam == 0) && ( pi.Item()->IsOptional()== false))
```

```
{
 sprintf(buf, "Non-optional parameter ’%s’ is not specified",
```

```
                pi.Item()->GetName().GetSZ());
            throw CInteropException(__FILE__, __LINE__, buf);
            break; // of for()
           }

          const char* value = ((PCIPString)iParam)->GetValue().Get();

          invoke->AppendParameter(name, value);
         }

       ActIPList()->ClearList();

  } // of if(v->HasParameter() == true)
```

Danach wird der eigentliche Invoke ausgelöst (siehe Interaktionsdiagramm). Wenn der Aufruf einer Methode einen Rückgabewert liefert oder der Wert eines Attributs abgefragt wurde, wird dieser Wert aus der Instanz von InvokeHandler in den Transferpuffer kopiert.

```
invoke->Invoke();

MyBoolean result = false;
if(v->HasParameter() == true)
 {
     if(((PCMethod)v)->HasResult()== true)
      {
       result = true;
      }
 }
else if(v->GetElemtype()== CVerb::Export)
  result = true;
```

Der Rückgabewert kann ein Feldparameter sein, daher wird invoke in einer Schleife abgefragt. Es folgen entsprechende Operationen je nach Typ des zurückgelieferten Wertes.

```
// Method or Attribute with result:
if(result == true)
 {
     for(unsigned index=0 ; index < invoke->GetResultCount(); index++)
       {
```

Wenn der Rückgabewert ein Laufzeitidentifikator zu einem bereits im SAP–System erzeugten *Business Objekt* ist, wird der entsprechende Objekttyp und ggf. Schlüssel beim SAP–System (via Factory) abgefragt und bei Erfolg das entsprechende Stellvertreterobjekt (BOProxy) erzeugt.
Die Klasse Coerce ist lokal im Modul definiert und nimmt ggf. Konvertierungen vor.
Die Rückgabe von Objekttyp und Laufzeitschlüssel an die OSA–Komponente erfolgt über den Transferpuffer.

```
          if(v->GetType()== CVerb::O)
           {
            Coerce coe;

            unsigned objHandle = coe.Unsigned(invoke->GetResult(index+1));

            if(objHandle == 0)
                 {
                  throw CInteropException(__FILE__, __LINE__, "Invalid Handle returned);
                 }

            MyString objType, objKey;
```

```
try
{
    _factory->ObjectTypeKey(objHandle, objType, objKey);
}
catch(CBOFactory::CBOFactoryException x)
{
    throw CInteropException(__FILE__, __LINE__, "ObjectTypeKey failed");
}

MyBoolean ret = Create(objType.Get(), objKey.Get(), objHandle);
if(ret == false)
     {
      throw CInteropException(__FILE__, __LINE__, "Creation failed");
     }
else
     {
      ActIPList()->Append(new CIP(new CIPString(objType.Get()), nameBOType));
      ActIPList()->Append(new CIP(new CIPUnsigned(objHandle), nameBOHandle));
     }
}
```

Wenn der Rückgabewert das Tupel (Objekttyp, Datenbankschlüssel) war, wird ein*Business Objekt* auf SAP–Seite erzeugt sowie im OSA–Server das entsprechende Stellvertreterobjekt.

```
else if(v->GetType() == CVerb::o)
 {
  Coerce coe;

  MyString objType((coe.ObjectId(invoke->GetResult(index+1))).GetType());
  MyString objKey((coe.ObjectId(invoke->GetResult(index+1))). GetKey());

  MyBoolean ret = Create(objType.Get(), objKey.Get(), ipList);
  if(ret == false)
     {
      throw CInteropException(__FILE__, __LINE__, "Creation failed");
     }
 }
```

Wenn der Rückgabewert ein Datum war, wird es in eine lokale Repräsentation konvertiert, die in der OSA–Komponente wiederum in das von OSA erwartete Format konvertiert wird (für die aktuelle Implementation des Transferpuffers siehe IPList).

```
else if(v->GetType() == CVerb::D)
 {
  Coerce coe;
  CTemplateInstance ti;

  ActIPList()->Append(ti.CreateIPMyDateType(coe.MyDateType(invoke->GetResult(index+1))));
 }
```

Die Übertragung von Daten der übrigen Datentypen ist bisher nicht implementiert.

```
  // else if ...
 }
  }
}
catch(CInvokeHandler::CInvokeHandlerException x)
{
    CInteropException(__FILE, __LINE__, "InvokeHandlerException raisded");
}
```

```
delete invoke;  // Wichtig!
```

```
    return true;
   }
 else
   throw CInteropException(__FILE__, __LINE__, "Invalid verb");

 return true;

} // CInterop::Invoke()
```

4.3 Auswertung: Anforderungen an eine Entwicklungsumgebung

Das im Anhang B beigefügte Diskussionspapier ist im Rahmen eines Kontakts mit der Firma microTOOL, Berlin entstanden. Bezogen auf deren Entwurfswerkzeug "objectiF" habe ich Anforderungen an eine Infrastruktur und an die Methodik für die Wiederverwendung von Entwurfskomponenten formuliert. Die Feststellungen aus dem Diskussionspapier möchte ich hier noch um einige weitere Aspekte ergänzen.
Durch die praktische Erprobung (Abschnitt 4.2) der in Abschnitt 4.1 dargestellten Notationen und Konzepte ergeben sich drei wesentliche Anforderungen an eine integrierte Infrastruktur zur Software–Entwicklung:

- **Konsistenzerhaltung**. Die Konsistenz zwischen den Entwurfsmodellen (Klassendiagramme, Implementierung, etc.) selbst und zu den sie beschreibenden Dokumenten ist eine ganz wichtige Anforderung. Deshalb ist zur Realisierung einer mehrbenutzerfähigen Infrastruktur eine unterliegende **Datenbank** erforderlich.
  Weiterhin ein **Benachrichtigungssystem**, das die durch Veränderungen entstehenden Inkonsistenzen registriert, Änderungsberichte zusammenstellt und diese Berichte den verantwortlichen Personen zustellt (per Email). Nötige Änderungen von Verweisen, die durch einfache Umbenennung des Ziels entstanden, sollten automatisch nachgezogen werden.
  Ohne die automatische Registrierung von Änderungen ist die Übersicht in einem realistischen Projekt nicht zu gewährleisten.
  Idealerweise könnten Änderungen vor ihrer Ausführung auf ihre Auswirkungen hin simuliert werden.
- **Erweiterbarkeit**. Das Thema Entwicklungsumgebungen und Objektorientierung wird in der Literatur und in Artikeln weiterhin von der Methodendiskussion beherrscht. Für den im werkzeuggestützten Entwurf unerfahrenen Leser erscheint die Wahl der richtigen Methode (Coad/Yourdon, Booch, Rumbaugh, bald die *Unified Method*) als die wichtigste Entscheidung bei der Auswahl unter den angebotenen Umgebungen. Beim Einsatz eines Werkzeugs bemerkt man aber schnell, daß andere Kriterien wesentlich wichtiger für die produktive Arbeit sind: Außer der schon angesprochenen Konsistenzerhaltung ist das die Erweiterbarkeit und Konfigurierbarkeit der Menge der angebotenen Werkzeuge, die zur Erstellung von strukturierten Graphiken verwendet werden.
  Ein Beispiel ist das Metawerkzeug VISIO (Visio GmbH, München, http://www.visio.com). Mit VISIO kann man graphische Editoren durch Definition der graphischen Sprache (das Aussehen und die Semantik der zulässigen Knoten und Kanten) neu erstellen oder erweitern bzw. anpassen. Ein Problem in diesem Kontext stellt die Integration des Metamodells eines neuen oder veränderten Werkzeugs in das Metamodell der unterliegenden Datenbank dar.
  Softwareentwicklung ist in Abgrenzung zur Variantenkonstruktion kein geradliniger sondern ein kreativer Prozeß, bei dem am Anfang noch keine vollständige Klarheit über den Weg zum Ziel besteht, auch wenn das durch das Wasserfallmodell suggeriert wird und natürlich der Wunsch eines jeden Projektleiters ist. Diesem Umstand muß in der Werkzeugumgebung Rechnung getragen werden. Ein iteratives Vorgehensmodell, wie das Cluster–Modell von Meyer

([Quibeldey–Cirkel, 1994, S.69f]) und die Unterstützung beim Austausch von Komponenten auch in späten Phasen der Entwicklung sind hier Stichworte.

- **Automatische Generierung**. Eine Entwicklungsumgebung sollte über ein änderbares Metamodell verfügen. So ist es möglich, eine grundlegende Strukturierung, wie sie durch die Abstraktionsebenen (Abbildung 13) erzeugt wird, vorzugeben. Auch die Definition von Rahmendokumenten (Formulare, Dokumentations–Templates), ist auf diesem Wege möglich (beispielsweise die Gliederungen, wie sie in den Abschnitten 4.1.3 und 4.1.5 dargestellt sind).
  Eine weitere sinnvolle Funktionalität ist die automatische Generierung von Verweisen zu Bezeichnern und Begriffen. Bezeichner sind entweder im Quellcode oder in Klassendiagrammen deklariert. Begriffe sind informeller und dienen dem Verständnis und der Bildung einer Begriffswelt der Problemdomäne. Bezeichner und Begriffe ergeben zusammen mit den jeweils anzulegenden Definitionen ein *data dictionary* (Datenlexikon). Die Verwendung eines Eintrags des *data dictionaries* in einem Dokument sollte automatisch einen Verweis auf die Definition erzeugen.
  Wird das Datenmodell des *data dictionaries* so gewählt, daß sich Einträge untereinander referenzieren können, so ist die "Entwicklungsgeschichte" eines Begriffes von einer ersten Definition über ggf. mehrere Abwandlungen und Konkretisierungen, bis hin zur Deklaration als Bezeichner nachvollziehbar.

Die Anforderungen an die Konsistenzerhaltung der Daten und die Erweiterbarkeit der Werkzeuge ist unterschiedlich (siehe Abbildung 13).
Je näher ein Dokument der eigentlichen Implementation ist (auf die Abstraktionshierarchie bezogen also in den unteren Ebenen liegt), desto feingranularer werden die Informationen und desto wichtiger wird die Konsistenzerhaltung, analog des steigenden Grads der Formalisierung. Was die Werkzeuge angeht, ist die Erweiterbarkeit um neue Notationen bzw. deren Anpassung nicht sinnvoll. Auf dieser Ebene kommt eine Erweiterung oder Veränderung der Notation einer Erweiterung oder Veränderung der Implementationssprache gleich. Ein Klassendiagramm beispielsweise visualisiert die selben Informationen, wie die rein textuelle Darstellung einer Klassendefinition.
Umgekehrt ist die Verfügbarkeit eines weiten Spektrums von Notationen und deren Anpassung an spezielle Erfordernisse sehr wichtig, wenn es um die Erstellung von halbformalen und informellen Dokumenten geht. Die Dokumente der Ebenen 1 und 2 fallen hierunter. Dort werden ggf. Modelle aus den unteren Ebenen importiert und um zusätzliche graphische und/oder textuelle Attribute erweitert. Ein Beispiel sind die grauen Rahmen in Abbildung 16 zur Eingrenzung der Entwurfsmusterinstanzen. Ein weiteres Beispiel sind die *hot spots* (Abschnitt 4.1), die in die Klassendiagramme und die anderen Graphiken eingefügt wurden, um die Verzweigung aus den Graphiken zu den entsprechenden Themen zu ermöglichen.
Die Konsistenzerhaltung ist auf diesen Ebenen mit Ausnahme von Verweisen nur grobgranular erwünscht. Dokumente dieser Ebenen sind änderungsstabiler; die Intention des Entwicklers liegt darin, Systemeigenschaften und strukturelle Zusammenhänge, also Dinge von längerem Bestand zu beschreiben. Änderungen auf der Implementationsebene, auch an der Schnittstelle einer Klasse, machen nicht notwendigerweise eine Anpassung dieser Dokumente nötig. Selbst die Tolerierung von Inkonsistenzen kann sinnvoll sein.

5 Ausblick

Auf dem Markt ist meines Wissens keine Entwicklungsumgebung verfügbar, die ein Dokumentationskonzept unterstützt, das den Anforderungen, wie sie in Abschnitt 4.3 und in Anhang B beschrieben sind, genügt. Dokumentation wird weiterhin generell als zweitrangig eingeordnet.
Durch die Entwicklungen im Bereich Internet/Intranet wächst momentan eine vereinheitlichte und plattformübergreifende weltumspannende Infrastruktur zur Informationsverwaltung heran. Diese Technologie paßt auch gut auf die Anforderungen der Software–Entwicklung: Hypertext, Datenbankanbindung, unterschiedliche Dokumenten– und Graphikformate.
Aus der konventionellen passiven (Papier–)dokumentation kann damit ein aktives Informationssystem werden, das an die evolvierende Eigenschaft eines Softwareprojekts angepaßt ist. Alle Dokumente befinden sich in einem gemeinsamen Informationsraum und können so mit unverhältnismäßig geringerem Aufwand konsistent gehalten werden.
Die Arbeit an der Hypertextdokumentation für den OSA–Server hat mich darin bestätigt, daß der Aufwand hierfür sinnvoll ist und durch eine adäquate Arbeitsumgebung stark reduziert werden könnte.
Entwurfsmuster eignen sich hervorragend zur Dokumentation, da sie (die Kenntnis der Mustersprache vorausgesetzt) Entwurfswissen von Domänenwissen sichtbar trennen – mit der Folge des leichteren Verstehens.

6 Glossar

ABAP/4 – Die *Advanced Business Application Language* ist die SAP–Programmiersprache. Sie ist für die Entwicklung betriebswirtschaftlicher Anwendungen im Client/Server–Umfeld optimiert (aus [SAP_1, 1994]).

ABAP/4 *Dictonary* – Es dient der aktiven Integration von Metadaten mit den Anwendungen, der Weitergabe von Tabellendefinitionen an die Datenbanken inkl. der automatischen Umsetzung nach Strukturänderungen und es stellt Services wie Views und Matchcodes zur Verfügung (aus [SAP_1, 1994]).

ALE – *Application Link Enabling* ist eine Eigenentwicklung von SAP zur losen Kopplung von R/3–Systemen zum Zweck des Datenaustauschs, (s. [Born, 1996] und [SAP_1, 1994]).

CILabs – (*Component Integration Labs*) Eine herstellerunabhängige Einrichtung, die den OpenDoc–Standard weiterentwickelt, (http://www.cilabs.org).

DII – Ein *Dynamic Invocation Interface* ist eine Systemkomponente, die die Zuordnung und Ausführung von Methoden auf Objekten zur Laufzeit regelt, (s. [Orfali et al., 1996] und Abschnitt 2.3.5).

EDI – Standardisierter Austausch von Geschäftsdaten zwischen verschiedenen Systemen mit Hilfe von definierten Dokumenten, wie Bestellung, Rechnung u.a. (aus [SAP_1, 1994]).

Funktionsbaustein – Programmodul, das in mehreren Programmen verwendet werden kann und eine definierte Schnittstelle besitzt. Im System sind u.a. häufig verwendete betriebswirtschaftliche Funktionen als Bausteine definiert (aus [SAP_1, 1994]).

OMG – (*Object Management Group*) Ein seit 1989 bestehendes Konsortium aus ca. 500 Firmen, die im Bereich Objekttechnologie aktiv sind. Die OMG koordiniert die Entwicklungen und Standardisierungsbestrebungen. Der Schwerpunkt liegt in der Entwicklung des Standards für einen Software–Bus, auf dem Objektkomponenten unterschiedlicher Hersteller über Netzwerk– und Betriebssystemgrenzen hinweg kooperieren können. Der wichtigste Standard der OMG ist die 1994 veröffentlichte CORBA 2.0 – Spezifikation.
Die Dokumente der OMG sind im Internet verfügbar: http://www.omg.org.public–doclist.html. Es gibt eine regelmäßig erscheinende Publikation der OMG (*First Class*).

OpenDoc – Siehe Glossar der Seminararbeit im Anhang D.

Release – Die Zwischenversionen von SAP–R/3, in der Fehler korrigiert wurden und ggf. Teilfunktionalität verbessert wurde.

– Ein weiteres Glossar ist im Anhang D zu finden –

7 Literatur

[Apple, 1992] Apple (Hg): *Apple Event Registry. Standard Suites*. Apple Computer Inc. 1992.

[Baryla, 1995] Baryla, Michael: *Setting the Stage for Object REXX Scripting*. In: The Developer Connection News. Reprint from Vol. 7. IBM Corp. 1995.

[Berthold, 1995] Berthold, Dr. Andreas: *SAP Business Workflow. Grundlagen und technischer Überblick*. Version 1.0. SAP AG, Walldorf: 17.1.95.

[Booch, 1991] Booch, Grady: *Object Oriented Design With Applications*. Redwood City: The Benjamin Cummings Publishing Company Inc. 1991.

[Born, 1996] Born, Achim: *Software von der Stange*. In: c't. Magazin für Computertechnik 2 (1996). S. 206ff. Hannover: Heise Verlag 1996.

[Brockschmidt, 1995] Brockschmidt, Kraig: *Inside OLE*. Redmont: Microsoft Press 1995.

[CILabs, 1995] CILabs (Hg): *Software Innovation and Opportunity*. 6 (1995) DocNo.CILL00020/A. CILabs Inc. 1995.

[Coad et al., 1991] Coad, Peter. Yourdon, Edward: *Object–Oriented Analysis*. Englewood Cliffs: Yourdon Press, Prentice–Hall Inc. 1991.

[Römer, 1996] Römer, Martin: *Autopiloten fürs Netz. Intelligente Agenten–Rettung aus der Datenflut*. In: c't Magazin für Computertechnik 3 (1996) S.156ff. Hannover: Heise–Verlag 1996.

[Duden, 1989] Lektorat des B.I. Wissenschaftsverlags (Hg): *Duden Informatik*. Mannheim: Dudenverlag 1988.

[Fritz, 1995] Dr. Fritz, Franz Josef: *Workflow–Gestaltung auf Basis des R/3–Referenzmodells*. SAP AG, Walldorf 1995.

[Gamma, 1992] Gamma, Erich: *Objektorientierte Software–Entwicklung am Beispiel von ET++*. Berlin, Heidelberg: Springer–Verlag 1992.

[Gamma et al., 1995] Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John: *Design Patterns. Elements of Reusable Object–Oriented Software*. Reading, Massachusetts: Addison–Wesley Publishing Company 1995.

[IBM_1, 1996] IBM (Hg): *Open Scripting Architecture. Guide And Reference*. IBM Corp. 1996.

[IBM_2, 1996] IBM (Hg): *Object REXX Reference Manual*. IBM Corp. 1996.

[Johnson, 1992] Johnson, Ralf: *Documenting Frameworks using Patterns*. In: ACM SIGPLAN Notices 27(19) 1992. S.63ff.

[Krane_1, 1995] Krane, Rolf: *Business Object Broker RFC Dynamic Invocation Interface*. Version 3.0A. SAP AG, Walldorf 10.8.1995.

[Krane_2, 1995] Krane, Rolf: Vortragsausarbeitung zur Object World 1995 (Datei objworld.doc).

[Meyer, 1988] Meyer, Bertrand: *Object–Oriented Software Construction*. Herfordshire: Printice–Hall International (UK) Ltd. 1988.

[Microsoft, 1995] Microsoft (Hg): *The Component Object Model Specification*. Draft 6.3.95. Microsoft Corporation 1995.

[OMG, 1991] OMG (Hg): *The Common Object Request Broker. Architecture and Specification*. OMG Document Number 91.12.1, Rev. 1.1. Object Management Group (OMG) 1991.

[OMG_2, 1995] Mobray, Tom. Malveau, Raphael: *The Relationship of Design Patterns to CORBA*. In: First Class (OMG) Issue 1, 2/3 (1996) Volume V.

[Orfali et al., 1996] Orfali, Robert. Harkey, Dan. Edwars, Jeri: *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons Inc. 1996.

[Pree, 1995]	Pree, Wolfgang: *Design Patterns for Object–Oriented Software Development*. Reading, Massachusetts: Addison–Wesley Publishing Company 1995.
[Quibeldey–Cirkel, 1994]	Dr. Quibeldey–Cirkel, Klaus:*Das Objekt–Paradigma in der Informatik*. Stuttgart: B.G. Teubner 1994.
[SAP_1, 1994]	SAP (Hg): *SAP R/3 Software–Architektur*. SAP AG, Walldorf 1994.
[SAP_2, 1995]	SAP (Hg): *R/3 Communication. Technical Documentation*. Basis Modeling (9/1995). SAP AG, Walldorf 1995.
[SAP_3, 1995]	SAP (Hg): *Workflow–Tutorial*. Stand: 3.0B. SAP AG, Walldorf 1995.
[SAP_4, 1995]	SAP (Hg): SAP Info, Business Reengineering (März 1995). SAP AG Walldorf 1995.
[SAP_1, 1996]	Jakob, Joachim: *Das Business Object Repository als Grundlage für BAPI* (28.3.96). SAP AG, Walldorf 1996.
[Scherhuhn, 1996]	Scherhuhn, Jürgen: *Maßgeschneidert. Referenzmodelle–Hilfe bei der Einführung von Standardsoftware*. In: iX 1 (1996). S.112ff. Hannover: Heise–Verlag 1996.
[Stroustrup, 1991]	Stroustrup, Bjarne: *The C++ Programming Language*. Second Edition. Reading, Massachusetts: Addison–Wesley Publishing Company 1991.
[Vossen, 1994]	Vossen, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbank–Management–Systeme*. 2.Auflage. Bonn: Addison–Wesley Publishing Company 1994.
[Winand, 1992]	Weinand, Andé: *Objektorientierte Architektur für graphische Benutzungsoberflächen*. Berlin, Heidelberg: Springer–Verlag 1992.
[Wirfs–Brock, 1990]	Wirfs–Brock R..Wikerson B.. Wiener L.: *Designing Object–Oriented Software*. Englewood Cliffs: Prentice Hall 1990.

8 Anhang A

Folgend der Artikel "Anmerkungen zum BOR und zur Objekt–Strategie der SAP", der während meiner Tätigkeit bei SAP entstanden ist.

Anmerkungen zum BOR und zur Objekt–Strategie der SAP

Der folgende Text basiert auf einer Auswertung von Reports der Ende 1995 erfolgten Evaluierungen des *Business Object Repositories* (BOR) durch Spezialisten der Firmen *Benchmarking Partners*, *Microsoft*, *Hewlett–Packard* und *SAP Amerika*. Die Auswertung wurde um Diskussionsbeiträge aus der Entwicklergruppe und einige Bemerkungen des Autors erweitert.
Ziel des Papiers ist es, konzeptionelle Einschränkungen in Bezug auf einen allgemeinen Standpunkt aufzuzeigen und einen Katalog möglicher Erweiterungen zu formulieren.

1. Konzeptbedingte Einschränkungen

In diesem Abschnitt werden Einschränkungen besprochen, die durch das Konzept des *Business Object Repositories* und des SAP–Objektmodells entstehen.

Interfaces mit Default–Implementierung

Das Interface–Konzept lehnt sich an die OLE/COM–Spezifikation von Microsoft [Microsoft, 1995] an. Im Gegensatz zu den dortigen rein abstrakten Interfaces wird im SAP–Objektmodell eine Default–Implementierung bereitgestellt.
Das aus Modellierungssicht sehr leistungsfähige Konzept der abstrakten Klasse sollte nicht verwässert werden. Durch die Bereitstellung der normalen Klassenvererbung in Verbindung mit Mehrfachvererbung können die Fälle, in denen eine Default–Implementierung sinnvoll ist, abgedeckt werden.

Delegation

Im Kontext des *Business Object Repositories* wird unter dem Begriff Delegation nur die statische Ausprägung verstanden: Delegation einer Nachricht an eine Subklasse. Delegation beinhaltet aber auch die dynamische Variante: Weiterleitung einer Botschaft an ein Objekt, das *in keiner Typrelation* zu dem Objekt stehen muß, das die Anforderung ursprünglich erhalten hat (siehe [Gamma, 1995, S.20]).
Desweiteren besteht folgende Einschränkung: Der Delegationsmechanismus kann immer nur an *einen* Subtyp delegieren – das stellt eine Restriktion für die kundenspezifische Anpassung durch Spezialisierung dar, falls mehrere Spezialisierungen nötig sind. Flexibler wären durch Vererbung erweiterbare Factory–Objekttypen (Kunden definieren ihre speziellen Objekt–Factories).

Polymorphie

Zunächst eine Definition: “Polymorphismus bedeutet, daß dieselbe Botschaft an Objekte verschiedener Klassen gesendet werden kann und daß die Empfängerobjekte jeder Klasse auf ihre eigene – evtl. ganz unterschiedliche – Art darauf reagieren. Das bedeutet, daß der Sender einer Botschaft nicht wissen muß, zu welcher Klasse das Empfängerobjekt gehört.” [Wirfs-Brock, 1990].
Im Zusammenhang mit dem Interface-Konzept wird Polymorphie analog der obigen Definition im *Business Object Repository* unterstützt.
Auch die in der neueren Literatur ([Pree, 1995], [Gamma, 1995]) dargestellten *Template–Hook*–Strukturen sind mit der BOR–Laufzeitumgebung realisierbar. Hierzu ist der Aufruf von Methoden (*Hook*) einer abgeleiteten Klasse aus einer Methode (*Template*) der Basisklasse nötig. Die Hook–Methoden überschreiben dabei evtl. Methoden der Basisklasse.
Eine Einschränkung besteht allerdings: Überschriebene Methoden einer Superklasse sind in der abgeleiteten Klasse nicht mehr sichtbar. Diese Einschränkung kann nur durch das Kopieren der Implementation aus der Superklassenmethode umgangen werden, mit der Konsequenz unnötiger Redundanzen.

Mehrfachvererbung

Die Mehrfachvererbung ist bisher kein Bestandteil des SAP–Objektmodells.
Die zur Auswahl stehenden Konzepte für die Kopplung von Klassenstrukturen sind die statische Verbindung von Klassen durch Mehrfachvererbung und die dynamische Komposition von Objektreferenzen in einer neuen Klasse. Die Komposition ist dabei in der Regel zu bevorzugen.
Für den Import externer Klassenbibliotheken kann die Mehrfachvererbung ein geeignetes Instrument sein. Die ausschließliche Bereitstellung des dynamischen Verfahrens der Komposition könnte sich für die Modellierung langfristig als zu unflexibel erweisen.

2. Einschränkungen durch Restriktionen der Umgebung

In diesem Abschnitt geht es um Einschränkungen, die durch die R/3–Dienste verursacht werden, auf denen das *Business Object Repository* aufbaut.

Strukturierte Datentypen

Strukturierte Datentypen stehen nicht zur Verfügung. Diese Restriktion ist durch die RFC–Schnittstelle begründet. Derzeit kann diese Schnittstelle nur zeichenbasierte ABAP/4–Basistypen und homogene Listen dieser Typen transportieren.

ABAP

ABAP/4 als Implementationssprache auch für das BOR bietet bisher keine Unterstützung für die Definition abstrakter Datentypen (Klassen) und den Aufruf von Objektmethoden.

Overloading

Overloading ist momentan nicht realisiert. Mit diesem Begriff wird eine Ausprägung der Polymorphie bezeichnet: Mehrere Methoden besitzen den gleichen Namen aber unterschiedliche Parameterlisten. Ein Beispiel wären mathematische Funktionen, die auf verschiedenen Datentypen operieren (z.B. Ganzzahlen, reelle Zahlen und komplexe Zahlen).

Inkonsistenzproblematik

Durch das Wrappen der Funktionsbausteine und der Daten in die Objekttypen entstehen verschiedene Inkonsistenzen:

- **Statische Inkonsistenz:** Solange die Objekttypen nur Wrapper darstellen ergibt sich ein Konsistenzproblem zwischen der unterliegenden und der in den Objekttypen abgebildeten Funktionalität, da beide auch ohne das BOR manipuliert werden können: Bei Veränderung des Datenbank–Schemas muß die Objekttyp–Definition aufgefrischt werden. Analogie: Die Funktionalität „gehört“ den Objekten nicht.
- **Dynamische Inkonsistenz I**: Objekttypen des BOR *kapseln* Daten und Funktionen. Diese wichtige Forderung des OO–Paradigmas ist somit prinzipiell erfüllt.

  Ist die Menge der referenzierten Daten zweier paarweise verschiedener Objekttypen nicht disjunkt, besteht die Gefahr der Dateninkonsistenz zur Laufzeit, da Objekte verschiedenen Typs die gleichen Daten manipulieren und lesen. Die Sperrmechanismen der unterliegenden Datenbank verhindern zwar den gleichzeitigen Zugriff, sie haben aber keinen Einfluß auf zeitlich nacheinander erfolgende Zugriffe, die aber das von einem Klienten erwartete Objektverhalten (Semantik) beeinflussen können.

  Die von einem Objekt referenzierten Daten sollten über dessen Lebensdauer implizit für schreibende Zugriffe, die nicht über Methoden des Objekts gehen, gesperrt werden.

- **Dynamische Inkonsistenz II**: Alle Instanzen eines Objekttyps werden in einer lokalen Tabelle im BOR verwaltet. Die Persistenz eines Objekts wird durch das Kopieren der im BOR lokal gehaltenen Daten in die durch das Objekt referenzierten Felder erreicht. Ein umgekehrter Mechanismus, der auf eine Veränderung der von einem Objekt referenzierten Daten reagiert und die lokale Kopie im BOR aktualisiert, ist nicht vorhanden. Betrifft die externe Veränderung ein von einem Objekt referenziertes Schlüsselfeld, so ist die Wirkung ungleich gravierender, da das Objekt dann nicht mehr existiert.

Aus den angesprochenen Problemen, die durch die "Objektifizierung" von R/3 entstehen, wird sich längerfristig der Bedarf nach einem neuen Transaktionskonzept ergeben.

3. Erweiterungsvorschläge

Folgend ein Katalog aus Erweiterungsvorschlägen, der keinen Anspruch auf Vollständigkeit erhebt.

Rechtzeitige Bereitstellung grundlegender Dienste

Die Bereitstellung einer Bibliothek aus grundlegenden Datenstrukturen (technische Klassen) ist eine wichtige Voraussetzung für die erfolgreiche Einführung der Objekttechnologie. Homogene Listen reichen nicht für alle Anwendungsfälle aus.

Außer grundlegenden technischen Datenstrukturen könnte die Bereitstellung von Datenstrukturen und Verfahren, die im Umfeld der betriebswirtschaftlichen Anwendungsentwicklung häufig benötigt werden, sinnvoll sein.

Je mehr grundlegende Konzepte als *ready-to-use*–Komponenten von Anfang an bereitgestellt werden, desto weniger Wildwuchs und Redundanz werden die Anwender erzeugen.

Parametrisierbare Typen

Wichtig ist unter anderem eine Technik zur Realisierung dynamisch–heterogener Strukturen von Daten. Hierzu eignen sich z.B. parametrisierbare Typen (im C++–Kontext als *Templates* bezeichnet). Diese Technologie ist derzeit nicht implementiert.

Mit Templates lassen sich sehr einfach generische Datenstrukturen definieren. Ein Beispiel sind Listen, die beliebige Datentypen verwalten können (List<integer>, List<MyType>). Die jeweils unveränderten Codeteile werden generisch, also ohne Angabe eines konkreten Typs, spezifiziert.

Laufzeitidentifikation

Die Laufzeitidentifikation von Typen sollte durchgängig, d.h. auch für die Basistypen und für benutzerdefinierte Typen realisiert werden. Die Implementierung und Nutzung heterogener Datenstrukturen wird damit stark erleichtert.

Ein Beispiel wäre eine heterogene Liste, die Elemente verschiedener Datentypen enthalten kann (beispielsweise verschiedene Materialien). Der Klient einer solchen Liste benötigt Informationen über die Typen der Elemente, die in beliebiger Reihenfolge in der Liste angeordnet sein können.

Überprüfung der statischen Semantik

Die Erweiterung von Parameterlisten in abgeleiteten Klassen führt im Zusammenhang mit der Definition von optionalen Parametern dazu, daß eine Signaturüberprüfung mit eindeutiger Zuordnung von Aufrufen zu Methoden nicht immer möglich ist.

Style Guide

Die Qualität der Implementierung kann durch die Einführung von *Pre–/Post–Conditions* (*Programming by Contracting*, [Meyer, 1988]) und *Exceptions* [Stroustrup, 1991] gesteigert werden, da mit diesen Konzepten eine einfache und kontrollierte Ausnahmebehandlung realisierbar ist. Auch Softwaremetriken können auf diese Merkmale angewendet werden.

Attributierbare Laufzeitbeziehungen

Laufzeit–Beziehungen sollten attributierbar sein. Inverse bzw. bidirektionale Beziehungen sollten Bestandteil des Objektmodells sein. Ferner die Möglichkeit, Methoden mittels Beziehungen zu propagieren.
Für die Attributierung von Beziehungen spricht beispielsweise Folgendes: Momentan werden Aggregationsbeziehungen von Objekten, die bezüglich ihrer Lebensdauer voneinander abhängig sind, mittels des Interface–Typs IFAGGREGATE modelliert: Aggregierbare Objekttypen müssen dieses Interface erben. Das aggregierende Objekt muß Auskunft über die Typen der aggregierten Objekte geben können. Dieser Zusammenhang ließe sich besser durch attributierte Aggregations-beziehungen modellieren.

Objektidentifikatoren

Das CORBA–Verfahren hierarchischer Namen [Orfali et al., 1996, S. 110ff], bietet eine intuitivere Namensgebung als das Modell des *Business Object Repositories* (Längenbeschränkung auf 10 Zeichen, kundenspezifische Typen müssen mit Y oder Z beginnen).
Hierarchisch strukturierte Identifikatoren haben gegenüber unstrukturierten Identifikatoren den weiteren Vorteil, daß sie sehr einfach durch Voranstellen oder Anhängen eines weiteren Spezifizierers in eine übergeordnete Domäne übernommen werden können. Auch eine Abbildung auf die Referenzierung von Objekten, die in Dateisystemen verwaltet werden, ist leicht möglich.
Der persistente Schlüssel abstrahiert nicht von der unterliegenden Systemarchitektur. Desweiteren wird zwischen einem Laufzeitschlüssel und einem persistenten Schlüssel unterschieden. Wünschenswert ist ein Objektidentifikator, der von der unterliegenden Architektur abstrahiert, sich in die von anderen *Object Request Brokern* (ORB) verwendeten Identifikatoren zumindest konvertieren läßt und auch den Transport zwischen verschiedenen ORBs ermöglicht.
Um einen Wechsel der Objektidentifikation in zukünftigen Versionen von R/3 zu ermöglichen, sollte parallel zum bestehenden Verfahren ein weiterer aufwärtskompatibler *Naming Service* angeboten werden. Die CORBA–Spezifikation [OMG, 1991] oder das GUID–Konzept von Microsoft [Brockschmidt, 1995] könnte hier als Vorbild dienen.
Das logische System als Bestandteil der persistenten Objektreferenz sollte erweitert und generalisiert werden, um zukünftigen Anforderungen erfüllen zu können: Replikation; Minimum–Revisions–Nummer; Eigentümerwechsel innerhalb einer Umgebung, aber auch zwischen lokalen Namensräumen; Objekte, die auf mehrere Server verteilt sind (*distributed objects*).

Nur lesender Attributzugriff

Gegenwärtig ist der Zugriff auf Attribute und somit auf die unterliegenden Tabellenfelder der Datenbank nur lesend möglich; schreibender Zugriff nur indirekt über vorhandene Funktionalität, die diese Daten modifiziert. Da die Definition von Objekttypen (bisher) keine weitere Funktionalität in das System einführt, sondern ausschließlich vorhandene Funktionalität wrappt, ist entweder die automatische Generierung von Update–Methoden für ausgewählte Attribute nötig oder diese Funktionalität muß manuell implementiert werden, wodurch das Wrapper–Konzept beeinträchtigt würde.
Als dritter Weg bietet sich die Einführung generischer Set–Methoden an.

Ereignisse

Das *Event*–Konzept sollte überarbeitet werden. Anstelle der *Callback*–Methoden sollte ein *Eventhandler*–Objekttyp definiert werden, der als Basisklasse geerbt wird. So verbessert sich die Übersicht und das Konzept wird flexibler und robuster. *Events* sollten ebenfalls als Objekttypen modelliert werden, um die für einen *Event* nötigen Daten und Funktionalitäten dort zu konzentrieren. Die Analyse erprobter Konzepte (z.B. ET++ oder *InterViews*) würde sich sicherlich lohnen.

Metamodell

Die Erweiterbarkeit des Metamodells (ohne Veränderungen am Laufzeitsystem) ist nicht realisiert (auch nicht für SAP–interne Erweiterungen).

Factory–Objekte

Die Instanziierung eines Objekts erfordert die Angabe eines Objekttyps – die Zuweisung eines persistenten Schlüssels ist hingegen optional. Ohne Schlüssel erzeugte Objekte erhalten über die instanzunabhängige Methode create *nachträglich* eine Zuordnung zu konkreten Daten. Solange diese Zuordnung nicht stattgefunden hat, sind die übrigen Methoden der Schnittstelle zwar sichtbar, der Aufruf würde aber Laufzeitfehler erzeugen.
Analog zur CORBA–Spezifikation sollten Objekte immer vollständig verfügbar sein. Das bereits realisierte Instanziierungskonzept mit *Factory*–Objekten, die nie Bezug auf konkrete Daten haben und ausschließlich zur Instanziierung von Objekten mit Datenbezug dienen, sollte durchgängig angewendet werden.

Dynamic Typing

Der Wechsel des Typs eines Objekts zur Laufzeit (entlang der Vererbungshierarchie) ist möglich aber nicht explizit unterstützt.

4. Allgemeine Anmerkungen

Abschließend einige allgemeine Anmerkungen zu den Voraussetzungen und den Folgen, die sich aus dem Einsatz objektorientierter Technologien ergeben.

Wrapping II

Die bis dato nicht realisierte Mehrfachvererbung (und wahrscheinlich auch komplexere Interaktionen von Objekten zur Laufzeit) erzeugen durch die Abbildung auf relationale Strukturen dort prinzipbedingt einen erhöhten Aufwand zur Bereitstellung der angeforderten Daten (temporäre *Joins*). Die so entstehenden Performanceverluste sind ohne praktische Erfahrungen in realen Projekten nicht abschätzbar.

Standard

Die marktbeherrschende Stellung von SAP würde es erleichtern einen Standard für eine Objekttyp–Bibliothek und für domänenspezifische Frameworks zu setzen.

Aufwand

Der personelle und zeitliche Aufwand für das Design und die Validierung einer Objekttyp–Bibliothek und besonders für Frameworks sollte nicht unterschätzt werden.

Paradigmenwechsel

Der Paradigmenwechsel hin zur Objektorientierung erfordert die Ausbildung der Entwickler und die Einbeziehung der Domänenexperten um ein fundiertes Verständnis über Objekt–Technologie im Business–Bereich zu erzeugen. Langfristiges Ziel muß die Entwicklung von domänenspezifischen Frameworks sein. Der Nutzen aus dem Paradigmenwechsel kann sich nur langfristig einstellen, wird aber neue Lösungswege für drängende Probleme, die mit dem bisherigen Paradigma nicht oder nur noch mit unwirtschaftlichem Aufwand lösbar sind, aufzeigen [Quibeldey-Cirkel, 1994].

Risiken

Bisher ist es versäumt worden Analyse– und Validierungsmethoden (z.B. Szenarios, *Use-Case*–Analyse, *Rapid Prototyping*) im Entwicklungsprozeß der Objekttyp–Bibliothek einzusetzen. Auch ein allgemeines Vorgehensmodell, das den Entwicklungsprozeß beschreibt, existiert bisher nicht.
Da die Bewegung hin zu Objektorientierung bei SAP mittlerweile den Kinderschuhen entwachsen ist, ergibt sich die Notwendigkeit, hier verstärkt tätig zu werden. Es besteht sonst die Gefahr, daß die bisherigen Aufwendungen in die Entwicklung vergebens waren, durch die verfrühte Veröffentlichung weiterer Schaden angerichtet wird und in absehbarer Zeit eine vollständige Neuentwicklung nötig sein wird, da sich die bestehenden Komponenten als nicht genügend tragfähig für die weitere Entwicklung erweisen.

Einschränkung

Die bisher überwiegend aus der prozessorientierten Workflow-Sicht getriebene Identifikation von Objekttypen birgt die Gefahr, daß aus der Warte eines generellen Standpunkts einseitige Abstraktionen und nur auf diesen Anwendungsfall spezialisierte Schnittstellen entstehen.

Performance

Die Performance ist mit einem relativ konstanten Verwaltungs–Overhead behaftet, der sich durch die Veränderung bzw. erstmalige Bereitstellung der notwendigen unterliegenden Funktionalität (ABAP/4) verringern wird.
Trotz allem läßt die zusätzliche Abstraktion durch Objekte neue Möglichkeiten der Modellierung zu, verbunden mit erhöhten Anforderungen an die tieferen Schichten.

5. Literatur

[Brockschmidt, 1995] Brockschmidt, Kraig: *Inside OLE*. Redmont: Microsoft Press 1995.

[Gamma et al., 1995] Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John:*Design Patterns. Elements of Reusable Object–Oriented Software*. Reading, Massachusetts: Addison–Wesley Publishing Company 1995.

[Meyer, 1988] Meyer, Bertrand: *Object–Oriented Software Construction*. Herfordshire: Printice–Hall International (UK) Ltd. 1988.

[Microsoft, 1995] Microsoft (Hg): *The Component Object Model Specification*. Draft 6.3.95. Microsoft Corporation 1995.

[OMG, 1991] OMG (Hg): *The Common Object Request Broker. Architecture and Specification*. OMG Document Number 91.12.1, Rev. 1.1. Object Management Group (OMG) 1991.

[Orfali et al., 1996] Orfali, Robert. Harkey, Dan. Edwars, Jeri: *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons Inc. 1996.

[Pree, 1995] Pree, Wolfgang: *Design Patterns for Object–Oriented Software Development*. Reading, Massachusetts: Addison–Wesley Publishing Company 1995.

[Quibeldey–Cirkel, 1994] Dr. Quibeldey–Cirkel, Klaus:*Das Objekt–Paradigma in der Informatik*. Stuttgart: B.G. Teubner 1994.

[Stroustrup, 1991] Stroustrup, Bjarne: *The C++ Programming Language*. Second Edition. Reading, Massachusetts: Addison–Wesley Publishing Company 1991.

[Wirfs–Brock, 1990] Wirfs–Brock R..Wikerson B.. Wiener L.:*Designing Object–Oriented Software*. Englewood Cliffs: Prentice Hall 1990.

9 Anhang B

Das Diskussionspapier "Wiederverwendung" ist im Rahmen meiner Kontakte zu der Berliner Firma microTOOL während der Arbeit an der Entwurfsmusterdokumentation entstanden. Ich habe deren Software–Entwicklungsumgebung "objectiF" zur Erstellung der Diagramme eingesetzt.

Wiederverwendung

Wiederverwendung ist ein gegenwärtig heiß diskutiertes Thema. Vielleicht aus dem Grund, daß für kaum einen anderen Bereich der Softwareentwicklung die Diskrepanz zwischen Wunsch und Realität so groß ist. Wohl aber auch, weil das Thema langsam existentiell wichtig wird, da der Konkurrenz– und Kostendruck in der Softwareentwicklung steigt und die Komplexität der Projekte gleichzeitig zunimmt.
Weiterhin sind aber fast ausschließlich Back-end–Entwicklungsumgebungen im Einsatz. Allen diesen Werkzeugen gemeinsam ist, daß der Wiederverwendungsaspekt vollkommen außer Acht gelassen oder zur Nebensache erklärt wird.
Die in letzter Zeit mit großem Erfolg eingeführten Werkzeuge der Klasse *Delphi*, *Visual Basic* oder *Visual Objects* wollen mit *Componentware* die ultimative Antwort auf das Problem der Wiederverwendung geben. Das ist meines Erachtens aber nur die halbe Wahrheit. Das Problem der qualitativ hochwertigen Dokumentation und der Suche nach geeigneten Komponenten bleibt bestehen.
Fazit: Es besteht weiterhin ein hohes Entwicklungspotential im Bereich Softwareentwicklungsumgebungen, das durch die Wiederverwendungsdebatte eher noch größer geworden ist. Eine verbreitete kommerzielle Lösung, die für die Wiederverwendungsproblematik gangbare Wege anbietet, gibt es bisher meines Wissens nicht.
Ganz wichtig für den Erfolg eines neuen Ansatzes ist, daß vom weit verbreiteten sequentiellen Vorgehen abgekommen wird. Dieses Vorgehen findet man in zwei Ausprägungen, meist abhängig von der Teamgröße:

- "Sequentiell ohne System": Erst Analyse, Entwurf, Implementation ohne jegliche Dokumentation, anschließend die Nach–Dokumentation, wenn sich das Projekt in einem ausreichend stabilen Zustand befindet (sprich: sich die Arbeit "lohnt").
  Der Nutzen der Dokumentation wird nicht erkannt (das Feedback fehlt) und es wird so ohne Motivation nur nutzlose Dokumentation von schlechter Qualität produziert, die dann natürlich nicht benutzt wird . . .
- "Sequentiell mit System": Vorgehen nach dem Wasserfallmodell, mit den bekannten Annahmen, daß Anforderungen objektiv spezifiziert werden können und ein Projektplan für den gesamten Zeitraum eines Projekts ohne große Veränderungen Bestand hat.

Dokumentation muß als integraler Bestandteil, deren möglichst gute Qualität als das eigentliche Ziel eines Projekts begriffen werden.
Ob das gelingt, hängt entscheidend von einer guten**Werkzeugunterstützung** ab:

- Qualität der Dokumentationswerkzeuge,
- Bereitstellung von Navigations– und Retrievaltechniken: statt passiver Dokumentation ein aktives Informationssystem,
- optimale Integration in eine Softwareentwicklungsumgebung (Konsistenz, Zugriff auf gemeinsames Repository).

Und ebenso hängt es von **konzeptionellen und organisatorischen Aspekten** ab:

- Von der Realisierung eines möglichst hohen Grades der Ähnlichkeit zwischen den Denkmustern bei der Softwareentwicklung und bei der Dokumentation (ich möchte das als "paradigmatische Nähe" bezeichnen),
- von einem möglichst hohen Anteil an halb– und vollautomatisch produzierten Dokumenten und vorstrukturierten Formularen (Doku–Templates),
- von der Existenz und der Anwendung eines Vorgehensmodells,

- und nicht zuletzt von einer guten Ausbildung und Motivation der Entwickler.

Eine weitere sehr wichtige Anforderung, die beide Aspekte betrifft, ist die **Skalierbarkeit** der Wiederverwendungsstrategie: Kleine Projektteams, bis hin zu Einzelpersonen, müssen gleichermaßen unterstützt werden wie größere Organisationsstrukturen.

Was bedeutet Wiederverwendung nun für objectiF?

- Unterstützung bei der Entwicklung von Frameworks,
- Werkzeuge zur Erstellung von Dokumentation,
- Werkzeuge zur Verwaltung, Navigation und zum Retrieval in der wachsenden Menge an Komponenten.

Im folgenden möchte ich auf diese drei Punkte näher eingehen.

1. Frameworks

Frameworks sind die potentiell am besten zur Wiederverwendung geeigneten Einheiten. Fast alle hier angestellten Überlegungen gelten aber genauso für die anderen Entwurfskomponenten.
Durch die *Subjekte* besteht in objectiF eine geeignete Verwaltungsinstanz für Frameworks. Was fehlt, ist die Erhebung eines Subjekts in den Status eines Frameworks, mit verschiedenen Konsequenzen.
Ein Framework stellt idealerweise eine Einheit aus verschiedenen Dokumenten und evtl. auch aktiven Komponenten dar (vgl. OBJEKTspektrum, 1-2/96, S.10ff):

- Evtl. getrennt vorliegende Klassenmodelle aus Analysephase und Design.
- Generell alle Modelle, die durch Werkzeuge der Umgebung produziert wurden und zum Verständnis des Entwurfs dienen können.
- Die Implementation.
- Eine Anleitung zur Verwendung des Frameworks, z.B. im *Cookbook–Style* oder durch die Beschreibung der *Hot* und *Frozen Spots* (Pree).
- Eine eingehende Darstellung der Erweiterungsmöglichkeiten (Veränderung des Frameworks).
- Entwurfsmusterdokumentation. Ausgehend von allgemein beschriebenen Mustern in einer Musterdatenbank (auch ein Hypertext) wird in die zu beschreibende Komponente verzweigt. So kann der Anwender, der die *Pattern Language* verinnerlicht hat, sehr viel einfacher einen unbekannten Entwurf explorieren.
- Speziell für Frameworks Dokumente, die deren konzeptionelle Eigenschaften beschreiben (beispielsweise Portabilität, Realisierung der Ausnahmebehandlung) und so eine Einschätzung für die Integrierbarkeit in bestehende Umgebungen bieten.
- Konventionelle Dokumentation der Klassen: Aufgabe, Beschreibung des Protokolls und der öffentlichen Methoden, Beschreibung der *uses*– und *used-by*–Beziehungen. Diese Dokumentation wird von den anderen Dokumenten häufig referenziert werden und stellt so ein Basisdokument dar.
- Eine *ready-to-use*–Umgebung, die die direkte Ausführung und somit ein interaktives Ausprobieren einer Komponente erlaubt.
- Interaktive Hilfen für die Nutzung eines Frameworks (Assistenten / Wizzards).

Diese Aufstellung ist wahrscheinlich nicht vollständig; desweiteren wird der Anwender für die Dokumentation einer spezifischen Komponente jeweils eine sinnvolle Untermenge bilden.
Die Entwicklung von Frameworks gestaltet sich iterativ; eine wirklich mehrfach einsetzbare Komponente entsteht erst nach mehreren Designzyklen. Diese Tatsache muß sich im Vorgehensmodell und also auch in den Werkzeugen niederschlagen.

Der zweite sehr wichtige Aspekt ist die Unterstützung des Anwenders bei der Nutzung einer Komponente. Ein gutes Beispiel bietet hier *Visual C++*: Kaum jemand wird darüber nachdenken, daß im Dialog mit dem *AppWizzard* ein *Application Framework* konfiguriert wird.

2. Dokumentation

Für eine auch unter dem Gesichtspunkt Wiederverwendung adäquate Dokumentationsqualität ist die alleinige Verwendung von WinWord, wie es bei objectiF der Fall ist, nach meiner Auffassung nicht ausreichend. Folgende Argumente lassen sich hierfür unter anderem anführen: Probleme entstehen bei der Konsistenzerhaltung zwischen den Komponenten und deren Dokumentation. Es ist mit vertretbarem Aufwand nicht möglich, Referenzen zu den in Dokumenten verwendeten Bezeichnern zu halten. Die Erweiterbarkeit der Dokumentation mit anschließendem Abgleich ist nicht geregelt möglich. Es besteht keine "paradigmatische Nähe" zwischen dem Applikationsmodell einer Textverarbeitung wie WinWord und der Denkwelt der Objekte. Navigation und Retrieval ist nicht akzeptabel realisierbar. Die Dokumentation von Frameworks schließlich würde dieses Konzept überfordern.
Die konventionelle sequentielle Dokumentation hat ihre Berechtigung, wenn es um die Produktion von gedruckten Auszügen geht; der Entwickler wird bei seiner Arbeit aber wesentlich besser durch ein aktives Repository, ein Informationssystem unterstützt.
In diesem Zusammenhang steht auch die Frage wie andere Dokumente (beispielsweise Handbücher als "sequentialiserte" Papierversion und als WinHelp) aus dem Repository erzeugt werden könnten. Bei der aktuell in objectiF realisierten Lösung besteht der Nachteil der Doppelverwaltung einzelner Komponentendokumente und eines Gesamtdokuments.
Nach meiner Ansicht stellt die Basistechnologie Hypertext den geeigneten Ansatz für die Softwaredokumentation dar. Durch Links können Beziehungen beliebiger Granularität zwischen verschiedenen Dokumenten (auch unterschiedlichen Typs – beispielsweise Textdokumente und Klassenmodell) hergestellt werden. Die Verzeigerung sollte für in der Objektbank definierte Bezeichner automatisch erfolgen (inklusive Konsistenzerhaltung); der Benutzer definiert die Links, die sich aus der Semantik ergeben. (Das Problem der – halbautomatischen – Verknüpfung "ähnlicher" Komponenten ist durch diesen Ansatz allerdings noch nicht gelöst; hier sollten die Erfahrungen aus einer ersten Realisierung eingehen).
Meine Vermutung ist, daß sich durch Hypertext eine an das OO–Paradigma angepaßte Dokumentationstechnik realisieren läßt: Es besteht zweifellos ein enger Zusammenhang zwischen der Denkwelt der Klassen, Beziehungen und Objekte und der Denkwelt der Knoten und Links eines Hyperdokuments. Inwieweit sich z.B. auch die Kapselung, Vererbung und Komposition für die Realisierung neuer Dokumentationstechniken eignen, sollte untersucht werden.
Was den Punkt Vererbung angeht, ist ein interessanter Ansatz in einem Papier beschrieben, daß ein Projekt der Universität Passau mit BMW in München beschreibt (B. Freitag: *A Hypertext–Based Tool for Large Scale Software Reuse* - ftp://ftp.uni-passau.de/pub/local/deductdb/papers/Fre94b.ps).

3. Navigation und Retrieval

Die bisher vorherrschende Arbeitsweise bei der Softwareentwicklung ist, wie schon angesprochen, überwiegend sequentiell. Die Arbeit in den Methodeneditoren findet unabhängig voneinander statt. Diese Vorgehensweise wird ja auch z.B. durch das Wasserfallmodell begründet. Mit objectiF 2.0 besteht nun die Möglichkeit des *iterativen* Arbeitens in Klassenmodell– und Implementationseditor; so ist ein Wechsel der Abstraktionsebenen möglich und damit eine wesentlich besser an die Realität angepaßte Arbeitstechnik. Die Vervollständigung hin zu einer Umgebung, die Wiederverwendung aktiv unterstützt, könnte von einem Werkzeug, dem ich den Arbeitstitel **Reuse–Navigator** geben möchte, erfüllt werden. Der Reuse–Navigator dient als *Integrationsplattform* für alle Werkzeuge

(Methodeneditoren, extern angebundene Tools) und bietet die Funktionalität zur Navigation und zum Retrieval im Datenbestand.

Zur Navigation auf der Suche nach einer Komponente, die bestimmten Anforderungen genügt, stellen sich die Daten idealerweise als Hyperdokument dar. Die Dokumente sind in dieser Sicht nicht veränderbar, der Benutzer sollte allerdings eigene, in einer Mehrbenutzerumgebung jeweils nur für ihn sichtbare Links und Kommentare einfügen können, um die Orientierung im "Hyperspace" zu verbessern (beispielsweise Lesezeichen oder Anmerkungen).
Ein Einstieg in die Daten sollte auf vielfältige Art möglich sein, um dem Anwender die Wahl einer individuellen Arbeitstechnik zu lassen. Prinzipiell lassen sich die schrittweise Annäherung durch Navigation und der direkte Einstieg durch Angabe von Suchworten zur Index– oder Volltextsuche unterscheiden.
Welche Zugriffstechniken realisiert werden, bedarf noch der genaueren Analyse.

- **Navigation**: Für sehr wichtig halte ich den sog. *Fisheye View*, der die Umgebung eines im Fokus stehenden Knotens anzeigt (Umgebung bedeutet in diesem Kontext thematisch verwandte Knoten). Auch dynamische Inhaltsverzeichnisse, wie sie z.B. im *Explorer* von Windows95 oder auf den *MS Developer–Network* CDs eingesetzt werden, sind ein geeignetes Mittel, um umfangreiche Datenbestände darzustellen.
- **Retrieval**: Die für diese Technik zu lösenden Probleme liegen in der Vorbereitung der Dokumente durch Indexierung, die weitestgehend ohne Benutzereingriff stattfinden sollte und einer möglichst optimalen Auswahl der Stichworte, um gute Trefferquoten zu erzielen.

Ist der Anwender am Ziel seiner Suche, wird er mehr über die gefundene Komponente erfahren wollen. Er kann eines der angebotenen Dokumente ansehen, das seinem Wissensstand und seiner Erwartungshaltung am meisten entspricht (z.B. eine Patterndokumentation oder eine Beschreibung der Klassen mit direktem Einstieg in den Sourcecode oder ein Kochbuch); er sollte eine *ready-to-use*–Umgebung ausprobieren können; evtl. ist auch eine Animation verfügbar (z.B. eine *Power Point Show*). Entspricht die Komponente seinen Erwartungen, wird er eine Referenz in seine aktuelle Arbeitsumgebung erzeugen und die Komponente mittels der gegebenen Anleitung integrieren können.

10 Anhang C

Der Ausdruck des Objekttyps APPOINTMNT aus dem SAP–R/3 *Business Object Repository*.

11 Anhang D

Die in Abschnitt 4 konzeptionell und an Beispielen dargestellte Dokumentation mit Entwurfsmustern ist eine inhaltliche Weiterführung und praktische Erprobung meiner im Rahmen des Entwurfsmusterseminars entstandenen Ausarbeitung. Diese Ausarbeitung ist hier zur weiterführenden Information angefügt.