G Fragmentierte Objekte

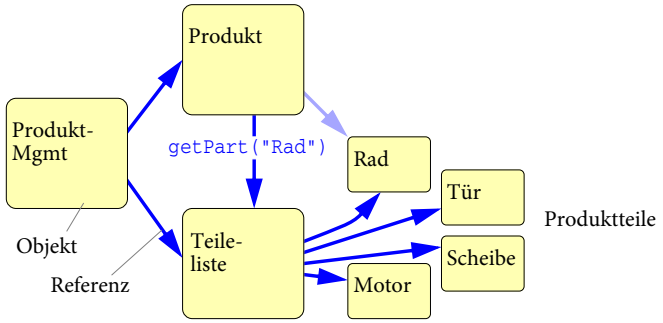
- Architektur 1- monolithische verteilte Objekte (Wiederholung)
- Architektur 2 - fragmentierte verteilte Objekte
- Aspectix - fragmentierte Objekte und CORBA

- Vorlage: AVO-Vorlesung Prof. Franz Hauck (Ulm)

1 Architektur (1)

1.1 Verteiltes objektbasiertes Programmiermodell

- Anwendung
  - Menge von kooperierenden Objekten



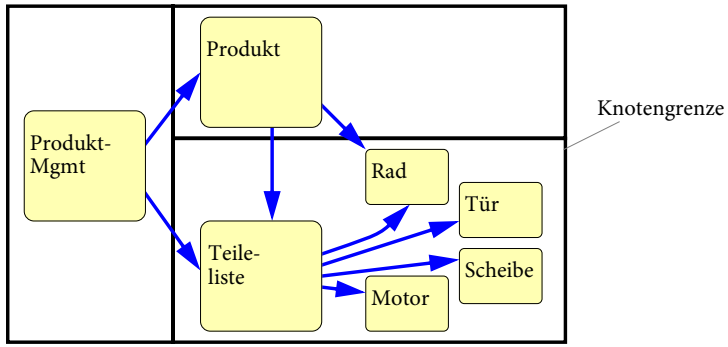
  - Interaktion über Methodenaufrufe
  - Austausch und Weitergabe von Objektreferenzen

1 Übergang zur Verteilung

- Objektbasierte Anwendung
  - Beibehalten der Designstruktur
  - Identifikation von Verteilungseinheiten
  - konkrete Verteilung für einen Anwendungslauf
- Wünschenswerte Transparenz
  - zugriffstransparenter Methodenaufruf
  - zugriffstransparente Parameterübergabe
  - ortstransparenter Methodenaufruf

2 Monolithisches Objektmodell

- Vorgehensweise bei Java RMI
  - Beispiel einer Verteilung auf drei Knoten:



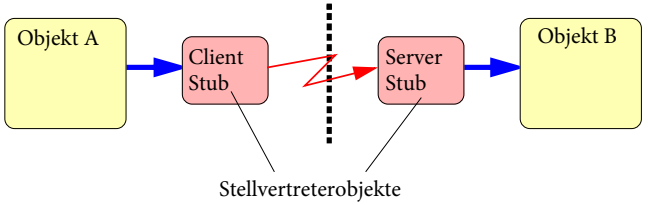
  - Ein Objekt liegt immer vollständig auf einem Knoten (monolithisches Objekt).

3 Entfernte Referenzen

- Beobachtung
  - Objektreferenzen sind lokal oder entfernt
- Zugriffstransparenz
  - lokaler Methodenaufruf soll wie entfernter Methodenaufruf aussehen
  - Anleihe beim RPC: Stellvertreterobjekte
  - bei Java RMI: Stubs und Skeletons

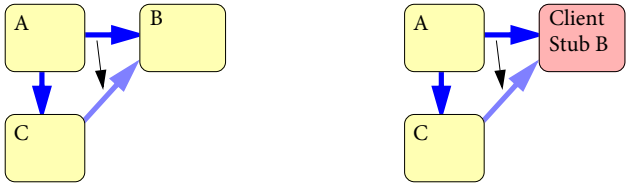
3 Entfernte Referenzen (2)

- ★ Lokales Stellvertreterobjekt
  - lokale Referenz auf Stellvertreterobjekt entspricht entfernter Referenz (Zugriffstransparenz)
  - Stellvertreter besitzt entsprechende Stubmethode für jede Objektmethode
    - Marshalling, Unmarshalling
    - Kommunikation mit verteiltem Objekt
  - Stellvertreter kennt aktuelle Kommunikationsadresse des verteilten Objekts (Ortstransparenz) und das Aufrufprotokoll



4 Referenzweitergabe

- Lokale Weitergabe
  - lokale Sprachreferenz wird immer weitergegeben



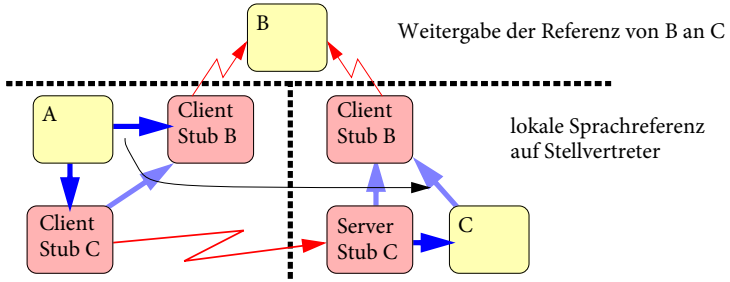
  - Semantik für lokale und entfernte Objektreferenzen gleich

4 Referenzweitergabe

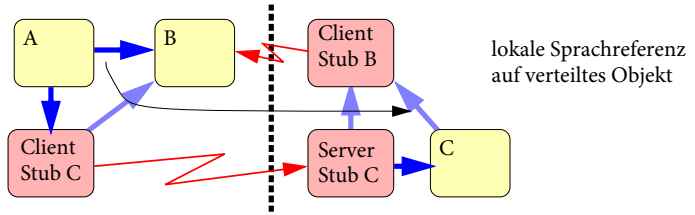
- Entfernte Weitergabe
  - Eingriff beim Marshalling
  - Weitergabe entfernter Objekte
    1. lokale Referenz auf ein Stellvertreterobjekt
    2. lokale Referenz auf ein lokal liegendes verteiltes Objekt
    - Java RMI: Serialisierung des Stubobjekts
      – enthält eindeutige Kommunikationsadresse und lokal gültigen Objektidentifikator
      – enthält Aufrufprotokollimplementierung
    - allgemein: Marshalling eines global gültigen Objektbezeichners (z.B. eindeutige Kommunikationsadresse und lokale Objekt-ID o.ä.)
  - Weitergabe lokaler Referenzen auf nichtverteiltes Objekt
    - Java RMI: Kopie serialisierbarer Objekte
    - allgemein: systemabhängiges Verhalten: z.B. „nicht möglich“ oder „wird kopiert“

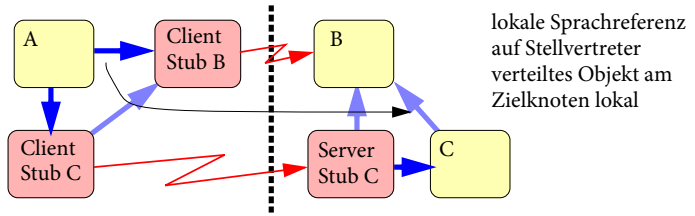
4 Referenzweitergabe (2)

- Entfernte Weitergabe (fortges.)
  - Eingriff beim Unmarshalling
    - aus übertragener Information muss Objektreferenz abgeleitet werden
    - Erzeugen eines neuen Stellvertreters
    - Referenz auf bereits bekannten Stellvertreter
    - Referenz auf lokal liegendes verteiltes Objekt (optional)



4 Referenzweitergabe (3)





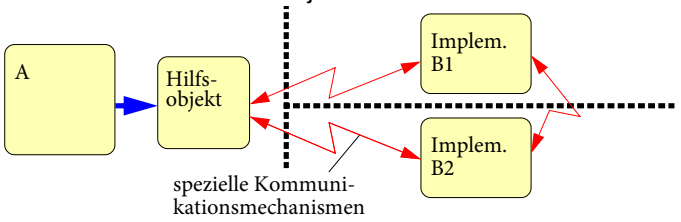
2 Architektur (Teil 2)

2.1 Probleme des monolithischen Ansatzes

- Fixierung auf RPC-basierte Kommunikation
  - Aufrufe werden transparent bis zum verteilten Objekt geleitet
- Beispiel: Fehlertoleranz
  - Objektreferenz verweist auf mehrere verteilte Einheiten
  - evtl. Multicast-Kommunikation
- Beispiel: Multimedia-Server
  - stromorientierte Kommunikation evtl. mit Bandbreitengarantie
  - z.B. Live-Radiodienst

1 Fehlertoleranz

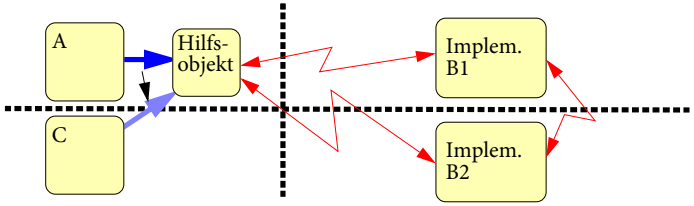
- Monolithischer Ansatz
  - Objektreferenz kann nur auf eine Instanz eines verteilten Objekts verweisen
  - fehlertolerantes Objekt: mehrere Referenzen → keine Zugriffstransparenz
- Abhilfe: Einführen von Hilfsobjekten



  - transparenter Zugriff auf fehlertolerantes Objekt
  - Fehlertoleranz gekapselt im Hilfsobjekt (Fassade)

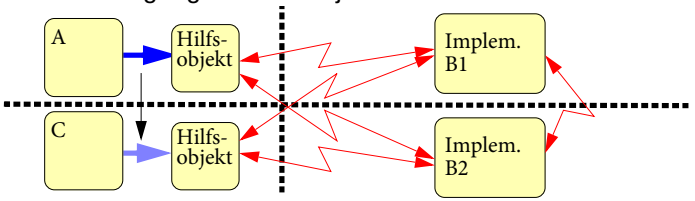
1 Fehlertoleranz (2)

- Fehlertoleranz bei Referenzweitergabe
  - Weitergabe der Referenz auf lokales Hilfsobjekt problematisch
  - Verlust der Fehlertoleranz



1 Fehlertoleranz (3)

- Abhilfe: Erzeugung des Hilfsobjekts



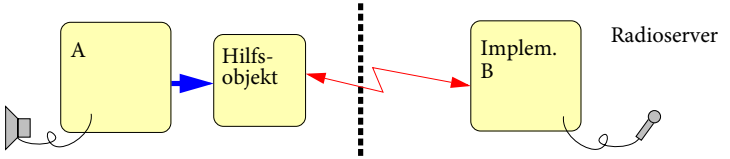
  - bei Referenzweitergabe neues Hilfsobjekt beim Empfänger
- Erzeugung eines weiteren lokalen Hilfsobjekts nicht zugriffstransparent
  - als Ergebnis bzw. Parameters eines Methodenaufrufs nicht transparent nutzbar

1 Fehlertoleranz (4)

- Mögliche Lösung
  - Integration in Middleware (vgl. FT CORBA)
    - Einführung fehlertoleranter Objekte mit speziellen Kommunikationsmustern
    - automatische Generierung von speziellen Stellvertretern
- Problem
  - verschiedene Varianten von Fehlertoleranz bedürfen verschiedener Integrationen

2 Multimedia-Server

- Monolithischer Ansatz
  - mit Server kann nur über RPC-basiertem Methodenaufruf kommuniziert werden
  - nicht realisierbar?
- Einführen von Hilfsobjekten



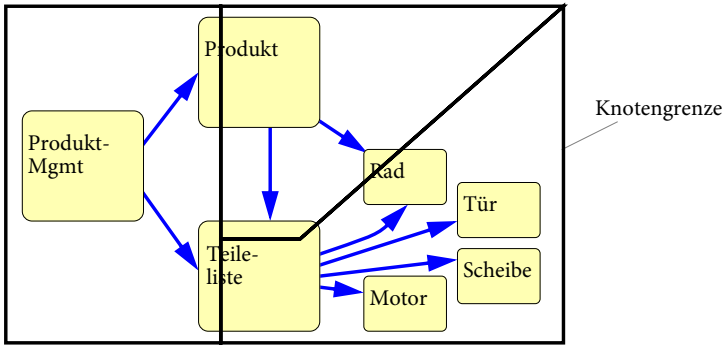
  - transparenter Zugriff auf Multimedia-Server
  - Multimedia-Protokolle gekapselt im Hilfsobjekt

2 Multimedia-Server (2)

- ▲ Referenzweitergabe nicht transparent
  - ◆ Weitergabe der Referenz auf Hilfsobjekt bringt nicht geforderte Kommunikationsleistung
  - ◆ Erzeugung eines neuen lokalen Hilfsobjekts nicht transparent
- ★ Mögliche Lösung
  - ◆ Integration in die Middleware
  - ◆ Einschubmodule für Kommunikationsprotokolle
    - Stellvertreter lassen sich andere Protokolle unterschieben
  - ◆ anwendungsabhängige Kommunikation schwierig realisierbar

2.2 Fragmentierter Ansatz

- Verteilungsschnitte durch Objekte



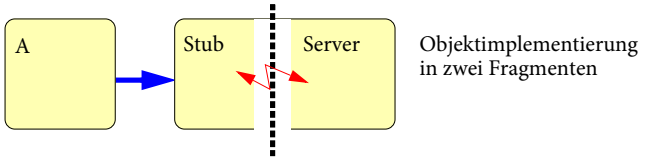
  - ◆ Referenzen sind immer lokal
  - ◆ verteilte Objekte sind an mehreren Orten präsent (Fragmente)

1 Referenzen

- Beobachtung
  - ◆ Verteiltes Objekt ist überall dort durch Fragment präsent, wo lokale Referenzen auf es existieren
  - ◆ es existieren keine entfernten Referenzen
- Zugriffstransparenz
  - ◆ immer lokaler Aufruf im lokalen Fragment
  - ◆ Kommunikation zwischen den Fragmenten ist gekapselt, d.h. hinter Schnittstelle verborgen
- Ortstransparenz
  - ◆ lokales Fragment kennt andere Fragmente
  - ◆ Client braucht deren Orte nicht zu wissen

2 Client-Server-Objekt

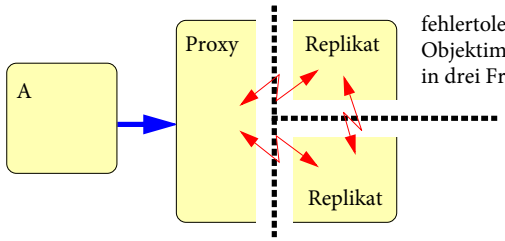
- Zwei Fragmenttypen
  - ◆ Serverfragment (einmal vorhanden)
  - ◆ Stubfragment (beliebig häufig vorhanden)
- Interaktion
  - ◆ Stubfragment führt RPC-basierte Kommunikation mit Serverfragment durch (ähnlich Stellvertreterobjekte)



  - ◆ identisches Vorgehen wie beim monolithischen Ansatz

3 Fehlertolerantes Objekt

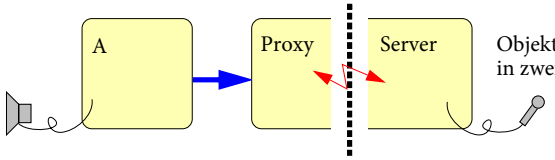
- Zwei Fragmenttypen
  - Replikatfragment (mehrfach vorhanden)
  - Proxyfragment (beliebig häufig vorhanden)
- Interaktion
  - Proxy spricht Replikate an
  - Transparenz der Fehler und der Fehlertoleranz



fehlertolerante Objektimplementierung in drei Fragmenten

3 Multimedia-Server

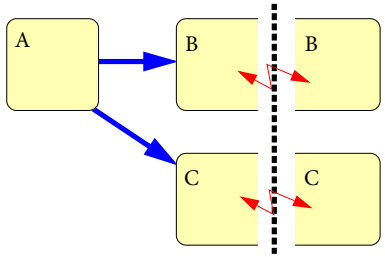
- Zwei Fragmenttypen
  - Serverfragment (einmal vorhanden)
  - Proxyfragment (beliebig häufig vorhanden)
- Interaktion
  - Proxyfragment erhält Multimediadaten über spezielles Protokoll vom Server
  - Transparenz der benutzten Protokolle



Objektimplementierung in zwei Fragmenten

4 Referenzweitergabe

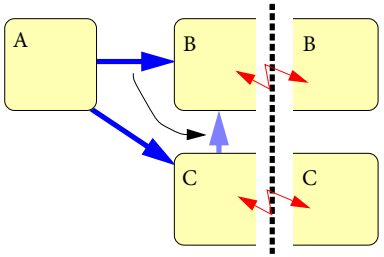
- Übergabe einer lokalen Referenz an das lokale Fragment



Objektimplementierungen in zwei Fragmenten

4 Referenzweitergabe

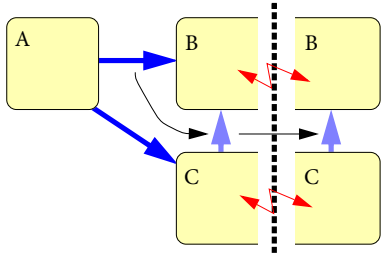
- Übergabe einer lokalen Referenz an das lokale Fragment



Objektimplementierungen in zwei Fragmenten

4 Referenzweitergabe

- Übergabe einer lokalen Referenz an das lokale Fragment



Objektimplementierungen in zwei Fragmenten

- Kommunikation zwischen Fragmenten
  - ◆ Ableiten des Objektbezeichners aus lokaler Referenz
  - ◆ Übertragung des Objektbezeichners
  - ◆ Erzeugen eines neuen lokalen Fragments auf der Empfängerseite

5 Erzeugung lokaler Fragmente

- Ableitung des lokalen Fragments aus Objektbezeichner
  - ◆ lokales Fragment ist objektspezifisch nicht schnittstellenspezifisch (!)
  - ◆ direkte Kodierung der Fragmentimplementierung im Objektbezeichner (statisch)
    - z.B. Klassenname
    - „Smart Proxy“
  - ◆ Befragung eines externen Dienstes, welche Implementierung instanziiert werden soll (dynamisch)
    - z.B. für Anpassung des Objektverhaltens an Vertrauenswürdigkeit, Bandbreiten, Leistungsfähigkeit o.ä. des lokalen Rechensystems
  - ◆ evtl. dynamisches Laden von Code notwendig

5 Erzeugung lokaler Fragmente (2)

- Bestehendes lokales Fragment
  - ◆ lokale Referenz auf bestehendes Fragment kann verwendet werden
    - interne Tabelle listet Fragmente und deren Objektbezeichner
  - ◆ neues Fragment wird erzeugt
    - mehrere lokale Fragmente zum selben Objekt

6 Kommunikation zwischen Fragmenten

- Fragmente eines Objekts sollten sich kennen
  - ◆ Kodierung von Kommunikationsadressen im Objektbezeichner (statisch)
  - ◆ Ortsdienst *(Location Service)*
    - liefert über Objektbezeichner Kommunikationsadressen laufender Fragmente (dynamisch)
    - Fragmente registrieren sich im Ortsdienst
    - ähnlich Namensdienst, jedoch andere Abstraktionsebene

2.3 Probleme des Fragmentierten Ansatzes

- Komplexeres Programmiermodell
  - Zugriffs- und Ortstransparenz von außen
  - keine automatische Kommunikation innen
  - Entwickler muss Kommunikation selbst implementieren
    - Implementierung mehrere zusammenarbeitender Fragmentimplementierung
    - Initialisierung von Ortsdienst, Diensten für dynamische Code-Laden etc.
- ★ Unterstützung durch Middleware
  - Vereinfachung häufiger Entwicklungsprozesse durch Automatisierung
  - Code-Generatoren
    - z.B. für RPC-basierte Fragmentkommunikation

2.4 Anforderungen an Middlewaresysteme

- Objektbasierte Middleware mit fragmentiertem Objektmodell
  - weltweit eindeutige Objektbezeichner zum Parametertransport zwischen Fragmenten
    - Erzeugung lokaler Fragmente aus Objektbezeichner
    - evtl. dynamisches Laden von Fragment-Code
  - Erzeugung neuer verteilter Objekte
  - Namensdienst zum Finden von Objekten
  - Ortsdienst zum Finden von Fragmenten
  - Kommunikationsdienste für Interfragmentkommunikation
    - Implementierung von adäquaten Protokollen z.B. RPC-basierte Aufrufprotokolle
  - Code-Generatoren zur Unterstützung der Fragmententwicklung

3 Aspectix

3.1 Projekt

- Universität Erlangen-Nürnberg, Universität Ulm
  - Projekt aus verschiedenen Einzelförderungen
- Ziel
  - Integration nichtfunktionaler Eigenschaften in verteilte Anwendungen
  - qualitätsbewusste verteilte Anwendungen
    - Qualitätsvorgabe an der Aufrufschnittstelle
    - Qualitätsgarantie einer Anwendungskomponente
    - Qualitätskontrolle in einer Anwendungskomponente
  - „große“ verteilte Anwendungen
- Im Rahmen der Vorlesung wird die Realisierung des fragmentierten Objektmodells vorgestellt

3.2 Fragmentierte Objekte

- Basis
  - fragmentierter Ansatz gut geeignet für Integration allerlei nichtfunktionaler Eigenschaften
  - CORBA als verbreite Verteilungsplattform (Middleware) für verteilte objektbasierte Anwendungen
- ★ Integration von CORBA und fragmentiertem Ansatz
  - Fragmente statt Servants und Stellvertreter
  - „Fragmentierungstransparenz“ für Aufrufer
    - Fragmente müssen für Aufrufer nicht sichtbar werden
- Prototypimplementierung
  - Java-basierte Implementierung
  - andere Sprachen denkbar

1 CORBA-Objektreferenzen

- Externe Referenzen: IOR
  - Profile lassen Spielraum für Zusatzinformationen
- Interne Referenzen: sprachabhängiger Verweis auf Stellvertreterobjekt
  - Verweis muss bestehen bleiben
  - in CORBA verschiedene Stellvertreter für ein Objekt lokal möglich
    - mehrere Stellvertreter mit gleichem Typ (z.B. als weiteres Ergebnis eines Aufrufs)
    - mehrere Stellvertreter mit verschiedenem Typ (z.B. als Ergebnis eines `narrow`-Aufrufs)
    - alle Objektreferenzen auf das selbe Objekt sind gleichwertig
- Interne Referenzen in Aspectix
  - müssen jeweils auf ein lokales Objektfragment verweisen

1 CORBA-Objektreferenzen

- CORBA-Objektreferenz gleich Fragment?
  - unterschiedliche Typen
    - Typ der Objektreferenz
    - Typ des Fragments
    - Typ des Objekts
    - es muss gelten: Referenztyp `<o` Fragmenttyp `<o` Objekttyp (`<o` entspricht Subtypbeziehung)
  - Austauschbarkeit des Fragments
    - Fragment soll zur Laufzeit transparent austauschbar sein (z.B. für Anpassung der Objektimplementierung an neue Umgebungsbedingungen oder neue Anforderungen)
    - Rückruf von Objektreferenzen?
  - NEIN!

1 CORBA-Objektreferenzen (2)

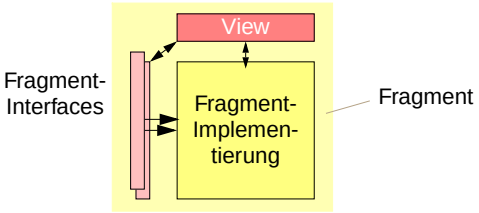
- Fragment-Interface
  - entspricht der CORBA-Objektreferenz
  - generische und typabhängige Implementierung
  - automatisch generiert durch IDL-Compiler
  - leitet Methodenaufrufe an eine Fragment-Implementierung weiter (vgl. lokal optimierter CORBA-Stub)
- Fragment-Implementierung
  - eigentlicher Code eines Fragments
- ▲ Koordinierung eines Austauschvorgangs?
  - Einführung sogenannter Views

2 Views

- Objektreferenzen sind nicht mehr gleichwertig!
  - „objektsemantisch“ gleichwertig, da immer mit dem selben Objekt verbunden
  - „fragmentsemantisch“ nur gleichwertig, wenn mit dem selben Fragment verbunden
- Unterscheidung
  - Beispiel: Fehlertoleranz
    - unterschiedliche Proxyfragmente müssen individuell für Fehlererholung sorgen, z.B. Umschaltung auf anderes Primärreplikat
    - Objektreferenzen auf das selbe Proxyfragment teilen sich Fehlererholung
- ★ Unterscheidung wird durch View greifbar
  - View ist eine Sicht auf ein verteiltes Objekt

2 Views (2)

- Views werden durch jeweils ein Fragment repräsentiert
  - interne Repräsentation durch ein View-Objekt



  - View-Objekt ist der statische Teil eines Fragments
    - Interfaces kommen dazu (z.B. durch narrow oder Parameter)
    - Interfaces verschwinden (z.B. durch Garbage-Collection)
    - Fragmentimplementierung wird ausgetauscht

2 Views (3)

- Zentrale Aufgaben des View-Objekts
  - interne Speicherung der Objekt-IOR
  - Koordination eines Fragmentaustausches
  - Verwaltung der Fragment-Interfaces
    - View kennt alle Fragment-Interfaces (d.h. Objektreferenzen)
    - View kann Methodenaufrufe blockieren und freigeben (wichtig für Fragmentaustausch)
  - Verwaltung der Qualitätsanforderungen gegenüber dem View bzw. dem Fragment

3.3 Objekterzeugung

- Factory-Objekt
  - gängiges Entwurfsmuster zur Objekterzeugung (Design-Pattern)
  - Methodenaufruf erzeugt neues Objekt
  - Wie entsteht Objekt tatsächlich?
- CORBA-Objekt entsteht mit Erzeugung der IOR
  - in FT-CORBA entsteht IOGR aus dem „Sumpf“ (ist nicht genau definiert)
  - in Aspectix entstehen IORs über einen Reference-Manager
  - Referenz auf Reference-Manager kann mit
    `resolve_initial_reference( "ReferenceManager" );`
    ermittelt werden
- Zunächst Betrachtung wie Fragmente entstehen ...

3.4 Fragmenterzeugung

- Annahme
  - IOR wird bei Kommunikation übermittelt: lokaler Stellvertreter bzw. lokales Fragment muss erzeugt werden oder
  - IOR wurde in String umgewandelt: `string_to_object` wurde aufgerufen
- Auswertung der IOR
  - Untersuchung der Profile
    - Erkennung von Aspectix-Profilen
    - notfalls Erkennung von IIOP-Profilen
- IIOP-Profil
  - Erzeugung eines CORBA-Stubs (klassische Objektreferenz)
- Aspectix-Profil
  - Erzeugung eines Fragmentes (Fragment-Interface als Objektreferenz)

3.4 Fragmenterzeugung (2)

- Erster Ansatz: View-Verwaltung
  - ist bereits lokales Fragment vorhanden wird nur neues Fragment-Interface erzeugt und zurückgegeben
  - nur ein lokales Fragment pro Objekt möglich
- Besserer Ansatz: Default-View-Verwaltung
  - ist bereits lokales Fragment vorhanden und als Default-View registriert, wird dazu ein neues Fragment-Interface zurückgegeben
  - Vergleich von Views auf Objektreferenzen möglich
  - Umschalten des Default-Views möglich
  - Duplikation einer Objektreferenz mit neuem View (neuem Fragment) möglich
  - Benutzer eines Objekts hat Wahlmöglichkeiten
    (Default entspricht ein lokales Fragment pro Objekt)

3.4 Fragmenterzeugung (3)

- Aspectix-IOR-Profile
  - zwei wesentliche Informationen müssen geliefert werden
    - Welche Fragmentimplementierung?
      (View- und Interface-Objekte sind generisch bzw. nur vom IDL-Typ abhängig)
    - Welche Kommunikationsadresse für Interfragmentkommunikation?

1 Spezifikation der Fragmentimplementierung

- Codierung des Implementierungstyps in der IOR
  - z.B. interessant für intelligente Proxies wie Radio-Dienst und statische Fehlertoleranz
  - Implementierungstyp kann nicht mehr verändert werden
  - Instanz der Implementierung wird erzeugt
- Zwei Spezifikationsvarianten
  - direkte Codierung: z.B. Klassenname
    - Problem der Sprachabhängigkeit
  - indirekte Codierung: z.B. eindeutiger Bezeichner eines Implementierungstyps
    - Dienst notwendig zum dynamischen Laden der Implementierung
    - in Aspectix: DLS (Dynamic Loading Service)
    - sprachunabhängig

1 Spezifikation der Fragmentimplementierung (2)

- Problem: Fragmentimplementierung soll dynamisch sein
  - z.B. dynamische Fehlertoleranz: Aufrufer soll evtl. auch ein Replikatfragment statt einem Proxy-Fragment bekommen
- Lösung: indirekte Codierung der Fragmentimplementierung in der IOR
  - Codierung eines Entscheidungsbezeichners in der IOR
  - Anfrage an einen Entscheidungsdienst → Rückgabe eines Implementierungstyps (direkt oder indirekt)
- Entscheidungsdienst (Policy Decision Service, PDS)
  - externer Dienst für strategische Entscheidungen
  - Entscheidung kann dynamisch in Abhängigkeit von Parametern fallen
    - z.B. Replikatfragment nur wenn sichere Umgebung und effizienter Rechner

2 Spezifikation der Kommunikationsadresse

- Direkte Codierung in der IOR
  - eine oder mehrere Kommunikationsadressen werden in IOR codiert
  - Protokoll und Adresse ist jeweils beliebig
  - Kommunikationsadressen sind statisch
    - z.B. einsetzbar bei Radiodienst
  - Änderung der IOR ähnlich IOGR bei FT-CORBA aufwändiger Vorgang
- Indirekte Codierung in der IOR
  - Codierung eines eindeutigen Objektbezeichners (in Aspectix sowieso Bestandteil der IOR)
  - Anfrage an einen Ortsdienst (Location Service)
    - Rückgabe von aktuellen Kommunikationsadressen
    - bei Abbruch der Verbindung erneute Anfrage
  - einsetzbar z.B. für fehlertolerante verteilte Objekte

3.5 Objekterzeugung

- Nutzung des Reference-Mangers
  - Erzeugung einer leeren IOR
  - Anfrage nach Profile-Managern (z.B. für IIOP-Profil, Aspectix-Profil ...)
  - Profile-Manager ist genau für ein Profil verantwortlich
    - kann Profil erzeugen und in eine IOR integrieren
  - Anwendung (z.B. Factory) konsultiert Profile-Manager
    - profilabhängige Schnittstelle zum Profile-Manager
    - z.B. für Anmeldung dynamisch ladbarer Code-Module etc.
  - nach Erzeugung der IOR muss sichergestellt werden, dass Fragmenterzeugung funktioniert

3.5 Objekterzeugung (2)

- Indirekte Objekterzeugung
  - Initialisierung externer Dienste
    - Dynamic-Loading-Service
    - Policy-Decision-Service
  - Erzeugung einer neuen IOR mit Hilfe des Reference-Managers und der Profile-Managern
  - erste Fragmenterzeugung durch Umwandeln der IOR in eine allererste Objektreferenz
- Initialisierung eines Location-Service
  - automatische Registrierung durch neue Fragmente
  - Abfrage durch Fragmente

3.5 Objekterzeugung (3)

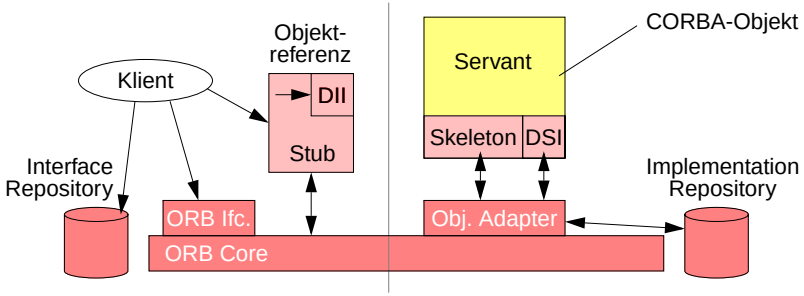
- Direkte Objekterzeugung
  - Erzeugung einer ersten Fragmentimplementierung
  - Initialisierung der Kommunikationsmöglichkeiten
    - z.B. Radiodienst: Initialisierung der Serverseite
  - Initialisierung externer Dienste (falls nötig)
  - Erzeugung einer IOR
    - Codierung der Kommunikationsadresse in die IOR
  - Herausgabe der IOR bzw. von Objektreferenzen
  - Umwandlung bisheriger Fragmentimplementierungen in Fragmente
  - vgl. mit Aktivierung eines Servants
- CORBA-kompatible Objekterzeugung
  - Erzeugung eines Servant
  - Aktivierung am POA

3.6 CORBA-Kompatibilität

- Aufrufer sieht keinen Unterschied zu normalen CORBA-Objekten
  - Objektreferenzen sind austauschbar
  - Views und damit Fragmente müssen nicht sichtbar sein (können aber)
  - Parameterübergabe und IDL-Beschreibung sind identisch
- CORBA-Objekte und Aspectix-ORB
  - Quellcode voll portabel
  - externe CORBA-Objekte über IIOP transparent ansprechbar
- Aspectix-Objekte und CORBA-ORB
  - Aspectix-Objekt auf normalem CORBA-ORB nicht lauffähig
  - externe Aspectix-Objekte über IIOP transparent ansprechbar
    - z.B. fehlertolerante Aspectix-Objekte enthalten IIOP-Profiles für Gateways

3.7 Architekturüberblick

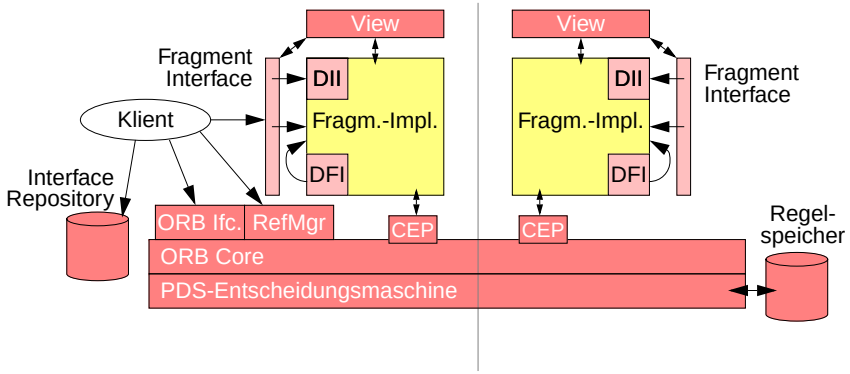
- Zur Erinnerung: CORBA-ORB-Architektur (etwas verändert dargestellt)



- Markierung wer die Softwarekomponenten bereitstellt

3.7 Architekturüberblick (2)

- Aspectix-ORB-Architektur



3.8 Zusammenfassung

- Aspectix hat den fragmentierten Ansatz
  - weltweit eindeutige Objektbezeichner zum Parametertransport zwischen Fragmenten
    - IOR mit IIOP- oder Aspectix-Profil
    - Aspectix-Profil identifiziert erforderliche Fragmentimplementierung
    - evtl. dynamisches Laden von Fragment-Code durch Dynamic-Loading-Service
    - Erzeugung lokaler Fragmente mit Fragment-Interface, Fragment-Implementierung, View-Objekt