

Christian Ey  
Fachhochschule Karlsruhe  
Fachbereich Wirtschaftsinformatik

# **Anbindung des Netscape Application Servers an CORBA**

ein Praxissemesterbericht in Zusammenarbeit mit  
Netscape Communications Corporation

## **Dankeschön!**

Ich möchte mich ganz herzlich bei Prof. Dr. Cosima Schmauch bedanken, die mir dieses Praxissemester bei Netscape ermöglicht hat. Ich habe in dieser Zeit ausgesprochen viel gelernt, Praxiserfahrung gesammelt und bin fachlich viel weitergekommen. Außerdem habe ich durch den Auslandsaufenthalt lernen dürfen, daß nicht alles überall „so ist wie es immer war“.

Danken möchte ich auch Paul Dreyfus, der als mein Supervisor immer ein offenes Ohr für mich hatte und neuen Ideen immer aufgeschlossen gegenüber stand.

Mike Lee, der uns bei unserem Projekt stets tatkräftig und unkompliziert zur Seite stand und professionelle Hilfe bot. Michelle Wyner, die genauso neu in der Abteilung war wie wir, und mit der auch nach Dienstscluß noch viel Spaß zu haben war.

Robert Husted, der den Netscape Application Server vertrat und uns immer sehr unterstützt hat. Eric Krock, der sein PalmPilot Fieber auf alle übertrug, die mit ihm zu tun hatten. Und allen @ Netscape, die mir in dieser Zeit geholfen haben und mit mir mitgearbeitet haben. Es war eine ganz besondere Zeit für mich! Danke!

## **Notizen:**

<b>Anbindung des Netscape Application Servers an CORBA</b>	4
Der Netscape Application Server (NAS)	4
Die Architektur des Netscape Application Servers	4
Projektbeschreibung	6
Der CORBA Server	6
Das Design der Bankapplikation	8
Die Aufgaben der AppLogics	9
Das Übertragen von Objektreferenzen	11
Der Inhalt der HTML-Seiten	12
Implementierung unseres Frameworks	12
Speicherung in der Session	12
Übertragen der Parameter	14
Verwendung des Frameworks	16
Programmieren der AppLogics	18
Die HTML Template Seiten	20
Verbinden zu einem CORBA Server mit NAS Extensions	21
Design einer CORBA Extension	22
Speicher für die Extensions	23
Das Objektmodell der Bank Extension	26
Java Exceptions in NAS Extensions?	27
Implementierung der Bank Extension	28
Verwendung der Bank Extension	29

## **Anbindung des Netscape Application Servers an CORBA**

### ***Der Netscape Application Server (NAS)***

Ein Netscape Application Server „sitzt“ im Internet hinter einem Web-Server und übernimmt die Funktion von CGI-Scripten und CGI-Programmen.

Die Benutzung eines Applikationsservers hat verschiedene Vorteile:

- Geschwindigkeit
- Skalierbarkeit
- Zuverlässigkeit
- Stabilität

Dies ist besonders interessant, wenn es sich um eine Applikation handelt, die sehr hoher Belastung ausgesetzt ist, eine Datenbank benutzt und/oder andere „teuere“ Dienstleistungen in Anspruch nimmt.

Um eine hohe Antwortgeschwindigkeit zu erreichen, verwendet der Netscape Application Server verschiedene Techniken:

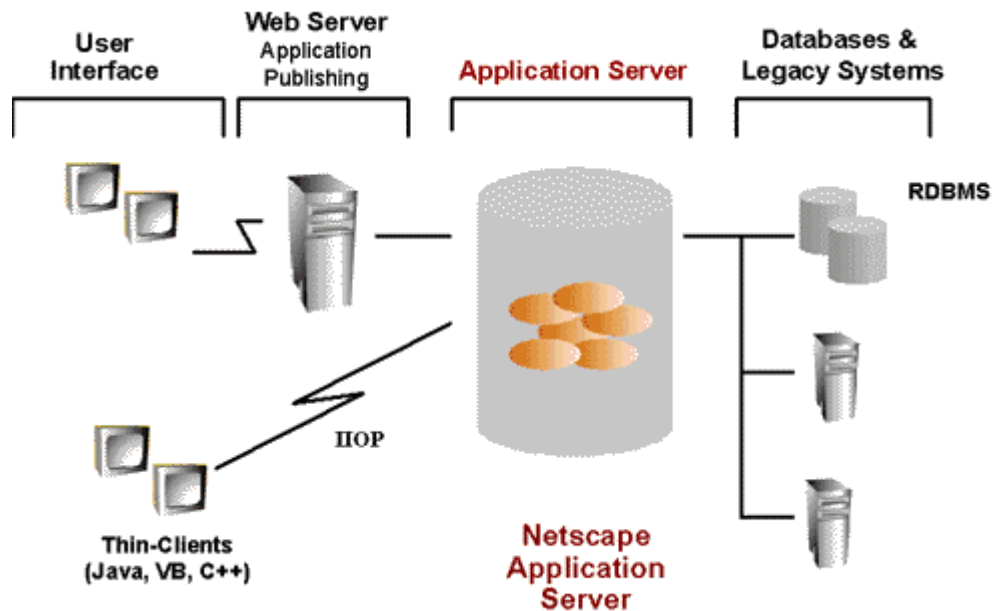
- Bereits errechnete Ergebnisse werden in einen Zwischenspeicher („cache“) geladen und bei wiederholtem Bedarf ohne Neuberechnung an den Kunden weitergegeben: „caching“.
- Verbindungen zu Datenbanken oder anderen externen Technologien werden nicht nur für einen Prozeß verwendet, sondern viele Prozesse können sich mit Hilfe des Applikationsservers wenige vorhandene Verbindungen teilen: „connection pooling“
- Um die Last besser bewältigen zu können, gibt es die Möglichkeit, diese auf mehrere Applikationsserver zu verteilen. Mehrere Applikationsserver können zu diesem Zweck zu einem Cluster zusammengeschlossen werden und jeder Applikationsserver kann wiederum bis zu 16 Server-Prozesse starten. Die Last wird nun, abhängig vom jeweiligen Belastungszustand, auf die vorhandenen Server-Prozesse verteilt: „load balancing“
- Last-Monitore und Management-Tools sorgen für bessere Kontrolle der Arbeitsabläufe

### ***Die Architektur des Netscape Application Servers***

NAS Applikationen basieren auf einer Client/Server Architektur. Wie Grafik 1 zeigt, gibt es in dieser Architektur drei oder vier Schichten, je nach Art der verwendeten Clients:

- I. Die Benutzerschnittstelle, das ist entweder eine HTML-Seite in einem Browser (HTML-Client), oder ein sogenannter „Thin-Client“, der per IIOP (Internet Inter-ORB Protokoll) direkt mit dem Application Server kommuniziert.
- II. Applikations-Publishing, hierbei handelt es sich um den Web-Server, der die Anfragen eines HTML-Clients an den Application Server weiterleitet (wird für Thin-Clients nicht benötigt).

- III. Application Server, der ggf. mit der IV. Schicht kommuniziert und ein errechnetes Ergebnis entweder als HTML-Code an den Web-Server oder mit IIOOP an den Thin-Client weitergibt.
- IV. Datenbanken und bereits existierende Systeme, die mit dem Netscape Application Server verbunden sind.



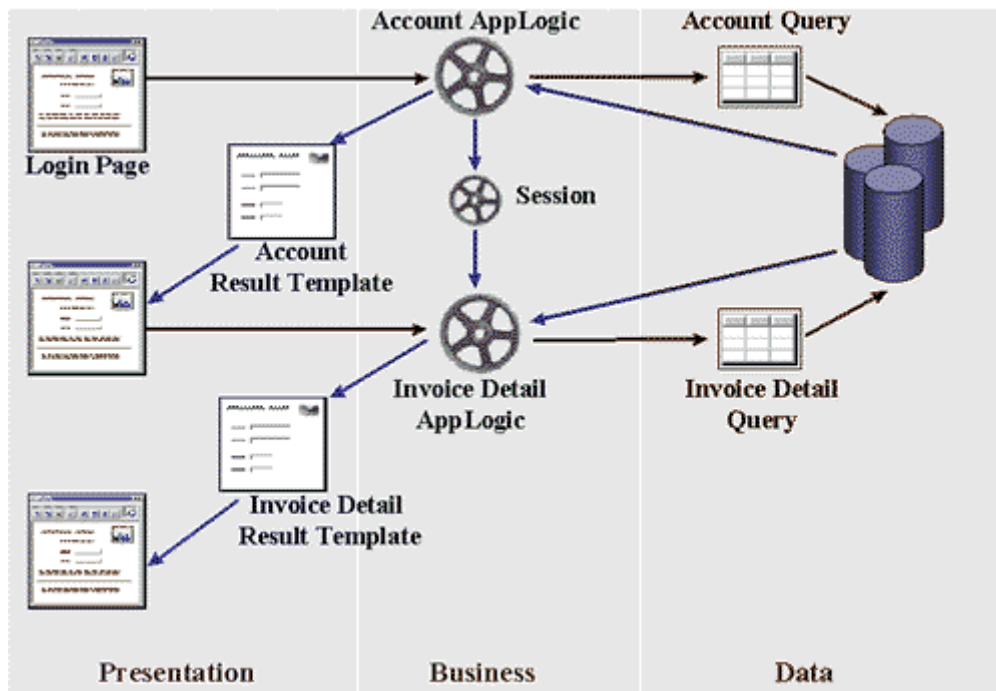
Grafik 1: Die Architektur des Netscape Applikation Servers (Quelle: Netscape Präsentationen)

Ich betrachte in meinem Bericht ausschließlich die Möglichkeit, den Netscape Application Server (NAS) mit HTML-Clients anzusprechen, das ist die Vier-Schicht-Methode.

Es ist darauf zu achten, daß hierbei die Präsentationsschicht von der Geschäftslogik getrennt ist. Web-basierte Clients benutzen einen Web-Browser, der HTML-Seiten anzeigt; während der Anteil der Applikation, der auf dem Server ist, aus Java oder C++ Objekten besteht. Diese Objekte haben jeweils *eine* Standardmethode, die „execute“ Methode, die aufgerufen wird, wenn ein Client eine Anfrage stellt. Die Java und C++ Objekte heißen „AppLogics“, und eine Applikation besteht aus einer Menge von AppLogics und HTML-Seiten.

Der Kontrollfluß einer Applikation ist ein Fluß durch eine Reihe von HTML-Seiten und AppLogics, die Daten untereinander austauschen können. Da ein AppLogic seine referierten Objekte verliert, wenn die execute-Methode endet, wird ein sog. „Session and state management“ zur Verfügung gestellt, das Daten speichert. Die Daten, die in Session- und State-Objekten gespeichert werden, können dadurch zwischen AppLogics und HTML-Seiten ausgetauscht werden.

Grafik 2 zeigt zwei AppLogics, die zu einem Session-Objekt verweisen, das auf die gemeinsamen Daten verweist.



Grafik 2: Kontroll- und Datenfluß zwischen zwei AppLogics (Quelle: Netscape Präsentationen)

## Projektbeschreibung

Bei diesem Projekt assistierte ich Prof. Dr. Schmauch der Fachhochschule Karlsruhe, Fachbereich Wirtschaftsinformatik, die ihr Sabbatical bei der Firma Netscape Communications Corporation in Mountain View, Kalifornien, USA, durchführte.

Unsere Aufgabe war es, den Netscape Application Server an ein CORBA-System anzubinden, d.h. eine Applikation für den NAS zu schreiben, die mit einem CORBA-Server kommuniziert und entfernte Objekte benutzt. Im Prinzip war es klar, daß dies funktionieren kann, aber bisher hatte noch keiner eine solche Anbindung programmiert. Es war noch nicht bekannt, welche Probleme es bei der Durchführung geben könnte und wie ein generisches Konzept für diese Aufgabe aussehen würde.

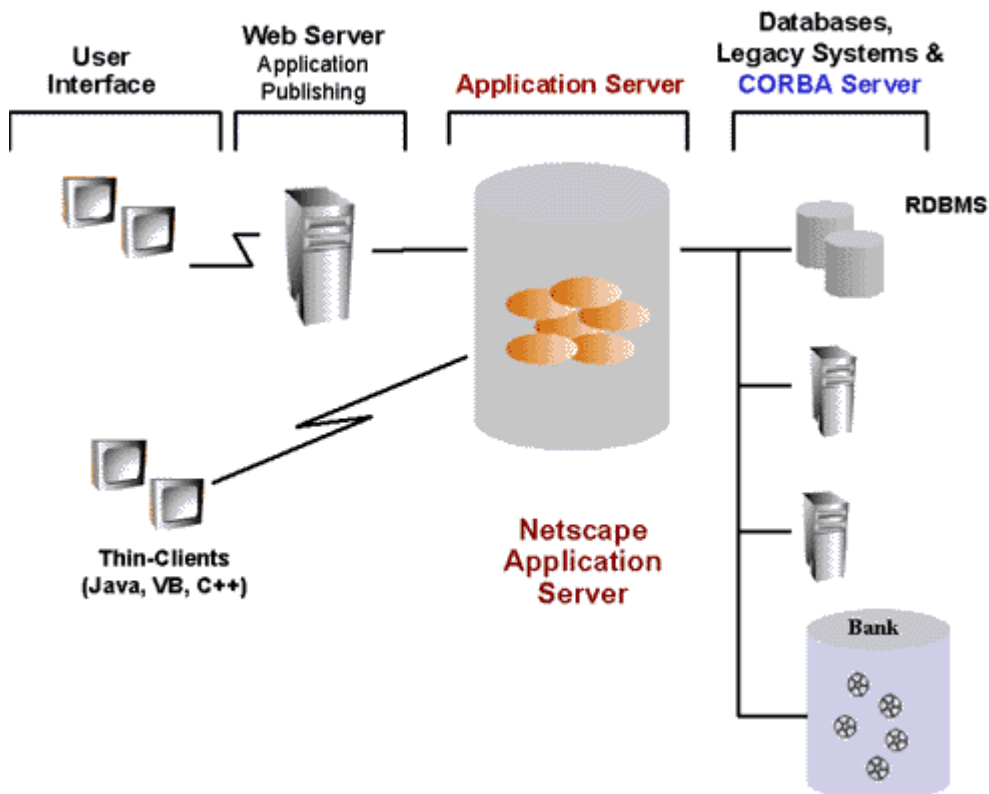
Als zweiten Schritt sollten wir überprüfen, ob es sinnvoll ist, NAS-Extensions (wird später erklärt) für eine solche Anbindung zu verwenden.

## Der CORBA Server

Als Anwendungsfall wählten wir eine Bank, die auf einem CORBA-System vom Typ OrbixWeb2.0 implementiert wurde. Diese Bank hat zwei Klassen von Objekten: Die Klassen *bank* und *account*. Die Klasse *bank* verwaltet *accounts* mit den Methoden *newAccount(.)*, *findAccount(.)* und *deleteAccount(.)*. Ein *account* hat die Attribute *name* und *balance* und die Methoden *makeLodgement(.)* (macheEinzahlung) und *makeWithdrawal(.)* (macheAuszahlung). Listing 1 zeigt die CORBA-IDL.

```
//  
// a simple description of a bank account  
//  
interface account {  
    readonly attribute string name;  
    readonly attribute float balance;  
  
    void makeLodgement (in float amount);  
    void makeWithdrawal (in float amount);  
  
};  
  
//  
// a bank simply manufactures accounts  
//  
// bank::reject is raised if a duplicate account name  
// is seen  
//  
interface bank {  
  
    exception reject {string reason;};  
  
    account newAccount (in string name) raises (reject);  
    account findAccount (in string name);  
    void deleteAccount (in account anAccount);  
  
};
```

Listing 1: CORBA-IDL des Bank-Servers (© Iona Technologies, verändert von Cosima Schmauch)



Grafik 3: NAS in Verbindung mit einem CORBA Server (Quelle: Präsentation von Prof. Dr. Schmauch)

Wenn wir nun CORBA zu einer Applikation hinzufügen, fügen wir eine zweite Client/Server Architektur neben der des NAS ein. Wir setzen den CORBA-Server in eine Reihe zusammen mit Datenbanken und bereits existierenden Systemen, wie in Grafik 3 ersichtlich.

Eine NAS-Applikation, die sich an einen CORBA Server bindet, tut dies durch AppLogics. Diese rufen Methoden an den entfernten CORBA Objekten auf, empfangen Ergebnisse, und bereiten die Antwortseiten vor, die diese Ergebnisse anzeigen. Im Fall unseres Bank-Servers könnte ein AppLogic die Methode *makeLodgement(.)* an einem vorher erhaltenen *account* aufrufen, um Geld einzuzahlen - und anschließend eine Seite zurückgeben, die den aktuellen Kontostand angibt. Das Erhalten des *account* Objektes wird gewöhnlich im Voraus von einem anderen AppLogic erledigt. Deshalb müssen NAS-Applikationen, die ja aus einer Menge von HTML-Seiten und AppLogics bestehen, die Referenzen auf CORBA Objekte über verschiedene Adreßräume hinweg verwalten: Im Browser des Clients, in den verschiedenen AppLogics, die zu einer Applikation gehören und sogar über mehrere NAS hinweg muß der aktuelle Zustand der Applikation bekannt sein.

Dies lösten wir, indem wir die CORBA Objektreferenz als Zeichenkette zusammen mit einem eindeutigen Identifizierer (UID) im Session-Objekt speicherten, das vom Session- und State-Management des NAS zur Verfügung gestellt wird. Anschließend leiteten wir die UID an die HTML-Seite weiter. Diese Verfahrensweise erlaubt dem Programmierer, die Applikation voll zu kontrollieren; sogar das viel gefürchtete Drücken des „zurück“-Knopfes im Browser, welches den Status einer Applikation durcheinanderbringen kann, ist somit kontrollierbar.

## **Das Design der Bankapplikation**

Als wir die Bankapplikation konzipierten, gingen wir in dieser Reihenfolge vor:

- Definieren der Aufgabe jedes AppLogics, die Reihenfolge der verschiedenen AppLogics und die Ausgabe jedes AppLogics:
  - *bankNewAccountAppLogic*: ruft *newAccount(.)* an einem *bank* Objekt auf, was ein *account* Objekt zurückgibt
  - *bankFindAccountAppLogic*: ruft *findAccount(.)* an einem *bank* Objekt auf, was ein *account* Objekt zurückgibt
  - *bankDeleteAccountAppLogic*: ruft *deleteAccount(.)* an einem *bank* Objekt auf, was das betreffende *account* Objekt auf dem Server löscht
  - *accountMakeLodgementAppLogic*: ruft *makeLodgement(.)* an einem *account* Objekt auf, was den eingegebenen Betrag zum Kontostand des Objektes hinzufügt
  - *accountMakeWithdrawalAppLogic*: ruft *makeWithdrawal(.)* an einem *account* Objekt auf, was den eingegebenen Betrag vom Kontostand des Objektes abzieht
  - *accountGetNameAppLogic*: ruft *get\_name()* an einem *account* Objekt auf, was den Namen des Objektes zurückgibt
  - *accountGetBalanceAppLogic*: ruft *get\_balance()* an einem *account* Objekt auf, was den aktuellen Kontostand des Objektes zurückgibt

- die beiden „get“ AppLogics wurden später zu einem *accountGetNameAndBalanceAppLogic* zusammengefaßt
- für jedes AppLogic muß entschieden werden, welche Objektreferenzen vorhanden sein müssen, um die spezifizierte Aufgabe zu erfüllen, Beispiele:
  - *bankNewAccountAppLogic*: benötigt eine Referenz auf ein *bank* Objekt
  - *bankDeleteAccountAppLogic*: benötigt eine Referenz auf ein *bank* Objekt und eine Referenz auf ein *account* Objekt
  - *accountMakeLodgementAppLogic*: benötigt eine Referenz auf ein *account* Objekt
- für jedes AppLogic muß entschieden werden, welche Objektreferenzen es an das nächste AppLogic weitergeben muß, Beispiele:
  - *bankNewAccountAppLogic*: gibt eine Referenz auf ein *bank* Objekt und eine Referenz auf ein *account* Objekt weiter
  - *bankDeleteAccountAppLogic*: gibt eine Referenz auf ein *bank* Objekt weiter
  - *accountMakeLodgementAppLogic*: gibt eine Referenz auf ein *account* Objekt weiter
  - hierzu ist zu sagen, daß das *accountMakeLodgementAppLogic* immer auch eine Referenz auf ein *bank* Objekt empfängt und weitergibt, da ansonsten zu einem späteren Zeitpunkt in der Applikation keine Methoden an einem *bank* Objekt mehr ausgeführt werden könnten, und daher neu an den *bank* Server gebunden werden müßte
- Definieren des Inhalts jeder HTML-Seite, die dem Benutzer präsentiert wird, Beispiele:
  - bevor ein neues account Objekt erstellt werden kann, muß dessen Name vom Benutzer eingegeben worden sein (Grafik 4)
  - danach wird, wenn keine Fehler auftraten, die das account Objekt betreffende Information und ein weiteres Auswahlmenü angezeigt (Grafik 5)



Grafik 4: New Account Form



Grafik 5: Account Menu for John

## Die Aufgaben der AppLogics

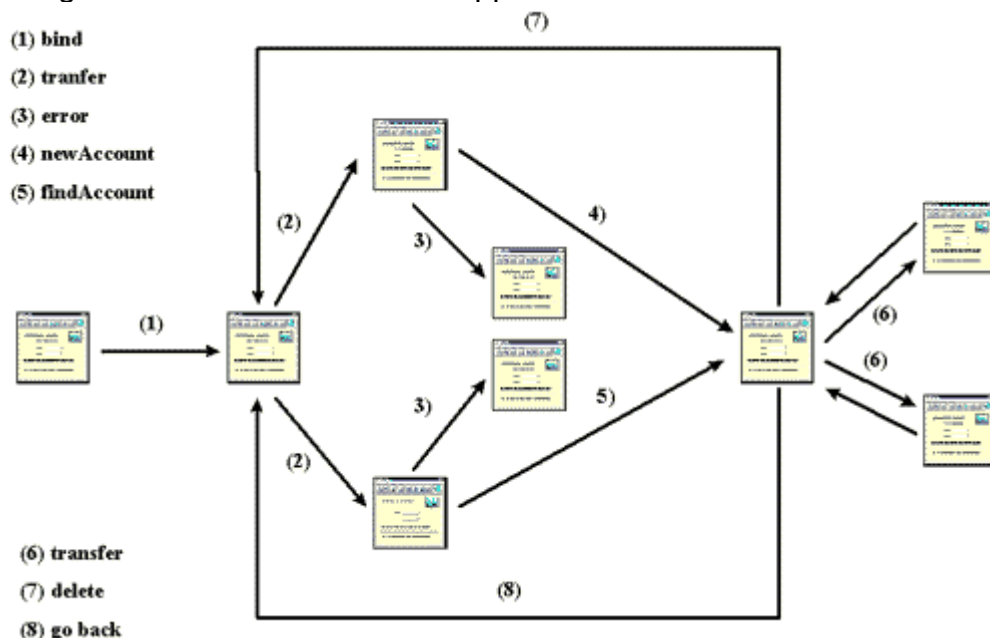
Jedes AppLogic muß zwei Aufgaben erfüllen: Es muß ein Teil der Geschäftslogik implementieren und den Aufruf einer Methode an einem entfernten Objekt auf dem

*bank* Server ausführen. Nachdem ein Benutzer einen Namen eingegeben hat in der „New Account Form“ und den OK-Knopf gedrückt hat (Grafik 4), muß das aufgerufene AppLogic die *newAccount(.)* Methode am entfernten *bank* Objekt aufrufen. Diese Methode gibt ein *account* Objekt zurück, dessen Name und Kontostand in der nächsten HTML-Seite angezeigt wird (Grafik 5). Jedes AppLogic ist verantwortlich für einen Hauptaufruf an einem Objekt, und der Name des AppLogics soll diesen Hauptaufruf benennen. Im Fall des Öffnens eines neuen Kontos, muß das AppLogic die *newAccount(.)* Methode an einem *bank* Objekt aufrufen (Hauptaufruf), aber auch die Methoden *get\_name()* und *get\_balance()* am zurückgegebenen *account* Objekt. Da nun das Öffnen eines neuen Kontos die Hauptaufgabe des AppLogics ist, und das Objekt, an dem diese Methode aufgerufen wird, ein *bank* Objekt ist, heißt dieses AppLogic *bankNewAccountAppLogic*.

Jeder CORBA Client muß sich mindestens *einmal* an ein Objekt im Server binden, um einen ersten Aufruf an einem entfernten Objekt machen zu können. Neben den bereits im vorigen Abschnitt genannten AppLogics benötigen wir daher noch ein weiteres, das sich an ein *bank* Objekt auf dem Server bindet, und nennen es *bankBindAppLogic*.

Für diese Applikation benötigen wir noch ein AppLogic, das keine Aufrufe an einem entfernten Objekt tätigt, wohl aber bereits vorhandene Objektreferenzen an ein nächstes AppLogic weitergibt und eine HTML-Seite an den Browser zurückgibt. Es ist gleichzusetzen mit einem Link in einer HTML-Seite, mit dem Unterschied, daß hierbei vorhandene Objektreferenzen weitergegeben werden. Wir nennen dieses AppLogic *transferAppLogic*.

Grafik 6 zeigt den Kontrollfluß der Bankapplikation.



Grafik 6: Kontrollfluß der Bankapplikation (Quelle: Präsentation von Prof. Dr. Schmauch)

Ein weiteres AppLogic, das für diese Anwendung benötigt wird, ist das *removeAccountFromSessionAppLogic*, das die Referenz auf ein *account* Objekt aus

der Session löscht, um sicherzustellen, daß der logische Fluß in der Applikation eingehalten wird.

Jedes AppLogic, das Methoden an einem entfernten Objekt aufruft, tut dies an einem Proxy-Objekt von einem entfernten CORBA-Objekt. Um zu verhindern, daß jedes AppLogic diese Referenz nach dem Abarbeiten des AppLogics wieder verliert, haben wir eine Strategie entwickelt, diese Referenzen zu verwalten.

## **Das Übertragen von Objektreferenzen**

Ein oft genutzter Ansatz, Daten unter AppLogics auszutauschen, ist es, das Session- und State-Management des NAS zu nutzen. Für unseren Fall kommt nur ein Session Objekt in Frage, da ein sog. State Objekt für alle AppLogics gleich ist, wir jedoch verschiedene gleichzeitige Benutzer haben, die jeweils einen eigenen Zustand der Applikation benötigen.

Bei der Benutzung des zur Verfügung gestellten Session Objektes treten jedoch Probleme auf. Ein Session Objekt des NAS kann nur Objekte vom Typ GXVAL speichern. Ein GXVAL wiederum kann ausschließlich Integers, Strings und BLOBs (Binary Large Objects) aufnehmen. Dies zwingt uns, die CORBA-Proxies zu Zeichenketten (Strings) zu verwandeln und die Zeichenketten in das Session Objekt zu speichern. Des weiteren muß es einem AppLogic möglich sein, die *richtige* Objektreferenz zu erhalten. Hierzu identifizieren wir jede Objektreferenz mit einem eindeutigen Identifizierer (unique ID → UID) und speichern ein Paar aus UID und Objektreferenz in dem Session Objekt. Dies tun wir auch, um unabhängig von der Applikation zu sein, da wir eine generische Lösung anbieten möchten.

Da die Objektreferenzen von einem AppLogic zum anderen weitergegeben werden müssen, geben wir die UID jeder benötigten Objektreferenz an die HTML-Seite weiter. Das nächste AppLogic kann nun mit Hilfe der UID und der Session ID die betreffenden Objektreferenzen aus der Session erhalten. Es stellt sich die Frage, warum wir nicht einfach die Objektreferenz an die HTML-Seiten weitergegeben haben. Zum Einen haben wir die Möglichkeit, eine Objektreferenz aus der Session zu löschen, und somit einen Zugriff auf dieses Objekt zu verhindern. Wenn Die Objektreferenz in der HTML-Seite wäre (z.B. durch Drücken des „zurück“-Knopfes im Browser), könnte ein AppLogic nicht entscheiden, ob auf dieses Objekt vom Benutzer noch zugegriffen werden darf oder nicht - es würde in jedem Fall den Zugriff gestatten. Zum anderen kann durch das Schützen der Objektreferenz ein eventueller direkter Zugriff des Benutzers auf das Objekt, ohne Kontrolle durch die AppLogics, verhindert werden.

Um zu verhindern, daß ein Objekt benutzt wird, auf das der Benutzer keine Zugriffsrechte mehr hat, wird einfach das entsprechende UID/Objektreferenz Paar aus dem Session Objekt gelöscht und ein späterer Zugriff auf dieses dann nicht mehr vorhandene Paar mit einer Fehlermeldung beantwortet.

Um die UIDs unter den AppLogics auszutauschen, senden wir diese als HIDDEN VALUES an die HTML-Seiten. Die HIDDEN VALUES müssen in jeder FORM der

HTML-Seite vorkommen, die ein weiteres AppLogic der Anwendung als FORM ACTION angegeben hat.

## Der Inhalt der HTML-Seiten

Die HTML-Seiten enthalten jeweils einen HIDDEN VALUE pro Objektreferenz. Es wird die UID gespeichert, zusammen mit dem Namen der jeweiligen Klasse, von der die entsprechende Objektreferenz ist. In unserem Fall sind die möglichen Klassen *account* und *bank*. Um den Namen der UID von anderen HIDDEN VALUES unterscheiden zu können, setzen wir vor Namen von Objektreferenzen die Kennung „\_OR\_“. Der Name einer UID einer *bank* Objektreferenz sieht so aus: „\_OR\_bank“, und die eines *accounts*: „\_OR\_account“, siehe auch Listing 2.

<pre>&lt;HTML&gt; &lt;HEAD&gt;   &lt;TITLE&gt;Bank Example - Bank Menu&lt;/TITLE&gt; &lt;/HEAD&gt; &lt;BODY BGCOLOR="#FFFFFF" LINK="#CC0000" ALINK="#003366"       VLINK="#FF3300" basefont=3&gt; &lt;H1&gt;&lt;CENTER&gt;Welcome to the Bank&lt;/CENTER&gt;&lt;/H1&gt; &lt;CENTER&gt;&lt;TABLE BORDER=0&gt;   &lt;TR&gt;     &lt;TD VALIGN=bottom&gt;       &lt;P&gt;&lt;FORM         ACTION="/cgi-bin/gx.cgi/GUIDGX-{9c3529f0...a0d1017dfc}"         METHOD=POST&gt;       &lt;P&gt;&lt;INPUT TYPE=hidden         NAME=target         VALUE="Bank_Example/Templates/new.html"&gt;       &lt;INPUT TYPE=hidden         NAME="_OR_bank"         VALUE="_UID_1f1508:d1643347cd:-7fe6"&gt;       &lt;INPUT TYPE=submit NAME=ok         VALUE="New Account"&gt;     &lt;/FORM&gt;&lt;/P&gt;     &lt;/TD&gt;     &lt;TD VALIGN=bottom&gt;       &lt;P&gt;&lt;FORM         ACTION="/cgi-bin/gx.cgi/GUIDGX-{9c3529f0...a0d1017dfc}"         METHOD=POST&gt;       &lt;P&gt;&lt;INPUT TYPE=hidden         NAME=target         VALUE="Bank_Example/Templates/find.html"&gt;       &lt;INPUT TYPE=hidden         NAME="_OR_bank"         VALUE="_UID_1f1508:d1643347cd:-7fe6"&gt;       &lt;INPUT TYPE=submit NAME=ok         VALUE="Find Account"&gt;     &lt;/FORM&gt;&lt;/P&gt;     &lt;/TD&gt;   &lt;/TR&gt; &lt;/TABLE&gt; &lt;/CENTER&gt; &lt;/BODY&gt; &lt;/HTML&gt;</pre>	<p><b>Erklärung:</b> Diese HTML-Seite speichert die UID der Objektreferenz „_UID_1f1508:d1...“ zusammen mit ihrem Namen „_OR_bank“ in beiden FORMS, die die beiden Knöpfe implementieren. Es handelt sich hier um den Aufruf des <i>transferAppLogics</i>, das die Zielseite nach Aufruf des AppLogics als Ergebnis zurückliefert. Diese Angabe ist in „target“ gespeichert und wird vom <i>transferAppLogic</i> auch so erwartet.</p>
--	--

Listing 2: HTML-Seite mit HIDDEN VALUES (Quelle: HTML-Code von Christian Ey, ©98 Netscape)

## Implementierung unseres Frameworks: Speicherung in der Session

Um unsere Objektreferenzen in der Session („Sitzung“) zu speichern, müssen wir die Objektreferenzen in Zeichenketten umwandeln und dann in einem GXVAL-Objekt

speichern. Dieses wird anschließend in einer von uns modifizierten Session gespeichert, die, wie schon zuvor beschrieben, nur GXVAL-Objekte aufnehmen kann.

Um dies zu bewerkstelligen, erzeugen wir eine neue Klasse namens *SessionStorage*, die die Klasse *Session2* beerbt (wird von NAS zur Verfügung gestellt). Dabei fügen wir die Methoden *addValue*, *setValue*, *getValue* und *removeValue* hinzu. Diese Methoden akzeptieren Objekte jeder Klasse als Ein- und Ausgabeparameter, und erfordern daher zusätzliche Konvertierungsmethoden. Diese heißen *convert* und *reconvert* und haben die Aufgabe, ein Objekt beliebigen Typs in ein GXVAL Objekt zu konvertieren und umgekehrt. Da wir an dieser Stelle nicht wissen können, von welchem Typ die in der Session zu speichernden Objekte tatsächlich sind, sind die Konvertierungsmethoden abstrakt und müssen von einer Unterklasse implementiert werden. *convert* und *reconvert* „werfen“ Exceptions, um Ausnahmebehandlungen zu ermöglichen. In Listing 3 wird die Klasse *SessionStorage* noch einmal erklärt. Beachten Sie hierbei bitte, daß der Code in diesem Listing verändert wurde, um die Auflistung zu verkürzen und trotzdem den logischen Fluß zu bewahren.

Listing	Erklärung
<pre> package StorageAndParameters; import com.kivasoft.*; import com.kivasoft.session.*; import com.kivasoft.types.*; abstract public class SessionStorage extends Session2 {     /*      * Function:      * -manages GXVAL objects in the session object      * -provides methods to add, set, get and remove the values      */     public SessionStorage(ISession2 session)     {         super(session);     } </pre>	<p><i>SessionStorage</i> ist eine Unterklasse der <i>Session2</i> Klasse. Sie stellt ein <i>IVallList</i> Objekt zur Verfügung, das <i>GXVAL</i> Objekte speichern kann. Wir definieren die Methoden <i>addValue</i> (Wertepaar hinzufügen), <i>setValue</i> (setzen), <i>getValue</i> (bekommen) und <i>removeValue</i> (löschen), um den Speicher zu manipulieren. <i>SessionStorage</i> ist eine abstrakte Klasse, da die Methoden <i>convert</i> und <i>reconvert</i> abstrakt sind.</p>
<pre> public String createMyOwnVariable() {     /* <b>Precondition:</b>      * - anObject is not null      * <b>Postcondition:</b>      * -this session object contains one more pair of      * (Variable name/GXVAL object)      * <b>Returns:</b>      * - a unique variable name      */     return new     VariableAndParameter().createVariable(); } </pre>	<p><i>createMyOwnVariable</i> ist eine Factory-Methode, die von <i>setValue</i> aufgerufen wird, um eine Variablenkennung zurückzugeben. Es muß von der Unterklasse überschrieben werden.</p>
<pre> abstract public GXVAL convert( Object anObject) throws Exception; /* <b>Precondition:</b>  * - anObject is not null </pre>	<p><i>convert</i> muß von der Unterklasse definiert werden. Es implementiert die Konvertierung eines Objektes in ein <i>GXVAL</i> Objekt. Weil die überschreibende</p>

<pre> * <b>Returns:</b> * - a GXVAL object * <b>Exception:</b> * - the overwriting method may throw an exception */ </pre>	<p>Methode eine Ausnahme „werfen“ könnte, müssen wir dies hier spezifizieren.</p>
<pre> abstract public Object reconvert( GXVAL aValue) throws Exception; </pre>	<p><i>reconvert</i> muß von einer Unterklasse definiert werden. Es implementiert die Konvertierung eines GXVAL Objektes in ein Objekt.</p>
<pre> <b>public String addValue( Object anObject) throws Exception { ... }</b> </pre>	<p><i>addValue</i> nimmt ein Objekt, erzeugt eine Variable und speichert das Paar im Session-Objekt durch Aufruf von <i>setValue</i></p>
<pre> <b>public boolean setValue( String keyString, Object anObject) throws Exception {</b> boolean result; [...] // convert the object GXVAL sOts = <b>convert( anObject);</b> // store the (key/GXVAL object) pair in this session result = (<b>this.getSessionData().setVal( keyString, sOts)</b> == GXE.SUCCESS) return result; <b>}</b> </pre>	<p><i>setValue</i> speichert die Variable zusammen mit dem konvertierten Objekt im Session Objekt</p>
<pre> <b>public Object getValue( String keyString) throws Exception { [...]</b> GXVAL sOR = <b>this.getSessionData().getVal(keyString);</b> return <b>reconvert( sOR);</b> <b>}</b> </pre>	<p><i>getValue</i> bekommt das GXVAL Objekt vom Session Objekt und benutzt die Variable als Suchschlüssel. Dann ruft es die <i>reconvert</i> Methode auf und konvertiert das GXVAL Objekt zu einem Objekt.</p>
<pre> <b>public int removeValue( String keyString) /* Precondition:</b> * - the key string is not null */ { [ ... ] return <b>this.getSessionData().removeVal( keyString);</b> <b>}</b> </pre>	<p><i>removeValue</i> löscht ein Variable/GXVAL-Objekt Paar vom Session Objekt und gibt folgende Werte zurück:</p> <ul style="list-style-type: none"> <li>• <i>GXE.SUCCESS</i>: alles OK</li> <li>• <i>GXE.INVALID_ARG</i>: Vorbedingung(en) nicht erfüllt</li> <li>• anderer GXE Fehler: wenn es nicht möglich war, das Paar zu löschen</li> </ul>

Listing 3: Die Klasse SessionStorage (Quelle: Code von Christian Ey, © 98 Netscape)

## Übertragen der Parameter

Das Übertragen der Parameter findet zwischen AppLogics und HTML-Seiten statt. Dies kann in NAS durch Verwendung zweier verschiedener Klassen getan werden: entweder *TemplateMapBasic* oder *TemplateDataBasic*. Die Vor- und Nachteile der beiden abzuwägen würde den Rahmen dieses Dokumentes sprengen, man kann dies jedoch in der Dokumentation des Netscape Application Servers nachlesen. Wir entwickelten die Klasse *ParameterPassing*, die Methoden für beide Template-Klassen zur Verfügung stellt. Eine Liste vom Typ *IValList* wird in die entsprechende Klasse gespeichert und je nach Methode als HIDDEN VALUE für HTML-Seiten oder als Rohdaten codiert.

Folgende Methoden sind in der Klasse *ParameterPassing* definiert:

- Methoden, die Feldnamen zur Bearbeitung in HTML-Template-Files zurückgeben (werden hauptsächlich innerhalb der Klasse verwendet)



Die Klasse *VariableAndParameter* enthält verschiedene Methoden, die beim Arbeiten mit Variablen und Parametern behilflich sind.

- `public String variablePrefix();`  
gibt den definierten Präfix zurück, der eine Variable als solche identifiziert: „\_VAR\_“
- `public String parameterPrefix();`  
gibt den definierten Präfix zurück, der einen Parameter als solchen identifiziert: „\_PAR\_“
- `public String createVariable();`  
erzeugt einen eindeutigen UID-String mit Präfix von *this.variablePrefix()*.
- `public boolean isVariable( String aString);`  
überprüft, ob *aString* den Variablen-Präfix hat.
- `public boolean isParameter( String aString);`  
überprüft, ob *aString* den Parameter-Präfix hat.
- `public String createParameter( String nameString);`  
erzeugt einen Parameter aus *nameString* durch Voranstellen des Präfixes, der von *this.parameterPrefix()* zurückgegeben wird. *nameString* wird ohne Veränderung zurückgegeben, wenn *nameString* schon ein Parameter ist.
- `public String parameterToString( String aParameter);`  
gibt *aParameter* als String zurück, d.h. es löscht den von *this.parameterPrefix()* zurückgegebenen Präfix von *aParameter* und gibt das Ergebnis zurück.

Das Framework besteht daher aus vier Klassen:

- *VariableAndParameter*; erzeugt eindeutige Identifizierer, denen ein Präfix vorangestellt wird; stellt Methoden zum Checken der Struktur einer Variable und eines Parameters zur Verfügung; erlaubt das Umwandeln von Strings in Parameter und zurück.
- *ParameterPassing*; stellt Methoden zum Extrahieren von Parametern von einer *IValList* und zum Erzeugen von *TemplateMapBasic* Objekten und *TemplateDataBasic* Objekten mit allen nötigen Daten für die nächste HTML-Seite.
- *SessionStorage*; zum Manipulieren des Session Speichers.
- *ExtensionStorage*; wird später zum Verbinden von NAS und CORBA mit Extensions benötigt, sehen Sie bitte hierzu das entsprechende Kapitel in diesem Bericht.

## **Verwendung des Frameworks**

Frameworks werden durch die Bildung von Unterklassen der einen oder anderen Framework-Klasse verwendet. Hierbei werden noch nicht definierte oder zu generisch gestaltete Stellen im Framework applikationsspezifisch überschrieben. Da die *SessionStorage* Klasse abstrakt ist, wegen der beiden abstrakten Methoden *convert(.)* und *reconvert(.)*, müssen wir unsere eigene Speicherklasse definieren, die mindestens diese beiden Methoden implementiert.

Zusätzliche Framework-Klassen und -Methoden, die beerbt werden:

- *VariableAndParameter*: durch *CorbaVariableAndParameter*,  
Methoden: *createMyOwnVariable*

- *SessionStorage*: durch *CorbaSession*;  
Methoden: *addValue*, *setValue*, *getValue*
- *ParameterPassing*: durch *CorbaParameterPassing*;  
Methoden: *templateDataName*, *isMyOwnParameter*

Eine Erklärung der Klasse *CorbaSession* befindet sich in Listing 4.

Listing	Erklärung
<pre> package NASCorba; import StorageAndParameters.*; import com.kivasoft.types.*; import com.kivasoft.util.*; import com.kivasoft.*; import IE.Iona.Orbix2._CORBA; import IE.Iona.Orbix2.CORBA.*; public class CorbaSession extends SessionStorage {     public CorbaSession(ISession2 session) {         super(session);     } </pre>	<p>Die <i>CorbaSession</i> Klasse ist eine Unterklasse der <i>SessionStorage</i> Klasse. Sie definiert, wie eine Variable zu erzeugen ist, wie ein Objekt in ein GXVAL Objekt zu konvertieren ist und umgekehrt. Sie spezifiziert auch, daß die Methoden <i>addValue</i>, <i>setValue</i> und <i>getValue</i> eine <i>CORBA SystemException</i> „werfen“ anstelle der generellen <i>Exception</i>.</p>
<pre> public String createMyOwnVariable() { [...]     return new CorbaVariableAndParameter().createVariable(); } </pre>	<p>erzeugt eine Variable durch Benutzung der Klasse <i>CorbaVariableAndParameter</i>.</p>
<pre> public GXVAL convert( Object anObject) throws SystemException {     // this turns a string into a GXVAL object     IValList temp = GX.CreateValList();     temp.setValString( "temp",         _CORBA.Orbix.object_to_string(             (_ObjectRef)anObject));     return temp.getVal( "temp"); } </pre>	<p>wandelt eine CORBA Objekt Referenz in eine Zeichenkette um und speichert diese in einem GXVAL Objekt. Der CORBA-Aufruf könnte eine <i>SystemException</i> „werfen“.</p>
<pre> public Object reconvert( GXVAL aValue) throws SystemException {     // this turns a GXVAL object into a string     IValList temp = GX.CreateValList();     temp.setVal( "temp", aValue);     return _CORBA.Orbix.string_to_object(         temp.getValString( "temp")); } </pre>	<p>wandelt eine in einem GXVAL Objekt gespeicherte Zeichenkette in eine CORBA Objekt Referenz um und gibt diese zurück. Der CORBA-Aufruf könnte eine <i>SystemException</i> „werfen“.</p>
<pre> public String addValue( Object anObject) throws SystemException {     String result;     try {         result = super.addValue( anObject);     }     catch (Exception e){         throw (SystemException)e;     }     return result; } </pre>	<p>wandelt die generelle Ausnahme, die von der Methode der Superklasse „geworfen“ wird um in eine CORBA <i>SystemException</i>.</p>

<b>public boolean setValue( String keyString, Object anObject) throws SystemException { [...] }</b>	wandelt Exception um, siehe <i>this.addValue(.)</i>
<b>public Object getValue( String keyString) throws SystemException { [...] }</b>	wandelt Exception um, siehe <i>this.addValue(.)</i>

Listing 4: Die Klasse CorbaSession (Quelle: Code von Christian Ey, © 98 Netscape)

Die Klasse *CorbaVariableAndParameter* ist eine Unterklasse von *VariableAndParameter* und überschreibt die Methoden *variablePrefix()* (gibt nun „\_UID\_“ zurück anstelle von „\_VAR\_“) und *parameterPrefix()* (gibt nun „\_OR\_“ zurück anstelle von „\_PAR\_“).

Die Klasse *CorbaParameterPassing* ist eine Unterklasse von *ParameterPassing* und überschreibt die Methoden *templateDataName()* (gibt nun „OREF“ zurück anstelle von „TDN“) und *isMyOwnParameter()* (ruft die Methode *isParameter()* an der Klasse *CorbaVariableAndParameter* auf anstatt an der Klasse *VariableAndParameter*).

Mit diesen drei Klassen können wir nun relativ einfach Objekt Referenzen im Session Objekt speichern, sie aus der Werteliste, die von den HTML-Seiten zurückgegeben wird, extrahieren und sie dann wieder aus dem Session Objekt herausnehmen. Nun müssen wir noch die AppLogics selbst programmieren.

## Programmieren der AppLogics

Bevor ein AppLogic eine Methode an einem entfernten Objekt aufrufen kann, muß es die Objektreferenzen vom Session Objekt bekommen. Danach sind verschiedene andere Schritte notwendig, um sicherzustellen, daß ein nachfolgendes AppLogic alle benötigten Objektreferenzen erhalten kann. Hier die Schritte, die ein AppLogic dazu ausführen muß:

- I. Anfordern des Session Objektes, das die UID/Zeichenketten-mit-Objektreferenzen Paare enthält.
- II. Überprüfen der Input Parameter, ob sie den Voraussetzungen entsprechen, z.B. Check auf NULL Werte.
- III. Anfordern der Objekt Referenzen vom Session Objekt durch Verwendung der UIDs aus der Input Parameter Liste.
- IV. Implementieren der Geschäftslogic: Die passenden Methoden an den Objektreferenzen aufrufen. Dies kann neue Objektreferenzen erzeugen oder bestehende löschen, was im nächsten Punkt verwaltet wird.
- V. Verwalten der Objekt Referenzen im Session Objekt: hinzufügen oder löschen. Speichern des Session Objektes.
- VI. Vorbereiten der nächsten HTML-Seite für den HTML Client oder der Datenliste für den Thin-Client.

Ein Beispiel hierzu ist in Listing 5 erläutert (Code gekürzt).

Listing	Erklärung
<b>public class bankNewAccountAppLogic</b>	

<pre><b>extends CorbaAppLogic {</b></pre>	
<pre><b>public int execute() {</b> TemplateMapBasic map; String newUID; _bankRef myBank; _accountRef myAccount;</pre>	Dies ist der Start der <i>execute</i> Methode mit der Variablendeklaration.
<pre>// get session object and test it for null CorbaSession <b>mySession =</b> <b>this.getSessionObject();</b> <b>if (mySession == null) {</b>     <b>return</b>         result("&lt;HTML&gt;Wrong session&lt;/HTML&gt;");     <b>}</b></pre>	I. Anfordern des Session Objektes, das die UID/Zeichenketten-mit-Objektreferenzen Paare enthält.
<pre>// verify correctness of precondition String <b>AccountName =</b>     <b>valIn.getValString("AccountName");</b> <b>if ( null == AccountName   </b>     <b>0 == AccountName.trim().length() ){</b>     <b>log("Input error on AccountName");</b>     <b>return result("&lt;HTML&gt;&lt;BODY&gt;AccountName</b>         <b>should not be null!&lt;/BODY&gt;&lt;/HTML&gt;");</b>     <b>}</b> // extract the ORs from valIn IValList <b>myValList=CorbaUtil.extractORs(valIn);</b> // get the UID-String for "bank" from myValList String <b>bankUID =</b>     <b>myValList.getValString( new</b>         <b>CorbaVariableAndParameter().createParameter(</b>             <b>"bank");</b> <b>if (bankUID == null)</b>     <b>return result("&lt;HTML&gt;&lt;BODY&gt;Lost</b>         <b>connection to the</b> <b>bank&lt;/BODY&gt;&lt;/HTML&gt;");</b></pre>	II. Überprüfen der Input Parameter, ob sie den Voraussetzungen entsprechen, z.B. Check auf NULL Werte. Die HTML Seite sollte eine UID für das <i>bank</i> Objekt beinhaltet haben. Wenn dies nicht der Fall ist, müssen wir die Applikation stoppen und eine Fehlermeldung zurückgeben: „Lost connection to the bank“, oder: „Die Verbindung zur Bank wurde verloren“.
<pre>// get the object reference(s) out of the // session object by using the UIDs // from the input parameter list <b>try {</b>     // get the _ObjectRef from the session     _ObjectRef <b>oRef =</b>         (ObjectRef)<b>mySession.getValue(bankUID);</b>     <b>if (oRef == null)</b>         <b>return result("&lt;HTML&gt;&lt;BODY&gt;Sorry,</b>             <b>action not allowed.&lt;/BODY&gt;&lt;/HTML&gt;");</b>     // narrow the _ObjectRef to _bankRef     <b>myBank = bank._narrow(oRef);</b> } <b>catch (SystemException e) {</b>     <b>return result("&lt;HTML&gt;Lost connection to the</b>         <b>bank&lt;/BODY&gt;&lt;/HTML&gt;");</b> }</pre>	III. Anfordern der Objekt Referenzen vom Session Objekt durch Verwendung der UIDs aus der Input Parameter Liste: Mit der UID das <i>bank</i> Objekt vom Session Objekt anfordern, wenn es nicht vorhanden ist (z.B. nach Drücken des „zurück“-Knopfes im Browser), Fehlermeldung ausgeben. Andernfalls die von der Session zurückgegebene CORBA Objektreferenz durch „narrow“ in eine <i>_bankRef</i> umwandeln, was das Proxy-Objekt erzeugt.
<pre>// business logic: get an account reference // from the bank <b>try {</b>     // get a new _accountRef from myBank     <b>myAccount =</b>         <b>myBank.newAccount(AccountName);</b></pre>	IV. Implementieren der Geschäftslogik, hier: <i>newAccount(.)</i> am <i>bank</i> Objekt aufrufen, was ein <i>account</i> Objekt zurückgibt. Mögliche Ausnahmen:

<pre> } catch (SystemException e) {     return result("&lt;HTML&gt;&lt;BODY&gt;Lost connection to the bank&lt;/BODY&gt;&lt;/HTML&gt;"); } catch (reject r) {     return result(" &lt;HTML&gt;&lt;BODY&gt;Attempt to create the account for "+AccountName +" failed: rejected&lt;/BODY&gt;&lt;/HTML&gt;"); } </pre>	<p><i>CORBA SystemException</i>; <i>reject</i>: benutzerdefinierte Ausnahme, die bei dem Versuch ausgelöst wird, ein neues <i>account</i> Objekt zu erzeugen, das den selben Namen erhalten soll wie ein schon bestehendes <i>account</i> Objekt.</p>
<pre> // manage the object reference(s) try {     // store the _accountRef in mySession and     // get back a unique ID     newUID = mySession.addValue(myAccount); } catch (SystemException e) {     return result("&lt;HTML&gt;&lt;BODY&gt;Lost connection to the bank&lt;/BODY&gt;&lt;/HTML&gt;"); } // save the session object this.saveSession(null); </pre>	<p>V. Verwalten der Objekt Referenzen im Session Objekt: hinzufügen oder löschen. Speichern des Session Objektes.</p>
<pre> // prepare the values passed back to the HTML page // add the account to myValList myValList.setValString( new CorbaVariableAndParameter().createParameter( "account"), newUID); // create the TemplateMapBasic map from myValList map = new CorbaParameterPassing( ).createParameterTemplateMap( myValList); try {     // adding NAME and BALANCE to the     // TemplateMapBasic     map.putString("NAME", myAccount.get_name());     map.putString("BALANCE", new Float(myAccount.get_balance()).toString()); } catch (SystemException e) {     return result("&lt;HTML&gt;&lt;BODY&gt;Lost connection to the bank&lt;/BODY&gt;&lt;/HTML&gt;"); } return evalOutput("Bank_Example/Templates/account.html" , (ITemplateData)null, (ITemplateMap)map, null, null); } </pre>	<p>VI. Vorbereiten der nächsten HTML-Seite für den HTML Client oder der Datenliste für den Thin-Client. In diesem Fall benötigt die HTML-Seite noch den Namen des aktuellen Kontos und den aktuellen Kontostand, was mit <i>map.putString(.)</i> an das von <i>createParameterTemplateMap(.)</i> vorbereitete <i>TemplateMapBasic</i> hinzugefügt wird. Die Ausgabe erfolgt über <i>return evalOutput(.)</i>, eine NAS Methode.</p>

Listing 5: Das AppLogic bankNewAccountAppLogic (Quelle: Code von Christian Ey, © 98 Netscape)

## Die HTML Template Seiten

Neben der statischen Information, die in einer HTML-Seite enthalten ist, enthalten HTML Template Seiten noch dynamische Information, die von den AppLogics geliefert wird. Die dynamische Information ist in GXML Code geschrieben, so daß der Netscape Application Server diese Stellen gezielt ersetzen kann.

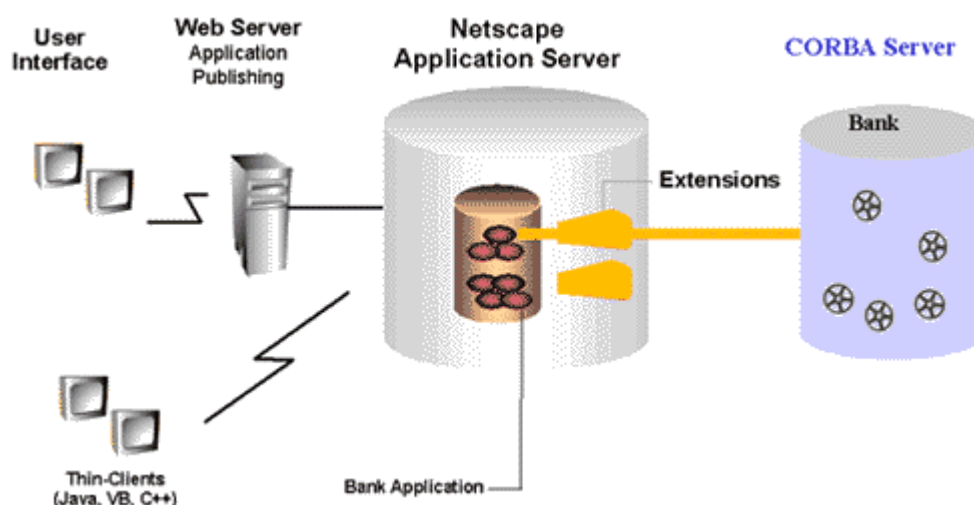
Das von unseren AppLogics verwendete *TemplateMapBasic* Objekt enthält z.B. nach Ablauf des *bankNewAccountAppLogic* folgende Felder:

- **NAME:** enthält den Namen des aktuellen *account* Objektes. Codierung zum Abrufen in der HTML Template Seite:  
`%gx type=cell id=NAME% %/gx%`  
Diese Stelle in der HTML Template Seite wird nun durch den Wert ersetzt, der zuvor im AppLogic durch  
`map.putString("NAME", myAccount.get_name());` gefüllt wurde.
- **BALANCE:** enthält den aktuellen Kontostand. Abruf im Template:  
`%gx type=cell id=BALANCE% %/gx%`
- **HIDDEN:** enthält alle Werte, die als HIDDEN VALUE in die Seite mit aufgenommen werden müssen, fertig als HTML-Code. Der Aufruf ist auch hier:  
`%gx type=cell id=HIDDEN% %/gx%`  
Zu beachten ist hierbei, daß in *jeder* HTML-FORM einer HTML-Seite, die ein AppLogic als FORM ACTION hat, d.h. ein AppLogic aufruft, dieses HIDDEN Feld angegeben werden muß, da ansonsten die notwendigen Objektreferenzen vom nächsten AppLogic nicht mehr angefordert werden können und die Applikation dann mit einer Fehlermeldung abbricht.

Dies schließt die Implementierung der Bank Applikation ohne Verwendung von NAS-Extensions ab. Für nähere Informationen ist das Studium des Source-Codes zu empfehlen.

## **Verbinden zu einem CORBA Server mit NAS Extensions**

Beim Verbinden zu einem CORBA Server mit NAS Extensions fügen wir eine weitere Schicht ein zwischen den AppLogics und dem CORBA Server: Die Extensions. In Grafik 7 sind die Extension als gelbe „Kuchenstückchen“ dargestellt, die den Kontakt zum CORBA Server herstellen.



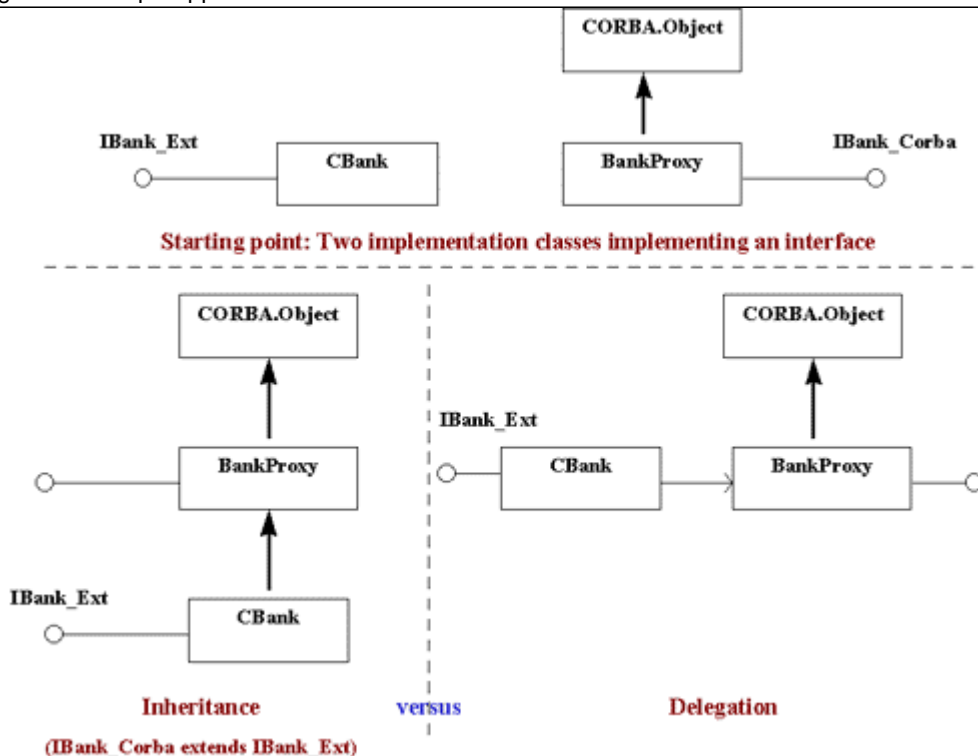
Das bedeutet auch, daß nun die AppLogics den CORBA Server nicht mehr direkt ansprechen, sondern immer über die Extensions gehen. Dies hat verschiedene Vor- und Nachteile.

Als Nachteil sehe ich, daß eine Applikation an Komplexität gewinnt, und die Konstruktion wird aufwendiger. Außerdem gibt es gewisse Probleme mit dem Umsetzen der CORBA-IDL in die Netscape-IDL, die von den Extensions verwendet wird. Hierdurch geht ein Stück der Flexibilität von CORBA verloren. Zum Beispiel ist das Konzept der Exceptions (Ausnahmen) in Netscape-IDL nicht vertreten. Wir haben jedoch speziell dafür ein eigenes Konzept entwickelt, das es ermöglicht, Exceptions trotz der Extensions letztendlich im AppLogic aufzufangen und zu bearbeiten.

Vorteile sind unter anderem, daß der CORBA Server durch Extensions abgekapselt ist. Somit kann ein NAS Programmierer eine Applikation schreiben, die mit einem CORBA Server kommuniziert, ohne selbst CORBA Elemente verwenden zu müssen. Daneben muß der Aufwand, einen CORBA Server zu kapseln, nur einmal betrieben werden und kann danach von allen AppLogics im NAS durch Extensions angesprochen werden.

### ***Design einer CORBA Extension***

Extensions stellen eine weitere Schicht zwischen AppLogics und dem entfernten Programm (in unserem Fall ein CORBA Server) dar, sie leiten also Aufrufe vom AppLogic an das entfernte Programm weiter. Andererseits sind die CORBA-Proxies auf der Client-Seite ursprünglich für diese Aufgabe verantwortlich. Um nun beide Technologien zu verbinden, müssen die Extensions die CORBA Aufrufe von den AppLogics an die CORBA-Proxies weiterleiten. Daher müssen wir die beiden Arten von Klassen auf eine sinnvolle Art und Weise miteinander verbinden. Grafik 8 zeigt verschiedene Möglichkeiten, zwei Objekte miteinander zu verknüpfen.



Grafik 8: Verknüpfen des Extension Objektes CBank mit dem BankProxy (Quelle: Schmauch)

Von den beiden Möglichkeiten Vererbung und Delegation wählten wir die zweite. Um die Extension Objekte zu verwalten, können wir das Object-Pooling von NAS verwenden oder die Managerklasse der Extensions diese Aufgabe übernehmen lassen. Das Object-Pooling eignet sich am Besten zur Verwaltung von wenigen, aufwendigen Objekten, die nur unter hohem Aufwand erzeugt und/oder zerstört werden können, z.B. bei Verbindungen zu Datenbanken.

Wir betrachten CORBA-Proxies jedoch als einfach zu erzeugende und zu löschende, und als in großer Anzahl vorhandene Objekte; daher erscheint uns das Verwenden der Managerklasse als angebracht.

Das funktioniert folgendermaßen: Ein AppLogic bindet sich zu Beginn an die Managerklasse der Extension, in unserem Fall der *BankManager*, kann sich mit ihrer Hilfe an den CORBA Server binden und erhält von dieser Klasse alle benötigten *CAccount* und *CBank* (Extension-)Objekte. Also befinden sich im Bankmanager neben der Methode zum Binden an den Bank Server auch die Verwaltung der Objektreferenzen, die zuvor von den AppLogics selbst in der Session vorgenommen wurde. Hierzu verwendet die Managerklasse entweder das Session Objekt oder nicht, in Abhängigkeit vom Design des Applikation Servers: Existiert nur *ein* Application Server Prozeß, kann auf das Verwenden der Session-Verwaltung verzichtet werden. Sobald es sich um mehrere Application Server Prozesse handelt, verwenden wir die Session in der Extension Managerklasse, um die Daten für alle Prozesse abrufbar zu machen. Das Session Management des NAS sorgt dann für das Verbreiten der Daten unter allen Prozessen.

## **Speicher für die Extensions**

Um die Verwaltung der zu speichernden Daten auch in der Extension zu vereinfachen, haben wir die *ExtensionStorage* Klasse entworfen. Sie übernimmt die Funktion der *CorbaSession*, die bei der Applikation ohne Extensions verwendet wurde. Sie stellt Methoden zum Hinzufügen (add), setzen (set), Bekommen (get) und Löschen (remove) von Werten zur Verfügung. Außerdem ist auch hier die Methode *createMyOwnVariable()* vorhanden, die wir in der Klasse *CorbaExtensionStorage* mit einem Aufruf der Methode *CorbaVariableAndParameter.createVariable()* überschreiben (vergleichbar mit Listing 4).

Die Managerklasse der Extension (*CBankManager*) verweist auf ein *CorbaExtensionStorage* Objekt. Da die AppLogics das Speicherobjekt durch die Managerklasse manipulieren, besitzt diese zusätzlich die vier Methoden *addObject*, *setObject*, *getObject* und *removeObject*. Diese Methoden unterscheiden sich von den Methoden der Speicherklasse *addValue*, *setValue*, *getValue* und *removeValue*, da sie einen zusätzlichen Parameter `boolean useSession` nehmen, der bestimmt, ob der *CBankManager* die benötigten UID/Objektreferenz Paare im Speicherobjekt der Extension oder in der Session speichert. Dies sollte davon abhängen, ob mehrere Application Server Prozesse verwendet werden (Session) oder nicht (Speicherobjekt der Extension). Diese Unterscheidung ist in „Design einer CORBA Extension“ erklärt.

Im Fall der Verwendung des Session Objektes zur Speicherung der Objektreferenz, muß diese in eine Zeichenkette umgewandelt werden, wie schon in „Speicherung in der Session“ erläutert. Dies wird in der Methode *externalize()* der Extensionklassen vorgenommen. Das Gegenstück hierzu ist die Methode *internalize(.)*, die den Zustand eines Objektes anhand eines Strings wiederherstellt. Da jede Extensionklasse außer der Managerklasse diese Funktionalität zur Verfügung stellen muß, werden diese beiden Methoden in der Klasse *CExtCorbaDelegateObject* definiert, welche *externalize()* schon implementieren kann. *internalize(.)* muß von der Unterklasse implementiert werden. Listing 6 zeigt die Definition der *externalize()* und *internalize(.)* Methoden der *CExtCorbaDelegateObject* Klasse.

Listing	Erklärung
<pre> public java.lang.String externalize(){     String result;     try {         result = CORBA.Orbix.object_to_string(proxy_object);     } catch (SystemException e) {         result = null;     }     return result; } </pre>	<p>das CORBA Proxy Objekt, dessen Verweis ein Attribut der <i>CExtCorbaDelegateObject</i> Klasse ist, wird zur Zeichenkette umgewandelt und so zurückgegeben.</p>
<pre> public abstract int internalize(     java.lang.String state,     bankCorbaExtDelegate.IBankManager mModule); </pre>	<p>diese Methode ist abstrakt und muß von der Unterklasse implementiert werden.</p>

Listing 6: Die CExtCorbaDelegateObject Klasse (Code von Christian Ey, ©'98 Netscape)

Um ein Extension Objekt wiederherstellen („internalize“) zu können, ist neben der Objektreferenz noch der Name der betreffenden Extensionklasse notwendig. Beim Wiederherstellen muß zuerst das korrekte Extension Objekt hergestellt werden und anschließend mit *internalize(.)* dessen Zustand (in unserem Fall betrifft dies nur die CORBA Objektreferenz).

All dies bewerkstelligt die Klasse *CorbaExtensionSession*, die die Klasse *CorbaSession* beerbt. Hierzu werden die Methoden *convert(.)* und *reconvert(.)* überschrieben. Listing 7 zeigt, wie wir es implementieren.

Listing	Erklärung
<pre> public GXVAL convert( Object anObject) throws SystemException {     // this turns a string into a GXVAL object     IValList temp = GX.CreateValList();     temp.setValString( "temp",         anObject.getClass().getName()         + "!" + ((CExtCorbaDelegateObject)anObject).externalize());     ;     return temp.getVal( "temp"); } </pre>	<p>Das zurückgegebene GXVAL Objekt enthält einen String, der den Namen der Extensionklasse, ein trennendes „!“ und die zur Zeichenkette umgewandelte CORBA Objektreferenz enthält. Das Format ist: <i>extensionKlasseName!Objektreferenz.</i></p>
<pre> public Object reconvert( GXVAL aValue) throws SystemException {     // this turns a GXVAL object into a string     IValList temp = GX.CreateValList();     CExtCorbaDelegateObject anObject;     temp.setVal( "temp", aValue);     String theString = temp.getValString( "temp");     int theInt = theString.indexOf('!');     try {         anObject = (CExtCorbaDelegateObject)Class.forName( theString.substring(0, </pre>	<p>Zuerst wird der String aus dem GXVAL Objekt extrahiert. Dann wird mit <i>Class.forName(.)</i> ein Extension Objekt erzeugt von der Klasse, die im String bis zum Zeichen „!“ niedergeschrieben ist. Wenn dies keine Exception verursachte, wird an dem neuen Extension Objekt die Methode <i>internalize(.)</i> mit der CORBA Objektreferenz aufgerufen.</p>

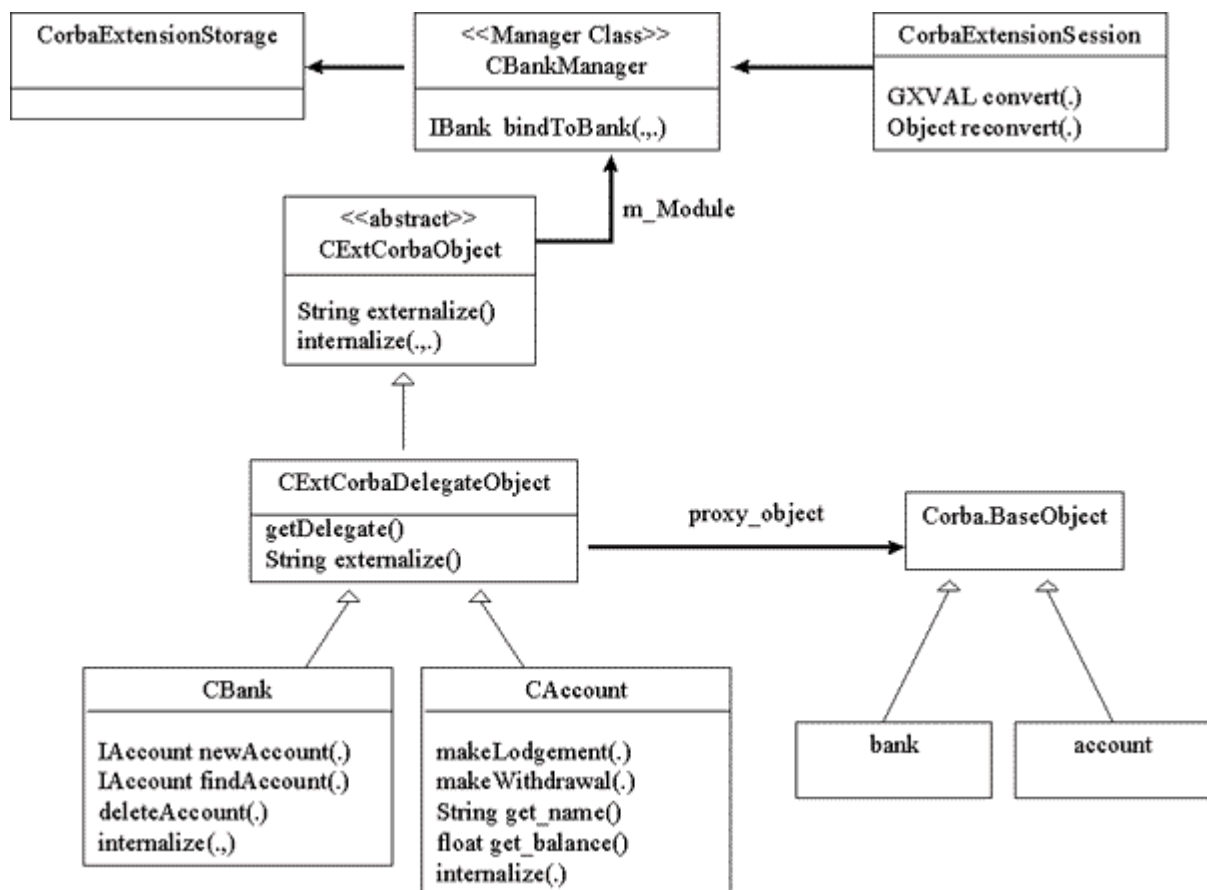
```

        theInt).newInstance();
    } catch (ClassNotFoundException e) {
        return null;
    } catch (InstantiationException e) {
        return null;
    } catch (IllegalAccessException e) {
        return null;
    }
    }
    anObject.internalize(theString.substring(
        theInt+1), m_Module);
    return anObject;
}
    
```

Listing 7: Die CorbaExtensionSession Klasse (Code von Christian Ey, ©'98 Netscape)

## Das Objektmodell der Bank Extension

In Grafik 9 sieht man das Objektmodell der Bank Extension.



Grafik 9: Objektmodell der Bank Extension (Quelle: Präsentation von Prof. Dr. Schmauch)

- *CExtCorbaObject* ist eine abstrakte Klasse, die sowohl bei der Methode der Delegation als auch bei der Vererbung als Basisklasse dienen kann. Es enthält einen Verweis auf das Manager Objekt (in *m\_Module*), da dies für jede Extension Klasse erforderlich ist.
- *CExtCorbaDelegateObject* ist eine abstrakte Klasse, die die Delegation der Aufrufe an das CORBA Proxy Objekt und die *externalize()* Methode implementiert.

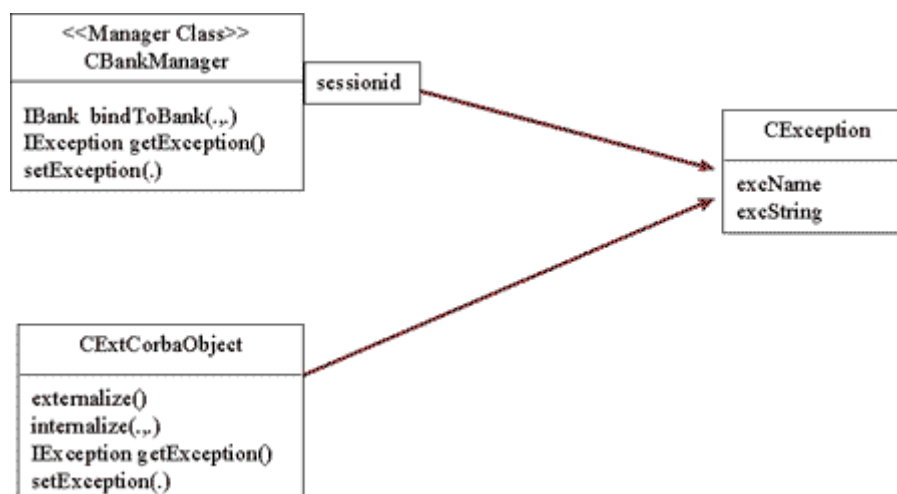
*externalize()* wird benötigt, da ein Extension Objekt nach Ablauf eines AppLogic ggf. in der Session gespeichert und hierzu in eine Zeichenkette umgewandelt wird.

- *CBank* und *CAccount* sind Unterklassen von *CExtCorbaDelegateObjekt* und implementieren die Methoden des CORBA Proxy Objekts durch Delegation, z.B. *newAccount(.)* von *CBank* ruft *newAccount* am *account* Proxy auf. Außer dem wird hier die *internalize(.)* Methode implementiert, die ein Extension Objekt aus einer Zeichenkette (z.B. aus der Session kommend) wiederherstellt.
- *bank* und *account* sind die CORBA Proxies, die von *Corba.BaseObjekt* abgeleitet sind.
- Die Konstruktion um *CBankManager* ist für das Funktionieren des Session Speichers erforderlich: *CBankManager* verweist auf ein *CorbaExtensionStorage* Objekt und das *CorbaExtensionSession* Objekt verweist auf den *CBankManager*.

## Java Exceptions in NAS Extensions?

Die Netscape IDL, in der Netscape Application Server Extensions definiert sind, verfügt, neben anderen Unterschieden zu CORBA IDL, nicht über das Konzept der Exceptions. Aufrufe von Methoden an entfernten CORBA Objekten können jedoch Exceptions werfen; eine oft auftretende Ausnahme ist z.B. die CORBA *SystemException*. Um nun Ausnahmen in den AppLogics behandeln zu können, haben wir ein Konzept entwickelt: Eine Ausnahme wird in der Extension aufgefangen und ein sogenanntes *CException* Objekt erzeugt, das den Namen der Ausnahme und ihren Fehlertext enthält. Ein AppLogic kann nach jedem Aufruf einer Methode an einem Extensionobjekt nachprüfen, ob während des Ablaufs der Methode eine Ausnahme aufgetreten ist. Wenn dies der Fall war, startet das AppLogic die Fehlerbehandlung.

Jedes Extensionobjekt verweist daher auf ein *CException* Objekt, die Managerklasse muß auf mehrere *CException* Objekte verweisen, die durch die ID der Session gekennzeichnet sind. Das hierdurch entstehende Objektmodell ist in Grafik 10 dargestellt.



Grafik 10: Objektmodell für die Exceptionbehandlung (Quelle: Prof. Dr. Schmauch)

## Implementierung der Bank Extension

Listing 8 zeigt und erklärt die wichtigsten Elemente der Extensionklasse *CBank*, die an die CORBA Klasse *bank* delegiert.

Listing	Erklärung
<pre> package bankCorbaExtDelegate.cBank; import [...] public class CBank   extends CExtCorbaDelegateObject   implements bankCorbaExtDelegate.IBank {   [...]   public IAccount newAccount(String name) {     _accountRef myAccount;     if (name != null) {       this.setException(null);       try {         // call the remote method on the proxy object         myAccount =           ((bankRef)proxy_object).newAccount(name);       } catch (SystemException e) {         this.setException(new CException(           m_Module,           "CorbaConnectionException",           e.toString()));         return null;       } catch (reject r) {         this.setException(new CException(           m_Module, "reject", null));         return null;       }       // create and return the extension object linked to       // the new proxy object       return new CAccount(m_Module,myAccount);     }     else return null;   } </pre>	<p>die Methode <i>newAccount(.)</i> delegiert den Aufruf an den <i>bank</i> Proxy <i>proxy_object</i>, und erhält ein <i>account</i> Proxy zurück. Dann erzeugt sie ein neues <i>CAccount</i> Objekt und gibt das <i>account</i> Proxy an den Konstruktor des <i>CAccount</i> Objektes weiter.</p> <p>Falls eine Ausnahme auftreten sollte, wird mit der Methode <i>this.setException(.)</i> eine Exception gesetzt, die später von einem AppLogic abgefragt werden kann.</p>
<pre> public void internalize(String state,   CBankManager mModule) {   m_Module = mModule;   if (state != null) {     try {       proxy_object = bank._narrow(         _CORBA.Orbix.string_to_object( state));     } catch (SystemException e) {       this.setException(new CException(m_Module,         "CorbaConnectionException",         e.toString()));     }     return;   }   else return; } </pre>	<p>das Proxy Objekt wird erzeugt, indem die übergebene Zeichenkette <i>state</i> in ein CORBA Proxy Objekt umgewandelt wird.</p>

## Verwendung der Bank Extension

Die AppLogics, die die Bank Extension benutzen, ähneln den AppLogics, die den CORBA Server direkt ansprechen. Die Geschäftslogik ändert sich nicht. Das erste AppLogic, *bankBindAppLogic*, ruft die Methode *bindToBank()* auf an der Managerklasse der Extension, *IBankManager*. Die Managerklasse gibt ein *IBank* Objekt zurück, wenn das Binden an den CORBA Server erfolgreich war. Wenn sie keinen Erfolg hatte, speichert die Managerklasse die Ausnahme in einem *IOException* Objekt in sich selbst. Das AppLogic kann nun nachprüfen, ob ein *IOException* Objekt vorhanden ist oder nicht und ggf. eine Fehlerbehandlung einzuleiten. Unser Verfahren zum Abfangen von Ausnahmen an Extensions hat im AppLogic eine ähnliche Struktur wie try-catch Blöcke in Java. In Listing 9 wird dieses Verfahren am Beispiel des *bankBindAppLogic* erklärt. Im AppLogic, das Extensions benutzt, sind alle CORBA spezifischen Elemente verschwunden, und alles wurde durch Aufrufe an Extension Objekten ersetzt.

Listing	Erklärung
<pre>package bankCorbaAppDelegate1; import [...] import bankCorbaExtDelegate.*;  public class bankBindAppLogic   extends CorbaAppLogic {   public int execute() {     String sUId;     IBank myBank = null;     IBankManager theBankManager;     boolean useSession = true;     IOException E;     boolean exceptionThrown;     // call getSessionObject() to create a     // CorbaExtensionSession     // for the current application     this.getSessionObject();     [...]</pre>	<p>das AppLogic importiert <i>bankCorbaExtDelegate</i>, erweitert <i>CorbaAppLogic</i>.</p> <p>Es werden je ein <i>IBank</i>, <i>IBankManager</i> und ein <i>IOException</i> Objekt deklariert.</p> <p><i>useSession</i> legt fest, daß in der Extension das Session Objekt zum Speichern verwendet wird.</p>
<pre>// bind to the manager object theBankManager = access_cBank.getcBank(   context, null, this); if (theBankManager == null)   return result( [...]("No extension access") );</pre>	<p>das AppLogic greift auf die Extension zu und bekommt deren Managerklasse.</p>
<pre>// business logic: bind to the bank exceptionThrown = false; tryblock: {   myBank = theBankManager.bindToBank(     bankserver, hostname);   E = theBankManager.getException();   if (E != null) {     exceptionThrown = true;     theBankManager.setException(null);     break tryblock;   } }</pre>	<p>hier die Geschäftslogik und der simulierte „try-Block“. Wenn bei <i>bindToBank(.)</i> eine Ausnahme aufgetreten ist, ist das <i>IOException</i> Objekt <i>E</i> nicht NULL und die bool'sche Variable <i>exceptionThrown</i> wird auf TRUE gesetzt. Anschließend wird das <i>IOException</i> Objekt an der Managerklasse gleich wieder auf NULL gesetzt. <i>break tryblock</i> wird sinnvoll, wenn mehrere Programmierschritte im <i>tryblock</i>:</p>

	vorhanden sind.
<pre> <b>if (exceptionThrown) {     if (E.getExceptionName().equals(         "CorbaConnectionException"))         return result(...)("Fehlermeldung");     } </b></pre>	hier das simulierte „catch“. Wenn <i>exceptionThrown</i> TRUE ist, wird eine Fehlerbehandlung eingeleitet.
<pre> [...] exceptionThrown = false; tryBlock: {     sUid = theBankManager.addObject(         myBank, useSession);     E = theBankManager.getException();     if (E != null) {         exceptionThrown = true;         theBankManager.setException(null);         break tryBlock;     } } </pre>	Speichern der Objektreferenz, Abfangen der möglichen Ausnahme.
<pre> <b>if (exceptionThrown) {     if (E.getExceptionName().equals(         "CorbaConnectionException"))         return result(...)("couldn't store the bank");     } [...]</b></pre>	Einleiten der Fehlerbehandlung

Listing 9: Auszüge aus dem bankBindAppLogic (Code: Christian Ey ©'98 Netscape)

Das Verbinden zu einem CORBA Server mit NAS Extensions ist hiermit abgeschlossen.

Autor:  
Christian Ey  
Studiengang Wirtschaftsinformatik  
Fachhochschule Karlsruhe  
Email: ey@inweb.de  
Homepage: <http://www.inweb.de/chetan/>

Vorlagen und Hilfen zu diesem Bericht:

### Integrating NAS, Netscape Extensions and CORBA

URL: <http://developer.netscape.com/tech/corba/index.html?content=/docs/manuals/appserv/cookbook/NASandCORBA.html>

Autorin: Prof. Dr. Cosima Schmauch

### Writing AppLogics Connecting to a CORBA Server

URL: <http://developer.netscape.com/tech/corba/index.html?content=/docs/manuals/appserv/cookbook/AppCorbaCookbook.html>

Autorin: Prof. Dr. Cosima Schmauch

### Writing Extensions Connecting to a CORBA Server

URL: <http://developer.netscape.com/tech/corba/index.html?content=/docs/manuals/appserv/cookbook/ExtCorbaCookbook.html>

Autorin: Prof. Dr. Cosima Schmauch