G.5 Object-Adaptor

- Aufgaben des Objektadapters
  - ◆ Generierung der Objektreferenzen
    - Objektreferenzen identifizieren CORBA-Objekte und enthalten die Adressierungsinformationen die ein Client im verteilten System benötigt, um Operationen an dem Objekt aufzurufen.
      - – Beispiel für eine Interoperable Object reference (IOR), die das Internet Inter-ORB-Protokoll unterstützt

`iiop:1.0/`	`/faui04a.cs.fau.de:10015`	`/P353bccdb00094ae8`	`/firstPOA`	`/myservant`
Protocol Id	Communication Endpoint (Rechnername und Portnummer)	Zeitstempel	Object Adapter ID	Object Id

  - ◆ Entgegennahme eingehender Methodenaufruf-Anfragen
  - ◆ Weiterleitung von Methodenaufrufen an den entsprechenden Servant
  - ◆ Authentisierung des Aufrufers (Sicherheitsfunktionalität)
  - ◆ Aktivierung und Deaktivierung von Servants
    - Ein Objekt ist eventuell zwar für das CORBA System präsent, aber noch nicht aktiv, so dass es direkt aufgerufen werden kann. Vor einem Aufruf wird der Object Adaptor dann das Objekt aktivieren.
  - ◆ Registriert Serverklassen im Implementation Repository
- Obligatorischer Adapter: Portable-Object-Adaptor
  - ◆ definierte Funktionalität für die häufigsten Aufgaben
- weiteres Beispiel: OODB-Adaptor
  - ◆ Anbindung einer objektorientierten Datenbank an CORBA
  - ◆ OODB-Adaptor repräsentiert alle Objekte in der Datenbank als CORBA-Objekte und vermittelt Aufrufe

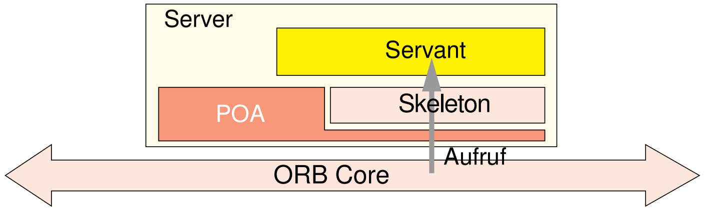
1 Portable-Object-Adaptor (POA): Ziele

- Portabilität von Objektimplementierungen zwischen verschiedenen ORBs
- Trennung von Objekt (CORBA-Objekt mit seiner Identität) und Objektimplementierung
  - ➤ eine Objektimplementierung kann mehrere CORBA-Objekte realisieren
  - ➤ Objektimplementierung kann bei Bedarf dynamisch aktiviert werden
  - ➤ persistente CORBA-Objekte (Objekte, die Laufzeit eines Servers überdauern)

  ➥ eine Object Reference beim Client steht für ein bestimmtes CORBA-Objekt — nicht unbedingt für einen bestimmten Servant!
- Mehrere POA-Instanzen (mit unterschiedlichen Strategien) innerhalb eines Servers möglich

2 Portable-Object-Adaptor (POA): Überblick

- Jeder Servant kennt seine POA-Instanz
  - ◆ POA-Instanz kennt die an ihm angemeldeten Objekte
  - ◆ POA stellt Kommunikationsplattform bereit und nimmt Aufrufe entgegen (für seine Objekte)



- Verhältnis ORB Core - Server - POA: Alternativen
  - ➤ ORB Core ist Teil des Server-Prozesses (Bibliothek), Kommunikation zu POA erfolgt lokal   oder
  - ➤ ORB-Serverprozess und lokale IPC zu den Server-Prozessen mit deren POAs

3 POA-Erzeugung

- Root-POA existiert in jedem ORB
  - ◆ voreingestellte Konfiguration
  - ◆ kann zur Aktivierung von Objekten verwandt werden
- Referenz auf Root-POA
  - ◆ Anfrage am ORB mit: `orb.get_initial_reference( "RootPOA" );`
- Weitere POAs mit unterschiedlichen Konfigurationen erzeugbar
  - ◆ Erzeugung am RootPOA bzw. an untergeordnetem POA (Vater-POA)
  - ◆ neuer POA bekommt eigenen Namen (String)
  - ◆ baumförmige, hierarchische POA-Struktur mit Wurzel beim Root-POA
  - ◆ POA-Instanzen teilen sich Kommunikationskanal
    - ➤ Objektreferenz enthält POA-Name: POA kann ermittelt werden

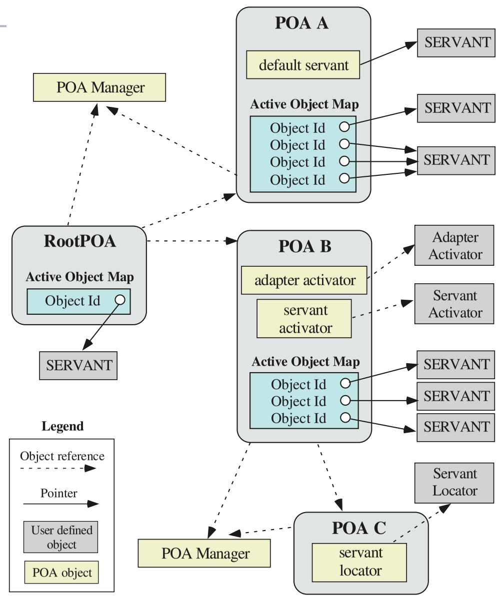
4 Erzeugung von CORBA-Referenzen

- Normalfall
  - Servant existiert und wird dem POA mit **`activate_object(...)`** gemeldet
  - Referenz wird dann mit `servant_to_reference(...)`-Aufruf am POA erzeugt
- Implizite Aktivierung
  - POA erzeugt automatisch eine *Object Reference* wenn ein (nicht aktivierter) Servant als Parameter oder Ergebnis "nach aussen" übergeben wird
- Explizite Erzeugung einer Referenz
  - *Object Reference* wird erzeugt, ohne dass ein Servant existiert
    - ➤ damit entsteht ein abstraktes CORBA-Objekt, zunächst ohne Implementierung
  - POA hat verschiedene Möglichkeiten, ankommende Aufruf zu bearbeiten:
    - ➤ "Default Servant" bearbeitet den Aufruf
      - z. B. wenn Datenbank-Einträge als CORBA-Objekte exportiert werden
    - ➤ Servant wird bei Bedarf dynamisch erzeugt
  - sie Abschnitt POA-Policies

5 Deaktivierung von Objekten, POA und Server

- Ressourcen-Problem
  - ➤ sehr viele CORBA-Objekte "in" einem Server
  - ➤ mehrere POAs in einem Server
  - ➤ Objekte oder auch POAs werden evtl. nur selten genutzt
  - ➤ Server-Prozess wird ggf. nur selten benötigt
- CORBA-Objekte können ihren Servant und ihre POA-Instanz überleben
  - ➤ persistente Objekte
  - Servants können deaktiviert werden
  - POA kann deaktiviert werden
  - Serverprozess kann beendet werden

6 POA-Architektur



- Implementation Repository
  - ➤ Basis für persistente Referenzen
  - ➤ kennt alle Server und zumindest die Root-POAs
- POA Manager
  - ➤ steuert POA
    - Anwendungen können einstellen, ob Aufrufe bearbeitet, zurückgehalten oder weggeworfen werden.
    - POA kann über POA Manager deaktiviert werden.
- Adapter Activator
  - ORB ruft Adapter Activator auf, wenn ein Aufruf für einen Kind-POA vorliegt, der noch nicht existiert
  - Adapter Activator kann entscheiden, ob er POA dann aktiviert
- Servant Manager
  - ORB benutzt Servant Manager, um Servants bei Bedarf zu aktivieren - oder zu deaktivieren.
  - verwaltet Beziehung zwischen Object Id (CORBA Reference) und Servant
  - entscheidet, ob in Objekt existiert
  - zwei Typen: `ServantActivator` und `ServantLocator` — welcher Typ genutzt wird ist von der POA Policy abhängig
- POA Policies
  - siehe eigener Abschnitt

7 Bearbeitung von ankommenden Aufrufen

- Ankommende Aufrufe enthalten *Object Key*
  (= Time Stamp + POA Id + Servant Id)

1. Server-Prozess finden

◆ ORB lokalisiert zuständigen Server-Prozess

- das kann automatisch passieren, indem der ORB-Code zu dem Server-Programm dazugebunden ist und der Server-Prozess Verbindungen auf dem Port, der in den von seinen POAs ausgegebenen Objektreferenzen angegeben ist
  - – Standardsituation bei transienten Objektreferenzen (die nur so lange gültig sind, wie der zugehörige Server aktiv ist)
- alternativ könnte ein generischer ORB Verbindungen auf allen Ports, die in Objektreferenzen ausgegeben wurden, entgegennehmen und dann die Aufrufe an den zuständigen Server-Prozess mittels lokaler IPC weiterleiten.

◆ falls Server deaktiviert: Aufrufe landen beim Implementation-Repository

- funktioniert nur für persistente Objektreferenzen
  - – persistente Objektreferenzen enthalten die Adresse:Portnummer des Implementation Repositories
  - ➤ Implementation-Repository hält dauerhafte Information über das CORBA-Objekt, insbesondere wie dieses wieder aktiviert werden kann
  - ➤ z. B. Programmdatei für Neustart
  - ➤ Starten eines neuen Servers und Aktivieren des Root-POAs
    - – Implementation Repository enthält Informationen zum Starten eines Serves (z. B. Kommandoname), den Namen der POAs in diesem Server (zumindest den Root-POA) und die Netzwerkadresse unter der der Server dann erreichbar ist
  - ➤ Aufruf wird an neue Kommunikationsadresse weitergeleitet *(Location Forward)*
    - – Implementation Repository sendet ein Location Forward Reply an den Aufrufer zurück, der Client wiederholt den Aufruf mit der neuen Adresse

7 Bearbeitung von ankommenden Aufrufen (2)

- Aktivierungsmechanismus
  - ➤ Speicherung von Daten und Protokoll zwischen Implementation-Repository und POA ist herstellerspezifisch
  - ➤ JDK 1.4: **`orbd`**-Prozess
  - ➤ Unterstützung im RPC-Protokoll
  - ➤ Location-Forward: Stubs werten Weiterleitung aus und kontaktieren Objekt an angegebener Kommunikationsadresse solange möglich

7 Bearbeitung von ankommenden Aufrufen (*2*)

2. POA finden
   - ORB lokalisiert zuständigen POA
     - Object Key enthält POA Id, Kommunikation zwischen ORB und POA ist implementierungsabhängig (normaler Funktionsaufruf falls ORB zum Server-Prozess gebunden ist)
   - falls POA deaktiviert: ORB ruft *Adapter Activator* bei "Vater-POA" auf
     - aus POA Id (Pfad-Struktur) kann der Name des Vater-POAs abgeleitet werden
     - am zugehörigen Adapter Activator wird die Methode `unknown_adapter` aufgerufen und der Name des fehlenden POA übergeben
3. Servant finden
   - Aufruf ist jetzt im richtigen Server angekommen oder ORB weiß zumindest wie er mit dem POA interagieren kann
   - ORB leitet Aufruf an zuständigen POA weiter
   - POA findet Servant entsprechend seiner *Servant Retention*- und *Request Processing*-Strategie
     - ➤ ggf. Servant neu instanziieren
     - ➤ sie Abschnitt POA-Policies
4. Skeleton finden
   - im letzten Schritt gibt der POA den Aufruf an das IDL Skeleton weiter
   - Skeleton entpackt Parameter und ruft die eigentliche Servant-Operation auf

8 POA Policies

- Konfiguration eines POA bei Erzeugung durch Policy-Objekt
  - ➤ nicht bei Root-POA!
  - ◆ bei Erzeugung eines POA werden vorab erzeugte Policy-Objekte als Konfiguration übergeben
- Threading policy
  - ◆ Aufrufbearbeitung multi-threaded oder single-threaded
- Lifespan policy
  - ◆ gibt an, ob CORBA-Objekte, die von POA erzeugt werden, transient oder persistent sind
    - ➤ **`TRANSIENT`** für Objekte, die Servant nicht überleben
    - ➤ **`PERSISTENT`** für Objekte, die POA und Servant überleben können
- ObjectId uniqueness policy
  - ◆ eindeutige IDs über alle Zeit oder nicht

8 POA Policies (2)

- ObjectId Assignment policy
  - ◆ system- oder benutzervorgegebene Objekt-IDs
- Implicit activation policy
  - ◆ falls implizite Aktivierung von Servants und Erzeugung von CORBA-Referenzen unterstützt werden soll
- Servant retention policy
  - ◆ Tabelle aktiver Objekte ordnet Servants den CORBA-Referenzen zu
  - ◆ Alternativ: bei ankommendem Aufruf wird jedesmal ein Servant dem angeforderten CORBA-Objekt zugeordnet
- Request processing policy
  - ◆ nur aktivierte Objekte sind bekannt
  - ◆ ein Default-Servant kann für alle nicht aktivierten Objekte einspringen
  - ◆ Servant-Manager kann für unbekannte Objekte einen Servant generieren

G.6 Dynamic-Invocation-Interface (DII)

1 Überblick

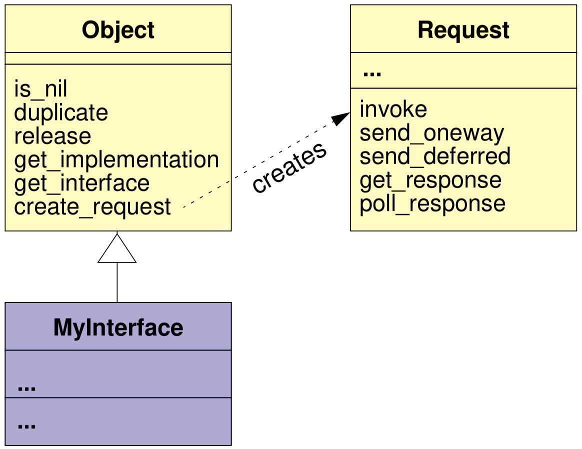
- Typ eines Objekts zur Entwicklungszeit einer Anwendung nicht bekannt
  - z. B. Name-Server-Implementierung, Verzeichnisdienst etc.
  - Wie aufrufen?
- Abfrage der Objektschnittstelle
  - Ermittlung erfolgt über das Interface Repository
  - Methode **`get_interface()`** bei jedem CORBA-Objekt
    - ➤ liefert Referenz auf CORBA-Objekte, die Schnittstelle eines Objekts beschreiben
    - ➤ Methoden, Parameter und deren Typen können erfragt werden

1 Überblick (2)

- Dynamische Konstruktion der Aufrufparameter
  - generische Aufrufschnittstelle **`create_request()`** bei jedem CORBA-Objekt
    - ➤ Benennung der Methode als String
    - ➤ Übergabe der Parameter eingepackt in Typ **`any`**
- Einzelne Schritte des Aufrufs (im Language Binding abgebildet)
  - ermittle Methodensignatur aus dem Repository
  - erstelle die Parameterliste
  - erstelle die Aufrufbeschreibung (Request)
  - führe Methodenaufruf aus
    - – als RPC
      - Es wird der Aufruf generiert und auf das Ergebnis gewartet.
    - – asynchroner RPC
      - Es wird der Aufruf generiert. Der Aufrufer kann weiterarbeiten und sich das Ergebnis irgendwann später abholen.
    - – über Datagramm-Kommunikation (ohne Antwort)
      - Diese Aufrufart funktioniert nur bei Methoden, die keine Parameter zurückerwarten. Der Aufruf wird generiert; auf ein Ergebnis bzw. den Abschluß der Methodenausführung kann nicht gewartet werden. Hinzu kommt, dass der Aufruf eventuell über einen Datagramm-dienst versandt wird. Das bedeutet, daß nicht sichergestellt ist, ob der Aufruf überhaupt durchgeführt wird.

1 Überblick (3)

- IDL-Definitionen für das DII



- Mit Hilfe der Methode create_request wird ein Requestobjekt erzeugt. Dabei werden diesem die bereits gesammelten Aufrufparameter übergeben.
- An dem Requestobjekt können dann mit verschiedenen Methoden die entsprechenden Aufrufe am eigentlichen Objekt ausgeführt werden:

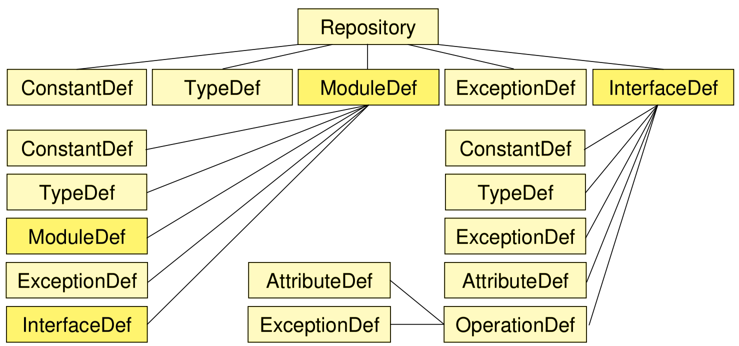
invoke	Aufruf als RPC
send_oneway	Aufruf als Datagramm ohne Antwort
send_deferred	asynchroner Aufruf
get_response	Warten auf Antwort
poll_response	Prüfen, ob Antwort bereits vorliegt

2 Interface Repository

- Datenbank für Schnittstellendefinitionen in IDL
  - Alle IDL Schnittstellen werden dort abgelegt.
- Abfrage der Datenbank
  - ◆ für Typechecking
    - Der ORB kann prüfen, ob die Schnittstelle des aufgerufenen Objekts mit der Schnittstelle übereinstimmt, die der Client-Stub des Objekts repräsentiert.
    - ➤ bei Inter-ORB-Operationen: Hinterlegung in mehreren Repositories
  - ◆ zum Abruf von Metadaten für Clients und Tools: dynamische Aufrufe, Debugging, Klassenbrowser
    - Beispielsweise für dynamische Aufrufe, d.h. für solche, bei denen der Typ des Objekts zur Compilezeit noch nicht feststand, kann mittels Interface Repository der genaue Typ der Schnittstelle, der Operationen und Parameter zur Laufzeit ermittelt werden.
  - ◆ Implementierung der Methode *get_interface* eines jeden Objekts
    - Jedes Objekt hat eine Methode get_interface mit dem sich ein Verweis auf die dazugehörige Schnittstellenbeschreibung ermitteln läßt. Diese Beschreibung kommt dann aus dem Interface Repository.
- Schreiben der Datenbank
  - ◆ durch IDL Compiler
    - Bei der Generierung von Stubs und Language Binding werden auch Informationen zum Laden des Interface Repositories erzeugt.
  - ◆ durch Schreibmethoden
    - Das Interface Repository besitzt auch Schreibmethoden, mit denen explizit Einträge in der Interface Datenbank vorgenommen werden können.

2 Interface Repository (2)

- IDL-Konstrukte werden jeweils als CORBA-Objekt realisiert
- Zusammenhänge zwischen den Repository-Objekttypen:



◆ Komponenten einer IDL-Datei werden als Objekte, die in IDL spezifiziert sind abgelegt

◆ Hierarchie der Komponenten bzw. des Interface Repositories
  - Die Grafik gibt den strukturellen Aufbau des Repositories wieder. Die Kästchen entsprechen Klassen bzw. Objekten. Die Verbindungslinien sind Assoziationen.
  - Ein Module kann beispielsweise Definitionen für Konstanten, Typen, Exceptions, Interfaces und wiederum weiterer Module enthalten.
  - Die Struktur entspricht damit dem generellen Aufbau einer IDL-Beschreibung.

2 Interface-Repository (3)

- Repräsentation von Typen durch TypeCode-Objekte

  ◆ Repräsentation der Basistypen:
  `int, float, boolean`

  ◆ Repräsentation zusammengesetzter Typen:
  `union, struct, enum, sequence, string, array`

  ◆ Repräsentation von IDL-basiertenObjekttypen: Interface-Repository-ID

- TypeCode-Objekte repräsentieren Typ und Struktur durch IDL-Typen

  ◆ TypeCode-Objekte haben Operation für Typvergleich
    - Typecodes werden zur Typprüfung eingesetzt und ergänzen damit die beschreibenden Objekte aus dem Interface Repository. Alle Typen, die sich durch IDL beschreiben lassen werden durch ein entsprechendes Objekt aus dem Interface-Repository selbst repräsentiert, alle Standardtypen durch ein geeignetes Typecode-Objekt.

  ◆ TypeCode-Objekte haben Operationen zur Strukturerkundung (z. B. welchen Elementtyp hat eine sequence)

3 Aufrufarten

- Normaler Aufruf
  - Aufruf von **`invoke`** am Request-Objekt
    - ➤ kehrt zum Aufrufer zurück, wenn eigentlicher Aufruf durchgeführt
- Asynchroner Aufruf
  - Aufruf von **`send_deferred`** am Request-Objekt
    - ➤ eigentlicher Aufruf wird angestoßen
    - ➤ Aufrufer kann weiterarbeiten
    - ➤ Ergebnissynchronisation durch **`get_response`**
    - ➤ Ergebnisabfrage (Polling) mit **`poll_response`**

3 Aufrufarten (2)

- Datagramm-Aufruf
  - Aufruf von **`send_oneway`** am Request-Objekt
    - ➤ Aufruf wird angestoßen
    - ➤ es wird nicht auf Ergebnis gewartet (Methode darf kein Ergebnis liefern)
    - ➤ Aufruf muss nicht sicher durchgeführt werden („May-be-Once-Semantik“)

  - **`oneway`**-Methoden werden mit dem Schlüsselwort **`oneway`** in IDL markiert

4 Dynamic Skeleton Interface (DSI)

- DSI ermöglicht das Entgegennehmen von Methodenaufrufen für Objekte, deren Schnittstelle nur dynamisch ermittelbar ist, z.B. für
  - ◆ Bridges zu anderen ORBs
    - Bridges stellen eine Verbindung zu anderen ORBs her. Sie haben ein „Bein“ auf jeder Seite und bilden eine „Brücke“.
  - ◆ CORBA-gekapselte Datenbank
  - ◆ dynamisch erzeugte Objekte und Schnittstellen
- Anhand der Parameter wird das Objekt und dessen Signatur ermittelt
  - In der Implementierung wird der Aufruf an eine registrierte Callback-Funktion zugestellt, die dann das entsprechende Objekt aufrufen muß.

★ **Beachte:**
DII und DSI sind jeweils von der Gegenseite nicht von statischen Stubs zu unterscheiden, d.h. voll interoperabel!

G.7 Inter-ORB-Kommunikation

- Innerhalb einer ORB-Instanz können Objekte mit herstellerspezifischem Protokoll kommunizieren
- Zwischen ORBs verschiedener Herstellern
  - ◆ GIOP – General Inter-ORB Protocol (standardisiert in CORBA)
    - ➤ Basisprotokoll für RPC-basierte Objektaufrufe
    - ➤ Common Data Representation (CDR) definiert Marshalling und Unmarshalling von IDL-Typen
  - ◆ IIOP – Internet Inter-ORB Protocol
    - ➤ GIOP über TCP/IP
    - ➤ Unterstützung durch den ORB für CORBA-Kompatibilität vorgeschrieben
  - ◆ GIOP-Implementierungen über andere Protokolle möglich

G.7 Inter-ORB-Kommunikation (2)

- ESIOP – Environment Specific Inter-ORB Protocols
  - z.B. DCE/ESIOP
  - Inter-ORB-Protokoll auf der Basis von DCE RPC

- IOR – Interoperable Object Reference
  - Darstellung einer Objektreferenz als IDL-Datenstruktur
  - muss verwendet werden für GIOP
  - Repräsentation als String möglich (**`object_to_string`**)
  - Profile in der IOR
    - für jedes Protokoll eigenes Profil möglich
    - Objekt kann theoretisch mehrere Protokolle unterstützen
      z.B. IIOP und herstellerspezifisches Protokoll

• Eine Objektreferenz besteht eventuell aus verschiedenen Profiles, d.h. aus verschiedenen gleichwertigen Beschreibungen einer Objektreferenz. Diese Beschreibungen sind jedoch meist für verschiedene Protokolle. Eine CORBA-Implementierung kann so beispielsweise ein lokales Profile benutzen, in dem eine proprietäre Protokolladresse für ein Objekt kodiert ist, und gleichzeitig ein IIOP-Profile angeben, das beschreibt, wie das gleiche Objekt über IIOP von außen erreichbar ist.

G.7 Inter-ORB-Kommunikation (3)

- IOR-Profil für IIOP
  - Eintragung eines Profils
    - Typ des Objekts: Interface-Repository-ID
    - Hostname des POA
    - TCP/IP-Portnummer des POA
    - Name des POA
    - ObjektID: eindeutiger Bezeichner innerhalb des POA
- Gängige Implementierung in Standard-ORBs
  - Nutzung von IIOP auch innerhalb des ORBs
  - IOR wird immer als eindeutiger Objektbezeichner benutzt

G.8 CORBA Services

- Basisdienste eines verteilten Systems – Erweiterungen des ORB

`Collection Service`	`Persistent Object Service`
`Concurrency Service`	`Property Service`
`Enhanced View of Time`	`Query Service`
`Event Service`	`Relationship Service`
`Externalization Service`	`Security Service`
`Naming Service`	`Time Service`
`Licensing Service`	`Trading Object Service`
`Life Cycle Service`	`Transaction Service`
`Notification Service`	

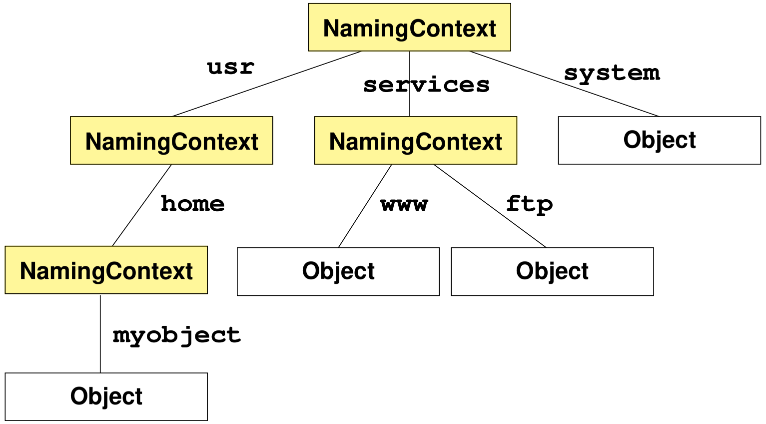
- Services sind über IDL-Schnittstellen aufrufbar
  - ◆ Keine zusätzlichen Konstrukte erforderlich; Methodenaufruf reicht aus

1 Naming Service

- CORBA definiert einen hierarchischen Namensdienst ähnlich dem UNIX Dateinamensdienst
  - ◆ Namensraum ist baumförmig
  - ◆ Name besteht aus mehreren Komponenten (Silben - syllables)
    z. B. < "usr"; "home"; "myobject" >
    (Entsprechung im Dateisystem wäre "**`/usr/home/myobject`**")
  - ◆ Schnittstelle des Namensdienstes ist in IDL beschrieben
    - ➤ ansprechbar als CORBA-Objekte

1 Namensdienst (2)

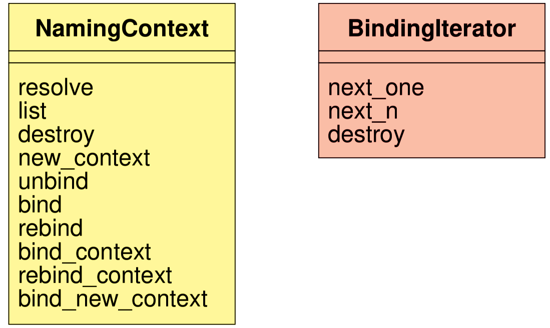
- Beispielbaum



- In diesem Baum gibt es beispielsweise den Namen
  < "usr" ; "home" ; "myobject" >
- Dieser Name besteht aus drei Komponenten. In unserer Notation wurden sie mit Strichpunkten abgetrennt. Es ist zu beachten, dass CORBA keine solche Notation definiert. Es ist lediglich festgelegt, dass der Name aus Komponenten besteht. Eine andere Notation könnte sein
  /usr/home/myobject
  wie aus UNIX bekannt.
- Bei der Auflösung dieses Namens erlangt man die Objektreferenz auf das entsprechende Objekt.
- Die Referenz auf den root-NamingContext bekommt man über eine Abfrage beim ORB
  `orb.resolve_initial_references("NameService");`

1 Namensdienst (3)

- Kontext- und Iteratorschnittstelle



➤ beliebige Objekte können gebunden (**`bind`**) und mit Namen wieder gesucht werden (**`resolve`**)

➤ Auflistung erfolgt durch Iterator-Pattern (**`list`**)

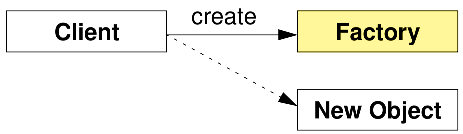
- Die Klasse NamingContext stellt so etwas wie ein Verzeichnis oder Directory dar. Man kann Namenskomponenten definieren und diese entweder mit einem Objekt oder einem anderen Kontext verbinden. Im Falle eines Kontexts entsteht der besagte Baum.
- Beim Auflisten eines Kontexts kann eine maximale Anzahl von Namenseinträgen angegeben werden, die der Aufrufer haben möchte. Überbleibende Einträge werden dann in einen BindingIterator gepackt, der zusätzlich zurückgegeben wird. Über diesen kann man dann auf die folgenden Einträge zugreifen.

2 Life Cycle Service

- Lebenszyklus eines Objekts
  - Erzeugung
  - Kopieren, Verlagern
  - Löschen
- Life Cycle Service definiert eine gemeinsame Schnittstelle für Operationen während des Lebenszyklus

★ Modell des Lebenszyklus

- Erzeugung eines Objekts:

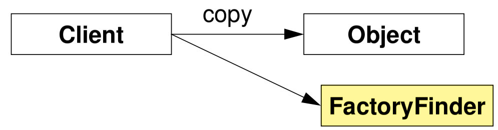


Aufruf einer create-Methode an einem Factory-Objekt

- Die Factory wird aufgerufen, erzeugt ein neues Objekt und gibt dessen Objektreferenz an den Aufrufer zurück.
- Das Protokoll bzw. das Interface der Factory ist in der Regel abhängig vom zu erzeugenden Objekt und nicht festgelegt.

2 Life Cycle Service (2)

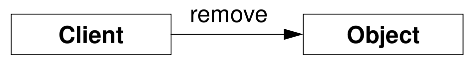
- Kopieren und Verlagern:



FactoryFinder bestimmt den Ort

- Client kennt einen FactoryFinder
- Der FactoryFinder bestimmt den Ort und oder die Region, in der die Kopie erzeugt wird bzw. in die das Objekt verlagert werden soll.
- Orte könnten sein: ein bestimmter Rechner, eine bestimmte Rechnergruppe etc.

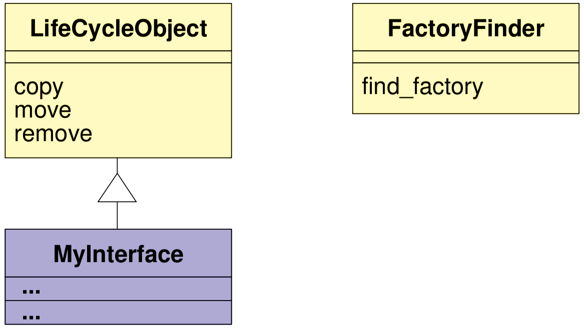
- Löschen:



- Mit dem Aufruf von remove löscht sich das Objekt.

2 Life Cycle Service (3)

- Objekte müssen das Interface LifeCycleObject implementieren



- ◆ **copy** und **move** benötigen ein **FactoryFinder**-Objekt um ein **Factory**-Objekt zu finden, das dann die Kopie bzw. das verlagerte Objekt erzeugt
- ◆ **remove** löscht ein Objekt

2 Life Cycle Service (4)

- Implementierung am Beispiel copy
  - ◆ Client ruft copy-Methode auf und übergibt einen FactoryFinder
  - ◆ Die copy-Methode stellt entweder der Programmierer des Objekts oder die CORBA-Implementierung bereit
  - ◆ Die copy-Methode ruft den FactoryFinder auf, sucht eine passende Factory aus und erzeugt mit ihrer Hilfe ein neues Objekt.
  - ◆ Das neue Objekt wird mit den Daten des bestehenden Objekts initialisiert.
- ▲ Nach aussen klare Schnittstelle
- ▲ Nach innen offen und implementationsabhängig,
  z. B. Protokoll zwischen copy-Methode und Factory
- In CORBA 2.0 Berücksichtigung von Teil-Ganze-Beziehungen und ähnlichen
  (Deep Copy und Shallow Copy)
  - Die Beziehungen zwischen Objekten können registriert werden. Bei einer Kopier- oder Löscheoperation wird dann ein ganzer Objektgraph kopiert oder gelöscht.

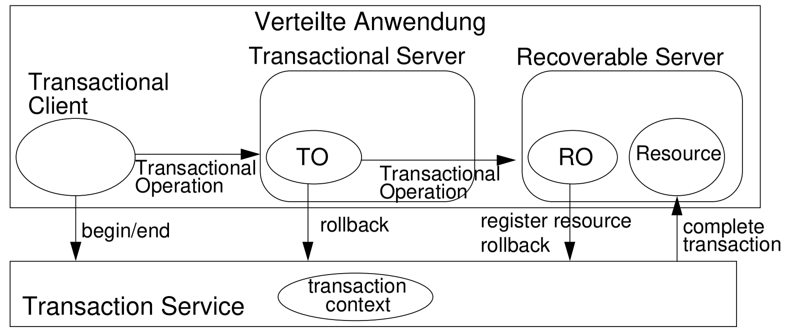
3 Object Transaction Service (OTS)

- Transaktionen: "ACID":
    - atomic (alles oder nichts)
    - consistent (Neuer Zustand erfüllt Konsistenzbedingungen)
    - isolated (Isolierung: keine Zwischenzustände sichtbar)
    - durable (Persistenz)
- Eine Transaktion => mehrere Objekte, mehrere Requests: Bindung an einen *"Transaction context"*
    - ◆ wird normalerweise *implizit* an alle Objekte weitergereicht
    - ◆ auch explizite Weitergabe durch Client möglich (Spez. in IDL)
- Typischer Ablauf:
    - ◆ Beginn der Transaktion erzeugt *Transaction context* (an den Client-Thread gebunden)
    - ◆ Ausführung von Methoden (implizit an die Transaktion gebunden)
    - ◆ Schliesslich: Client beendet die Transaktion (commit/roll back)

G.8 CORBA Services

3 Object Transaction Service (OTS) (2)

- Bestandteile einer Anwendung, die vom OTS unterstützt wird:
    - ◆ Transactional Client
    - ◆ Transactional Objects (TO)
    - ◆ Recoverable Objects (RO)
    - ◆ Transactional Servers
    - ◆ Recoverable Servers



3 Object Transaction Service (OTS) (3)

- Zugriff auf Transaction Service über **`Current`**-Objekt
  - **`orb.resolve_initial_reference("TransactionCurrent");`**

```
interface Current : CORBA::Current {
   void begin() raises(SubtransactionsUnavailable);
   void commit(in boolean report_heuristics)
      raises(NoTransaction,HeuristicMixed,HeuristicHazard);
   void rollback() raises(NoTransaction);
   void rollback_only() raises(NoTransaction);

   Status get_status();
   string get_transaction_name();
   void set_timeout(in unsigned long seconds);

   Control get_control();
   Control suspend();
   void resume(in Control which) raises(InvalidControl);
};
```


3 Object Transaction Service (OTS) (4)

- *Transaction Context*: **`Control`**-Objekt

```
interface TransactionFactory {
   Control create(in unsigned long time_out);
   Control recreate(in PropagationContext ctx);
};

interface Control {
   Terminator get_terminator() raises(Unavailable);
   Coordinator get_coordinator() raises(Unavailable);
};

interface Terminator {
   void commit(in boolean report_heuristics) raises(...);
   void rollback();
};
```

4 CORBA Services - Zusammenfassung

- Von OMG standardisierte Spezifikationen von Dienste für Probleme, die in verteilten Systemen häufig auftreten
- Spezifikationen legen fest:
  - Immer: Einheitliche Schnittstelle (IDL)
  - Meistens: generelle Vorgehensweise bei der Problemlösung
  - Manchmal: konkrete Strategien (oft auch offen gelassen bzw. implementierungsabhängig)
- Weitere Dokumentation/Informationen:
  - http://www.omg.org/technology/documents/formal/corbaservices.htm
  - http://developer.java.sun.com/developer/onlineTraining/corba/corba.html